

Solutions for cs154 Extra Credit Problems

Homework #1

Part (a) Let x and y be strings and L be any language. We say that x and y are *distinguishable* if some string z exists whereby exactly one of the strings xz and yz are members of L ; otherwise, for every string z , $xz \in L$ whenever $yz \in L$ and we say that x and y are *indistinguishable* by L . If x and y are indistinguishable by L , we write $x \equiv_L y$. Show that \equiv_L is an equivalence relation.

Solution: Straightforward to show that \equiv_L is reflexive, symmetric and transitive.

Part (b) Let L be a language and let X be a set of strings. We say that X is *pairwise-distinguishable* by L if every two distinct strings in X are distinguishable by L . Define the *index* of L to be the maximum number of elements in any set that are pairwise distinguishable by L . The index of L may be finite or infinite. Show that if L is recognized by a DFA with k states, L has index at most k .

Solution: Proof by contradiction. Let us assume that set X consists of at least $k + 1$ strings, pairwise distinguishable by some regular language L (with a k -state DFA). For each string in X , let us trace the path in the DFA, beginning at the start-state of the DFA. The number of DFA states is only k while $|X| \geq k + 1$. Therefore, by Pigeonhole Principle, at least two strings in X must lead to the same state. We claim that these two strings are indistinguishable from each other with respect to L . This is because for any suffix z , the extensions of both strings end up in the same state. This means that either both extensions are accepted or both are rejected. This contradicts the assumption that all strings in X are pairwise distinguishable with respect to L .

Part (c) Let L be a language. Show that if the index of L is a finite number k , it is recognized by a DFA with k states.

Solution: From Part (a), we know that \equiv_L is an equivalent relation. Let $X = \Sigma^*$. For string $x \in \Sigma^*$, let $[x]$ denote the equivalent class of x .

Claim: If $[x] = [y]$ then $\forall a \in \Sigma : [xa] = [ya]$

Proof: By contradiction. If $[x] = [y]$ then strings x and y are indistinguishable from each other with respect to L . Let us assume that for some $a \in \Sigma$, it is true that $[xa] \neq [ya]$. However, if $[xa] \neq [ya]$ then there is some string z such that $xaz \not\equiv_L yaz$. This means that $w = az$ has the property that $xw \not\equiv_L yw$, a contradiction.

The DFA that recognizes L has one state per equivalence class defined by \equiv_L . The transition function is simple: $\forall a \in \Sigma : \delta([x], a) = [xa]$. This is well defined because of the claim above. The start state is $[\epsilon]$. The final states is the set of equivalence classes $\{[w] \mid w \in L\}$. Thus L is recognized by a k -state DFA.

Part (d) From (b) and (c), conclude that L is regular iff it has finite index. Moreover, its index is the size of the smallest DFA recognizing it.

Solution: From (b) and (c), it immediately follows that L is regular iff it has finite index.

Proof that the index of L is the size of the *smallest DFA* accepting it: Saying that a set X has index k with respect to a language L is equivalent to saying that the largest set of strings, pairwise distinguishable by L , has size k . Let us assume that a DFA with fewer than k states accepts L . Then from part (a), we know that its index is strictly less than k . However, the existence of X presents a clear contradiction.

Homework #2

Prove that if CFG G has no *self-embedding* non-terminal, then $L(G)$ is regular. A non-terminal A is said to be self-embedding if there exists a derivation $A \xRightarrow{*} uAv$, where u and v are both non-empty strings of terminals and non-terminals.

Solution: This problem appears to be complex. We present just the key ideas here. Proofs of claims and lemmas are omitted for brevity.

We first remove all ϵ -productions from G to arrive at a grammar G' . The algorithm for removal of ϵ -productions is the same that is used when converting G into Chomsky Normal Form.

Claim: A non-terminal A is self-embedding in G iff A is self-embedding in G' .

We now analyze the productions of the grammar. Here are a couple of simple observations:

Claim: If $A \xRightarrow{*} Ax$ where $x \neq \epsilon$, then it is not possible that $A \xRightarrow{*} yA$ where $y \neq \epsilon$. Loosely speaking, A cannot have both right- and left-recursion with non-empty w 's. If it does, then A will be self-embedding. Similarly, if $A \xRightarrow{*} Bx$ where $x \neq \epsilon$, then it is not possible that $B \xRightarrow{*} yA$ where $y \neq \epsilon$. The proof is along similar lines.

Let us define a relation \Leftrightarrow^* between pairs of non-terminals as follows. We will say that $A \Leftrightarrow^* B$ iff $A \xRightarrow{*} B\alpha_1 \xRightarrow{*} A\alpha_2$ or $A \xRightarrow{*} \alpha_3B \xRightarrow{*} \alpha_4A$, where α_2 and α_4 are non-empty strings of terminals. Using the claims above, it can be shown that \Leftrightarrow^* is an equivalence relation.

Let us define another relation \rightarrow^* between pairs of non-terminals. We will say that $A \rightarrow^* B$ iff $A \xRightarrow{*} \alpha_1B\alpha_2$ where α_1 and α_2 are (possibly empty) strings of terminals. It is clear that relation \Leftrightarrow^* is a subset of relation \rightarrow^* . Further, it can be argued that the equivalence classes defined by \Leftrightarrow^* can be partially ordered by using the relation \rightarrow^* . This is equivalent to computing the Strongly Connected Components (SCCs) of the graph defined by relation \rightarrow^* . Each SCC corresponds to an equivalence class of \Leftrightarrow^* . The partial order among the various SCC's corresponds to the partial order of equivalence classes of \Leftrightarrow^* .

Given that a partial order over the SCCs exists, we topologically sort the SCCs (equivalence classes of \Leftrightarrow^*) and traverse the SCCs bottom up. We claim that we can create regular expressions for all non-terminals of an SCC as we traverse the topologically sorted list of SCCs bottom up. Each SCC corresponds to either *left-recursion* or *right-recursion*. Let us see how we handle SCCs with right-recursion. Left recursion is handled similarly. Let A_1, A_2, \dots, A_k belong to the current SCC. We create an NFA with one state per A_i . Every production that expands A_i has the form $A_i \rightarrow \alpha$ or $A_i \rightarrow \alpha A_j$ where A_j belongs to the current SCC and α is a string of terminals and non-terminals such that each non-terminal is below A_i in topological ordering. Let $\alpha = x_1X_1x_2X_2\dots x_kX_k$ where x_1, x_2, \dots are strings of terminals and X_1, X_2, \dots are non-terminals. We create an NFA that recognizes x_1 , makes an ϵ -transition to a copy of the NFA for X_1 out of which there is an ϵ -transition to an NFA that recognizes x_2 , and so on. Finally, if the last non-terminal A_j belongs to the current equivalence class, there is an ϵ -transition to the state for A_j .

Homework #3

Prove the following stronger form of the pumping lemma, wherein we require that both pieces v and y to be non-empty when w is broken up:

If L is a context-free language and $G = (V, \Sigma, R, S)$ is a CFG in CNF for L , then there is a number p where, if w is any string in L of length $|w| \geq p$, then w may be divided into five pieces $w = wvxyz$ such that:

- $\forall i \geq 0 : wv^ixy^iz \in L$

- $v \neq \epsilon$ and $y \neq \epsilon$
- $|vxy| \leq p$

In your proof, make sure to demonstrate the pumping length p that works for language L , in terms of the grammar G (e.g., in terms of the sizes $|V|$, $|R|$, etc.)

Solution: We know that G is in CNF. Consider a string in $L(G)$ so large that some non-terminal in its parse tree *must occur at least thrice* along a path from the root to some leaf. How large a string do we need? If the length is $p = 2^{2^{|V|+1}}$, then every path from root to leaf has length at least $2|V| + 1$. By Pigeonhole Principle, some non-terminal must appear at least three times.

Consider the last three occurrences of some non-terminal A that appears at least three times in the parse tree. The string derived by the topmost (among the last three) A cannot be more than p in length. Let us denote the string derived by the last occurrence of A by s . Let us denote the string derived by the last but one occurrence of A by r_2sr_3 . Finally, let us denote the string derived by the topmost (among the last three) occurrence of A by $r_1r_2sr_3r_4$. We know that $|r_1r_2sr_3r_4| \leq p$. Let the overall string be $qr_1r_2sr_3r_4t$. There are two cases:

- $r_1r_2 \neq \epsilon$ and $r_3r_4 \neq \epsilon$. Then we can interleave the first two occurrences of A repeatedly to claim that the string $q(r_1r_2)^k s (r_3r_4)^k t \in L(G)$ for any $k \geq 0$. This satisfies the strong form of the Pumping Lemma.
- $r_1r_2 = \epsilon$. Then $r_3r_4 \neq \epsilon$ because the grammar is in CNF (which has no ϵ -production except for the start symbol). In this case we can repeat the topmost (among the last three) A in the parse tree k times. Then we repeat the middle A k times, followed by one derivation of the bottommost A . This generates the string $qr_1r_2s(r_3)^k(r_4)^k t \in L(G)$. Once again, the strong form of the Pumping Lemma is satisfied.

Homework #4

Part (a) Say that a write-once TM is a single-tape TM that can alter each tape square at most once (including the input portion of the tape). Show that this variant TM model is equivalent to the ordinary TM model.

Solution: We show how a Turing Machine M can be simulated by a write-once TM. The new TM will use four additional symbols: $\{\#, X, \pounds, \}\}$.

Place a marker $\#$ immediately after the end of the input string. Copy the input string (onto the blanks to the right of $\#$) leaving *three blanks* between each symbol. Move over to the first character of the copy of the input just made and start simulating M . Whenever M moves one square right or left, the new machine's head *skips blanks* and moves *four squares* right or left. Whenever M needs to *write* a symbol onto the tape, we invoke a routine called COPY.

COPY places a marker \pounds in the blank immediately preceding the current symbol. The role of \pounds is simply to remember the current head position. The machine then moves over to the rightmost character of the current copy of the string and places a marker $\#$ at the end. Then it starts copying the string between the last two $\#$ symbols (onto the blank tape following the last $\#$). While making the copy, the TM has to cross out the portion of the source string that has already been copied. We cross out characters using the symbol X . Note that we do not overwrite any character; we place the X 's on one of the three blanks between successive characters. While copying, if the \pounds symbol is encountered, the transition function of the original machine M is simulated as follows. Instead of copying the old character, the *new character* is written. Also, the new *head position* of M in the

new copy is remembered by using a marker § in one of the three blanks. When COPY has finished, the TM searches for the last occurrence of § and resumes its simulation of M .

The three blanks between characters are used for the three markers: £, X and §. No tape square is written more than once.

Part (b) (Rice's Theorem) Let \mathcal{P} be any problem about Turing machines that satisfies the following two properties. As usual we express \mathcal{P} as a language.

- a. For any TMs M_1 and M_2 , where $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in \mathcal{P}$ iff $\langle M_2 \rangle \in \mathcal{P}$. In other words, the membership of a TM M in \mathcal{P} depends only on the language of M .
- b. There exist TMs M_1 and M_2 , where $\langle M_1 \rangle \in \mathcal{P}$ and $\langle M_2 \rangle \notin \mathcal{P}$. In other words, \mathcal{P} is non-trivial — it holds for some, but not all, TMs.

Show that \mathcal{P} is undecidable.

Solution: For any non-trivial property \mathcal{P} , we can always find two languages L_1 and L_2 such that $L_1 \subset L_2$ and $\mathcal{P}(L_1) \neq \mathcal{P}(L_2)$ where $L(M_1) = L_1$ and $L(M_2) = L_2$. For example, we could choose $L_1 = \{\}$. Then L_2 exists since \mathcal{P} is non-trivial (this is the choice made in Hopcroft-Motwani book). Alternatively, we could choose $L_2 = \Sigma^*$. Then L_1 exists since \mathcal{P} is non-trivial (this is the choice Sipser makes in his proofs). There might be other choices for L_1 and L_2 .

We reduce A_{TM} to \mathcal{P} . For convenience, let us denote the machines for languages A_{TM} and \mathcal{P} by the same symbols, namely A_{TM} and \mathcal{P} . Here is the reduction: Upon receiving the input $\langle M, w \rangle$, A_{TM} constructs a machine X we will shortly describe. It then feeds $\langle X \rangle$ to \mathcal{P} . If $\mathcal{P}(L_1)$ is true, A_{TM} accepts if and only if \mathcal{P} rejects. Otherwise, $\mathcal{P}(L_2)$ is true and A_{TM} accepts if and only if \mathcal{P} accepts. Assuming \mathcal{P} is decidable, A_{TM} becomes decidable, which is a contradiction. Therefore, \mathcal{P} must be undecidable.

Machine X is a three-tape TM into which $\langle M, w \rangle$ is hard-coded. Let us denote the input to X by string x . Machine X first makes a copy of x onto the second tape. It then checks whether $x \in L_1$. For this purpose, it simulates M_1 on x on the first tape. If M_1 accepts, then X writes w onto the third tape and starts simulating M on w . If M accepts w , then X checks whether $x \in L_2$ (by simulating M_2 on x sitting on the second tape). Now X accepts iff M_2 accepts. X has the following crucial property: If M accepts w , then X accepts L_2 , otherwise X accepts L_1 .

Homework #5

(Cat and Mouse Game) A two-player game is played on a grid of $N \times N$ squares that resembles a maze. Each square has at most four neighbours: in the North, South, East and West directions. Two neighbours are either *connected* or there is a *wall* separating them. At the beginning of the game, a *cat*, a *mouse* and a *piece of cheese* are placed in three different squares. Further, one of the corner squares is a *hole* where the mouse can hide.

The game is played as follows: Mouse moves first, then cat, then mouse, and so on. The cat wins if it catches the mouse, that is, if it lands on the same square as the mouse. The mouse wins if it (a) takes the cheese, and then (b) reaches its hiding place. If at any point the mouse reaches the hole without first getting the cheese, or if the position of cat and mouse repeats, the game is a draw.

An instance of the cat and mouse game G consists of the following: (a) description of the maze (whether two neighbours are connected or separated by a wall), (b) the position of the hole and (c) the initial positions of cat, mouse and cheese. If each player has unlimited time to think, then we can predict the outcome of the game from the beginning. Consider the following languages:

$L_{win} = \{G \mid G \text{ is a } N \times N \text{ cat and mouse game and the cat wins} \}$
 $L_{draw} = \{G \mid G \text{ is a } N \times N \text{ cat and mouse game and the result is a draw} \}$
 $L_{lose} = \{G \mid G \text{ is a } N \times N \text{ cat and mouse game and the mouse wins} \}$
 Prove that all three languages are in P.

Solution: Yet to write. *Key idea:* Game tree evaluation.

Homework #6(a)

A **2cnf-formula** is an AND of clauses, where each clause is an OR of at most two literals. Let $2SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 2cnf-formula}\}$. Show that $2SAT \in P$.

For a 2cnf-formula over n variables and k clauses, we construct a directed graph with $2n$ variables and $2k$ edges. For every variable x , there are two nodes denoted by x and \bar{x} respectively. For every clause $(\ell_1 \vee \ell_2)$, there are two edges $(\bar{\ell}_1, \ell_2)$ and $(\bar{\ell}_2, \ell_1)$.

Claim: The 2cnf-formula has no satisfying assignment iff there exists some node x in the graph such that there are paths from both x to \bar{x} and from \bar{x} to x .

Proof: The edges in the graph correspond to *implications*, two per clause. For example, if the clause is $(\ell_1 \vee \ell_2)$, then the edge $(\bar{\ell}_1, \ell_2)$ is equivalent to $\bar{\ell}_1 \Rightarrow \ell_2$. In other words, *if $\bar{\ell}_1$ is TRUE, then ℓ_2 must be TRUE in order to satisfy the clause $(\ell_1 \vee \ell_2)$* . Therefore, if there is a path from x to \bar{x} and also from \bar{x} to x , then by transitivity, we derive $(x \Rightarrow \bar{x}) \wedge (\bar{x} \Rightarrow x)$, a contradiction.

Algorithm: For each variable x , see if there are paths from x to \bar{x} and from \bar{x} to x . If both paths exist for any variable, then reject the input; otherwise accept. Checking for existence of a path between a pair of nodes can be done in linear time. Thus the algorithm is polynomial overall.

Note: Actually, the problem is equivalent to computing Strongly Connected Components (SCCs) of the graph and checking if any pair of complementary vertices belong to the same SCC. There is a linear time algorithm to compute SCCs. Thus 2cnf-formula can actually be recognized in linear time.

Homework #6(b)

Prove that PRIMES is in NP.

Solution: (Fermat's Little Theorem) For any natural number n , if there exists b such that (a) $b^{n-1} = 1 \pmod n$ and (b) $b^d \neq 1 \pmod n$ for any proper divisor d of $n-1$, then n is prime.

Certificate for primality of n :

- A number b such that $b^{n-1} = 1 \pmod n$
- The prime factorization of $n-1 = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$
- Certificates that p_1, p_2, \dots, p_k are primes.

The certificate is defined recursively. However, the recursion depth is only $\log n$.

Verification of the Certificate:

- Verify that $b^{n-1} = 1 \pmod n$
- Verify that p_1, \dots, p_k are primes using the certificates bundled as the certificate of n 's primality.
- Check that $b^{(n-1)/p_i} \neq 1 \pmod n$, for any $1 \leq i \leq k$.

The whole verification is actually linear time.

Notes: PRIMES was actually shown to be in P recently by a team of two undergraduates and a professor at Indian Institute of Technology, Delhi. This is said to be one of the neatest results in CS Theory in the past decade.