Midterm 2 Solutions

Important note. This exam was a good deal harder than I'd intended it to be. The mean mark was around 45% and only one student scored above 70%.

Grading notes. In all cases, full marks for any correct solution. Subtract 2 marks for mistakes that aren't too big and don't affect the overall solution.

Question 1

The key point here about the language L is that, if $\langle M \rangle \in L$, then whenever M halts, it correctly tells you whether its input is in HALT.

a) We reduce the halting problem to L as follows. Given a string x, let M_x be the Turing machine that accepts x and loops for all other inputs. This machine clearly exists and a description of it can be computed from x.

Now, let T be any Turing machine and let w be an input. $M_{\langle T \rangle w}$ approximates the halting problem if, and only if, T halts on input w (since it accepts $\langle T \rangle w$, so $T(w)$ must halt; it loops for all other inputs so says nothing about their halting behaviour). Therefore, $\langle T \rangle w \in \text{HALT}$ if and only if $M_{\langle T \rangle w} \in L$.

Grading notes. Score around 8 for a reasonable attempt. Subtract 3 marks if the answer assumes that a machine that loops on all inputs is not in L. Subtract 1 mark for describing the reduction as being *to*, e.g., HALT when it's actually *from* it.

Comments. Several students tried to do a reduction by converting $\langle M \rangle w$ to $\langle M_w \rangle$, where M_w is the TM that deletes its input, replaces it with w, and then acts like M. This usually doesn't work because it produces a machine that either accepts every input, rejects every input, or loops on every input. The only TM in L that has this property is the machine that loops for every input, which is in L because it never gives a false answer to the halting problem, because it never gives any answer at all. Thus, the only way that I can see to use M_w is to reduce $\overline{\text{HALT}}$ to L using the fact that $M_w \in L$ if, and only if, M does not halt on w. In fact, reducing $\overline{\text{HALT}}$ to L would also prove non-semi-decidability for part b).

b) Similarly reduce $\overline{\text{HALT}}$ to L. To do this, we let M_x be the machine that *rejects* x and loops on any input different from x. Now, consider a string x. $x \in \overline{\text{HALT}}$ if, and only if either x is not of the form $\langle T \rangle w$ or $x = \langle T \rangle w$ for some Turing machine T that loops on input w. The second case occurs if, and only if, $M_{\langle T \rangle w} \in L$. $\overline{\text{HALT}}$ is not semi-decidable by Corollary 42, so L is not semi-decidable by Lemma 46.

Alternatively, we can reduce $\text{HALT}_\forall$ to L as follows. Given an encoding $\langle M\rangle$ of a Turing machine, we can produce a machine P_M that accepts every input beginning with $\langle M\rangle$ (i.e., every input $\langle M\rangle w$ for some w) and loops on every other input. Now, $P_M \in L$ if, and only if, M halts for every input. $\text{HALT}_\forall$ is not semi-decidable, as (I think) was stated in class.

Grading notes. "Yes"/"No" is not enough: some justification is required to get marks. 5 marks if an intuitive explanation (along the lines of needing to check that the maching has the right behaviour on infinitely many inputs before you can say that $\langle M\rangle \in L$) but a proper proof is what we're looking for.

Question 2

a) Consider a string xy and let $\langle M\rangle v$ and $\langle N\rangle w$ be shortest representations of x and y. We can give a representation of xy informally as follows: "Compute the output of $M(v)$ and append the output of $N(w)$ to that." We want a Turing machine T that takes $\langle M\rangle v$ and $\langle N\rangle w$ and outputs $M(v)N(w)$ but, to do this, we need to know when $\langle M\rangle v$ ends and $\langle N\rangle w$ begins. Observe that $|\langle M\rangle v| \leq |x| + k$ for some constant k. We can specify this in $\log(|x|+k)$ bits but now we have the same problem: we don't know when this number ends and $\langle M\rangle v$ begins! But we can deal with this by, e.g., writing $\log|\langle M\rangle v|$ 0s, followed by a 1, followed by $|\langle M\rangle v|$: this tells the machine how many bits the number takes.

It follows that there is a Turing machine T such that $\langle T\rangle 0^{\log|\langle M\rangle v|}1\text{bin}(|\langle M\rangle v|)\langle M\rangle v\langle N\rangle w$ is a representation of xy. This has length at most

$$\begin{aligned} &|\langle T\rangle| + 2\log(|xy|+k) + 1 + K(x) + K(y) \\ &\leq |\langle T\rangle| + 2\log|xy| + 2\log k + 1 + K(x) + K(y) \\ &= K(x) + K(y) + 2\log|xy| + c\,, \end{aligned}$$

for some constant c. The inequality uses the fact that, for $a, b \geq 1$, $\log(a+b) \leq \log(ab) = \log a + \log b$.

Grading notes. 5 marks for recognizing the basic idea, which is "concatenate the representations"; 2 marks for recognizing that you need to know where the representation of x ends and the representation of y begins; 3 marks for recognizing that log-something bits is relevant to that. For more than 10 marks, it's necessary to take the log of the right thing and show why that's useful.

Comments. Several students tried to say that, if r_x is a representation of x and r_y is a representation of y, then $r_x\#r_y$ is a representation of xy. This supposes that # cannot appear within r_x or r_y since, if they did, how would we know which of the #s in $r_x\#r_y$ was the delimiter? But, if # cannot appear in the representation of x or y, why should it be allowed in the representation of xy? To avoid these problems, we insist that all representations be binary strings. (I suppose you could get around the problem by using up to $2\log|xy|$ bits to encode the number of #s in r_x but, if you're going to do that, you may as well just encode $|r_x|$.)

b) If v is a substring of x, we can write $x = uvw$ which we can consider to be the concatenation of

uv and w. By part a), we have

$$\begin{aligned}|x| \le K(x) &\le K(uv) + K(w) + 2\log|uvw| + c_1 \\ &\le K(u) + K(v) + 2\log|uv| + c_1 + K(w) + 2\log|uvw| + c_1 \\ &\le |u| + |w| + K(v) + 2\log|uv| + 2\log|uvw| + c \\ &\le |x| - |v| + K(v) + 4\log|x| + c\,,\end{aligned}$$

for some constant c, so $K(v) \ge |v| - 4\log|x| - c$.

Grading notes. Similar grading to part a). It's fine to assume the result from that part, even if it wasn't successfully proved. Quite a few people tried to do this by contradiction ("Assume that, for some substring v, $K(v) < |v| - 4\log|x| - c$. Then, ..."). This doesn't seem to work, probably because it commits to a particular value of c too early, but a decent attempt at this will score most of the marks.

Question 3

a) If $\mathbf{P} = \mathbf{NP}$, then every language in $\mathbf{P}$ (i.e., in $\mathbf{NP}$), except for $\emptyset$ and Σ^*, is $\mathbf{NP}$-complete. Let Y be any language in $\mathbf{NP} \setminus \{\emptyset, \Sigma^*\}$. We can choose strings $w_{\text{in}} \in Y$ and $w_{\text{out}} \notin Y$ and reduce any language $X \in \mathbf{NP}$ to Y using the function

$$f(w) = \begin{cases} w_{\text{in}} & \text{if } w \in X \\ w_{\text{out}} & \text{if } w \notin X\,. \end{cases}$$

This function can be computed in polynomial time because we can determine whether $w \in X$ or not in polynomial time by the assumption that $\mathbf{P} = \mathbf{NP}$. Neither $\emptyset$ nor Σ^* can be $\mathbf{NP}$-complete because the definition a function f being a many-one reduction from a language X to $\emptyset$ requires that, for any $w \in X$, $f(w) \in \emptyset$, which cannot happen for $X \neq \emptyset$.

Comments. w_{in} and w_{out} are fixed constants and we don't need to compute them every time we compute f: they would be "hard-coded" into the Turing machine that computes the reduction. Note that, if $Y = \emptyset$ then w_{in} does not exist and, if $Y = \Sigma^*$, then w_{out} does not exist.

Grading notes. The answer given is more detailed than necessary; it's enough to say something like "Every language in $\mathbf{P}$ [or $\mathbf{NP}$] except for $\emptyset$ and Σ^* would be $\mathbf{NP}$-complete because we could solve the problem in the reduction." Subtract 1 mark for missing the exception for $\emptyset$ and Σ^*; subtract 2 marks for no justification. "Every $\mathbf{P}$-complete language would be $\mathbf{NP}$-complete" isn't an answer: it's just a tautology.

b) First, we show that VERTEXCOVER $\in \mathbf{NP}$. This follows because a vertex cover is a succinct certificate. A vertex cover can have at most $|V(G)|$ vertices and, given a set $v_1, \dots, v_\ell$ of vertices, we can check deterministically in polynomial time that $\ell \le k$ and that every edge has at least one of the given vertices as an endpoint.

Now, we prove $\mathbf{NP}$-completeness. Suppose that S is a vertex cover of a graph $G = (V, E)$. Then every edge must have at least one endpoint in S, so no edge has both endpoints in $V \setminus S$, so

$V \setminus S$ is an independent set. It follows that G has a vertex cover of size at most k if, and only if, it has an independent set of size at least $|V| - k$. We reduce IndSet to VertexCover by mapping input G, k to $G, |V| - k$.

Grading notes. 3 marks for proving membership in **NP**. 6 marks for describing any correct reduction and 6 marks for proving that it works. If the reduction doesn't work, around 6 marks if it's at least plausible and a there's a reasonable attempt at a proof. Note that a bare statement such as "Reduce from IndSet" is not enough: since VertexCover really is **NP**-complete, there is a reduction to it from every problem in **NP** by definition.

Comments. For this part and the following part, it's crucial to reduce some **NP**-complete language *to* the problem in the question. Reducing VertexCover to, say, Sat only establishes that VertexCover $\in$ **NP**. ("If I could solve this hard problem, I could solve VertexCover" isn't nearly as strong a statement as "If I could solve VertexCover, I could solve this hard problem.")

c) We show that the problem is in **NP** by showing a succinct certificate. It can't be necessary to remove more than $|V(G)|$ vertices and, given vertices $v_1, \ldots, v_\ell$, we check that $\ell < k$ and then use depth-first search to check deterministically in polynomial time that deleting those vertices leaves a graph with no directed cycles.

We now reduce from VertexCover to this problem. Given an undirected graph $G = (V, E)$, we produce the directed graph $G' = (V, E')$ by replacing every undirected edge xy with the 2-cycle consisting of directed edges (x, y) and (y, x). If G has a vertex cover S of size at most k, then deleting these vertices from G removes all edges from G. Similarly, deleting the same vertices from G' removes all edges and, hence, all directed cycles from G'. Conversely, if G has no vertex cover of size at most k, then deleting any k vertices from G must leave at least one edge, so deleting the same set of vertices from G must leave at least one 2-cycle.

Grading notes. Same as part b). For membership in **NP**, it's necessary to give at least a brief description of how you'd check in polynomial time that a graph is acyclic (something like "by DFS" is enough).

Comments. This problem is known as FeedbackVertexSet. It's tempting to try to reduce from HamiltonianCycle but this is unlikely to get anywhere: typically, the only problems that have easy reductions from HamiltonianCycle are problems about visiting every vertex of a graph. It's possible for a graph to have many short cycles but no Hamiltonian cycle so it's not obvious how to use that here.