# Effectiveness of the tree test for validity of arguments in SL

Effectiveness of a test is the property of always giving an answer. We must prove that the tree test will always give an answer. It will not run on forever. (In contrast, the tree test for validity of arguments in Predicate Logic can run on forever.)

Notice that this really has nothing special to do with arguments. What we need to show, is that the tree for any list of sentences will eventually finish.

First let's do a simplified version which ignores biconditionals.

THEOREM: If a list of sentences has no biconditionals, and has exactly b binary connectives and n negations, then the tree for that list will be finished in at most b+ (b+n)/2 steps. That is, b plus half of b+n.

PROOF IN THREE STEPS:

Step One: Each use of a binary connective rule leaves one less binary connective, and at most one more negation than before. You must verify that the rules for &, v, and ~⊃ each eliminate one binary connective and leave no more negations than before. The rules for ~&, ~v, and ⊃ each eliminate one binary connective and leave one more negation.

Step Two: Each time we use the ~~ rule there are two less negations and no more binary connectives.

Step Three: So, for our original list of sentences, we will apply at most b steps of binary connective rules (usually less, since some branches will close). The most negations we could possibly have to work with in the whole tree is b+n. that is, the original n, plus b new ones introduced by using binary connective rules. Since each use of the double negation rule eliminates two negations, we can use it at most (b+n)/2 times.

Trivial Exercise: How many steps does it take to finish the tree for ~(~A&~B)?

More interesting exercise: Find other trees that take exactly b+((b+n)/2) steps to finish.

Don’t read this side of the handout unless you really enjoyed the first side and you like a little math. There are actually two oddities of that proof on the first side. It ignores biconditionals. And worse, notice that it counts one "step" for each binary connective in the original sentences. So for example it counts three "steps" to finish the list (AvB), (CvD), (EvF). But when you do the tree, the "step" for the second sentence actually takes place twice, that is on two different branches, and the "step" for the third takes place on four different branches. In a way it makes more sense to count separately each "step on a branch". That list of sentences takes 7 steps on branches: One for the first sentence we handle, 2 for the second, and 4 for the third.

The way to be more precise about it, and include biconditionals, is to keep track of two things: the number of levels in the tree (that is, how lines there are on each branch), and the breadth of the tree (how many branches). But then we have to decide how close an estimate to try for.

We will use very simple overestimates: We will estimate that each time we check off a binary connective, we produce two new branches, each with two new lines. This is an overestimate for all but biconditionals. We will also estimate that each new branch contains exactly one less binary connective than the sentence we checked off (often an overestimate, and never under), plus two new negations (ditto).

So assume our original list of sentences has exactly b binary connectives and n negations. Then by the time we have checked off all the binaries we will have generated (at most) 2b new lines on any one branch, so each branch is (at most) 3b lines long. And we will have generated (at most) 2b new negations. So we can apply at most (n+2b)/2 double negation rules. No branch can be longer than 3n plus (n+2b)/2 lines long. That is (7n+2b)/2. And the tree will have doubled its branches (at most) b times. So it will be (at most) $2^b$ branches wide at the bottom.

So, for example, our tree could be written in a grid that is $2^b$ wide and (7n+2b)/2 deep. That means it will finish in less than $2^{b-1}$.((7n+2b) steps.

This is always an overestimate, and in most cases a really huge one. For one, it calculates a rectangular grid the tree would fit in, but the tree will clearly not fill that grid. One improvement would be to figure out what part of a rectangular grid the tree could fill--you will see this is like calculating the area under an exponential curve.

More detailed improvements would involve correcting the overestimates we made about how much connectives branch and deepen the tree. But notice we cannot, for example, say that "&" will not branch the tree. An & might lie inside some negation, or more deceptively it might lie in the antecedent of a conditional, so it will have a ~ applied to it when we use the conditional rule. If you want a further improvement based on knowing when branches will close, then in effect you are just running the tree itself.