Overview of Section from Chapters 1-3

February 27, 2023

Abstract

Your test will resemble the problems from your homework assignments, quizzes, and problems from previous tests I've given. You will probably have six equally weighted questions or so (one every twelve minutes or so).

1 Section 1.1

We are introduced to statements, logical connectives, and wffs.

An implication is an argument, or theorem, which we may seek to prove. It is false if and only if the hypothesis (antecedent) is true while the conclusion (consequent) is false. The truth table for this logical connective is very important for understanding much of what follows.

Truth tables can prove tautologies (statements which are always true).

TautologyTest can prove tautologies of the form $P \to Q$, which it does by **contradiction**: assume both P and Q', and then break down each until all statement letters have truth values. If a statement letter is both true and false (a contradiction) then $P \wedge Q'$ is false, and the implication is true - a tautology.

2 Section 1.2

Propositional logic allows us to test arguments

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \to Q$$

to see if they're valid (tautologies).

Create a proof sequence using hypothesis or derivation rules (e.g. modus ponens). There are equivalence rules (such as DeMorgan's laws), and inference rules (e.g. modus tollens) which only operate in one direction.

The deduction method helps us prove implications: the antecedent joins the list of hypotheses, and we simply prove the consequent of the implication. (A related technique is called temporary hypothesis, and is explored in a later section.)

It is sometimes a challenging task to convert English arguments into wffs; but doing so allows us to cut through the verbiage to more clearly understand the argument, and whether it is true of not.

3 Section 1.3

We add a variable to statements to create predicate wffs. We then consider statements like "for all integers....", or "there is an integer such that....": that is, we **quantify** the predicate, using $\forall$ and $\exists$.

By introducing a variable we require a domain, called the domain of interpretation (non-empty).

Quantifiers have a scope, which indicates the part of a wff to which the quantifier applies.

Once again, translating English arguments into wffs is one of the tough challenges.

A few rules of thumb:

- $\forall$ tends to go with $\rightarrow$
- $\exists$ tends to go with $\wedge$

4 Section 1.4

We use predicate logic to prove predicate wffs, including new rules such as instantiation and generalization (as well as all the old familiar propositional logic rules).

Big Idea: strip off the quantifiers, use derivation rules on the wffs, and put quantifiers back on as necessary. Table 1.17 outlines limitations on stripping and putting.

A few rules of thumb:

- ei before ui.
- Don't use ug on $P(x)$ deduced from a hypothesis in which x is free, or by ei from another wff in which x is free:

$1.P(x)$	hyp
$2.(\forall x)P(x)$	incorrect ug

$1.(\forall x)(\exists y)Q(x,y)$	hyp
$2.(\exists y)Q(x,y)$	$1, ui$
$3.Q(x,a)$	$2, ei$
$4.(\forall x)Q(x,a)$	incorrect ug

Table 1: Summary of useful proof techniques, from Gersting, p. 91.

Proof Technique	**Approach to Prove** $P \to Q$	**Remarks**
Exhaustive Proof	Demonstrate $P \to Q$ for all cases.	Cases finite
Direct Proof	Assume P, deduce Q.	Standard approach
Contraposition	Assume Q', deduce P'.	Q' gives more ammo?
Contradiction	Assume $P \wedge Q'$, deduce contradiction.	

5 Section 2.1

We look at a variety of proof techniques, including exhaustion, by contradiction, by contraposition, direct; and one "disproof" technique: counterexample.

6 Section 2.2

Induction is a proof technique which is useful for demonstrating a property, or assertion of fact ($P(n)$), for an infinite ladder of propositions (think of our assertion as being indexed by n, as in $P(n)$, for n a natural number). Induction begins with a base case (or an anchor), frequently $n = 1$, and then proceeds via an inductive case (often $P(k) \to P(k+1)$).

I like to say that this is just dominoes falling (or "infinite *modus ponens*"): we know $P(1)$ (the first domino falls), and that $P(k) \to P(k+1)$ (if the k^{th} domino falls, so does the $(k+1)^{th}$ domino). Therefore we know $P(2)$; and so on. All the dominoes are falling down.

There are two different (but equivalent) principles of induction, the first and second. The second appears to assume more than the first: the inductive hypothesis in the second principle is that the property is true for all cases up to and including the k^{th} case.

In terms of dominoes, the second asserts that if all the dominoes up to the kth have fallen, then the (k+1)th domino will also fall.

7 Sections 3.1-3.3

Recursion in section 3.1 looks very much like induction: the idea is that we have a base case (or cases), and from there we generate additional cases. Unlike induction, the set of things we generate may not be easily indexed to the integers. For example,

- the palindromes on binary strings, or
- the construction of wffs using the logical connectives.

In this section we see how to solve one particular recurrence relation, the linear, first-order, constant-coefficient recurrence relation.

Once we deduce this formula (and prove it by induction), we needn't ever solve another linear, first-order, constant-coefficient recurrence relation from scratch: we can just invoke the formula. This is our quest, the holy grail!

The general method is to "guess, check, and verify": compute some terms, look for a pattern, characterize the pattern, then prove it. We used that strategy to prove the general formula for the first-order, constant-coefficient recurrence relation.

A different variety of recurrence relation occurs in the analysis of algorithms, when we consider "divide and conquer" algorithms (such as *BinarySearch*).

By a change of variables, we get a closed-form solution for the number of operations for these "divide and conquer" algorithms.

In the analysis of algorithms we are interested in efficiency, and will count operations in order to compare competing algorithms. We can sometimes count operations directly, but may resort to recursion to count. We're interested in the worst-case scenario.