Week 9

CSE 390Z
November 22, 2022

Welcome in!

Grab your nametag and the workshop problems from the front

Announcements

- Quick Check 8 & Imposter Syndrome Reflection due Sunday (late deadline Tuesday). They're already posted so you can finish them before Thanksgiving if you'd like.
- Imposter Syndrome Reflection assignment is required! We'll be doing a panel about it next week. It's one of the most popular workshops in the past.
- Please check Gradescope Hw Corrections 1. If you received an N, please resubmit via email. Required for course credit.

Conceptual Review

Structural Induction

- A lot of students are confused by Structural Induction when they first see it, but it is the same in principle as Weak/Strong Induction.
- We prove that a statement is true about a set by assuming that it holds for an arbitrary member of the set and then showing that it must also hold for the next element in the set.

- However, in Structural Induction, we are working with Recursively Defined sets!
- Structural Induction problems have many interesting variations often involving things like Binary Trees, Strings, CFGs (foreshadowing…)

Refresher: Recursively Defined Sets

- A recursively defined set has three components:
- Basis Step – Essentially the base case for the set, often this will be 0 for a set of integers, the empty string for a set of strings, a root node for a binary tree, etc.
- Recursive Step – The rules by which we go from step K to step K+1. Note that there can be more than one!
- Exclusion Rule – This rule states that every element of the set is generated either by the basis step alone or the basis step and some combination of applications of the recursive step
- Typically, we don’t explicitly state the exclusion rule.
- These recursively defined sets are also the basis for RegEx and CFGs!

Example: Spring 22 Lecture 19

Define a set S as follows:

Basis Step: $0 \in S$

Recursive Step: If $x \in S$ then $x + 2 \in S$.

Exclusion Rule: Every element of S is in S from the basis step (alone) or a finite number of recursive steps starting from a basis step.

What is S?

Back to Structural Induction

- As we said, Structural Induction is very similar to Weak and Strong Induction conceptually but there are a few key things to keep in mind:
- Your Base Case is going to correspond with the Basis Step of the recursive set definition. We have to prove the predicate holds at the lowest level and build our argument up!
- It is possible that our recursive set will have more than one Recursive Step/Rule, if that's the case then we will need to have more than one Inductive Step in our proof. **You must have one inductive step for each recursive step/rule.**
- So how does Structural Induction even work???
- Think of each element of the set S as being generated by some K applications of the recursive rule. If we assume the predicate is true for some arbitrary element of the set and then show that the predicate still holds after every possible recursive rule application, we have proven the predicate for all members of the set.
- This is analogous to showing P(K) implies P(K+1)

Here's a template! Thanks again, Spring 22!

1. Define $P()$ Show that $P(x)$ holds for all $x \in S$. State your proof is by structural induction.

2. Base Case: Show $P(x)$ for all base cases x in S.

3. Inductive Hypothesis: Suppose $P(x)$ for all x listed as in S in the recursive rules.

4. Inductive Step: Show $P()$ holds for the "new element" given.

You will need a separate step for every rule.

5. Therefore $P(x)$ holds for all $x \in S$ by the principle of induction.

Regular Expressions (RegEx)

- Reminder: A set of strings is called a language.
- What Regular Expressions do is define a language for us, namely, the set of all strings matching some particular pattern.
- Regular Expressions are recursively defined, meaning they have a basis rule, a recursive rule, and obey the exclusion rule.
- Basis: empty string, no string, and a single character.
- The recursive rule is where the magic happens! There are three recursive rules for RegEx:

RegEx Recursive Rules (Operations)

- Union: If A and B are regular expressions then (A U B) is a regular expression that matches either A or B or both.
- Concatenation: If A and B are regular expressions then AB is a regular expression that matches the contents of A + the contents of B
- Wildcard (*): If A is a regular expression, A* is a regular expression which matches any string which can be subdivided into 0 (remember this) or more strings that match A

- Examples:
- (a U bc)
- 0(0 U 1)1
- 0*

Notes about RegEx

- While RegEx is powerful, they can’t define every possible language. For example, the set of all palindromes can’t be defined by regular expressions.
- The CSE311, simple machine version of Regular Expressions are not interchangeable with the regular expressions some of you may be familiar with in some programming languages. These languages extend our definition of regular expressions with features that make these regular expressions more powerful.
- As a result, our definition of RegEx may have some limitations that don’t exist in other versions of RegEx, just FYI!

Context-Free Grammars (CFGs): The Basics

- We can think of CFGs as being String “generators”
- Consist of an alphabet of “terminal symbols,” meaning symbols that don’t create another substitution into the resulting string, and a finite set V of nonterminal symbols which do create further substitutions.
- Each CFG has a start symbol which is one of the nonterminal symbols in V and is usually denoted S.
- Each of the nonterminal symbols has a production rule in the form of A -> w1|w2|w3..
- To generate a string using a CFG simply start with the start symbol S, then choose a nonterminal in the string and a production rule, then substitute in an element w(i) from the production rule into the nonterminal and rinse and repeat until you’re out of nonterminal characters!
- Example:

$$S \rightarrow 0S0|1S1|0|1|\varepsilon$$

Notes about CFGs

- CFGs can define more complex languages than Regular Expressions can. For example, they can be used to define the set of all palindromes, expressions with matched parentheses, arithmetic expressions, and even the syntax of the Java programming language!
- In fact, CFGs define a superset of Regular Expressions which will be proved in lecture!
- The main takeaway for RegEx and CFGs is that they give us ways of succinctly representing sets of strings.

Example Problem

Prove that every string generated by the CFG “S -> SS | 0S1 | 1S0 | ε “ contains an equal number of 1’s and 0’s

Prove that every string generated by the CFG “S -> SS | 0S1 | 1S0 | ε “ contains an equal number of 1’s and 0’s

- First things first, let’s translate this CFG to a recursively defined set
- What will our basis rule be?
- What about the recursive rule? How many will there be?

Example: Spring 22 Lecture 19

Define a set S as follows:

Basis Step: $0 \in S$

Recursive Step: If $x \in S$ then $x + 2 \in S$.

Exclusion Rule: Every element of S is in S from the basis step (alone) or a finite number of recursive steps starting from a basis step.

What is S?

Prove that every string generated by the CFG “S -> SS | 0S1 | 1S0 | ε “ contains an equal number of 1’s and 0’s

Prove that every string generated by the CFG “S -> SS | 0S1 | 1S0 | ε “ contains an equal number of 1’s and 0’s

- Now that we’ve got a recursively defined set we can set up our induction proof
- What should our predicate be?
- What is our base case?
- How many Inductive Steps will we need to complete?

Here's a template! Thanks again, Spring 22!

1. Define $P()$ Show that $P(x)$ holds for all $x \in S$. State your proof is by structural induction.

2. Base Case: Show $P(x)$ for all base cases x in S.

3. Inductive Hypothesis: Suppose $P(x)$ for all x listed as in S in the recursive rules.

4. Inductive Step: Show $P()$ holds for the "new element" given.

You will need a separate step for every rule.

5. Therefore $P(x)$ holds for all $x \in S$ by the principle of induction.

Workshop Problems