Computer Science 313

David Ng

Spring 2016

Contents

1 May 9, 2016

1.1 Strings and Languages

Definition. An **alphabet** is an arbitrary finite set containing elements called symbols, characters, or letters. An alphabet will be denoted by Σ.

Example. *For instance, $\Sigma = \{a, b\}$ is an alphabet containing two elements.*

Definition. A **string** is a finite sequence of zero or more symbols from Σ. The symbol Σ^* denotes the set of all possible strings over Σ. The **length** of a string s is denoted by $|s|$. The **concatenation** of strings $s_1 = ab$ and $s_2 = ba$ is $s_b = s_1 \cdot s_2 = abba$. It may also be denoted by $s_b = s_1 s_2$.

Remark. The empty string is denoted by ϵ.

Example. *$s_1 = abab$ and $s_2 = 001$ are examples of strings.*

Definition. A **substring** of s is a string obtained from s by deleting zero or more symbols from the beginning or the end of s. A **subsequence** of s is a string obtained from s by deleting zero or more symbols from anywhere in s.

Remark. We note that a substring is a continuous sequence of characters from the string, whereas this is not the case with a subsequence.

Example. *Given the string "$EXCITINGCOURSE$", "$EXCITING$" is a substring and "$EXCITCOURSE$" is a subsequence.*

Definition. A **language** is a set of string over some finite alphabet.

Example. *For instance, $L_1 = \{0, 1\}^*$, $L_2 = \{\epsilon\}$ and $L_3 = \{a^n b^m : m \geq 0\}$, are examples of languages.*

Remark. We note that a character a or string s raised to a power m indicates that character or string concatenated m times.

Definition. The **concatenation** of two languages A and B is $A \cdot B = \{xy : x \in A \text{ and } y \in B\}$.

Example. *Let $A = \{hocus, abraca\}$ and $B = \{pocus, dabra\}$. Then $A \cdot B = \{hocuspocus, hocusdabra, abracapocus, abracadabra\}$. We note that the operation is not commutative.*

Remark. The empty set concatenated with a string A is the empty set, while the set of the empty string concatenated with a string A is A. These properties are commutative.

Definition. The **kleene star** L^* of a language L is the set of all the strings obtained by concatenating a sequence of zero or more strings from the language L.

Example. $\epsilon^* = \epsilon$.

Example. *Let* $\Sigma = \{a, b, c\}$, $L_1 = \{a^i b^i : i \geq 0\}$ *and* $L_2 = \{a^i b^i c^i : i \geq 0\}$. *Is it the case that* $L_1 \cap L_2 = \varnothing$*?*

This is false since the empty string $\epsilon \in L_1 \cap L_2$.

Example. *Let* $\Sigma = \{a, b, c, d\}$, $L_1 = \{a^i b^i : i \geq 0\}$ *and* $L_2 = \{c^i d^i : i \geq 0\}$. *Is it the case that* $L_1 L_2 = \{a^i b^i c^i d^i : i \geq 0\}$*?*

This is false since the exponents may be different.

2 May 11, 2016

2.1 Regular Languages

Remark. Note that all finite languages are regular???????

3 May 16, 2016

3.1 Building Languages

Example. *Let* L *be any language. Prove that* $L = L^+$ *if and only if* $LL \subseteq L$.

In the forward direction, we first assume that $L = L^+$. Because $LL \subseteq L^+$, then $LL \subseteq L$.

In the reverse direction, we suppose that $LL \subseteq L$. Now, we will show that $L = L^+$. $L \subseteq L^+$ by definition of L^+. To prove that $L^+ \subseteq L$, we let $w \in L^+$. Then w is $s_1 s_2 ... s_n$ such that $n \geq 1$ and $s_i \in L$. We can reduce the number of strings needed to represent w since we know that $s_1 \in L$ and $s_2 \in L$. Thus, $s_1 s_2 \in LL$. But $LL \subseteq L$, so $s_1 s_2 \in L$. We repeat this process to show that w in a string from L.

3.2 Regular Expressions

Definition. A language L is **regular** if and only if it satisfies one of the following conditions:

- L is empty.
- L contains a single string (which could be the empty string ϵ).
- L is the union of two regular languages.
- L is the concatenation of two regular languages.
- L is the Kleene star of a regular language.

Example. *Let L be the language such that every pair of adjacent 0's appear before every pair of adjacent 1's. We wish to define a regular expression for L.*

Let L_1 be a language such that is does not contain 11, and let L_2 be a language such that it does not contain 00. Then we note that $R(L) = R(L_1) + R(L_2)$. Furthermore, the we can deduce that $R(L_1) = (0+10)^*(1+\epsilon)$ while $R(L_2) = (1+01)^*(0+\epsilon)$. Therefore

$$R(L) = (0+10)^*(1+\epsilon) + (1+01)^*(0+\epsilon)$$

3.3 Finite Automata

Definition. A **deterministic finite automaton** is a state machine such that each node has exactly one outgoing transition for each character in the alphabet. For each state, there must be exactly one transition for each letter in Σ.

Remark. Note that in each state, the DFA remembers only the current state.

Theorem. *Palindromes are not regular. That is, a DFA cannot be constructed to accept only palindromes.*

Proof. Suppose to the contrary that there were a DFA for palindromes. Let N be the number of states. Then, there necessarily exists at least two palindromes x and y where $x \neq y$ such that after reading half of each palindrome, the DFA is in the same state. Let us denote this state as q.

Since y is a palindrome, then there is a path from q to a final state using the last half of y. But then a string which is formed with the first half of x and the last half of y would reach the final state despite not being a palindrome. This is a contradiction. $\square$

4 May 18, 2016

4.1 Finite Automata Cont'd

Definition. A **non-deterministic finite automaton** is a state machine such that each node has zero or more outgoing transitions for each character in the alphabet. For each state, there may be 0, 1, or more transitions for each letter in Σ.

Remark. We may have ϵ-transitions in NFA's, where there are no transitions for a given character in the alphabet.

In the case of NFA's, we may accept a particular sequence of inputs if at least one of the series of choices of states dictated by the input leads to an accepting state. We also note that any DFA is indeed an NFA.

4.2 Properties of Non-Deterministic Finite Automaton

An NFA may be described by the tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states.
- Σ is the alphabet.
- $\delta : Q \times \Sigma_\epsilon \to 2^Q$ is the transition function.
- q_0 is the starting state.
- $F \subseteq Q$ is the set of final states.

Note that 2^Q is the number of subsets of Q, and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. An NFA $N = (Q, \Sigma, \delta, q_0, F)$ accepts $w \in \Sigma^*$ if there is a sequence of states $n_0, n_1, ..., n_k \in Q$ and $w = w_1 w_2 ... w_k$ with $w_i \in \Sigma_\epsilon$ such that

1. $n_0 = q_0$
2. $n_{i+v} \in \delta(n_i, w_{i+v})$ for $i = 0, 1, ..., k-1$
3. $n_k \in F$

Definition. A language is considered **regular** if there is an NFA which accepts it. We shall refer to this definition for the following theorems.

Theorem. *The set of regular languages is closed under the union operation.*

Proof. Let L_1 and L_2 be two languages. Then there are NFA's which accept each of them. We now provide a starting state with ϵ-transitions to the original starting states of L_1 and L_2. We obtain a new NFA such that it represents the union of $L_1 \cup L_2$. $\square$

Theorem. *The set of regular languages is closed under concatenation.*

Proof. Let L_1 and L_2 be two languages. Then there are NFA's which accept each of them. We attach ϵ-transitions from each accepting state of L_1 to the starting state of L_2. We obtain a new NFA such that it represents the union of L_1 with L_2. $\square$

Theorem. *The set of regular languages is closed under the kleene star operation .*

Proof. Let L be a regular language. We need to show that L^* is also a regular language. It follows that an NFA representing L could be made to represent L^* by including ϵ-transitions from the accepting states to the starting state. Furthermore, we need to accept the empty string, so we may include epsilon transitions from the starting state to the accepting states. Therefore $L^* = \bigcup_{i=0}^{\infty} L^i$ is regular. $\square$

Remark. Note that although $L^0 = \{\epsilon\}$ is regular, $L^1 = L$ is regular, $L^2 = LL$ is regular (by concatenation), and so on... It is not the case that this shows L^* is regular. Thus we note that infinite union and concatenation are not regular.

Theorem. *For every NFA, N, there exits a DFA, M, such that $L(N) = L(M)$. That is, the language accepted by N and M is the same.*

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$ and $M = (Q', \Sigma, \delta', q_0', F')$. We will define $\epsilon(s)$ as

$$\epsilon(s) = \{q \in Q | q \text{ is reachable from some } s \in S \text{ by taking 0 or more } \epsilon\text{-transitions}\}$$

Now, we have

1. $Q' = 2^Q$ is the number of subsets of Q
2. $q_0' = \epsilon(\{q_0\})$
3. $F' = \{R \in Q' | f \in R \text{ for some } f \in F\}$
4. $\delta' : Q' \times \Sigma \to Q'$, such that $\delta'(R, a) = \bigcup_{\rho \in R} \epsilon(\delta(\rho, a))$

□

Remark. An NFA may be represented by a DFA with an exponential amount of states.

4.3 Kleene's Theorem

Theorem. *A language L can be described by a regular expression $\iff$ L is accepted by a DFA.*

Proof. Proof of this theorem follows from the following established properties:

1. Every DFA can be transformed into an equivalent NFA.
2. Every NFA can be transformed into an equivalent DFA.
3. Every regular expression can be transformed into an equivalent NFA (by closure theorems).
4. Every NFA can be transformed into an equivalent regular expression.

□

Theorem. *Every language described by a regular expression is accepted by an NFA.*

Proof. We prove by induction that every language described by a regular expression is accepted by an NFA with exactly one accepting state different from the starting state. Let R be any regular expression over some alphabet Σ.

Base Case: When $R = \varnothing$, $L(R) = \varnothing$. this corresponds to a starting state and a final state, where there is no transition from the starting state to the accepting state. When $R = \epsilon$, $L(R) = \{\epsilon\}$. This corresponds to an NFA where there is an ϵ-transition from the starting state to the accepting state. When $R = a$ where $a \in \Sigma$, $L(R) = \{a\}$. This corresponds to an NFA with a starting state which transitions to the accepting state on input a.

Induction Hypothesis: Let R be of the form $R = ST$, so that $L(R) = L(S)L(T)$. We simply follow the rules for concatenation and use an ϵ-transition from the accepting state of S to the starting state of T. The cases for union and kleene star are similar. □

5 May 25, 2016

5.1 Properties of Non-Deterministic Finite Automaton Cont'd

Definition. The **reversal** of a language L is denoted and defined by

$$L^R = \{w^R | w \in L\}$$

which consists of all the strings in L reversed.

Example. *Given that $w = abc$, find w^R.*

w^R would be cba.

Definition. The **compliment** of a language L is denoted and defined by

$$\bar{L} = \Sigma^* \backslash L$$

Example. *Find the compliment of $L = \{a, abc\}$.*

$\bar{L} = \Sigma^* \backslash \{a, abc\}$.

Theorem. *The set of regular languages is closed under reversal.*

Proof. Let N be an NFA for a language L. We shall assume that N has a single final state. To construct an NFA which accepts L^R, we simply reverse the direction of the transitions from each state. We obtain an NFA such that the accepting state becomes the starting state, and vice versa. In the case that N has multiple final states, we may simply construct an NFA with a single final state by taking ϵ-transitions from the original final states to this final state. □

Theorem. *The counting language $L = \{0^n 1^n | n \geq 0\}$ is not regular.*

Proof. Let M be a DFA for a language L with k states. Let us assume that this language is regular. We will show by contradiction that the language is not regular. We choose an $n > 2k$. Since $n > 2k$ and M must accept 0^n1^n, then there exists a state which is traversed at least twice by the sequence of 0's. But then since the current state is the only memory of a DFA, the DFA cannot distinguish between these two strings and hence cannot correctly recognize the language. $\square$

Example. *Let $A \subseteq \{0,1\}^*$ be a regular language, and let $B = \{uv|u, v \in \{0,1\}^*$ and $u\sigma v \in A$ for some $\sigma \in \{0,1\}\}$. Show that B is regular.*

Let M be a DFA for A. We will take two copies of M to form M_0 and M_1. We construct an NFA for B which we denote N. We take the starting state of M_0 to be the starting state of N, and take the accepting state of M_1 to be the accepting state of N. We then take ϵ-transitions from each state in M_0 to each state in M_1 in which there exists a transition to in M_0 (This simulates the deletion of a character). This is an NFA which describes B. Therefore, B is regular.

5.2 Context-Free Grammar

Definition. A **context-free language** is a language that can be built from strings using unions, concatenation, kleene star and recursion.

Definition. A **context-free grammar (CFG)** is a structure given by $G = (V, \Sigma, R, S)$ where

- V is a set of non-terminal symbols (uppercase Latin).
- Σ is a set of terminal symbols, disjoint from V.
- R is a set of production rules detailing how each non-terminal can be converted to a string of terminals and non-terminals.
- S is a start symbol that is an element of V.

Example. *Given the rule of replacing $S \to 0S1$, we obtain* $00S11 \to 000S111 \to 0000S1111...$

Let G by a CGF, and S be a start symbol. Then the language accepted by G is denoted and defined by

$$L(G) = \{w \in \Sigma^* | S \overset{*}{\implies} w\}$$

where $\overset{*}{\implies}$ means that we can transform S into w by a sequence of productions.

Theorem. *The language accepted by the grammar $G|(S \to 0S1,\ S \to \epsilon)$ is the language $L(G) = \{0^n1^n | n \geq 0\}$.*

Proof. a) We prove that $S \overset{*}{\Longrightarrow} 0^n1^n$ for any $n \geq 0$ by induction on n.

In the base case, $S \to \epsilon$. This is true since ϵ is a part of the language. Now, suppose that $S \overset{*}{\Longrightarrow} 0^k1^k$ for any $0 \leq k < n$. Then

$$S \to 0S1 \to 0(0^{n-1}1^{n-1})1 = 0^n1^n$$

b) For every $w \in \Sigma^*$ such that $S \overset{*}{\Longrightarrow} w$, we have $w = 0^n1^n$ for some n. We again prove by induction. Suppose that for any $x \in \Sigma^*$ with $|x| < |w|$ and $S \overset{*}{\Longrightarrow} x$, then $x = 0^k1^k$ for some k. In the case that $w = \epsilon$, this implies that $w = 0^01^0$. Otherwise, $w = 0x1$ for some x. Furthermore, $|x| < |w| \implies x = 0^k1^k$ for some k. This implies that $w = 0^{k+1}1^{k+1}$.

□

Example. *Generate all palindromes given the language* $PAL = \{w \in \{0,1\}^n | w = w^R\}$.

We note that it would be of the form $S \to (0S0)|(1S1)|0|1|\epsilon$.

Example. *Generate the language* $L = \{w \in \{0,1\}^* |$ *w starts and ends with the same symbol and* $|w| \geq 2\}$.

We note that it would be of the form $S \to (0T0)|(1T1)$ where $T \to (0T)|(1T)|\epsilon$.

Example. *Generate the language* $L = \{0^n1^n0^m1^m | n \geq 0,\ m \geq 0\}$.

First, we separate the language into two languages where $L_1 = \{0^n1^n | n \geq 0\}$ and $L_1 = \{0^m1^m | m \geq 0\}$ where $L = L_1L_2$. For L_1 we have $S_1 \to (0S_11)|\epsilon$ and for L_2 we have $S_2 \to (0S_21)|\epsilon$. For language L then, it becomes $S \to S_1S_2$.

Example. *Generate the language* $L = \{0^n1^m0^m1^n | n \geq 0,\ m \geq 0\}$.

We will proceed by solving for the outer part, and then the inner part by grouping $0^n(1^m0^m)1^n$. The outer part is generated by $S \to (0S1)|T$. The inner part is generated by $T \to (1T0)|\epsilon$.

Example. *Generate the language* $L = \{w | \#(0,w) = \#(1,w)\}$.

We note that this is the language where the number of 0's is the same as the number of 1's. It is of the form $S \to (0S1)|(1S0)|(SS)|\epsilon$.

Example. *Generate all numbers without leading 0's.*

It would be of the form $S \to 0|(TN)$, $T \to 1|2|3|4|5|6|7|8|9$, $N \to (DN)|\epsilon$ and $D \to 0|1|2|3|4|5|6|7|8|9$.

6 Midterm Questions

The midterm is out of 31 marks, and contains the following form of questions:

1. 8 short answer questions.
2. 6 true/false questions with justification.
3. Construct a DFA and corresponding regular expression.
4. Construct an NFA.
5. Construct a context free grammar (bonus question).

Remark. We note that the counting language and the language of palindromes are two examples of non-regular languages.

7 June 6, 2016

7.1 Context-Free Grammar Cont'd

We note that in practice, context-free grammars are used in programming languages where parsing by the compiler is used to interpret the language.

Definition. A **parse tree** is a tree encoding the steps in a derivation of a context-free grammar. The internal nodes represent the non-terminals used in the derivation, while the leaf nodes represent the terminals. By employing inorder traversal, we generate the string of the particular derivation.

Definition. A CFG is **ambiguous** if there exists a string can be derived in at least two different ways. We note that we may show that a CFG is ambiguous by constructing two distinct parse trees which generate the same string.

Remark. It may be possible to eliminate ambiguity by rewriting the CFG, though there is no fixed procedure for doing this. There are some CFG's that are inherently ambiguous and cannot be rewritten without ambiguity.

7.2 Chomsky Normal Form

We note that the following grammar is ambiguous.

$$S \to SSSS|\epsilon$$

This is because we can form the empty string in many different ways. To reduce the possibility of ambiguity, we utilize the Chomsky Normal Form.

Definition. A CFG is in **Chomsky Normal Form** if it satisfies all of the following conditions:

1. The starting non-terminal S does not appear on the right side of any production rule.
2. The starting non-terminal S may have the production rule $S \to \epsilon$.
3. The right side of every other production rule is either a terminal of a string of exactly two non-terminals.

Remark. We note that the parse tree of a CNF is a full binary tree.

Definition. A sequence of parentheses is **balanced** if the total number of left parentheses equals the total number of right parentheses, and for all prefixes of the sequence, the number of left parentheses traversed is greater than or equal to the number of right parentheses traversed.

Example. *Generate the language of balanced parentheses.*

We note that it would be of the form

$$S \to [S]|SS|\epsilon$$

where [ ] denotes the left and right parentheses.

Theorem. *If x is balanced, then $S \overset{*}{\Longrightarrow} x$.*

Proof. Suppose that x is balanced. For the base case, when the length of x is 0, we have that $x = \epsilon$. But this is generated by $S \to \epsilon$. Now, suppose that we can generate any balanced string of length $< n$. Let x be such that $|x| = n$. There are two cases to consider: whether there is a proper prefix y of x such that it is balanced, or not. In the first case, we note that using the fact that x is balanced and y is balanced, we take z to be the part of x which comes after y which can be generated by S since $|z| < n$ and z is balanced. Thus, x is balanced. In the second case, then $x = [z]$ for some z. In this case, x can be generated by the rule $S \to [S]$. Furthermore, z is balanced since the left parentheses and right parentheses is simply the number of those is x subtract and in every prefix of z, the number of left parentheses is greater than or equal to the number of right parentheses due to x being balanced.

□

7.3 Dynamic Programming (Optional)

Dynamic programming is the practice of dividing a problem into multiple sub-problems. Each sub-problem is solved independently and the result is stored, the results are combined for the full solution to the problem.

Example. *Given a sequence of integers, find the length of the longest increasing subsequence (Note that this may differ from the longest substring).*

We begin by splitting the problem into the smaller sub-problem of determining the longest increasing subsequence that starts at each index of the sequence. We then find the largest of these to determine the solution.

8 June 13, 2016

8.1 Midterm Review

Example. *If $x = a_1a_2...a_n$ and $y = b_1b_2...b_n$ are two strings of the same length n, define $alt(x, y)$ to be the string in which the symbols of x and y alternate, starting with the first symbol of x, that is*

$$alt(x, y) = a_1b_1a_2b_2...a_nb_n$$

If L and M are languages, define $alt(L, M)$ to be the language of all strings of the form $alt(x, y)$ where x is a string in L and y is a string in M of the same length. If L and M are regular languages, prove that $alt(L, M)$ is regular.

Since L and M are regular, suppose that we have a DFA for each such that $L(D_1) = L$ and $L(D_2) = M$. Let us consider the ordered triple of (q_1, q_2, b), where $q_1 \in D_1$, $q_2 \in D_2$ and $b = 0/1$. If $b = 0$, the first DFA takes a step, and if $b = 1$, then the second DFA takes a step. The start state is given by $(q_{01}, q_{02}, 0)$ and the final state is therefore given by $(f_1, f_2, 0)$, where $f_1 \in F_1$ and $f_2 \in F_2$. The transition function is given

$$\delta((q_1, q_2, b), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), q_2, 1) \text{ if } b = 0 \\ (q_1, \delta_2(q_2, 0), 0) \text{ if } b = 1 \end{cases}$$

8.2 Properties of Context-Free Grammar

Theorem. *Given CFG's A and B over the same alphabet then $A \cup B$, AB and A^* are context-free.*

Proof. We may formalize the CFG's as $G_A = (V_A, \Sigma, R_A, S_A)$ and $G_B = (V_B, \Sigma, R_B, S_B)$. Thus, $L(G_A) = A$ and $L(G_B) = B$. We also let $V_A \cap V_B = \varnothing$, which we may construct by simply using new terminals to represent the values.

1. $A \cup B$ can be produce by adding a new start variable S, and the following production rules

$$S \to S_A | S_B$$

$$R_A$$

$$R_B$$

2. AB can be produced by adding a new start variable S, and the following production rules

$$S \to S_AS_B$$

$$R_A$$

$$R_B$$

3. A^* can be produced by adding a new start variable S, and the following production rules

$$S \to SS_A|\epsilon$$

$$R_A$$

$\square$

Theorem. *Every regular language is context-free.*

Proof. We note that for any regular expression over an alphabet Σ, we can construct a context-free grammar:

1. $R = \varnothing$: $S \to S$
2. $R = \epsilon$: $S \to \epsilon$
3. $R = a$: $S \to a$

By structural induction, we may use the closure properties for union, concatenation and kleene star to show that every regular expression may be expressed as a context-free grammar. $\square$

Example. *Let A be a regular language, and let M be the corresponding DFA for A. We have $M = (Q, \Sigma, \delta, q_0, F)$. Define a CFG for A.*

We may define a corresponding CFG for A as $G = (V, \Sigma, S, R)$ as follows. For each state $q_i \in Q$, we introduce a non-terminal X_i. We define the following production rules for R such that $\forall k, m \in \{0, 1, ...m - 1\}$ and $\forall \sigma \in \Sigma$ we have the following:

$$\delta(q_k, \sigma) = q_m \implies X_k \to \sigma X_m$$
$$\forall q_f \in F \implies X_f \to \epsilon$$

Remark. An example of a language that is not context-free is $L = \{a^m b^m c^m | m \geq 0\}$.

Theorem. *The family of context-free languages is not closed under intersection or complementation.*

Proof. We give a counterexample. Let $L_1 = \{a^n b^n c^m | m \geq 0, n \geq 0\}$ and $L_2 = \{a^n b^m c^m | m \geq 0, n \geq 0\}$. We note that both of these languages are context-free, since they are formed by using the concatenation of the counting language with another context-free language. We note that the intersection of $L_1 \cap L_2 = L$, where $L = \{a^m b^m c^m | m \geq 0\}$ which is not a context-free language.

We use De Morgan's law to note that

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

We assume to the contrary that the context-free languages are closed under complement. This means that $\overline{L_1}$ and $\overline{L_2}$ are context-free. But we know that context-free languages are closed under union, so $\overline{L_1} \cup \overline{L_2}$ is context-free. But then $\overline{\overline{L_1} \cup \overline{L_2}}$ is context-free. By De Morgan's law, this is $L_1 \cap L_2$, which we know is not context-free. $\square$

8.3 Dynamic Programming Cont'd (Optional)

Example. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Does M accept a string w?*

We represent the states in a boolean array such that $F[q_i] = 1 \iff q_i \in F$, and $F[q_i] = 0$ otherwise. We represent the transition function as a two-dimensional array such that $D(q_k, \sigma, q_m) = 1 \iff q_m \in \delta(q_k, \sigma)$ and $D(q_k, \sigma, q_m) = 0$ otherwise. We can then write a function *Accept* defined as follows

$$Accept(q_i, w) = \begin{cases} 1 \text{ if } w = \epsilon \text{ and } q_i \in F \\ 0 \text{ if } w = \epsilon \text{ and } q_i \notin F \\ Accept(n, x) \text{ if } w = ax, \text{ where } a \in \Sigma \text{ and } \forall n \in \delta(q_i, k) \end{cases}$$

However, we note that this recursive solution takes an exponential amount of running time to compute. This recursive algorithm takes exponential time since we are computing results many times, instead of storing the results of each sub-problem.

9 June 15, 2016

9.1 Algorithms

Definition. The **Floyd-Warshall Algorithm** is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices.

Let the input be a directed graph $G = (V, E)$ and $w(u \to v)$ denote the weight of the edge from u to v. We find the shortest path between all pairs of nodes. Assuming that there are no negative cycles, $\pi(u, v, n)$ denotes the shortest path between u and v where all nodes have indices at most n (except u and v). Thus, given vertices $1, 2, ..., |V|$, we need to consider all vertices $\pi(u, v, |V|)$.

When $n = 0$, we have $\pi(u, v, 0) = w(u \to v)$. When $n > 0$, $\pi(u, v, n)$ either contains n, or it does not. In the case that it does not contain n, then

$$\pi(u, v, n) = \pi(u, v, n-1)$$

In the case that it does contain n, then

$$\pi(u, v, n) = \pi(u, n, n-1) + \pi(n, v, n-1)$$

We use these recursive definitions to find the shortest path between any pair of vertices, by calling the minimum of the first expression with the second.

Definition. The **CYK Algorithm** is a parsing algorithm for context-free grammars, named after its inventors, John Cocke, Daniel Younger and Tadao Kasami.

Let the input be a string w of length n and a CFG $G = (V, \Sigma, S, R)$ in CNF. That is, we have the production rules $A \to BC$ and $A \to a$. The output is *true* if $w \in L(G)$ and $false$ otherwise.

We split the problem into a smaller problems, so we let $Generates(A, x)$ be a function which outputs *true* if $A \overset{*}{\Longrightarrow} x$.

$$Generates(A, x) = \begin{cases} 1 \text{ if } A \to x \text{ and } |x| = 1 \\ 0 \text{ if } A \not\to x \text{ and } |x| = 1 \\ \forall_{A \to BC} \forall_{x=yz} Generates(B, y) \wedge Generates(C, z) \text{ otherwise} \end{cases}$$

9.2 Class of Languages

We note that regular languages are a subset of context-free languages. Context-free languages are a subset of P (The set of languages for which we may decide membership in polynomial time). P is a subset of NP, which is the complexity class for which we may verify in polynomial time. It is not known whether P = NP.

9.3 Turing Machines

Early attempts to rigorously define an algorithm resulted in Alonzo Church's λ-calculus. In 1936, Alan Turing presented the Turing machine, encapsulating the idea of what is an algorithm. A Turing machine is defined as a one way infinite tape, on which is written symbols from a finite alphabet, with a read/write head. Formally, a Turing machine is represented as $M = (Q, q_0, q_{accept}, q_{reject}, \Sigma, \Gamma, \delta)$:

1. Q: A finite set of states.
2. q_0: The starting state.
3. q_{accept}: The accepting states.
4. q_{reject}: The rejecting states.
5. Σ: A finite input alphabet (Does not include blank).
6. Γ: A finite tape alphabet (Σ with blank).

7. δ: A transition function of the form $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R\}$ where $Q' = Q \setminus \{q_{accept}, q_{reject}\}$.

The **configuration** of a Turing machine presents the position of the head on the tape, the content of the tape, and the state of control.

$$C = uqv$$

where

1. $u \in \Gamma^*$ is the string before the head.
2. $q \in Q$.
3. $v \in \Gamma^*$ is the string after the head.

A Turing machine M accepts an input $w \in \Sigma^*$ if there is a sequence of states from q_0w (the initial configuration) to a configuration that contains q_{accept} in a finite number of steps. A Turing machine M rejects an input $w \in \Sigma^*$ if there is a sequence of states from q_0w to a configuration that contains q_{reject} in a finite number of steps.

10 June, 20, 2016

10.1 Turing Machines Cont'd

The **language** of a Turing machine M is denoted and defined as

$$L(M) = \{w \in \Sigma^* | M \text{ accepts } w\}$$

1. $L \subseteq \Sigma^*$ is **recognizable (recursive enumerable)** by a Turing machine if there exists a Turing machine M with $L(M) = L$.
2. $L \subseteq \Sigma^*$ is **decidable (recursive)** if there is a Turing machine M that accepts every $w \in L$ and rejects every $w \notin L$.

Remark. We note that every decidable language is recognizable, but the converse is not true. If $w \notin L$, where L is recognized by a Turing machine, it may either reject or loop. A Turing machine that halts on all inputs is **total**.

Example. *Determine whether* $L = \{0^{2^n} | n \geq 0\}$ *is decidable.*

We note that on input w, M rejects w if it is not of the form 0^+. We then repeat the following steps:

1. Accept if there is exactly one 0 is on the tape.

2. Reject if the number of 0's on the tape is odd and greater than 1.

3. Cross off every second 0 on the tape using a new symbol.

Thus, if $w = 0^n$, the machine halts after k iterations of the repeated steps, where k is the largest value such that $2^k \leq n$.

10.2 The Church-Turing Thesis

Definition. Let $\langle M \rangle$ denote the **encoding** of M as a binary string.

Theorem. *Every multi-tape Turing machine has an equivalent single-tape Turing machine.*

Proof. Suppose we have multiple tapes with independent read/write heads. Initially, we have input on the first tape. The transition function is $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}$. Let $k = 2$, where k indicates that a Turing machine M has two tapes. We can encode the configuration of M into a string C of the form

$$C = u_1 q a_1 v_1 \# u_2 q a_2 v_2$$

where a_i denotes the symbol that the head is reading.

Now, we define a Turing machine M', which operates on input w. First, we write the starting configuration of $C_0 = q_0 w \# q_0 \text{\textvisiblespace}$ of M onto the tape of M'. We then repeat the following:

1. Read the configuration C of M on the tape M'.

2. Accept if C is an accepting configuration of M.

3. Reject if C is a rejecting configuration of M.

4. Make a second pass to update the tape of M' such that it becomes the next configuration of M.

□

Example. *Let M be a single tape Turing machine with q states in the control and let w be an input of length m such that when processing w, the machine does not move its head left in the first $m + q + 1$ steps. Prove that M never moves its head left on input w.*

We note that we need $m - 1$ steps to traverse the initial input. For the next $q + 2$ steps, the machine reads only blanks. By the pigeonhole principle, there is a state q_i which M enters at least twice after traversing the input. But this means that when q_i encounters a blank, it deterministically reaches a new state q_k. But then this is a loop, so the head

11 Final Questions

1. True and False questions.
2. Short answer with justification.
3. DFA/NFA.
4. CFG/TM.
5. Reduction.
6. Bonus.

12 June 22, 2016

12.1 Universal Turing Machine

Theorem. *There exists a Turing machine U such that on input $\langle M, w\rangle$ simulates the Turing machine M on w.*

Proof. We may consider a multi-tape Turing machine where one tape is the input, while the other is the output. We construct a Turing machine U_0 with two tapes. On input $\langle M, w\rangle$, U_0 does the following:

1. Write the start configuration of M, $C_0 = q_0 w$ as a binary string $\langle C_0\rangle$ on the first tape.
2. Write the description of M as a binary string on the second tape.
3. Locate on the first tape the state of M.
4. Locate on the first tape the symbol under the read/write head of M, which we denote a.
5. Look up on the second tape the part from the transition function that is required, $\delta(q, a)$.
6. Accept or reject if the current state is an accepting or rejecting state for M.
7. Update the first tape with the current state of M.
8. Repeat steps 3 to 7 as necessary.

□

Theorem. *The Acceptance problem*

$$\mathit{Accept} = \{\langle M, w\rangle | M \text{ accepts on } w\}$$

is recognizable.

Proof. We utilize the universal Turing machine U which accepts the input $\langle M, w\rangle$ if and only if M accepts w. But then, U is a recognizer for *Accept*. □

Example. *Find a single tape Turing machine capable of recognizing $L = \{0^n 1^n | n \geq 0\}$ in O(n log n) steps.*

We first ensure that w is not of the form $0^k 1^l$ for some integers k and l. Until all characters are crossed out, we make sure that the parity of 0's is the same a the parity of 1's. We cross out every other 0 and every other 1. Repeat the last two steps until either the string is rejected, or every character is crossed out. We accept the string if there are no more characters.

12.2 Diagonalization

Definition. Two sets A and B have the same **cardinality** if there exists a bijective function $f : A \to B$.

Definition. A set is **countable** if it has the same cardinality as as the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, ...\}$.

Theorem. *Every set S has a cardinality less than that of their power set.*

Proof. Suppose that S is a countably infinite set. We use diagonalization to show that $P(S)$ is uncountably infinite. We list the elements of S horizontally. Suppose to the contrary that $P(S)$ is countable. Then there is a way to list all possible subsets of S as $X = (x_1, x_2, x_3...)$. We encode each subset as a binary string, with 1 meaning that the element is in the subset, and 0 meaning that the element is not in the subset. But then, we can construct a new subset of S by flipping the values along the diagonal, call x_n. Since this is a subset of $P(S)$, then it should have been in the list. This is a contradiction from construction of this new subset since $x_n \in X$ and $x_n \notin X$ so $P(S)$ is uncountable. □

12.3 Undecidability

Theorem. *Not every language is decidable.*

Proof. We want to show that there exists a language that is not decided by any Turing machine. Note that every turing machine may be represented as a binary string. But we know that languages are elements of the power set of all binary strings. Thus, there are languages that are not decided by any Turing machine. □

Let us define the following languages:

$$\begin{aligned}
SelfAccept &= \{\langle M\rangle | M \text{ accepts } \langle M\rangle\} \\
SelfReject &= \{\langle M\rangle | M \text{ rejects } \langle M\rangle\} \\
SelfHalt &= \{\langle M\rangle | M \text{ halts on } \langle M\rangle\}
\end{aligned}$$

Theorem. *$SelfAccept$ is undecidable.*

Proof. Suppose to the contrary that there exists a Turing machine SA that decides $SelfAccept$. That is

$$\begin{aligned} SA \text{ accepts } \langle M\rangle &\iff M \text{ accepts } \langle M\rangle \\ SA \text{ rejects } \langle M\rangle &\iff M \text{ does not accept } \langle M\rangle \end{aligned}$$

We let SA^R be the Turing machine obtained from SA by swapping its accept and reject states. That is, SA^R rejects whenever SA accepts and vice versa. Now we consider SA^R to get

$$\begin{aligned} SA \text{ accepts } \langle SA^R\rangle &\iff SA^R \text{ accepts } \langle SA^R\rangle \\ SA \text{ rejects } \langle SA^R\rangle &\iff SA^R \text{ does not accept } \langle SA^R\rangle \end{aligned}$$

But this is a contradiction due to the definition of SA^R. Thus, $SelfAccept$ is undecidable. □

Theorem. *$SelfReject$ is undecidable.*

Proof. Suppose that there exists a Turing machine SR that decides $SelfReject$. That is, for any Turing machine M

$$SR \text{ accepts } \langle M\rangle \iff M \text{ rejects } \langle M\rangle$$

In particular, we consider SR to get that

$$SR \text{ accepts } \langle SR\rangle \iff SR \text{ rejects } \langle SR\rangle$$

This is a contradiction. Therefore, $SelfReject$ is undecidable. □

Theorem. *$SelfHalt$ is undecidable.*

Proof. Suppose there is a Turing machine SH that decides $SelfHalt$. Then, for any Turing machine M

$$\begin{aligned} SH \text{ accepts } \langle M\rangle &\iff M \text{ halts on } \langle M\rangle \\ SH \text{ rejects } \langle M\rangle &\iff M \text{ does not halt on } \langle M\rangle \end{aligned}$$

We now construct a Turing machine SH^X from SH such that it redirects any transitions to accept to a new state that hangs, and redirects any transitions to reject to accept. Thus, we consider SH^X

$$SH \text{ accepts } \langle SH^X \rangle \iff SH^X \text{ halts on } \langle SH^X \rangle$$
$$SH \text{ rejects } \langle SH^X \rangle \iff SH^X \text{ does not halt on } \langle SH^X \rangle$$

But this leads to a contradiction since SH^X hangs if and only if SH^X halts, so $SelfHalt$ is undecidable. □

Theorem. *The Halting problem is undecidable.*

Proof. Suppose H is a Turing machine that decides $Halt$ (the Halting language), where $Halt = \{\langle M, w, \rangle | M \text{ halts on } w\}$. Then, we use H to build a Turing machine SH that decides $SelfHalt$. We note that on input w, SH first checks if w is a valid description of a Turing machine. It then duplicates w on the tape and passes control to H. Thus, SH decides $SelfHalt$. But since $SelfHalt$ is undecidable, so is $Halt$. □

13 June 27, 2016

13.1 Recognizable Languages

Theorem. *$SelfAccept$ is recognizable.*

Proof. We describe a Turing machine SA which on input w, first checks that w is the encoding of a Turing machine. If not, then SA hangs (or rejects). SA writes the string ww, which is the encoding for $\langle M, M \rangle$ and passes control to the universal Turing machine U. Thus, we get that

$$U \text{ accepts } \iff M \text{ accepts } \langle M \rangle$$

Therefore, $SelfAccept$ is recognizable. □

Theorem. *A language L is decidable if and only if both L and $\overline{L}$ are recognizable.*

Proof. In the forward direction, we suppose that L is decidable. But since L is decidable, it is also recognizable. We can construct a Turing machine for $\overline{L}$ by swapping the accepting and rejecting states of the Turing machine of L.

In the reverse direction, let M_1 be a recognizer for L and M_2 be a recognizer for $\overline{L}$. We will build a Turing machine M that decides L. On input w, M does the following:

1. Run both M_1 and M_2 on input w in parallel.

2. If M_1 accepts, then M accepts. If M_2 accepts, then M rejects.

By running in parallel, this means that M has two tapes, one for simulating each of M_1 or M_2. M alternates simulation of one step of each machine until one of them accepts.

Alternatively, we can define M such that on input w, it does the following:

1. Set $k = 0$.
2. Simulate M_1 for k steps on w. Accept if M_1 accepts in k steps.
3. Simulate M_2 for k steps on w. Reject if M_1 accepts in k steps.
4. Increment k by 1.
5. Repeat Steps 2-4 as necessary.

□

Theorem. *$\overline{Halt}$ is not recognizable, where $\overline{Halt} = \{\langle M, w\rangle | M$ does not halt on $w\}$.*

Proof. We know that $Halt$ is recognizable, but not decidable. Thus, we apply the previous theorem to show that $\overline{Halt}$ is not recognizable. □

13.2 Reductions

Definition. $f : \Sigma^* \to \Sigma^*$ is a **reduction** from language A to language B if

$$w \in A \iff f(w) \in B$$

for all $w \in \Sigma^*$.

Definition. $f : \Sigma^* \to \Sigma^*$ is **computable** if there exists a Turing machine that for any $w \in \Sigma^*$, halts with $f(w)$ written on the tape.

Remark. A language A reduces to a language B, denoted as $A \leq_m B$, if there is a computable reduction from A to B.

Theorem. *If $A \leq_m B$ and A is undecidable, then so is B.*

Proof. Suppose that B is decidable and let $f : \Sigma^* \to \Sigma^*$ such that $w \in A \iff f(w) \in B$ for every $w \in \Sigma^*$. Let M_B be a Turing machine that decides B and M_f be a Turing machine that computes f. Then we can construct a Turing machine M that decides A on input w. We will define M as follows:

1. Simulate M_f on w to compute $f(w)$.
2. Simulate M_B on $f(w)$.
3. Accept if M_B accepts and reject if M_B rejects.

However, this contradicts the fact that A is undecidable. Therefore, B is also undecidable. □

Remark. To prove that a language L is undecidable, we reduce a known undecidable language to L.

Theorem. *The language*

$$Any = \{\langle M\rangle | L(M) \textit{ contains at least one string } \}$$

is undecidable.

Proof. Suppose that we can decide the language Any. We show that this implies that we can decide $Halt$. Given any input $\langle M, w\rangle$ to the Halting problem, we construct a machine $M' = f(\langle M, w\rangle)$ that on any input y does the following:

1. Erases the input y.
2. Writes $\langle M, w\rangle$ on its tape.
3. Runs M on input w (using the universal Turing machine).
4. Accepts if and only if M halts on w.

Then

$$L(M') = \begin{cases} \Sigma^* \text{ if M halts on } w \\ \varnothing \text{ if M does not halt on } w \end{cases}$$

If we can decide whether a machine accepts any string at all, we an apply this decision procedure to M'. Therefore, this would determine whether$M$ halts on w. That is, we would be able to decide $Halt$, which is a contradiction. □

Theorem. *The language*

$$Rev = \{\langle M\rangle | M \textit{ is a Turing machine that accepts } w^R \textit{ whenever it accepts } w\}$$

is undecidable.

Proof. Suppose that we can decide the language Rev. We show that this implies we can decide $Halt$. Given any input $\langle M, w\rangle$ to the Halting problem, we construct a machine $M' = f(\langle M, w\rangle)$ that on any input $y$does the following:

1. If $y = 01$, we accept.
2. if $y \neq 10$, we reject.
3. if $y = 10$, simulate M on w and accept if M halts.

Thus, if M halts on w, then $L(M') = \{01, 10\}$, so $\langle M' \rangle \in Rev$. Conversely, if $\langle M, w \rangle \notin Halt$, then $L(m') = \{01\}$, so $\langle M' \rangle \notin Rev$. This shows that $f(\langle M, w \rangle) = \langle M' \rangle$ is a reduction from $Halt$ to Rev. Since f is computable, we conclude that Rev is necessarily undecidable. □

Theorem. *The language*

$$Reg = \{\langle M \rangle | M \text{ is a Turing machine and } L(M) \text{ is regular}\}$$

is undecidable.

Proof. Suppose that we can decide the language Reg. .We show that this implies that we can decide $Halt$. Given any inout $\langle M, w \rangle$ to the Halting problem, we construct the machine $M' = f(\langle M, w \rangle)$ that on input y does the following:

1. if y has the form $0^n 1^n$, then accept.
2. if y does not have this form, then we run M on input w and accept if M halts.

If M halts on w, then $L(M') = \Sigma^*$, which is a regular language, so $\langle M' \rangle \in Reg$. Conversely, if $\langle M, w \rangle \notin Halt$, then $L(M') = \{0^n 1^n | n \geq 0\}$, which is not regular so $\langle M' \rangle \notin Reg$. This shows that $f(\langle M, w \rangle) = \langle M' \rangle$ is a reduction from $Halt$ to Reg. □

14 June 29, 2016

14.1 Selected Problems

Example. *Prove that*

$$SameC = \{\langle M_1, M_2 \rangle | M_1, M_2 \text{ are Turing machines and } L(M_1) = \overline{L(M_2)}\}$$

is undecidable.

We will reduce this to the Halting problem. Suppose to the contrary that we can decide $SameC$. Given any input $\langle M, w \rangle$ to the Halting problem, we construct an input to $SameC$ by making sure that $L(M_1) = \Sigma^*$ only if input M halts on w, and makes $L(M_1) = \varnothing$ otherwise. We ensure that $L(M_2) = \varnothing$. But then, our Turing machine for $SameC$ decides the Halting problem, which is a contradiction.

Example. *Prove that the language*

$$NonEmpty = \{\langle M \rangle | M \text{ accepts some string}\}$$

is recognizable.

Suppose that $s_1, s_2, s_3, ...$ is a list of all possible strings over the alphabet. We construct a machine R such that for $i = 0, 1, 2, ...$ we run M for i steps on each input $s_1, s_2, ..., s_i$. We note that by this way of computation, we are able to traverse all strings over the alphabet. If any computation accepts, then R accepts.

Example. *Let*

$$Fin = \{\langle M\rangle | L(M) \textit{ is finite}\}$$

Prove that Fin *and* $\overline{Fin}$ *are both not recognizable.*

Suppose to the contrary that $\overline{Fin}$ is recognizable by the Turing machine F. We will reduce $\overline{Halt}$ to $\overline{Fin}$. On any input $\langle M, w\rangle$ to $\overline{Halt}$, we will construct a Turing machine $M' = f(\langle M, w\rangle)$, where M' on input y, first saves y onto a second tape, then runs M on w for $|y|$ steps. If M does not halt, M' accepts, and rejects otherwise. We note that if $\langle M, w\rangle \in \overline{Halt}$, then $L(M') = \Sigma^*$. Otherwise, in the case that M halts on w in n steps, M' would accept if $|y| < n$. In this case, $L(M') = \{y \in \Sigma^* || y| < n\}$.

If $\langle M, w\rangle \in \overline{Halt}$, then $M' \in \overline{Fin}$. If $\langle M, w\rangle \notin \overline{Halt}$, then $M' \notin \overline{Fin}$. Since we know that $\overline{Halt}$ is unrecognizable, then $\overline{Fin}$ is not recognizable.

Example. *Let* L *be a regular language. Let*

$$prefmax(L) = \{x \in L | xy \in L \iff y = \epsilon\}$$

Prove that $prefmax(L)$ *is regular.*

Since L is a regular language, then there is a DFA for L. We construct a DFA for $prefmax(L)$ by simply removing all outgoing transitions from the accepting states of L and removing all accepting states which have outgoing transitions which eventually lead to an accepting state.

Example. *Prove that*

$$L = \{\langle M_1, M_2, M_3\rangle | L(M_1) = L(M_2)L(M_3)\}$$

is unrecognizable.

Suppose to the contrary that L is recognizable. We will show by reduction that $\overline{Halt}$ is recognizable, thus leading to a contradiction. Given any input $\langle M, w\rangle$ to $\overline{Halt}$, we can construct an input $\langle M_1, M_2, M_3\rangle$ to L. M_3 is set to be the empty set. M is first run on input w. If M halts, then we accept. That is, $L(M_1) = \Sigma^*$ and $L(M_3) = \varnothing$. In the case that it does not halt, then we have both $L(M_1) = L(M_3) = \varnothing$.