Chapter 10: Truth Trees

Recall that in doing truth tables the long way we were reconstructing truth values for a sentence or set of sentences in every possible truth value assignment—and that in doing that a good deal of our work was wasted. In testing for consistency, for example, we were just looking for a row of the truth table in which all the sentences were true. If we found such a row all the other rows were irrelevant. But we had to do the complete truth table because if there was *no* row in which all the sentences were true we needed to determine that. Getting a 'yes' answer to the consistency question was easy: just find a row in which all the sentences get **T**. Getting a 'no' answer was tedious: we had to consider all the possibilities—and some of them were clearly irrelevant. Consider the truth table we did to test for consistency:

All True!

A	v	B	/	~	(A	≡	B)	/	~	B
T	T	T		F	T	T	T		F	T
T	T	F		T	T	F	F		T	F
F	T	T		T	F	F	T		F	T
F	F	F		T	F	T	F		T	F

Consistent or inconsistent? Consistent

We could have seen straightaway that Row 4 was a non-starter because in that row 'A v B' was false. No point in working through that row, plugging in **T**s and **F**s. Similarly, we could have seen that computing truth values for Row 1 was also a waste of time because '~ (A ≡ B)' is false in the truth value assignment that that row represents. We're only interested in truth value assignments that make all sentences true so, for starts we could have thrown out every row in which *any* of the sentences was are false, namely Rows 1, 3, and 4.

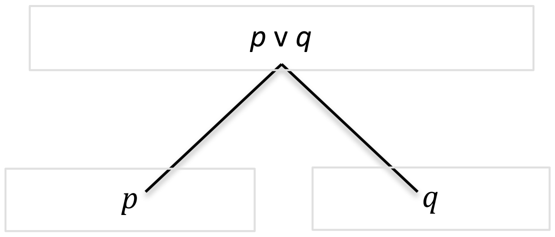
This is essentially what truth trees do. They systematically select all and only those truth value assignments in which each sentence individually is true and put them together, so that we can determine whether there's a truth value assignment in which *all* of them are true together. To do this, we grow a tree with branches that represent truth value assignments that make the sentence(s) we're testing true according to *tree rules* that represent the ways of making each of the sentences true. With one exception, these rules essentially represent conjunction or disjunction. We can think of them as logic gates through which Truth flows up the tree.

To see how this works, first consider a conjunction, '*p* • *q*'. To make it true we need Truth flowing to both *p* and *q*. So, given that Truth flows up the tree, we can represent the truth conditions for conjunction like this:

$$\begin{array}{c} p \bullet q \\ p \\ q \end{array}$$

Since Truth has to flow through both conjuncts to make the conjunction true, we hook them up in series, so to speak. You've may have seen this arrangement in Real Life. You have a light at the top of the stairs, with a switch down stairs and another up stairs hooked up in series so that both switches have to be on for the light to go one. If even one of them is off current can't flow through, so no light. You are upstairs and want to turn on the light, but no go. 'Oh ----!', you exclaim, 'the downstairs switch if off'. This is the cheap way of wiring a light and switches. This is conjunction.

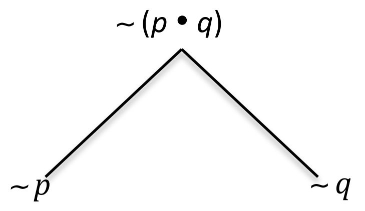
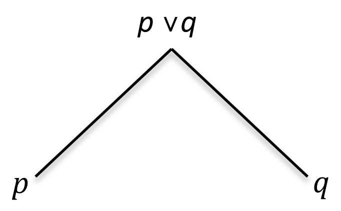
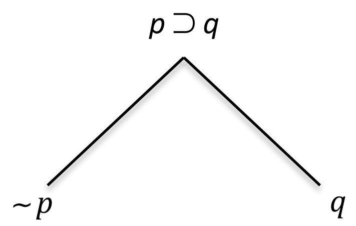
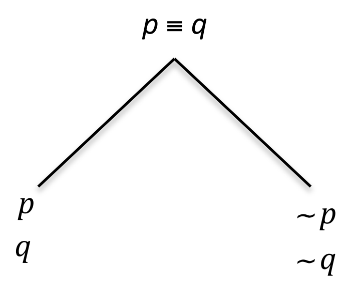
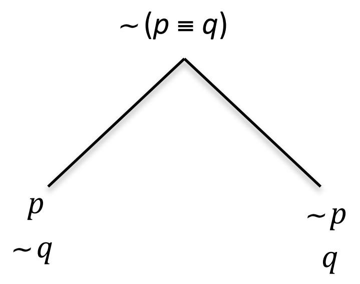
Then there is the expensive way: you hook up those switches in parallel, so that if either one of the switches is on current flows through and the light goes on. This is disjunction:



To make '$p \vee q$' true we just need truth flowing through p or through q or through both so this represents the truth conditions for disjunction. Note, we don't need an additional branch with both p and q on it. This arrangement represents all three ways of making disjunction true: p true, q true or both p and q true.

So here, in these diagrams, you have the tree rules for conjunction and disjunction respectively. It's easy to see that these tree rules come from the truth tables that define conjunction and disjunction. The tree rule for conjunction is non-branching: it shows that there's only one way to make conjunction true, viz. if both of the conjuncts are true. The tree rule for disjunction, above, is branching: it shows that there is more than one way to make disjunction true. The branches represent the three rows of the truth table for disjunction in which it's true. The complete tree rules follow. With the exception of the Double Negation rule, each is either a conjunction rule or disjunction rule.

1.1 Truth Tree Rules

Double Negation $\sim\sim p$ p	
Conjunction $p \bullet q$ p q	**Negated-Conjunction** 
Disjunction 	**Negated-Disjunction** $\sim(p \vee q)$ $\sim p$ $\sim q$
Conditional 	**Negated-Conditional** $\sim(p \supset q)$ p $\sim q$
Biconditional 	**Negated-Biconditional** 

1.2 About the Tree Rules

The rationale for the Double Negation Rule is obvious. Negation reverses truth value. Since there are just two truth values, reversing it twice gets you back to where you started: ~ ~ *p* is equivalent to *p*. Each of the other rules is either a conjunction or disjunction: non-branching rules are conjuctions; branching rules are disjunctions.

Consider, for example, the Negated-Disjunction Rule. It says that ~ (*p* v *q*) is made true by the conjunction of ~ *p* and ~ *q*. And this is exactly what we should expect: '~ (*p* v *q*)' is DeMorgan equivalent to '~ *p* • ~ *q*'. Likewise for the Negated-Conjunction Rule: '~ (*p* • *q*)' is equivalent to '~ *p* v ~ *q*'. We can read the Conditional and Negated-Conditional Rules off of the truth table for '*p* ⊃ *q*'. Conditional is true if either the antecedent is false or the consequent is true, that is, if we either have '~ *p*' or '*q*' or both: because we have just two truth values '~ *p*' says '*p* is false'. The Negated-Conditional Rule is non-branching because there is just one way to make a conditional false: true antecedent and false consequent. Finally, the Biconditional and Negated-Biconditional rules are both disjunctions of conjunctions, representing the rows of the truth table for biconditional in which it is true and false respectively. Biconditional is true if both sides have the same truth value—either both true or both false, as represented by the first and fourth rows of its truth table. Biconditional is false in the two middle rows, which represent the sides as having opposite truth value.

Exercise 3.2

Write tree rules for these two connectives:

Sheffer Stroke

p	*q*	*p* | *q*
T	**T**	**F**
T	**F**	**T**
F	**T**	**T**
F	**F**	**T**

Down Arrow

p	*q*	*p* ↓ *q*
T	**T**	**F**
T	**F**	**F**
F	**T**	**F**
F	**F**	**T**

1.3 Growing a Tree

To grow a tree, first list the sentence or sentences from which the tree will grow. This is the trunk of the tree: the *initial sentence(s)*. Unlike conventional trees, truth trees branch downwards. We apply tree rules to the sentence to grow the tree. Once a rule has been applied to a sentence the sentence is checked (√) to indicate that we're done with it. When all sentences to which tree rules can be applied—that is, all sentences other than sentence letters and their negations—are checked the tree is complete. Each branch wants to represent a truth value assignment—a row of a truth table. But what we want isn't always what we get. Recall that for each row of a truth table, all occurrences of the same sentence letter get the same truth value: you can't have a truth value assignment where a sentence is both true and false. On a truth tree we represent the falsity of a sentence by its negation. If a sentence, *p*, appears on a branch, that is, if it occupies the entire branch at a given point, then that branch assigns TRUE to *p*. If ~ *p*, appears on a branch, the branch assigns false to *p*. If both a sentence and its negation appear on a branch, then that branch hasn't succeeded in representing a truth value assignment so we chop it off, and put an 'X' under it, indicating that the branch is *closed*. To see how this works, we shall consider truth tree tests for consistency, tautologousness and validity.

1.4 TRUTH TREE TESTS

1.4.1 Consistency

Recall that a set of sentences is *consistent* iff there is some truth value assignment in which all of the sentences are true—otherwise inconsistent. To test set of sentences for consistency, therefore, we list the sentences and grow a tree from them in order to determine whether there is some truth value assignment that makes all of them true. The sentences from which the tree grows are the trunk of the tree—the *initial sentences*.

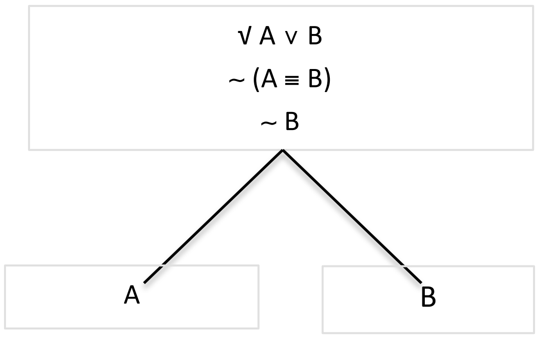
Example: Test the following set of sentences for consistency: {A ∨ B, ~ (A ≡ B), ~ B}

Step 1: List the sentences to be tested.

A ∨ B
~ (A ≡ B)
~ B

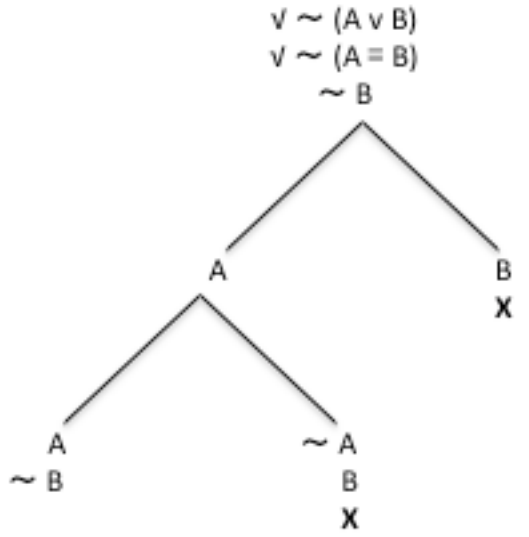
Step 2: Apply tree rules to the sentences until the tree stops growing. The order in which the rules applied doesn't matter, however, for convenience we apply non-branching rules first, so that the true doesn't get too big. Every time we apply a rule we check to see whether there is a contradiction, that is a sentence and its negation, on the branch that the rule grew. If there is we chop off the branch and put an 'X' under it.

There are no non-branching rules that can be applied to the sentences we have so let's apply the disjunction rule to the first sentence and check it off:



Note: branches go all the way from the bottom through the trunk to the top. So the branches overlap in the trunk: the left branch includes the initial sentences and 'A' while the right branch includes all the initial sentences and 'B'.

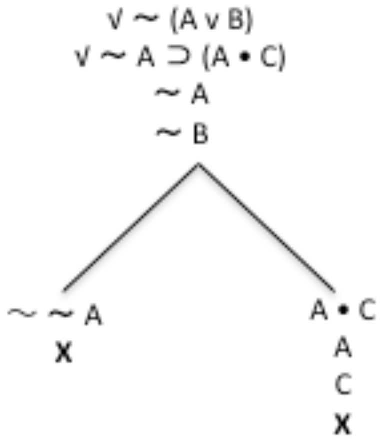
Next, we apply the Negated-Biconditional Rule to '~ (A ≡ B)' to complete the tree:



Consistent or inconsistent? *Consistent*

Now the only unchecked sentences are sentence letters or their negations, so there are no more unchecked sentences to which tree rules apply. The tree, therefore, is complete. There are four branches. Following them from bottom to the top we see that there are two branches that have contradictions on them: 'A' and '~ A' on the branch second to the left, and 'B' and '~ B' on the fourth branch. So we put 'X's on the bottom of those two branches. The other two branches however are open. That means they each of them represents a truth value assignment that makes all initial sentences true: that set of sentences is, therefore, consistent. Moreover, looking at those branches we can see what truth value assignment it is. Since each branch includes 'A' and '~ B' the truth value assignment that makes all initial sentences is true is: A – TRUE and B – FALSE. As you can see from the truth table we did earlier to check this set of sentences for consistency, that is exactly the truth value assignment represented by the second row of the truth table.

Example: Test the following set of sentences for consistency: {~ (A ∨ B), ~ A ⊃ (A • C)}



Consistent or inconsistent? *Inconsistent*

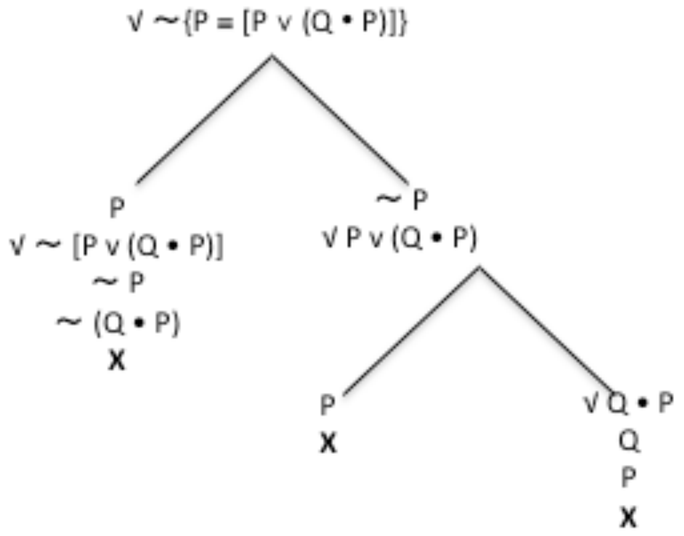
1.4.2 Tautologousness

We can also use truth trees to test sentences for tautologousness. The negation of any tautology is self-contradictory: whatever makes a tautology true makes its negation false; whatever makes its negation true makes the tautology false. This gives us an idea about how to test a sentence for tautologousness using the tree method: start with the negation of the sentence and grow the tree. If all the branches close then we know that there is no truth value assignment that makes it true, that is, that it is self-contradictory, and hence that the sentence itself is tautologous. When all the branches of a tree close we say that the tree is *closed*. So this is the test for tautologousness: grow a tree from the *negation* of the sentence to be tested; if the completed tree is closed the sentence is a tautology.

Example: Test the following sentence for tautologousness.:

$$P \equiv [P \vee (Q \bullet P)]$$

To do this, grow a tree from the *negation* of the sentence. If the tree closes, this shows that there is no truth value assignment that makes the sentence false, hence that it is a tautology. This tree shows that the sentence is a tautology:



Note that on the left-most branch we didn't apply any rule to '~ (Q • P)'. We didn't have to, because once we applied the Negation-Disjunction Rule to '~ {P ≡ [P ∨ (Q • R)]}' we got a contradiction: 'P' and '~ P' on the same branch. That closed the branch: we didn't have to go any further.

Exercise 3.4.2. Why are we going about this is such a roundabout way? Why don't we just grow a tree from the sentence itself? If all the branches were open, wouldn't that show that the sentence was a tautology? Explain why this would or wouldn't work. And if it wouldn't work, give an example of a sentence that has a tree in which all branches are open but isn't a tautology.

1.4.3 Validity

An argument is valid iff there is no truth value assignment that makes all the premises true and the conclusion is false. Recall that when we tested arguments for validity using indirect truth tables we began by assigning TRUE to each of the premises and FALSE to the conclusion and then worked through the problem in order to see whether we could construct a truth value assignment that produced these truth values for the premises and the conclusion. If we could, the argument was invalid; if not, it was valid.

When we test arguments using the truth tree method we do the same thing. We look for a truth value assignment that makes all the premises true and the conclusion false. In a truth tree, however, we don't tag sentences with **T**s or **F**s. Instead, a sentence is assigned TRUE if it appears on a branch of the tree and FALSE if its negation appears on a branch. So to see if there is some truth value assignment that makes all the premises true and the conclusion false, we begin by listing the premises and the *negation* of the conclusion. These are the *initial sentences* from which we grow the tree, which are part of every branch. If the tree closes, the argument is valid; if it remains open, the argument is invalid. What we are doing here is testing the premises + *negation* of conclusion for consistency so we grow the tree in the usual way. If the premises +

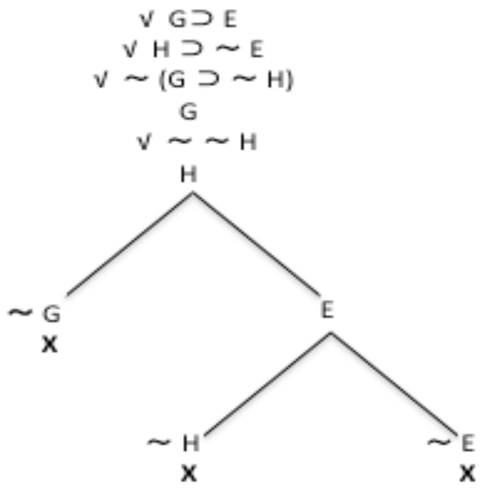
negation of conclusion are consistent (open tree) the argument is invalid; if inconsistent (closed tree) the argument is valid.

Like the method of indirect truth tables, this is an indirect proof method, a.k.a. 'reductio ad absurdum' or 'proof by contradiction'. When we do a reductio proof we begin by assuming that the premises are true and trying out the additional assumption that the conclusion is false. If this gets us into trouble it shows that we can't have the premises true and the conclusion false, which is to say the argument is valid. When doing an indirect truth table, trouble is when truth values that are forced produce a goof, e.g. a conditional assigned TRUE, with TRUE antecedent and FALSE consequent, a FALSE disjunction with TRUE disjuncts, etc. When we do a reductio proof by natural deduction—as we shall do—trouble is deriving a *contradiction*, a sentence of the form '*p* • ~ *p*. And when we use truth trees to test arguments, trouble is a closed tree—which shows that there is no truth value assignment that makes the premises + negation of the conclusion true together, hence that the argument is valid.

Example: Test the following argument for validity using a truth tree:

G ⊃ E
H ⊃ ~ E   / G ⊃ ~ H

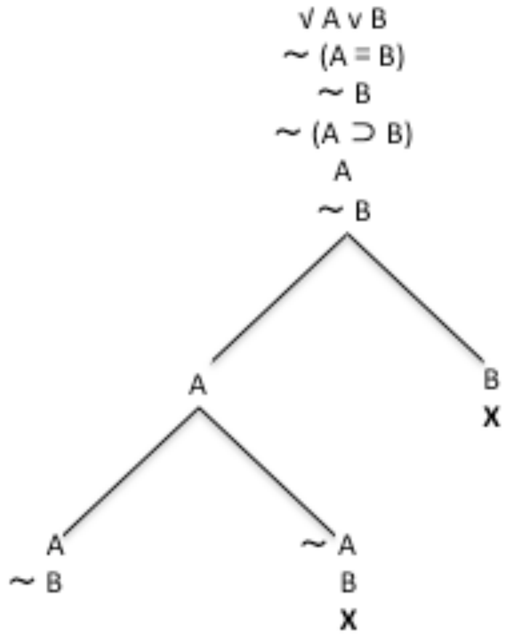
We put the conclusion after a slash to the right of the last premise. To test for validity, we list the premises and the *negation* of the conclusion, and grow the tree in the usual way:



Valid or invalid? Valid

'~ (G ⊃ ~ H)', the negation of the conclusion is the last of the initial sentences. The tree grown from these initial sentences closes, so the argument is valid.

Example: Test the following argument for validity using a truth tree.



Valid or invalid? *Invalid*

The tree is open so the argument is invalid—that is, there is some truth value assignment that makes all the premises true and the conclusion false. The open branch—the leftmost one on the tree—represents that truth value assignment. Since 'A' in on that branch 'A' is true in that truth value assignment; since '~ B' is on that branch, 'B' is false in that truth value assignment. And that is exactly what we saw when we tested that argument using an indirect truth table!

A	v	B	/	~	(A	≡	B)	/	~	B	//	A	⊃	B
T	T	F		T	T	F	F		T	F		T	F	F

Valid or invalid? *Invalid*

In general, if a truth tree is open there is some (at least one) truth value assignment that makes all initial sentences true. To determine what it is, look at the open branch(es). If a sentence letter is on a branch, that sentence letter is assigned true; if the negation of a sentence letter appears on a branch, that sentence letter is assigned false. What about the closed branches? Irrelevant. They don't represent legitimate truth value assignments since a branch closes precisely because includes a sentence and its negation.