MIT Primes 2020 Computer Science Problem Set

1 Problem 1

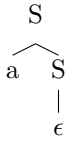
1.1 Question 1

String: ϵ

$S \to \epsilon$

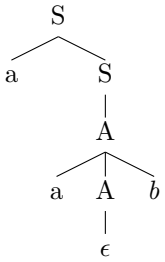
String: a

$S \to aS \to a$



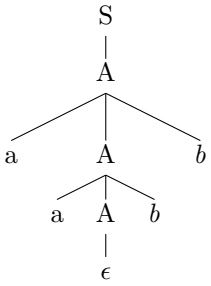
String aab

$S \to aS \to aA \to aaAb \to aab$



String $aabb$

$S \to A \to aAb \to aaAbb \to aabb$



1.2 Question 2

We see that the start symbol generates either the empty string, or a string containing a non-negative integer number of consecutive a, followed by A. The non-terminal A generates a non-negative number of a, followed by an equal number of b. Every time the terminal b is generated, the terminal a is generated directly before it. Therefore, we can see that the number of a terminals is greater than or equal to the number of b terminals. The language of the CFG is the set of all strings over a, b that are composed of a string of m number of a, followed by n number of b, where m, n are integers ≥ 0, and $m \geq n$.

2 Problem 2

2.1 Question 1

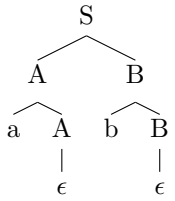
Yes, there is another derivation of ab in G_1:
$S \to AB \to aAB \to aAbB \to aAb \to ab$

2.2 Question 2

$S \to AB \to aAB \to aB \to abB \to ab$

2.3 Question 3

The parse tree for ab is as follows:



2.4 Question 4

The non-terminal A generates a string of consecutive a terminals of non-negative integer length. The non-terminal B similarly generates a string of consecutive b terminals of non-negative integer length. Since the start symbol only generates AB, we see that the language consists of all strings over a, b consisting of a string of consecutive a terminals of non-negative integer length, followed by a string of consecutive b terminals of non-negative integer length.

3 Problem 3

3.1

ab is a string derived in G that has more than one leftmost derivation:

1. $S \to aA \to aAb \to ab$
2. $S \to aA \to abA \to ab$

3.2

Let us call the grammar given in this example G_3, and the grammar given in example 1 as G_1. Let us define L_1 to be the language derived by G_1, and L_3 to be the language derived by G_3. We then prove that G_3 defines the same language as G_1, or that L_1 and L_3 are equivalent.

The set of rules in G_1 is a proper subset of the set of rules in G_3. This implies that L_1 is a subset of L_3. From example 1, we know that G_1 derives all strings over a, b that begin with the terminal a. The start symbol of G_3 only generates aA. Therefore all strings derived from G_3 must begin with the terminal a and contain only terminals a and b. L_3 is therefore a subset of L_1. Since L_1 and L_3 are subsets of each other, they must be equal, and G_1 and G_3 derive the same language.

3.3

Every rule in G_1 appends exactly 1 unique character to the end of the currently generating string. After the S generates aA, there is a bijection between the character being added to the string and the rule being used (the empty string corresponds to $A \to \epsilon$, a corresponds to $A \to aA$, and b corresponds to $A \to bA$). Building a string by adding a sequence of characters to the end of the string step by step corresponds to a unique sequence of rules. Since every unique string is a unique sequence of characters, this implies every string the L_1 corresponds to a unique sequence of rules. Each string corresponds to a unique leftmost derivation, so G_1 is unambiguous.

4 Problem 4

We define our CFG as follows:

$S \to E = S \mid E$
$E \to E + F \mid E - F \mid F$
$F \to V \mid (S)$
$V \to a \mid b \mid c \ ...$

The grammar has the following properties [1]:

1. E denotes an expression, F denotes a term (or expression surrounded by parentheses), and V denotes a variable.

2. Rules $E \to E + F$ and $E \to E - F$ make addition/subtraction left-associative, as the left-side symbol is to the left of the operator

3. The rule $S \to E = S$ makes $=$ right-associative, as the start symbol is to the right of the equality sign

4. This grammar is unambiguous, as we do not have symmetric rules like $E \to E+E$. Every rule creates a symbol with higher precedence (i.e. $E \to E+F$, the nonterminal for an expression, E, generates a new nonterminal for a term, F)

5. The rule $T \to (S)$ gives us any number of nested parentheses. Since the start symbol is within the parentheses, we see that any type of equation/expression can exist within parentheses.

Leftmost derivations:

1. $a = b + c$

   $S \to E = S \to F = S \to V = S \to a = S \to a = E \to a = E + F \to a = F + F \to a = V + F \to a = b + F \to a = b + V \to$ **a=b+c**

2. $a + b - c$

   $S \to E \to E - F \to E + F - F \to F + F - F \to V + F - F \to a + F - F \to a + V - F \to a + b - F \to a + b - V \to$ **a+b-c**

3. $a - b - c$

   $S \to E \to E - F \to E - F - F \to F - F - F \to V - F - F \to a - F - F \to a - V - F \to a - b - F \to a - b - V \to$ **a-b-c**

4. $a = b = c$

   $S \to E = S \to F = S \to V = S \to a = S \to a = E = S \to a = F = S \to a = V = S \to a = b = S \to a = b = E \to a = b = F \to a = b = V \to$ **a=b=c**

5. $a = (b + (c = d))$

   $S \to E = S \to F = S \to V = S \to a = S \to a = E \to a = F \to a = (S) \to a = (E) \to a = (E + F) \to a = (F + F) \to a = (V + F) \to a = (b + F) \to a = (b + (S)) \to a = (b + (E = S)) \to a = (b + (F = S)) \to a = (b + (V = S)) \to a = (b + (c = S)) \to a = (b + (c = E)) \to a = (b + (c = F)) \to a = (b + (c = V)) \to$ **a=(b+(c=d))**

6. $a + b = c - d$

   $S \to E = S \to E + F = S \to F + F = S \to V + F = S \to a + F = S \to a + V = S \to a + b = S \to a + b = E \to a + b = E - F \to a + b = F - F \to a + b = V - F \to a + b = c - F \to a + b = c - V \to$ **a+b=c-d**

7. $a - (b = c - d) = e$

   $S \to E = S \to E - F = S \to F - F = S \to V - F = S \to a - F = S \to a - (S) = S \to a - (E = S) = S \to a - (F = S) = S \to a - (V = S) = S \to a - (b = S) = S \to a - (b = E) = S \to a - (b = E - F) = S \to a - (b = F - F) = S \to a - (b = V - F) = S \to a - (b = c - F) = S \to a - (b = c - V) = S \to a - (b = c - d) = S \to a - (b = c - d) = E \to a - (b = c - d) = F \to a - (b = c - d) = V \to$ **a-(b=c-d)=e**

5 Problem 5

5.1 Question 1

The only nonterminal that can generate ϵ in a grammar in CNF is the start symbol. Therefore if the rule $S \to \epsilon$ exists, the grammar derives ϵ. If not, the grammar does not derive ϵ.

5.2 Question 2

We can think of a derivation as a sequential generation of the derived string. Our starting "string" after 0 derivations, the start symbol, consists of 1 nonterminal and 0 terminals. Since our final string has length n, which is greater than 0, we can ignore the rule $S \to \epsilon$.

Our desired string of size n consists of exactly n terminals. Since each terminal can only be generated with exactly one nonterminal (using a rule $V_1 \to a_1$, which we will call a "terminal rule"), a string of n nonterminals must first be generated.

We see that the only way to generate additional nonterminals is with rules of the form $V_1 \to V_2V_3$ (let us call this a "nonterminal rule"), which adds to the total number of nonterminals in a string by exactly 1. Since we start with 1 nonterminal after 0 derivations (S), it takes exactly $n-1$ steps to generate of string of n nonterminals. Converting this string of n nonterminals to a string of n terminals takes n steps, one step per nonterminal.

We see that it takes a total of $n-1+n = 2n-1$ steps. Therefore, generating a string of size n in a CNF grammar takes a derivation of length $2n-1$.

5.3 Question 3

We have noted above that every string can be generated by a sequence of $2n-1$ rules. Let us denote m to be the maximum number of rules for a *single* nonterminal in the grammar. To generate a sequence of $2n-1$ rules, we consider building a string using a leftmost derivation. At each of the $2n-1$ steps, we consider the leftmost nonterminal. There will be a maximum of m rules to choose from to replace this specific nonterminal.

Thus, in a sequence of $2n-1$ rules, our total runtime complexity is $O(m^{2n-1})$. This worst-case runtime is achieved when there are exactly m rules for each nonterminal in the grammar.

6 Problem 6

(see problem6.java for main class)

The inputted grammar is first converted into CNF form [5]. The program then uses the CYK algorithm [6] to determine whether or not the string can be derived from the inputted language.

7 Problem 7

(see problem7.java for main class)

We define our "alphabet" to be all terminal characters that exist in our grammar. The program generates strings sequentially in shortlex order given our alphabet. It checks each string using the CYK algorithm from problem 6, and outputs it if the string is defined by our grammar.

7.1 Additional Analysis

One problem that I came across during my testing is when the user inputs a number of strings (n) greater than the number of strings in our language. With the initial implementation of the algorithm given above, the program will continue forever. I decided to fix this problem in two ways:

1. The first method I thought of was to merely terminate the program after our algorithm stops finding strings in the language for a sufficiently large amount of time. This of course, is not an exact method to determine if no more strings exist in the language, but for most cases it is sufficient.

2. The second method I thought of was to find an exact upper limit on the number of strings in a CFL if the language is finite, and to terminate if we exceed this number. The Pumping Lemma for Context-Free Languages states that any string in a CFL larger than some integer size p can be generated (or "pumped") using the method in [4], if such strings exist. The application of the Pumping Lemma to our algorithm is that if we do not find any strings between sizes p and $2p$, there do not exist any strings with size $\geq p$ [3]. A sufficient value of p is $p = 2^{m-1} + 1$, where m is the number of non-terminal symbols in our grammar [2].

   If we do find a string between sizes p and $2p$, we can determine that an infinite number of strings can be "pumped" for our CFL. If we do not find any strings, we can safely terminate our program, as no other strings exist to be found.

Since the value of $p = 2^{m-1} + 1$ can be an incredibly large size for a string if there are more than a few non-terminals, I chose to use both methods to terminate my program. The program will end whichever comes first:

1. We find the sufficient number of strings that the user inputs

2. The Pumping Lemma proves that no more strings exist in the language (note that this is unlikely to occur if their are more than a few nonterminals in the CNF conversion of the grammar)

3. We generate more than 5000 strings in a row in shortlex order that are all not in the language.

7.2 Additional Method

I actually tried an additional method for this problem. The method was the "naive" method of finding all derivations of length $2n-1$, seeing if a valid string is produced, and then sorting all valid strings in shortlex order. We begin by generating strings of size 0, then 1, then 2, etc, incrementing the value of n until the correct amount of strings have been generated.

After testing, I came to find that this method had an average runtime slower than my currently submitted method for my test cases due to its high exponential complexity runtime, so I decided to not use it.

This method also has the additional downside of exponential memory complexity, as all strings need to be stored to be sorted, using much more memory than my current implementation. My current method does not actually store any strings in memory, but instead just prints them out as they are generated.

8 Problem 8

Every parse tree corresponds to a unique leftmost derivation. We list out all possible leftmost derivations for each string, and find the sum of the probabilities for each derivation.

String: a
$S \to A \to a$
Probability: $0.3 * 0.4 = \mathbf{0.12}$

String: aa
$S \to A \to aA \to aa$
Probability: $0.3 * 0.3 * 0.4 = \mathbf{0.036}$

String: abb
$S \to B \to aB \to abB \to abb$
Probability: $0.7 * 0.3 * 0.3 * 0.4 = \mathbf{0.0252}$

9 Problem 9

I believe it is impossible to develop a PCFG for a language over all strings $\{a, b\}$ that satisfies all three of the given conditions. In particular, the following two conditions are contradictory:

1. Groups of repetition of the same symbol in a row even number of times (2 and larger) appear 30% of the time groups of odd number of repetitions of the same symbol.

2. Within each group the probability of a sequence of length n is twice higher than of all lengths higher than n combined.

9.1 Proof

Since the first rule above applies to both groups of a and groups of b, we can deduce that groups of even size of **any** character appear 30% of the time groups of odd size appear for **any** character.

From the second rule above, the probability that a group of characters is of size n is twice the probability that the group is of a size greater than n. Let us denote the probability that a group of size n to be p. From the second rule, we see that the probability of a group being of size greater than n is $\frac{p}{2}$.

We can also see that the probability of a group being of size $n+1$ is twice as large as the probability of a group being of size greater than $n+1$. Since those two probabilities sum to $\frac{p}{2}$, we calculate that the probability of a group being size $n+1$ is $\frac{2}{3} * \frac{p}{2} = \frac{p}{3}$, or $\frac{1}{3}$ a group has a size of n. In general, the probability a group is of size n is 3 times the probability that it has a size of $n+1$.

The probability a group is of size 1 is twice as large as a group of size greater than 1, which is every group not of size 1. From this, we can see that the probability of a group being size 1 is $\frac{2}{3}$. The probability of a group of size 2 is thus $\frac{2}{9}$, of size 3 is $\frac{2}{27}$, etc. In general, the probability of a group being size n is $\frac{2}{3^n}$.

The probability of a group having an odd size is thus:

$$\sum_{k=0}^{\infty} \frac{2}{3^{2k+1}} = \frac{2/3}{1-(1/9)} = \frac{3}{4}$$

The probability of a group having an even size is thus:

$$\sum_{k=0}^{\infty} \frac{2}{3^{2k+2}} = \frac{2/9}{1-(1/9)} = \frac{1}{4}$$

We see that groups of an even number of repetitions appear $\frac{1}{3}$ the amount of times groups of an odd number of repetitions appear. This, however, contradicts the first rule, which claims that groups of an even number of repetitions appear 30% of the time groups of an odd number of repetitions appear.

9.2 Partially satisfied PCFG

We now construct some PCFGs that satisfy some, but not all, of the conditions in the problem. To make the conditions non-contradictory, we remove one of the two contradictory conditions, and create a PCFG that satisfies the remaining conditions.

We eliminate the following condition: "Groups of repetition of the same symbol in a row even number of times (2 and larger) appear 30% of the time groups of odd number of repetitions of the same symbol."

The two remaining conditions are as follows:

1. Within each group the probability of a sequence of length n is twice higher than of all lengths higher than n combined.

2. The probability of a string with n groups of repetition is also twice higher than of all of the strings with the larger number of groups combined.

We construct our CFG (without probabilities) as follows:

$S \to A \mid B$
$A \to aA \mid aB \mid a$
$B \to bB \mid bA \mid b$

The A nonterminal generates a group of a terminals, and the B nonterminal generates a group of b terminals. Within the three derivations of the A and B nonterminals, the first rule represents adding an additional character to the group, the second rule represents changing to a new group with a different character, and the third rule represents a termination of the string. The grammar is strategically designed such that every rule of A or B represents the generation of a single character.

Given our new set of conditions, there is no distinction between groups of a or b. Therefore, we set $S \to A$ to be equally likely as $S \to B$, so both of them have probability of 0.5.

The condition "Within each group the probability of a sequence of length n is twice higher than of all lengths higher than n combined" can be satisfied if at every step the group is twice as likely to terminate as it is to continue to grow larger. If we consider the A nonterminal, the rules $A \to aB$ and $A \to a$, both terminate the currently generated group of a, while the rule $A \to aA$ increases the size of the currently generated group without changing groups. Thus, the probability of $A \to aA$ should be $\frac{1}{3}$, and the other two rules should have probabilities that sum to $\frac{2}{3}$.

The condition "The probability of a string with n groups of repetition is also twice higher than of all of the strings with the larger number of groups combined" can be satisfied if at every step the string is twice as likely to terminate as it is to create another group. If we consider the A nonterminal, the rules $A \to aB$ creates another group, while the rule $A \to a$ terminates the string. We can ignore the rule $A \to aA$, as it neither terminates the string or creates another group. No matter how many times the rule $A \to aA$ is used, the ratio between the probabilities of terminating the string and creating another group remain the same. We can see that the rule $A \to a$ must be twice as likely as the rule $A \to aB$.

Since the probabilities for $A \to a$ and $A \to aB$ sum to $\frac{2}{3}$, the probability of $A \to a$ is thus $\frac{4}{9}$, and the probability of $A \to aB$ is $\frac{2}{9}$.

Since this grammar has identical conditions for the A and B nonterminals, the probabilities for the rules of B can be derived identically as A.

Our final PCFG is as follows:

$S \to A\ \frac{1}{2} \mid B\ \frac{1}{2}$

$A \to aA\ \frac{1}{3} \mid aB\ \frac{2}{9} \mid a\ \frac{4}{9}$

$B \to bB\ \frac{1}{3} \mid bA\ \frac{2}{9} \mid b\ \frac{4}{9}$

10 Problem 10

(see problem10.java for main class)

We simulate a leftmost derivation when we generate each string. For each string, at each step we find the leftmost nonterminal of the string. We then randomly choose a rule relative to their respective probabilities to apply to it, replacing that nonterminal with a new set of characters. This process is repeated until the string no longer has any nonterminals, at which point it has completed its derivation.

11 Problem 11

(see problem11.java for main class)

In this program we utilize the CYK algorithm generalized for PCFGs [6] to create the initial array/parse forest. We also create an array of backpointers that keeps track of the paths travelled down the CYK array. These backpointers allow us to retrace our path and generate all leftmost derivation and probabilities for each string.

For our parsing algorithm, we build the parse forest (tree of parse trees of the string) bottom-up. At each node in the parse forest, we store all partial derivations of the node (what the nonterminal represented by the node itself generates). We also utilize a method that combines all combinations of partial derivations at two child nodes into a single set of derivations at the parent node, allowing us to build the parse forest. For example if there are m partial derivations at one child node and n at the other, the parent node will store $m*n$ total derivations.

The parse forest eventually builds up to the start symbol node, which will contain all derivations of the string once the algorithm terminates.

11.1 Additional Analysis

My program is designed in a way such that it stores all possible derivations as strings at the root node of the parse forest. Given that the number of derivations grows exponentially relative to size of the string, I have found through testing that inputting very large strings can overload Java's memory capacities. I have found through testing that strings of size greater than around 20, with a grammar with many rules, can cause this error to occur.

It may be possible to use a different parsing algorithm than the one I have chosen to use. More specifically, an algorithm that parses the entire forest without actually storing all possible derivations may be able to circumvent the memory overloading error.

A possible advantage of my algorithm, however, is that by storing all partial derivations at the nodes, we only traverse each node in the forest exactly once. In a potential algorithm that doesn't store derivations in memory, it is likely

that it would have to search every single node for every derivation that passes it. This is generally an exponential number of calls for each node relative to the height, which *may* take longer time than my algorithm.

Therefore my algorithm is most likely faster than potential alternatives, even though it has a tradeoff through poor memory usage.

This issue also applies to any derivations read in Problem 12, as Problem 12 utilizes the same algorithm for parsing strings as Problem 11.

12 Problem 12

(see problem12.java for main class)

This problem uses a unique algorithm to calculate probabilities for each rule:

For each nonterminal, we begin by setting each rule associated with it to be equally likely. For example, if a specific nonterminal V_1 has 5 rules associated with it, we set the probability of each rule occurring to be 0.2. By doing this for each nonterminal, we create an intial PCFG, which we will denote as G_0.

To derive a new PCFG G_{n+1} from G_n, we use the following method:

For each of the 1000 inputted strings:

1. We use the method described in Problem 11 to find the probability of each leftmost derivation of a given string.
2. If there are multiple leftmost derivations, we choose a single one randomly according to their relative probabilities (i.e. if derivation A occurs with probability 0.1 and derivation B occurs with probability 0.3, we pick derivation A 25% of the time and derivation B 75% of the time)
3. Now that we have a leftmost derivation, we count how many times each rule is used throughout this derivation. For each rule, we add the total number of occurrences to a "total" counter.

We end with the number of occurrences for each rule throughout the derivations of all 1000 strings. Our last step is to recalculate our probabilities to be directly proportional to the distribution of the occurrences (i.e. if our final distribution for nonterminal V is:

$V \to a$: 1000 times $V \to AB$: 3000 times

We recalculate the probabilities for nonterminal V in our PCFG to be: $V \to a$ 0.25 | AB 0.75

Repeat this for every nonterminal.

As we repeatedly run this algorithm, the probabilities in our PCFG approach the actual probabilities, as they replicate the distribution of rules in the derivations for our strings. In my program, I calculate a total of 10 repetitions. Therefore the program outputs the values in G_{10}, a very close approximation to theoretical probabilities.

References

[1] CSCI 5220. *6.5. Precedence and Associativity.* URL: http://www.cs.ecu.edu/karl/5220/spr16/Notes/CFG/precedence.html. (accessed: 11.30.2019).

[2] Michal Forišek. *In the proof of the pumping lemma for context-free languages, how do we determine the minimum pumping length for which the proof works?* URL: https://www.quora.com/In-the-proof-of-the-pumping-lemma-for-context-free-languages-how-do-we-determine-the-minimum-pumping-length-for-which-the-proof-works. (accessed: 11.29.2019).

[3] Ran G. *Decidablity of Languages of Grammars and Automata.* URL: https://cs.stackexchange.com/questions/627/decidablity-of-languages-of-grammars-and-automata. (accessed: 11.29.2019).

[4] GeeksforGeeks. *Pumping Lemma in Theory of Computation.* URL: https://www.geeksforgeeks.org/pumping-lemma-in-theory-of-computation. (accessed: 11.29.2019).

[5] Sariel Har-Peled. *CS 273, Lecture 15 Chomsky Normal form.* URL: https://sarielhp.org/teach/07/b_spring_08/Lectures/lect_15.pdf. (accessed: 11.30.2019).

[6] Wikipedia. *CYK algorithm.* URL: https://en.wikipedia.org/wiki/CYK_algorithm. (accessed: 11.30.2019).