COMS W3261
Computer Science Theory
Homework #4

Alexander Roth

2014 – 11 – 12

Problems

1. Classify each of the following languages as being (i) recursive, (ii) recursively enumerable but not recursive, or (iii) not recursively enumerable. In each case briefly justify your answer.

   (a) $\{ D \mid D$ is a deterministic finite automaton that accepts at least one string with an equal number of a's and b's $\}$.

Solution: This language is recursive. We can simulate a DFA D that accepts at least one string with an equal number of a's and b's, and we can create a Turing machine M that simulates this DFA. Thus, M will always halt whenever D halts, which occurs either when it accepts the string or rejects said string.

   (b) $\{ (D_1, D_2) \mid D_1$ and D_2 are deterministic finite automata and $L(D_1) = L(D_2) \}$

Solution: This language is recursive. We can construct a Turing machine M that simulates the table-filling algorithm that determines if two DFA's are equivalent. Thus, we can check for equivalence between D_1 and D_2; if we find that $L(D_1) = L(D_2)$, M will halt and accept, otherwise it will halt without accepting.

   (c) $\{ (M, w) \mid M$ is a Turing machine that halts on input $w \}$.

Solution: This language is recursively enumerable. We can construct a Turing machine Q that simulates the Turing machine M. Thus, we can simulate M on Q. However, we do not know if M will ever halt; thus, Q may never halt unless M halts and accepts. Therefore, it is recursively enumerable.

   (d) $\{ (M_1, M_2) \mid M_1$ and M_2 are Turing machines and $L(M_1) = L(M_2) \}$

Solution: This language is not recursively-enumerable. To test the equivalency between M_1 and M_2, we must check that M_1 and M_2 accept and

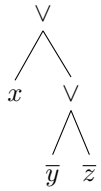
reject the same input strings. Typically, the scenario to test equivalence between M_1 and M_2 would be recursively enumerable if there would most likely be a finite set of accepting strings. However, suppose that one of the machines, M_1 in this case, has an infinite set of strings to which it halts and accepts. Thus, we cannot even say for certain if the machines are equivalent since we will continue to check halting strings in an infinite set. Thus, the language is not recursively-enumerable.

2. Boolean expressions

   (a) Describe an algorithm in high-level pseudocode to evaluate a boolean expression given a truth assignment to its variables. Describe how your algorithm evaluates the boolean expression $x \vee \neg(y \wedge z)$ for the truth assignment $T(x) = 0$, $T(y) = 0$, $T(z) = 1$. (1 is true, 0 is false)

Solution: We can construct a parse tree where the leafs of the tree are all literals of the boolean expression, and any negation we have found on the way gets pushed down to the literals. From there, we can traverse up the tree, evaluating the truth values as we go. Once we reach the top of the tree, we have figured out the truth values for the whole boolean expression.

We can construct a parse tree for $x \vee \neg(y \wedge z)$ as follows:



Now, we read up the tree. Thus, $(\overline{y} \vee \overline{z})$ becomes $(1 \vee 0)$, which evaluates to 1. From there, we have $(x \vee 1)$, which is true.

   (b) What is the time complexity of your algorithm?

Solution: The time complexity for this algorithm is $O(n)$. We must first construct a parse tree, which takes $O(n)$, and the read up the parse tree, which is again $O(n)$ time. Thus, $O(n + n)$ is $O(n)$ overall.

   (c) Describe an algorithm in high-level pseudocode to determine whether a boolean expression always evaluates to true on all truth assignments to its variables (is a tautology). Illustrate how your algorithm performs on the boolean expression $x \vee \neg(y \wedge x)$.

Solution: In order to determine if a boolean expression always evaluates to true, we shall test with every possible truth assignment for every variable using the algorithm described in 3a. If in the course of this algorithm, we find that one of the combinations does not evaluate to true overall, we know that it cannot be a tautology.

For example, $x \vee \neg(y \wedge x)$ has 4 possible solutions that prove it is a tautology

x	y	$\neg(y \wedge x)$	$x \vee \neg(y \wedge x)$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	1

(d) What is the time complexity of your algorithm?

Solution: The time complexity to build the parse tree is $O(2^n)$ as it is necessary to test each truth value for n variables. Therefore, there are two possible choices for each variable.

3. The game PEBBLES is played on an $k \times n$ chessboard. Initially each square of the chessboard has a black pebble, or a white pebble, or no pebble. You play the game by removing pebbles one at a time. You win the game if you can end up with a board in which each column contains only pebbles of a single color and each row contains at least one pebble.

   (a) Show that the set of winnable PEBBLES game is in NP. Hint: Describe a nondeterministic polynomial-time algorithm to determine whether a given PEBBLES board is winnable.

Solution: The set of winnable PEBBLES boards is in $\mathcal{NP}$. Construct a nondeterministic polynomial algorithm that will determine if a color should be completely removed from a given row. After each of the rows has been guessed, the algorithm determines if the board is winnable. This algorithm removes every possible subset of pebbles and accepts the only winnable one.

   (b) Given a boolean expression E in 3-CNF with k clauses and n variables, construct the following $k \times n$ board: If literal x_i is in clause c_j, put a black pebble in column x_i, row c_j. If literal $\neg x_i$ is in clause c_j, put a white pebble in column x_i, row c_j. Show that E is satisfiable if and only if this PEBBLES game is winnable.

Solution: We must create a mapping between the PEBBLES game and the 3-CNF E and show that the map is bidirectional. If we were to map the PEBBLE game to a CNF, each row would be a clause while each column would be a literal. A literal cannot have more than one truth value, which corresponds to either having one column of a single color (black or white). From the 3-CNF E, we see that a black pebble will represent true and the white pebble will represent false. In the game, we will remove all white pebbles when the literal is true and remove all black pebble when the literal is false.

If a row in the game is empty, this is equivalent to the saying the clause is a fallacy. If a column is empty, then this literal does not appear in boolean expression; however, this cannot be the case as there must be a truth assignment for all n variables, as per the instructions.

Therefore, we see that the PEBBLES game is equivalent to the 3-CNF E through a mapping of its characteristics and requirements for satisfiability.

(c) What can you conclude from showing (a) and (b)?

Solution: Any 3-CNF problem is $\mathcal{NP}$-complete; thus, E is NP-complete. From section (a), we know that the game is in $\mathcal{NP}$. Since we were able to reduce the PEBBLES game to the 3-CNF satisfiability problem, we are able to show that the PEBBLES game is also $\mathcal{NP}$-complete.

4. A polynomial-time verifier for a language L is an algorithm V such that $L = \{\, w \mid V \text{ accepts } (w, c) \text{ in polynomial time for some polynomial-length string } c \text{ called a certificate or proof of membership in } L \,\}$. Show that NP is equivalent to the class of languages that have polynomial-time verifiers.

Solution: We must convert the polynomial time verifier into a polynomial time Nondeterministic Turing machine, and we must convert the Nondeterministic Turing machine into a polynomial time verifier. The Turing machine simulates the verifier by guessing the certificate. Likewise, we shall use the accepting branch of the machine as the accepting branch for the verifier.

Let us assume that L is in $\mathcal{NP}$. We have a verifier V that verifies L in polynomial time. We can construct a Nondeterministic Turing machine M such that given input w, we nondeterministically select a certificate c which is at most polynomial in length. Thus, we test V on the input of (w, c). If V accepts, M should accept; otherwise, M will reject.

Assume that L is decided by a polynomial time Nondeterministic Turing machine M. We can construct a verifier V such that, given input (w, c), V checks whether c becomes an accepting branch of M given input w. If c is an accepting branch, M accepts; otherwise, M rejects w.

5. The classes P and NP

(a) Show that if language A is $\mathcal{NP}$-complete and A is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Solution: Suppose A is both $\mathcal{NP}$-complete and in $\mathcal{P}$. Then all languages L is $\mathcal{NP}$ reduces in polynomial- time to A. If A is in $\mathcal{P}$, then L is in $\mathcal{P}$.

(b) Show that if A is $\mathcal{NP}$-complete and A is polynomially reducible to a language B in $\mathcal{NP}$, then B is $\mathcal{NP}$-complete.

Solution: We need to show that every language L in $\mathcal{NP}$ polynomial-time reduces to B. We know that there is a polynomial-time reduction of L to A, since A is already NP-complete. Thus, a string w in L of length N is converted to a string x in A of length at most $p(n)$ where $p(n)$ is the polynomial time reduction.

We know that there is a polynomial-time reduction of A to B; this reduction takes polynomial time $q(m)$. This reduction transforms x to some string y in B, taking time at most $q(p(n))$. Thus, the transformation of w to y takes time at most $p(n) + q(p(n))$, which is

a polynomial. Therefore, L is polynomial-time reducible to B. Since L could be any language in $\mathcal{NP}$, we know that all of $\mathcal{NP}$ polynomial-time reduces to B; thus, B is NP-complete.