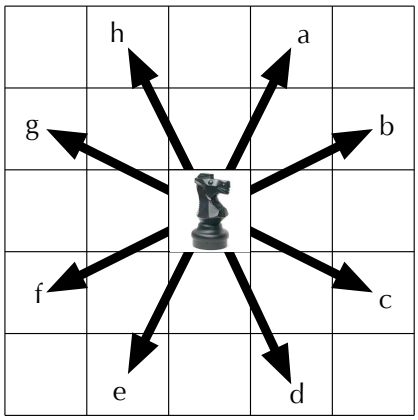
CS500, Theory of Computation: Midterm Solutions

1. (30 points) In chess, a knight can move in eight directions. Label these with eight symbols $\Sigma = \{a, b, c, d, e, f, g, h\}$ as in this figure:



Let $L \subset \Sigma^*$ be the set of paths that return a knight to its original location on an infinite board. Then:

(a) Prove that L is not regular.

(b) Prove that L cannot be recognized in real time by a one-counter machine, i.e.,

i. M reads its input from left to right with no ϵ-transitions.

ii. M has a finite set of states Q. At each step it may update its state and may increment or decrement its counter by any integer.

iii. M's only dependence on the current value of its counter is whether or not it is zero; thus its transition function takes the form $\delta : Q \times \Sigma \times \{\text{zero}, \text{nonzero}\} \rightarrow Q \times \mathbb{Z}$ where $\mathbb{Z}$ denotes the integers.

iv. M accepts the input if the counter is zero and/or its final state is in some accepting subset $Q_{\text{accept}} \subseteq Q$.

(c) Prove that L is not context-free.

Note: don't just prove part (c) and then point out that it implies the first two parts! Use proof methods that are "native" to each class of languages or machines, such as closure properties, pumping lemmas, inequivalent states, etc.

Answers. (a) The regular languages are closed under intersection. Therefore, if L is regular, so is

$$L_1 = L \cap a^* e^* \ .$$

But the only paths composed of a string of a's followed by a string of e's are those with an equal number of each, so

$$L_1 = \{a^n e^n\} \ .$$

Sipser (Example 1.38) shows using the Pumping Lemma that this language (with a and e replaced by 0 and 1) is not regular, so L is not regular. □

(b) Let us consider the number of possible states a 1-counter machine M can get into after reading ℓ symbols. Since it cannot make ϵ-transitions, this means it takes exactly ℓ steps. Since it has a finite transition table, there is some constant C bounding the amount by which it increments its counter, so after ℓ steps its counter is in the range $[0, C\ell]$. Including its finite state, M can be in at most

$$|Q|(C\ell + 1) = O(\ell)$$

states after ℓ steps.

On the other hand, we can show that any machine that recognizes ℓ has to be able to get into $\Theta(\ell^2)$ different states after ℓ or fewer steps. Define the equivalence class

$$u \sim_L v \text{ if } \forall w : (uw \in L \Leftrightarrow vw \in L) \ .$$

Now we claim that there are $\Theta(\ell^2)$ inequivalent strings of length ℓ or less. For instance, let $u = a^i c^j$ and $v = a^{i'} c^{j'}$, and let $w = e^i g^j$. Clearly if $i \neq i'$ or $j \neq j'$, we have $uw \in L$ but $vw \notin L$. Therefore, all words of the form $a^i c^j$ are inequivalent. There are $\ell(\ell+1)/2 = \Theta(\ell^2)$ such strings of length ℓ or less, i.e. with $i + j \leq \ell$.

Therefore, for sufficiently large ℓ, by the pigeonhole principle M will arrive in the same state after reading two different words u, v of the form $a^i c^j$. But then reading an appropriate w causes M to make a mistake by either accepting both uw and vw or rejecting both. (Note that L can be recognized by an automaton with *two* counters, since these can keep track of the x and y coordinates of the knight's position.) □

(c) The context-free languages are closed under intersection with regular languages. Therefore, if L is context-free, so is

$$L_2 = L \cap a^* c^* e^* g^* \ .$$

But, since the e's must cancel the a's and the g's must cancel the c's, we have

$$L_2 = \{a^i c^j e^k g^\ell \mid i = k \wedge j = \ell\} \ .$$

In homework #2, problem #1, we showed using the Pumping Lemma that L_2 is not context-free (with a, c, e, g replaced by a, b, c, d). □

2. (20 points) Let L be the set of descriptions of Turing machines M such that M accepts an infinite number of different input strings. Show that L is undecidable. Is L recursively enumerable? Explain, and for extra credit provide a proof.

   Answers. We show that L is undecidable by reducing the Halting Problem to it. Suppose I have an instance of the Halting Problem which asks whether a machine M halts on an input x. Construct a new machine M' with x hard-coded into it; M' ignores its input, writes x on the tape, simulates M on x, and accepts if M halts. Then M' accepts all inputs if M halts on x, and accepts no inputs if M does not halt on x, so $M' \in L$ if and only if M halts on x. Thus if we could decide membership in L, we could decide the Halting Problem.

   This shows that L is *at least* as hard as the Halting Problem; but we only used a very special case of L, namely the set of machines that accept everything or nothing. In general, L seems much harder than the Halting Problem, since telling whether M is in L or not involves asking an infinite number of halting questions, not just one. Formally, we can write the statement that $M \in L$ as

$$\forall y : \exists x > y : M \text{ accepts } x \ .$$

   In other words, the statement that M accepts an infinite number of x's is equivalent to the statement that, for any y, there is an $x > y$ that M accepts. Thus it has two nested quantifiers of the form $\forall\exists$, while recursively enumerable languages have a single $\exists$. This suggests that L is not recursively enumerable. But how do we prove this?

   We showed in homework #3, problem #10 that IsDecider, the property that a Turing machine halts on all inputs, is not r.e. We will prove that L is not r.e. by reducing IsDecider to it. Suppose I want to know whether M is a decider. Construct a new machine M' that does the following: given input x, M' does a `for` loop, and simulates M on all inputs y ranging from 0 to x (or if you prefer, all inputs of length less than $|x|$). If M halts on all these inputs, M' accepts x.

   Now note that if M is a decider, M' accepts all inputs. On the other hand, if M is not a decider, let y be the smallest (or shortest) input for which M does not halt; then M' only halts for inputs $x < y$, and this is a finite set. Thus $M' \in L$ if and only if M is a decider. This proves that L is at least as hard as IsDecider, and is not r.e. $\square$

3. (30 points) In the little-known 51st state, each district contains 3 voters, and the districts may overlap. Political parties there try to win all the district elections by convincing a majority of voters in each district to vote with them. Of course, your party has a limited budget, and you can only afford enough ads to convince up to k voters out of the entire population n, so this might be tricky. Formally:

DISTRICT ELECTIONS
Input: An integer n, an integer k, and a set of m subsets $S_1, \ldots, S_m \subset \{1, \ldots, n\}$ such that $|S_i| = 3$ for all i.
Question: Does there exist a subset $V \subseteq \{1, \ldots, n\}$ with $|V| \leq k$ such that $|S_i \cap V| \geq 2$ for all i?

(a) Prove that DISTRICT ELECTIONS is NP-complete.

(b) EXACT DISTRICT ELECTIONS is a variant of this problem which tells, given k, whether it is the case that I can win all the elections if I convince the right set of k voters, but that I can't do it with fewer than k (as opposed to DISTRICT ELECTIONS as defined above, which asks if I can win with k or fewer). Explain why EXACT DISTRICT ELECTIONS might not be in NP.

(c) However, show that if I am given an oracle which answers yes or no to DISTRICT ELECTIONS as defined above, then I can solve EXACT DISTRICT ELECTIONS with a reasonable number of calls to this oracle. How many calls do I need as a function of the input size? For extra credit, also comment on the variant MINIMUM AD BUDGET that tells us, for a given set of S_i, the smallest number k such that there is a solution of size k.

Note: you can do parts (b) and (c) even if you have trouble with (a).

Answers. (a) To prove membership in NP, the certificate is the set V. We can clearly check in polynomial time whether $|V| \leq k$, and whether $|S_i \cap V| \geq 2$ for all i.

To prove NP-completeness, we reduce Vertex Cover to District Elections. Recall that an instance of Vertex Cover consists of a graph G and an integer k, and asks whether there is a subset U of G's vertices of size k or less such that every edge has at least one endpoint in U. We convert this to an instance of District Elections with n' voters and a threshold k' as follows. If G has n vertices, add a single new vertex $w = n + 1$, and let $n' = n + 1$. Let m be the number of edges in G, and let $S_i = \{u, v, w\}$ where (u, v) is the i'th edge of G. In other words, let each district consist of the two endpoints of one of G's edges, plus the new vertex. Finally, set $k' = k + 1$. We can clearly carry out this reduction in polynomial (or linear) time.

To prove that this reduction works, note that if U is a vertex cover of size k, then $V = U \cup \{w\}$ is a winning set of size $k + 1$, and vice versa; adding w covers a majority of $\{u, v, w\}$ if and only U already covered at least one of $\{u, v\}$. Conversely, if there is a winning set V of size $k + 1$, then either V contains w, in which case removing w gives a vertex cover of size k, or V contains *every* vertex of G, and removing any one of them gives a vertex cover of size $k = n - 1$. Thus G has a vertex cover of size k if and only if we have a winning set of size k'. □

(b) Exact District Elections makes a claim of the form

$$(\exists V : V \text{ is a solution of size } k) \bigwedge (\neg \exists V' : V' \text{ is a solution of size less than } k)$$

or equivalently

$$(\exists V : V \text{ is a solution of size } k) \bigwedge (\forall V' : \text{if } |V'| < k \text{ then } V' \text{ is not a solution})$$

(Note that the "for all" in the definition of a solution, "$|S_i \cap V| \geq 2$ for all i," only involves checking a polynomial number of things, while the $\exists$ and $\forall$ here range over the 2^n possible subsets V.)

Thus it is not clear how Exact District Elections could be expressed with a single $\exists$. It makes a different kind of logical claim than an NP problem; specifically, it simultaneously makes an NP claim and a co-NP claim. (Note, however, that this is not the same as saying that Exact District Elections is in NP $\cap$ coNP!) □

(c) If we have an oracle for District Elections, we can answer Exact District Elections by asking it just two questions, one for k and one for $k - 1$; if it answers these "yes" and "no" respectively, we know that there is a solution of size k, but no smaller.

(Extra credit) To solve Minimum Ad Budget, we perform a binary search: first ask the District Elections oracle about $k = n/2$, then $k = n/4$ or $3n/4$ depending on the answer to the first question, and so on. This determines the smallest value of k for which the oracle answers "yes" with $O(\log n)$ queries. □

4. (20 points) *Geography* is a two-player game played on a directed graph where we take turns deciding which edge to follow. Whoever gets stuck in a "dead end", with no outgoing edge to follow, loses. The problem GEOGRAPHY is whether, given a directed graph G and an initial vertex u, the first player has a winning strategy. With the restriction that we cannot visit a vertex twice, GEOGRAPHY is PSPACE-complete.

   Now consider removing this restriction. Show that the resulting version of GEOGRAPHY is in P. Hint: find a way to iteratively label vertices of G as "winning" or "losing" vertices, i.e., places you want to be, or places you want to avoid. (Since we can visit the same vertex many times, it might be the case that neither player has a winning strategy and optimal play leads to a draw, i.e., an infinite or cyclic game.)

   Finally, try to explain why the can't-visit-the-same-vertex-twice restriction makes the difference in complexity between P and PSPACE.

   Answers. We use the following algorithm.

```
Start with all vertices unlabeled.
Do until we reach a fixed point:
  For all unlabeled vertices v:
    If v has an out-neighbor labeled Losing,
      label v Winning.
    If all of v's out-neighbors are labeled Winning,
      label v Losing.
```

   Note that this algorithm labels all vertices with out-degree zero Losing on its first loop, since all their out-neighbors (of which there are none) are labeled Winning.

   This algorithm takes optimal play into account, since a winning position is one where there *exists* a move that puts our opponent in a losing position, and a losing position is one where *all* moves (if there are any) put our opponent in a winning position. The game is a win or loss for the first player if u is Winning or Losing respectively, and a draw if it is unlabeled. Since the loop runs at most n times, the algorithm runs in polynomial time.

   One reason the can't-visit-the-same-vertex-twice restriction makes such a big difference in the computational complexity is the amount of information we have to keep track of as we play the game. Without the restriction we only need to know our current position, which is a $(\log n)$-bit number. With the restriction we also have to remember for every vertex whether we visited it before or not, which requires n bits of memory. In other words, in the two versions of the game there are either n or 2^n states we can be in. Without the restriction we simply evaluate all n states (with the labeling scheme above) and determine whether they are winning or losing, but with the restriction it doesn't seem possible to evaluate all 2^n possible states in polynomial time. □