6.541/18.405 Problem Set 1

due on March 5, 11:59pm

Rules: You may discuss homework problems with other students and you may work in groups, but we require that you *try to solve the problems by yourself before discussing them with others.* Think about all the problems on your own and do your best to solve them, before starting a collaboration. If you work in a group, include the names of the other people in the group in your written solution. **Write up your *own* solution to every problem**; don't copy answers from another student or any other source. You should be able to explain and write your solutions all by yourself. Cite **all** references that you use in a solution (books, papers, people, etc) at the end of each solution.

We encourage you to use LaTeX, to compose your solutions. The source of this file is also available on Piazza, to get you started!

How to submit: Use Gradescope entry code **2P3PEN**.

Please use a separate page for each problem.

Problem 1: NP vs coNP (1 point)

Question

Prove that if $\mathbf{NP} \subseteq \mathbf{coNP}$, then $\mathbf{NP} = \mathbf{coNP}$.

Answer

Let $L \in \mathbf{coNP}$. Then there is a language $L' \in \mathbf{NP}$ such that $L = \overline{L'}$ (the complement of L'). If $\mathbf{NP} \subseteq \mathbf{coNP}$, then $L' \in \mathbf{coNP}$. Thus $\overline{L'} \in \mathbf{NP}$. Since $\overline{L'} = L$, this means $L \in \mathbf{NP}$. Thus $\mathbf{NP} \subseteq \mathbf{coNP} \implies \mathbf{coNP} \subseteq \mathbf{NP}$.

Problem 2: P to the NP (1 point)

Question

Define a function f that treats its binary input as the encoding of a 3CNF Boolean formula, where $f(\phi) = 1$ if ϕ is a 3-CNF formula with exactly one satisfying assignment, and $f(\phi) = 0$ otherwise. Prove that $f \in \mathbf{P}^{\mathbf{NP}}$.

Answer

We can write f as

$$f(\phi) = 1 \iff \exists x.\forall y.(\phi(x) = 1 \wedge (x \neq y \implies \phi(y) = 0))$$

Where the quantifiers for x, y range over all bit strings describing assignments to the variables in ϕ. These strings are polynomial in the description length of ϕ, which has size linear in the number of variables referenced in the formula.

Let $g : \{0,1\}^* \to \{0,1\}$ where

$$g(\phi) = 1 \iff \exists x.\phi(x) = 1$$

Let $h : \{0,1\}^* \to \{0,1\}$ where

$$h((\phi, x)) = 1 \iff \exists y.x \neq y \wedge \phi(y) = 1$$

Observe that g and h are in NP. Let k be the function which either returns the value of g or h depending on the first bit:

$$k(0 \cdot z) = 1 \iff g(z) = 1; \quad k(1 \cdot z) = 1 \iff h(z) = 1$$

Then of course k is in NP.

Let T be the following Turing machine with oracle access to k. It first queries g through k to determine if ϕ is satisfiable, and returns false if not. If it is satisfiable, using its access to g, it finds a satisfying assignment x to ϕ in polynomial time (using the reduction of Search-SAT to SAT from Pset 0). Then it queries $h(\phi, x)$ through k to see if there exists a satisfying assignment y to ϕ, distinct from x. If so, it rejects. Otherwise, it accepts.

T runs in polynomial time (given k-oracle access) and decides f. This proves $f \in \mathbf{P}^{\mathbf{NP}}$.

Problem 3: An Interesting Problem in PH (2 points)

Question

An important concept in machine learning theory is that of *VC dimension.* Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a collection of subsets over a universe $U = \{0,1\}^n$. We define the *VC dimension of* $\mathcal{S}$, abbreviated as $VC(\mathcal{S})$, to be the size of the largest set $X \subseteq U$ such that for every possible subset $X' \subseteq X$, there is some $i \in \{1, \ldots, m\}$ such that $S_i \cap X = X'$. (In other words, X is a maximal subset of the universe such that for every possible subset X' of X—every possible way to "classify" the elements of X as 0 or 1—there is some $S_i \in \mathcal{S}$ whose intersection with X is precisely X'.)

Let C be a CNF formula over two collections of Boolean variables $i = (i_1, \ldots, i_t)$ and $x = (x_1, \ldots, x_n)$, where $t = \log_2(m) + O(1)$ and i is treated as a number in $\{1, \ldots, m\}$. We say that C *represents a collection* $\mathcal{S} = \{S_1, \ldots, S_m\}$ if for all $i \in \{1, \ldots, m\}$, $S_i = \{x \in \{0,1\}^n \mid C(i,x) \text{ is true}\}$. (In other words, the CNF C encodes the sets of the collection $\mathcal{S}$ in a natural way.)

Finally, we define the language

$$\text{VCD} = \{\langle C, k\rangle \mid k \text{ encoded in binary, } C \text{ represents a collection } \mathcal{S} \text{ such that } VC(\mathcal{S}) \geq k\}.$$

(a) (1 point) Show that for any CNF C over variables i and x as above, the VC dimension of the collection $\mathcal{S}$ represented by C is at most $\log_2(m)$.

(b) (1 point) Show that VCD is in $\Sigma_3\mathbf{P}$.

As a consequence, $\mathbf{P} = \mathbf{NP}$ would also imply that computing the VC dimension can be done in $\mathbf{P}$!

Answer to (a)

For contradiction say the VC dimension of $\mathcal{S}$ were $k > \log_2(m)$. Let $X \subseteq U$ be a set of size k such that $\{S_i \cap X : S_i \in \mathcal{S}\} = 2^X$. Then there are 2^k distinct subsets of X, such that for each subset X'_a for $a \in \{1, 2, \ldots, 2^k\}$, there exists $S_{m_a} \in \mathcal{S}$ so $X'_a \cap S_{m_a} = X'_a$. This means there exists a number i_{m_a} such that $\forall x \in X, C(x, i_{m_a}) = 1 \iff x \in X'_a$.

Because there are only m values of i treated as distinct by the formula C, and there are $2^k > 2^{\log_2(m)}$ values of a, we must have $i_{m_a} = i_{m_{a'}}$ for some distinct a, a'. But then $X'_a = X'_{a'}$, contradicting the fact that the X' were defined to be the 2^k *distinct* subsets of X.

Answer to (b)

Observe that

$$\langle C, k\rangle \in \text{VCD} \iff \exists X.\forall X'.\forall x.\exists i.$$
$$k \leq t(C) \quad \wedge |X| \geq k \quad \wedge (X' \subseteq X) \implies [C(i,x) = 1 \iff x \in X']$$

Here, we write $t(C)$ to denote the function which takes a description of formula C as input, and outputs the number t of input bits in the first collection i of input bits to C.

Note that the first clause in this formula, $k \leq t$, always holds if k is the VC dimension of C, by part (a) and the fact that $t \geq \log(m)$.

The values which are quantified over are as follows,

- X and X' are subsets of U, each of size $\leq \log m$, represented as a list of integers in binary (giving the values in the subsets). This requires $O(\log(m)n) = O(tn)$ bits (n bits for each element in the set).
- x and u are bit strings of length n, describing elements of U
- i is a bit string of length t

Since the description of a CNF formula C is at least linear in the number of variables appearing in it (which in this case includes x_1 through x_n and $i_1, \ldots, i_t$), the input $\langle C, k \rangle$ has length $\geq n + t$. Thus each of the variables which is quantified over has length polynomial in the input.

We only need to enumerate subsets X of size $\leq \log m$ because we only need to consider cases where $k \leq \log m$, due to part (a).

It is easy to see that computing the proposition after the quantifiers in the quantified formula can be done in polynomial time.

Thus, VCD $\in \Sigma_3 P$.

Problem 4: Polynomial Hierarchy and Oracles (4 Points, 2 for each sub-problem)

Question

This problem is intended to help you learn about *relativization*: how to re-prove a known theorem when oracles are stuck in everywhere.

We say a language $L \in (\mathbf{\Sigma}_k\mathbf{P})^A$ iff there is a polynomial time oracle Turing machine M^A and polynomial $q(n)$ such that for all strings x,

$$x \in L \Leftrightarrow (\exists\ x_1)(\forall\ x_2)\cdots(Q_k\ x_k)[M^A(x, x_1, \ldots, x_k) = 1]$$

where $|x_j| \le q(|x|)$ for all $1 \le j \le i$, and $Q_i = \exists$ if i is odd, else $Q_i = \forall$. Define

$$\mathbf{PH}^A := \bigcup_{k>0}(\mathbf{\Sigma}_k\mathbf{P})^A.$$

Let $k \ge 0$ be arbitrary, and let A be any language such that $(\mathbf{\Sigma}_{k+1}\mathbf{P})^A = (\mathbf{\Sigma}_k\mathbf{P})^A$.

(a) Prove that $\mathbf{PH}^A \subseteq (\mathbf{\Sigma}_k\mathbf{P})^A$.

(b) Use part (a) to prove that $\mathbf{P} = \mathbf{NP}$ implies $\mathbf{PH} = \mathbf{P}$ is true "relative to all oracles A", i.e. prove that $\mathbf{P}^A = \mathbf{NP}^A$ implies $\mathbf{PH}^A = \mathbf{P}^A$, for all A.

Answer to (a)

First, I give a lemma and a corollary.

Lemma 1 *$(\mathbf{\Pi}_m\mathbf{P})^A$ and $(\mathbf{\Sigma}_n m\mathbf{P})^A$ are complements.*
Let $(\mathbf{\Pi}_m\mathbf{P})^A$ be the class of languages L such that there exists a machine M^A where $x \in L \iff (\forall x_1)(\exists x_2)\ldots(Q'_m x_m)[M^A(x, x_1, \ldots, x_m)]$, where Q'_m is whichever of the symbols $\forall, \exists$ that Q_m is not. Then

$$(\mathbf{\Pi}_m\mathbf{P})^A = \{\bar{L} : L \in (\mathbf{\Sigma}_m\mathbf{P})^A\}$$

and

$$(\mathbf{\Sigma}_m\mathbf{P})^A = \{\bar{L} : L \in (\mathbf{\Pi}_m\mathbf{P})^A\}$$

Proof: For f as in the lemma statement, $f(x) = 1 \iff \neg(\exists x_1)(\forall x_2)\ldots(Q_m x_m)[\neg M^A(x, x_1, \ldots, x_m)]$. And $\neg M^A$ can certainly be implemented with a Turing machine given oracle access to A. The proof of the second equation is symmetric.

Corollary 1 *If $(\mathbf{\Sigma}_n\mathbf{P})^A = (\mathbf{\Sigma}_k\mathbf{P})^A$, then $(\mathbf{\Pi}_n\mathbf{P})^A = (\mathbf{\Pi}_k\mathbf{P})^A$.*

Proof: Say $L \in (\mathbf{\Pi}_n\mathbf{P})^A$. Then $L = \bar{L}'$ for some $L' \in (\mathbf{\Sigma}_n\mathbf{P})^A = (\mathbf{\Sigma}_k\mathbf{P})^A$. Thus L is the complement of a language in $(\mathbf{\Sigma}_k\mathbf{P})^A$, so it is in $(\mathbf{\Pi}_k\mathbf{P})^A$.

Proof for question (a):

Now I will prove by induction that $(\mathbf{\Sigma}_n\mathbf{P})^A \subseteq (\mathbf{\Sigma}_k\mathbf{P})^A$ for all $n \ge k$. Since $\mathbf{PH}^A = \cup_n(\mathbf{\Sigma}_n\mathbf{P})^A$, this proves $\mathbf{PH}^A \subseteq (\mathbf{\Sigma}_k\mathbf{P})^A$.

The cases $n = k$ is trivial and the case $n = k+1$ is given.

For the inductive step where $n \ge k+2$, assume the theorem has been proven for $n-1$ and $n-2$.

Consider a language $L \in (\mathbf{\Sigma}_n\mathbf{P})^A$, where this inclusion is witnessed by Turing machine M^A. For any x, let L' be the language

$$(x, x') \in L' \iff (\forall x_2)(\exists x_3)(\forall x_4)\ldots(Q_n x_n)[M^A(x, x', x_2, x_3, \ldots, x_n)]$$

Note that $x \in L \iff (\exists x_1)[(x, x_1) \in L']$.

Observe that $L' \in (\mathbf{\Pi}_{n-1}\mathbf{P})^A$. By the inductive hypothesis, and the corollary above, $L' \in (\mathbf{\Pi}_k\mathbf{P})^A$. Thus there exists an oracle Turing machine T^A so that

$$(x, x') \in L' \iff (\forall x_1)(\exists x_2) \ldots (Q'_k x_k)[T^A(x, x', x_1, \ldots, x_k)]$$

Then $x \in L \iff (\exists x')(\forall x_1)(\exists x_2) \ldots (Q'_k x_k)[T^A(x, x', x_1, \ldots, x_k)]$. This proves that $L \in (\mathbf{\Sigma}_{k+1}\mathbf{P})^A = (\mathbf{\Sigma}_k\mathbf{P})^A$.

Answer to (b)

Given part (a), it suffices to prove that $\mathbf{P}^A = \mathbf{NP}^A \implies (\mathbf{\Sigma}_0\mathbf{P})^A = (\mathbf{\Sigma}_1\mathbf{P})^A$. For this it certainly suffices to show that $\mathbf{P}^A = (\mathbf{\Sigma}_0\mathbf{P})^A$ and $\mathbf{NP}^A = (\mathbf{\Sigma}_1\mathbf{P})^A$. And these follow immediately from our definitions. (Ie. for each of these equalities, our definition of the object on the LHS is identical to our definition of our object on the RHS.)

Problem 5: Alternation! (8 Points, 2 for each sub-problem)

Question

There is a generalization of both nondeterminism and conondeterminism called *alternation*, which is a natural machine model for massive parallelism. In this problem we will explore why alternation is cool. **(Note: Arora and Barak also discuss alternation. Feel free to use the textbook to help you answer the questions below, but please write your answers in your own words!)**

Here's the setup. An *alternating Turing machine* M works just like a non-deterministic machine, except the states of M have "labels" and the accepting condition is different. Each state of an alternating Turing machine M, other than q_{acc} and q_{rej}, has an associated *state label* which can be $\forall$ or $\exists$. Configurations of M which have an $\forall$ state are called *universal configurations*, and configurations involving an $\exists$ state are *existential configurations.*

The configuration graph $G_{M,x}$ for a machine M on input x is the same as for non-deterministic machines. (If you haven't seen these before, we will recall them on 2/24.) The acceptance condition is more complicated, and is defined recursively. For a configuration node v in the graph $G_{M,x}$ we say v *leads to acceptance* if

1. v contains q_{acc}, or
2. v is an existential configuration and there is an edge (v, u) in $G_{M,x}$ such that the node u leads to acceptance, or
3. v is a universal configuration and for all edges (v, u) in $G_{M,x}$, the node u leads to acceptance.

Then, we define that M *accepts* x if the initial configuration of $G_{M,x}$ leads to acceptance.

Note that a machine with only existential configurations gives exactly the acceptance condition for a non-deterministic machine, and a machine with only universal configurations works just like a co-nondeterministic machine.

We define the running time of an alternating machine M on input x to be the length of the longest path in $G_{M,x}$. The space used by M on x is the size (in bits) of the largest configuration in $G_{M,x}$. Finally, we define

$$\mathsf{ATIME}(t(n)) = \{L \mid \text{there is an alternating machine that accepts } L \text{ in time } O(t(n))\},$$

and

$$\mathsf{ASPACE}(s(n)) = \{L \mid \text{there is an alternating machine that accepts } L \text{ in space } O(s(n))\}.$$

We define $\mathbf{AP} := \bigcup_c \mathsf{ATIME}(n^c)$, (alternating poly-time), $\mathbf{AL} := \mathsf{ASPACE}(\log n)$ (alternating logspace), and $\mathbf{ASPACE} := \bigcup_c \mathsf{ASPACE}(n^c)$ (alternating polynomial space). In the following, assume the functions t and s are time and space constructible, respectively.

(a) Prove that $\mathsf{ATIME}(t(n)) \subseteq \mathsf{SPACE}(t(n))$.

(b) Prove that $\mathsf{SPACE}(s(n)) \subseteq \mathsf{ATIME}(s(n)^2)$ for $s(n) \geq n$.

Hint: Use the idea of Savitch's theorem that $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{SPACE}(s(n)^2)$.

(c) Define $\mathsf{TIME}[2^{O(s(n))}] := \bigcup_{c \geq 1} \mathsf{TIME}(c^{s(n)})$.
Prove that for $s(n) \geq \log n$, $\mathsf{ASPACE}(s(n)) \subseteq \mathsf{TIME}[2^{O(s(n))}]$.

Hint: Note this generalizes the proof that $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{TIME}(c^{s(n)})$.

(d) Prove that for $s(n) \geq \log n$, $\mathsf{TIME}[2^{O(s(n))}] \subseteq \mathsf{ASPACE}(s(n))$.

Hint: In alternating space $O(s(n))$, you could have computation paths of length up to $2^{O(s(n))}$, with many alternations.

From parts (a), (b), (c), (d), it follows immediately that **AL** = **P**, **AP** = **PSPACE**, and **APSPACE** = **EXP**. Adding alternation turns a time class into the corresponding space class, and a space class into an exponentially larger time class! So for example, the **P** vs **PSPACE** question boils down to whether there are problems solvable in alternating polynomial time which cannot be solved in *deterministic* polynomial time.

Answer to (a)

First, observe that for any alternating Turing machine A, there is an equivalent machine A' with a unique accepting configuration c_{acc}. (This can be constructed from A by replacing the accept state with a state that deterministically clears the memory tape and then accepts.) If the maximum paths in the configuration graph for A have length $t(n)$, the path lengths in A' are bounded by $O(t(n))$ (since there are at most $t(n)$ values on the memory tape to clear).

Let $L \in \mathsf{ATIME}(t(n))$. Then there exists an alternating Turing maching A deciding L, with a unique accepting configuration c_{acc}, such that all paths in the computation graph for A have size $\leq ct(n)$ for some c.

I will now prove that there exists a deterministic Turing machine M for L which uses space $O(t(n))$.

Let B be the maximum branching factor of any state in A.

In this proof, I will sometimes say that a configuration node *c leads to acceptance within k steps.* For $k = 0$, this means that c is c_{acc}. For $k \geq 1$, if c is an existential node this means that there exists a c' with an edge $c \to c'$ where c' leads to acceptance within $k-1$ steps, and if c is an universal node, this means that for all c' with an edge $c \to c'$, c' leads to acceptance within $k-1$ steps.

Given an input x to A, of size n, since we know that all paths in the computation graph have length less than $ct(n)$, we know that $x \in L$ iff the initial configuration for input x leads to acceptance within $ct(n)$ steps.

Consider the following recursive algorithm to decide whether a configuration c leads to acceptance within k steps. This algorithm will utilize a memory layout of the following form. Let K be the value of k at the top of the recursion. We will use $\log(K)$ bits to store the current recursive level's value of k. We will use $\log|c_0|$ bits to write down the configuration c_0 we are checking at the top level of the recursion. And we will maintain a stack of values, each containing $\log(B)$ bits. Each level of the recursion will add a value to this stack. Each element in this stack will store the index of the child of a configuration in the configuration graph.

Given c_0 and a sequence $i_1, \ldots, i_j$ of integers in $\{1, 2, \ldots, B\}$, for each $c > 0$, let c_t be the i_tth child of c_{t-1} in the configuration graph. Observe that we can compute c_t in constant $O(t(n))$ memory. (We first compute c_1, then c_2, and so on. Each transition $c_t \to c_{t+1}$ can be computed by simply finding the i_{t+1}th state successor of the Turing machine state in c_t. The only memory this needs is $O(t(n))$ bits to store the configuration c_t [we can overwrite c_{t+1} into the same slots as we had used to store c_t each time we transition].)

The recursion works as follows. Say the stack has size J. Compute c_J as described in the preceding paragraph. Let k denote the value written in the slot of $\log(K)$ bits. The goal at this level of the recursion is to determine if c_J leads to acceptance within k steps. If $k = 0$, simply return 1 if $c_J = c_{\text{acc}}$ and return 0 otherwise. If $k > 1$, and c is existential, add a level to the recursion stack. Write down $i = 1$ at a new level on the recursion stack. For each value $i = 1, \ldots, B$, consider the ith state c' which c leads to in the configuration graph (if i such states exist). Overwrite the current storage of the value k with the value $k-1$, and then recurse the algorithm to determine if c' leads to acceptance within $k-1$ steps. Then overwrite the value $k-1$ with the value k (by incrementing it by 1). If c' was found to lead to acceptance in $k-1$ steps, return that c leads to acceptance within k steps. Otherwise continue to the next i. Conversely, if c is a universal state, we can perform the same recursion, but now if we find that c' does not lead to acceptance, we immediately return false, while we continue looping otherwise.

This recursion uses $O(\log(K) + K\log(B) + s) = O(K + s)$ bits of storage, where s is the number of bits it takes to store a configuration in the graph.

To determine whether a string $x \in L$, it suffices to compute the initial configuration c_{acc} and determine if c_{acc} leads to acceptance in $ct(n)$ steps. Per the above, this can be done by a deterministic Turing machine in $O(t(n)+s)$ time, where s is the size of a description of a configuration. Since there are only at most $2^{t(n)}$ configurations, $s \in O(t(n))$. Thus, $L \in \text{SPACE}(t(n))$.

Answer to (b)

Let $L \in \mathsf{SPACE}(s(n))$. Then certainly $L \in \mathsf{TIME}(2^{O(s(n))})$. WLOG there is a Turing machine M such that $x \in L$ iff there is a path of length $\leq 2^{cs(n)}$ for some c from the initial configuration on input x, c_{init}, to a unique accepting configuration c_{acc} in the Turing machine.

Let L' be the language containing tuples (G, s, t, k) of graphs with upper-bounded branching factor, and with nodes s, t, where there is a path $s \to t$ of length $\leq 2^k$. The above shows that L can be decided by determining whether $(G_{M,x}, c_{\text{init}}, c_{\text{acc}}, cs(n)) \in L'$. Observe that $G_{M,x}$ has no more than $2^{cs(n)}$ vertices. Thus, if for any graph G with $\leq 2^{O(k)}$ it is possible to determine whether $(G, s, t, k) \in L'$ using an alternating TM using space $O(k^2)$, then $L \in \mathsf{ASPACE}(s(n)^2)$.

Observe

$$\begin{aligned}(G,s,t,k) \in L' \iff \exists v \in V(G).\forall z \in \{0,1\}.& \\ [(z=0) \wedge (G,s,v,k-1) \in L'] \vee [(z=1) \wedge (G,v,t,k-1) \in L']&\end{aligned}$$

That is, $(G, s, t, k) \in L'$ if there exists an intermediate vertex v in the set $V(G)$ of vertices in the graph, such that $s \to v$ in 2^{k-1} steps and $v \to t$ in 2^{k-1} steps. We use universal quantification over the binary value z to determine whether to check that path $s \to v$ or path $v \to t$ exists. When $k = 0$, the final clause in this formula (where we either check path $s \to v$ or path $v \to t$) can be computed simply by checking if $s \to v$ or $v \to t$ (whichever we are checking) is an edge in graph G. This recursive definition can be implemented by an alternating Turing machine which must store $\log|V(G)| + 1$ bits at each step of the recursion (where $|V(G)|$ is the number of vertices in the graph), to write down the value of vertex v and the value z. The recursion will have k steps. Thus, we can check $(G, s, t, k) \in L'$ with an alternating Turing using space $O(k \log|V(G)|)$. If $|V(G)| \leq 2^{O(k)}$, as it is in the configuration graph for the $\mathsf{SPACE}(s(n))$ Turing machine (as we noted above), this is contained in $O(k^2)$. This completes the proof.

Answer to (c)

To show that $\mathsf{ASPACE}(s(n)) \subseteq \mathsf{TIME}(c^{s(n)})$, I will show that for any language $L \in \mathsf{ASPACE}(s(n))$ and for any x, it is possible to determine whether $x \in L$ by running an $O(n\log(n))$ graph search algorithm on the configuration graph of an alternating Turing machine for L, which has size $2^{O(s(n))}$.

In particular, the graph search problem we need to solve is one I will call $\mathsf{AGraphSearch}$. The input to an $\mathsf{AGraphSearch}$ problem is a graph G where each vertex $v \in V(G)$ is labeled with a universal or existential quantifier. We will write $u(v) = 1$ if the quantifier for v is universal and $u(v) = 0$ if it is existential. The solution to this problem is given as follows.

$$\begin{aligned}(G,s,t) \in \mathsf{AGraphSearch} \iff &G \text{ is a directed acyclic graph with edges} \\ &\text{labeled with quantifiers, with a bounded branching factor,} \\ &\text{and } s,t \in V(G), \text{ and } (s = t \vee \\ &[u(s) = 1 \wedge \forall s' \text{s.t.}(s \to s') \in E(G).(G,s',t) \in \mathsf{AGraphSearch}] \vee \\ &[u(s) = 0 \wedge \exists s' \text{s.t.}(s \to s') \in E(G),(G,S',t) \in \mathsf{AGraphSearch}])\end{aligned}$$

For any language $L \in \mathsf{ASPACE}(s(n))$, any alternating Turing machine A which decides this language, and any string x of length n, let $G_{A,x}$ be the configuration graph of A on x, and WLOG let c_{init} and c_{acc} be the unique initial and accepting configurations. Per the discussion on Piazza, WLOG we can take $G_{A,x}$ to be acyclic. The number of vertices in $G_{A,x}$ is in $2^{O(s(n))}$, and since the branching factor of $G_{A,x}$ is bounded above by a constant B, the number of edges is also $2^{O(s(n))}$. It is immediate from our definitions that $x \in L$ if $(G_{A,x}, c_{\text{init}}, c_{\text{acc}}) \in \mathsf{AGraphSearch}$. If $\mathsf{AGraphSearch}$ can be decided in time $O(n\log(n))$, then $x \in L$ can be decided in $O(s(n)2^{O(s(n))}) = 2^{O(s(n))}$.

The final step to the proof is to provide an $O(n\log(n))$ algorithm for $\mathsf{AGraphSearch}$. The algorithm is depth first search, using a lookup table T (implemented, e.g., as a tree map) to track the accept/reject values for each node the depth first search has already visited. The algorithm $\mathsf{AGraphSearch}$ initializes T to the 1-element dictionary $\{t \mapsto 1\}$ to indicate that t is an acceptance state. It then calls $\mathsf{AGraphSearchHelper}$, which performs the depth first search using dictionary T. The helper algorithm, called on vertex s, returns immediately if $s \in \text{keys}(T)$. Otherwise, it recurses on each child of s in the graph. If s is universal, it rejects on the first child that rejects (and accepts if no children reject), and if it is existential, it accepts on the first accepting child (and rejects if no children accept).

```
AGraphSearch($G, s, t$)
  $T \leftarrow \{t \mapsto 1\}$
  **return** AGraphSearchHelper($G, s, T$)
```

```
AGraphSearchHelper($G, s, T$)
  **if** $s \in \text{keys}(T)$, return $T[s]$
  **for** vertex $s'$ with $(s \to s') \in E(G)$ **do**
    $a \leftarrow$ AGraphSearchHelper($G, s', T$)
    Add key $s'$ to table $T$ and set $T[s'] \leftarrow a$.
    **if** $a = 0$ and $u(s) = 1$, REJECT.
    **if** $a = 1$ and $u(s) = 0$, ACCEPT.
  **end for**
  **if** $u(s) = 1$, ACCEPT
  **if** $u(s) = 0$, REJECT
```

It is immediate from our definitions that this algorithm solves $\mathsf{AGraphSearch}$. All that remains is to verify that it can be run in $O(n\log(n))$ time First, note that the for loop in $\mathsf{AGraphSearchHelper}$ is entered at most once on a call call for any vertex s each vertex, since after the first time $\mathsf{AGraphSearchHelper}$ is called on a vertex s, the result is added to the table T, and future calls just perform a lookup in this table and terminate immediately. Second, note that the for loop only ever iterates over up to B values, which is a constant independent of the graph size. This means that the number of calls to $\mathsf{AGraphSearchHelper}$ is bounded by $B|V(G)|$. Third, note that the amount of work performed in any call to $\mathsf{AGraphSearchHelper}$, other than recursing, takes time $O(B\log(n))$, since it just requires (1) checking if $s \in \text{keys}(T)$, (2) querying the children of s in the graph, (3) adding up to B keys to T, and (4) checking B times if it should accept or reject. Steps (1) and (3) can be done in $\log(n)$ and $B\log(n)$ time respectively, if we represent T as a binary tree map, and step (4) takes $O(B)$ time. Step (2) can also be done in $O(\log(n))$ time if the graph's edges are represented in a data structure like a tree map supporting $O(\log(n))$ accessing of the array of children. Thus the cost of the algorithm is $O(|V(G)|B^2\log(|V(G)|))$. Since B is a constant, this is $O(n\log(n))$ in the graph size.

This essentially completes the proof. The last thing to note is just that it is possible to efficiently encode the configuration graph $G_{A,x}$ in such a way that vertices can be represented in $O(s(n))$ bits, and where querying the children vertices of any vertex can be done in $O(s(n))$ time. There are a number of ways this can be implemented. One is to represent each vertex as a bit string directly encoding the Turing machine configuration, and to provide the edge set by providing a Turing machine which, given a configuration, writes down a list of up to B children configurations. (A Turing machine exists which can do this in $O(B|A|s(n)) = O(s(n))$ time, where $|A|$ is the size of the description of Turing machine A, which is a constant.)

Answer to (d)

Say we are given a Turing machine T which runs in time $2^{cs(n)}$.

I will construct a Turing machine A which uses $O(s(n))$ space, and simulates T.

The Turing machine A will have two modes. The first mode will be used to answer the question: "In executing T on x, was $b \in \{0,1\}$ the bit value on the i'th slot of the read/write tape, at time t' of the computation?" The second mode will answer the question: "If at time t, the Turing machine T run on x had its head at position i on the tape, and was in state q, does it ultimately accept?"

The second mode is implemented by existentially guessing the bit value b which T reads from slot i at time t, and then computing the new state q_{t+1}, the new head position i_{t+1}, and the new time $t+1$ which would occur if T read bit b. A then uses universal quantification to accept if and only if (1) b actually *is* the value which T would read from slot i at time t (this can be verified using mode 1), and (2) at time $t+1$, T on x accepts if it is in state q_{t+1} with its head at position i_{t+1} (this can be verified using mode 2). Naturally, if the second mode is ever entered to ask if T would accept on state $q = q_{\text{accept}}$, then A immediately returns yes rather than recursing. This is implemented in algorithm AcceptsOn below.

Mode 1 is implemented recursively as well. It is implemented by effectively running a simulation of the Turing machine T, to verify this bit value. An implementation of Mode 1 is given in the algorithm box BitAtTimeIs$(x, i_0, t_0, q_0, b_0, i_1, t_1, b_1)$. This algorithm runs a simulation of T on x, tracking the current head position i_0, the current simulation timestep t_0, the current state q_0, and the current bit b_0 at position i_1 on the tape. It ultimately runs the simulation to time t_1, to determine if the bit at slot i_1 at time t_1 is the value b_1. Mode 1, to verify that bit i at time t is b, can be implemented by calling BitAtTime1$(x, 1, 1, q_{\text{init}}, \text{InitBitVal}(i), t, i, b)$, to initialize a simulation with the head at position 1, at time 1, from the initual state. The function InitBitVal(i) is used to determine the initial bit on the tape at position i.

```
AcceptsOn($x, t_0, q_0, i_0$)
  { Decide if $T$ ultimately accepts on $x$, given that at time $t_0$ its state was $q_0$ and its head was at position $i_0$. }
  **if** $q_0 = q_{\text{accept}}$ **then**
    **return** 1
  **else if** $q_0 = q_{\text{reject}}$ **then**
    **return** 0
  **else**
    Existentially guess $b \in \{0,1\}$.
    Universally select $m \in \{0,1\}$
    **if** m = 0 **then**
      **return** BitAtTimeIs($x, 1, 1, q_{\text{init}}$, InitBitVal($i_0$)$, t_0, i_0, b$)
    **else**
      Get next head position, state $(q_1, i_1, \_) \leftarrow T(q_0, i_0, b)$
      **return** AcceptsOn($x, t_0 + 1, q_1, i_1$)
    **end if**
  **end if**
```

The final step in the proof is simply to verify that this can all be done in $O(s(n))$ space. To verify this, simply observe that only things any of these algorithms need to track are: (1) x (of length n), (2) time values t_0 and t_1 which must be of size $2^{O(s(n))}$ and can thus be stored in $O(s(n))$ bits, (3) indices i_0 and i_1 onto the read/write tape, which can only grow to size $2^{O(s(n))}$ in the execution of T and thus can be encoded in $O(s(n))$ bits, and (4) states q and bit values b which can be encoded in constant space.

```
BitAtTimeIs($x, i_0, t_0, q_0, b_0, i_1, t_1, b_1$)
  {BitAtTimeIs: Given that when $T$ was run on $x$, at time $t_0$ its head was at position $i_0$ and it was in state $q_0$, and that the value of the bit at position $i_1$ was $b_0$, determine whether $b_1$ is the value of the bit at position $i_1$ at time $t_1$. }
  If $t_1 = 1$ and $i_1 = 1$ and $b = x[1]$, return 1.
  If $t_0 = t_1$ and $i_0 = i_1$, return $b_0 == b_1$.
  If $t_0 = t_1$ and $i_0 \neq i_1$, print "this will never happen in the execution of AcceptsOn."
  Existentially guess $b' \in \{0, 1\}$.
  Universally select $m \in \{0, 1\}$.
  if m = 0 then
    return BitAtTimeIs($x, 1, 1, q_{\text{init}}$, InitBitVal($x, i_0$)$, i_0, t_0, b_0$) {Check if $b'$ was the right guess.}
  else
    {Continue the simulation, assuming $b'$ was the right guess.}
    $(i', q', b'') \leftarrow T(q_0, b')$ {Compute the next head position, write bit, and state of $T$, given current position $q_0$ and read bit $b'$.}
    if $i_0 = i_1$ then
      $b_0 \leftarrow b''$
    end if
    return BitAtTimeIs($x, i', t_0 + 1, q', b_0, i_1, t_1, b_1$)
  end if
```

```
InitBitVal($x, i$)
  if $i \leq |x|$ then
    return $x[i]$
  else
    return 0
  end if
```

Problem 6: Algorithms versus Oracles (2 Points)

Question

Here is an apparent *paradox*. Given a polynomial time algorithm for SAT, we know how to construct a polynomial time algorithm for any problem in the polynomial hierarchy, e.g., MIN-FORMULA and VCD (problem 3). (In particular, we proved that **P** = **NP** implies **P** = **PH**.) On the other hand, given an *oracle* for SAT, we *do not* know how to use it to solve MIN FORMULA in polynomial time (that would put MIN-FORMULA in $\mathbf{P}^{\mathbf{NP}}$, which is an open problem). The oracle and the algorithm have the same input/output behavior; how can one be more useful than the other? Your answer should be less than ten lines of LaTeX.

Answer

The MIN-FORMULA problem on ϕ is naturally expressed as $\forall\psi.\exists x.\psi(x) \neq \phi(x)$. If all we have is a SAT oracle, we can solve the inner $\exists$ in one step, but we may need to do this exponentially many times. Conversely, we could reduce the inner $\exists$ clause to formula without a quantifier, and use SAT solving to solve the outer $\forall$, but the inner clause may have exponential length if the quantifier is removed.

On the other hand, if we have a polynomial time SAT algorithm, that guarantees that we can in fact reduce the inner $\exists$ clause to a polynomial-length quantifier-free formula, and then solve the outer $\forall$ by solving a polynomial-size SAT problem.

In generally, a SAT algorithm lets us reduce each quantifier to a polynomial-sized quantifier-free formula, while the oracle only lets us circumvent one layer of quantification.