Homework–2B
Fall 2024: CS 313: Computational Complexity Theory

Due: Friday, November 15, 2024. Total Marks: 50

This homework can be discussed in groups of two, but must be **attempted individually**.

Question 1 [10 points]

Let A be an **NP**-complete problem and B be a **coNP**-complete problem.

Show that, "**NP** $=$ **coNP** if and only if $A \leq_{\mathbf{P}} B$ and $B \leq_{\mathbf{P}} A$."

Solution: We know that A is a **NP**-complete problem, which implies that every problem in **NP**can be reduced to A in polynomial time. Similarly, B is a **coNP**-complete problem, which implies that every problem in **coNP**can be reduced to B in polynomial time.

Then we show that if **NP** $=$ **coNP**, then $A \leq_{\mathbf{P}} B$ and $B \leq_{\mathbf{P}} A$ as so:

1. **If NP= coNP, then $\mathbf{A} \leq_{\mathbf{P}} \mathbf{B}$ and $\mathbf{B} \leq_{\mathbf{P}} \mathbf{A}$:**

   Assume that **NP**$=$ **coNP**. Then every problem in **NP**also exists in **coNP**and vice versa. Since A is **NP**-complete, then by definition, every problem in **NP**can be reduced to A in polynomial time. Since **NP**$=$ **coNP**, then every problem in **coNP**can also be reduced to A in polynomial time. Since B is a **coNP**-complete problem, then every problem in **coNP**can be reduced to B in polynomial time. Therefore, every problem in **NP**can be reduced to B in polynomial time. This implies that $A \leq_{\mathbf{P}} B$. Further, since B is a **coNP**-complete problem, then every problem in **coNP**can be reduced to B in polynomial time. Since **NP**$=$ **coNP**, then every problem in **NP**can also be reduced to B in polynomial time. Therefore, every problem in **coNP**can be reduced to A in polynomial time. This implies that $B \leq_{\mathbf{P}} A$.

2. **If $\mathbf{A} \leq_{\mathbf{P}} \mathbf{B}$ and $\mathbf{B} \leq_{\mathbf{P}} \mathbf{A}$, then NP= coNP:**

   Since A is **NP**-complete, and can be reduced to B in polynomial time, and B is **coNP**-complete, then every problem in **NP**can be reduced to B in polynomial time. Similarly, since B is **coNP**-complete, and can be reduced to A in polynomial time, and A is **NP**-complete, then every problem in **coNP**can be reduced to A in polynomial time. This implies that every problem in **NP**can be reduced to B in polynomial time, and every problem in **coNP**can be reduced to A in polynomial time. Therefore, **NP**$=$ **coNP**.

Thus, "**NP**$=$ **coNP**if and only if $A \leq_{\mathbf{P}} B$ and $B \leq_{\mathbf{P}} A$."

■

Question 2 [10 points]

The game of **Nim** is played with a collection of piles of sticks. In one move, a player may remove any nonzero number of sticks from a single pile. The players alternately take turns making moves. The player who removes the very last stick loses. Say that we have a game position in **Nim** with k piles containing $s_1, \ldots, s_k$ sticks. Call the position *balanced* if each column of bits contains an even number of 1s when each of the numbers s_i is written in binary, and the binary numbers are written as rows of a matrix aligned at the low order bits. The following facts hold:

- Starting in an *unbalanced* position, a single move exists that changes the position into a *balanced* one.
- Starting in a *balanced* position, every single move changes the position into an *unbalanced* one.

Let **NIM** $= \{ < s1, ..., s_k > |$ each s_i is a binary number and Player 1 has a winning strategy in the **Nim** game starting at this position $\}$.

Use the preceding facts about balanced positions to show that **NIM** $\in$ **L**.

Solution: A game of **NIM** consists of two players, and k piles containing $s_1, s_2, ..., s_k$ number of sticks. The outcome of the game depends on two factors:

1. The player whos starts first,
2. The initial configuration of the piles.

Moreover, the winner of the game can be determined by the NIM-SUM; the bitwise XOR of the number of sticks in each pile. If the NIM-SUM is zero, then the second player wins, otherwise the first player wins.

Thus, the binary equivalent of each of the numbers s_i is taken and arranged row-wise, and aligned at the low order bits. If the number of 1's in each column is even, then the position is said to be *balanced*. Otherwise, it is said to be *unbalanced*. Thus, by the definition of the *balanced* position, it can be said that if the NIM-SUM is zero, then the position is *balanced*, and the second player wins. Otherwise, the position is *unbalanced*, and the first player wins.

The facts provided state that:

1. From an unbalanced position, a single move exists that changes the position into a balanced one.
2. From a balanced position, every single move changes the position into an unbalanced one.

This aligns with the standard winning strategy of NIM; if the NIM-SUM (xor of all sizes) is zero (balanced), the next player to move is in a losing position assuming optimal play. If the NIM-SUM is not zero (unbalanced), the next player to move is in a winning position assuming optimal play.

To determine whether a given configuration $s_1, s_2, ..., s_k$ belongs to NIM, we compute the NIM-SUM $NIM_s = s_1 \oplus s_2 \oplus ... \oplus s_k$. If the result is non-zero, the first player has a winning strategy from this position, and thus the configuration beongs to NIM. Conversely, if the result is zero, the second player has a winning strategy from this position, and thus the configuration does not belong to NIM.

Calculating the NIM-SUM involves taking the XOR of the binary representations of k numbers. This can be done in a fixed amount of space for each number as it is processed, since each number can be read and XOR-ed sequentially, and the space required for intermediate results does not depend on k but only on the number of bits in the largest number, which would be logarithmic in the size of the number. The space required to store each number and the result of the XOR operation is logarithmic in the size of the numbers, thus, the problem can be solved in logarithmic space.

Thus, **NIM** $\in$ **L**.

■

Question 3 [15 points]

Let **BOTH-NFA** $= \{ < M_1, M_2 > \mid M_1$ and M_2 are NFAs where $L(M_1) \cap L(M_2) \neq \emptyset \}$. Show that **BOTH-NFA** is **NL-COMPLETE**.

Solution: To show that **BOTH-NFA** is **NL-COMPLETE**, we need to show that **BOTH-NFA** is in **NL** and that every problem in **NL** can be reduced to **BOTH-NFA** in logarithmic space, that is, **BOTH-NFA** is **NL**-hard.

1. **BOTH-NFA is in NL:**

   We can construct a non-deterministic machine M that decides **BOTH-NFA** in logarithmic space. Consider M_1 has n states and M_2 has m states. The machine M works as follows:

   > $M =$ "On input $< M_1, M_2 >$:
   >
   > (a) Mark the initial states of M_1 and M_2.
   >
   > (b) Repeat:
   >
   > (c) Non-deterministically select an input symbol and accordingly change the marker positions on M_1 and M_2 to simulate the reading of that symbol.
   >
   > (d) If some string is found in (b) and (c) on which both M_1 and M_2 reaches any accept state, that is, if the marker is on an accept state of both M_1 and M_2, then accept. Otherwise, reject."

   The only space requirement for the algorithm to run is to store the location of the markers and repeat the counter. If we consider the binary representation of the counter, and store the index of the states of the NFAs in binary, then the space required to store the counter and the states is logarithmic in the size of the input. Thus, the algorithm runs in logarithmic space. Thus, **BOTH-NFA** is in **NL**.

2. **BOTH-NFA is NL-hard:**

   To show that **BOTH-NFA** is **NL**-hard, we show that another **NL-COMPLETE**problem can be reduced to **BOTH-NFA**.

   Given a directed graph $G = (V, E)$, and two nodes $s, t \in V$, the **PATH**problem is to determine whether there exists a path from s to t in G.

   We can construct two NFAs M_1 and M_2 such that M_1 accepts all paths starting form s, and M_2 accepts all paths ending at t. Specifically:

   - Construct M_1 to accept any sequence of edges that can be traversed from s. M_1 will be set up so that q_s is the start state and it will have transitions corresponding to edges in G.
   - Construct M_2 to accept any sequence of edges that can be traversed to reach t. Here M_2 will have transitions corresponding to edges in G and q_t will be the accept state.

   Then if there is a path from s to t, then there exists some sequence of edges accepted by both M_1 (starting from s) and M_2 (ending at t). Meaning, $L(M_1) \cap L(M_2) \neq \emptyset$. Conversely, if $L(M_1) \cap L(M_2) \neq \emptyset$, then there exists a path from s to t in G.

   The reduction only requires constructing the NFAs M_1 and M_2 from the input graph G. So to construct the graph, we only need to store the current vertex and can sequentially scan the edges to determine which transitions are possible from each vertex which can be done in logarithmic space by reusing the same space to store the current vertex and the edges. Thus, the reduction can be done in logarithmic space.

Since **BOTH-NFA** is in **NL** and is **NL**-hard, then **BOTH-NFA** is **NL-COMPLETE**.

■

Question 4 [15 points]

Prove that 2-SAT is **NL-COMPLETE**.

Solution: To show that a language is **NL-COMPLETE**, we need to show that the language is in **NL** and that it is **NL**-hard.

1. **2-SATis in NL:**

   To show this, we are going to be showing that $\overline{\textbf{2-SAT}}$, which is the complement of **2-SAT**exists in **NL**since we know that **NL**= **coNL**, therefore, showing that $\overline{\textbf{2-SAT}}$is in **NL**implies that **2-SAT**is also in **NL**.

   To show that $\overline{\textbf{2-SAT}}$is in **NL**, we construct an implication graph of the formula ϕ. Then for each variable x in the formula, create nodes for x and $\neg x$, and for each clause $(a \vee b)$, add two directed edges to the graph from $\neg a$ to b, and from $\neg b$ to a. Since the formula is in 2-CNF, then its unsatisfiable if for any variable x, there is a path from x to $\neg x$ and from $\neg x$ to x. This indicates that both x and $\neg x$ would need to be true simultaneously, which is a contradiciton.

   To check the unsatisfiability, we can construct a non-deterministic machine M that decides $\overline{\textbf{2-SAT}}$in logarithmic space. The machine M non-deterministically guesses the paths from x to $\neg x$ and from $\neg x$ to x for each variable x. If such a path exists, then the formula is unsatisfiable, and M accepts. Otherwise, the formula is satisfiable, and M rejects. We only need to keep track of the current variables, thus the space required is logarithmic in the size of the input. Thus, $\overline{\textbf{2-SAT}} \in$ **NL**, and hence, **2-SAT**$\in$ **NL**.

2. **2-SATis NL-hard:**

   To show that **2-SAT**is **NL**-hard, we need to show that every problem in **NL**can be reduced to **2-SAT**in logarithmic space. We can use the **PATH**problem to show this reduction.

   Given a directed graph $G = (V, E)$, and two nodes $s, t \in V$, the **PATH**problem is to determine whether there exists a path from s to t in G. Then we construct a **2-SAT**instance from the graph G. For each node $u \in G$, introduce a variable u in our **2-SAT**instance. For each edge $(u, v) \in G$, add a clause $(\neg u \vee v)$ to the **2-SAT**instance. This clause ensures that if u is reachable, then v must also be reachable, that is, if we are at u, and there is an edge from u to v, then v must also be true. Additionally, add a clause asserting that s is true, as we want to check if t can be reached from s. Then the **2-SAT**instance is satisfiable if and only if there exists a path from s to t in G.

   If s is true, and there is a path from s to u in G, then the variable u must be true by a chain of implications, thus, if t is reachable from s, then we can propagate truth values through the clauses to make t true as well, hence, the **2-SAT**instance is satisfiable. Conversely, if the **2-SAT**instance is satisfiable, then there is a satisfying assignment for every clause in the instance, which means that there must be a path from s to t in G.

   The reduction can be done in logarithmic space as we only need to store the current node while traversing the graph and the current clause while constructing the **2-SAT**instance. Thus, the reduction can be done in logarithmic space.

Since **2-SAT**is in **NL**and is **NL**-hard, then **2-SAT**is **NL-COMPLETE**.