Lab 06: Logical Normal Forms, Part 1

1 Overview

The goal of this two-part lab is to write an algorithm that converts any proposition to conjunctive normal form (CNF). It’s valuable to have propositions in CNF because we can design algorithms that are specialized for this format, such as the previous two labs, Satisfiability Solver.

2 Propositions as Trees

In this lab, we will be abstracting propositions using Binary Search Trees – a recursively defined structure. Previously, you’ve most likely seen BSTs used to store some comparable type of data (such as ints); but our proposition trees will work a bit differently. Specifically, the nodes of our tree will follow the following design:

- **internal nodes** (meaning nodes with at least one child) represent logical operations, such as ∨, ∧, ¬, and ⇒
- **leaf nodes** (meaning nodes with no children) represent atomic propositions – variables such as p, q, r, etc.

Below is an example of a tree representation of a proposition:

\[(r \implies (p \lor \neg q)) \lor \neg (q \land s)\]
Let’s break down how the proposition above is abstracted into the tree:

- each non-leaf node represents some logical operator (\(\land\), \(\lor\), \(\Rightarrow\), and \(\neg\))
- each operator (aside from \(\neg\)) has two children, representing the propositions on either side of the operator
- each leaf node represents some variable
- think about how the proposition is split amongst the branches of the tree – the root note effectively splits the original proposition into its two sub-propositions (as two sub-trees), conjoined with the \(\lor\):
  - we have \(r \Rightarrow (p \lor \neg q)\) as the left child sub-tree
  - and we have \(\neg(q \land s)\) as the right child sub-tree

3 Provided Code

Provided to you are six starter files (five classes plus the tester file). A high level description of the (non-tester) classes is provided below:

- **Proposition.java**: an abstract class representing all propositions
- **NegateProp.java**: a proposition made up the negation of some proposition. You can visualize this as \(\neg\alpha\), where \(\alpha\) could be a single variable, ex: \(p\), or a larger proposition, ex: \((p \lor q \lor r)\).
- **AtomicProp.java**: an atomic proposition (i.e. a proposition consisting of a single variable with no operator)
- **BinaryProp.java**: a proposition made up of two proposition connected by some binary logical connective (ex: \(a \land b\))
- **Build.java**: a collection of useful helper methods for writing complex propositions; also helpful in understanding how the pieces come together.

These classes are used together to create tree abstractions of our prepositions. Relating to the example tree above, you can think of:

- Blue colored nodes as BinaryProp objects
- Red colored nodes as NegateProp objects
- Green colored nodes as AtomicProp objects
- thanks to the wonders of polymorphism, all nodes are a Proposition

4 Your Task

As described previously, the end-goal of this project is to write an algorithm that can convert any proposition to CNF. You will ultimately accomplish this in three steps:

1. Given a proposition \(\varphi\), simplify \(\varphi\) so that it only contains \(\land\) and \(\neg\) operators. We’ll call this "simplified" \(\varphi\).
2. Convert "simplified" \(\varphi\) into negation normal form (NNF). We’ll call this "NNF" \(\varphi\). (Note: this may introduce some \(\lor\)’s because of DeMorgan’s law)
3. Convert "NNF" \(\varphi\) into CNF. The key idea here is to distribute \(\lor\) over \(\land\).

Each step takes proposition and recursively manipulates it, producing a new proposition that is logically equivalent to the input but in a desired form. In this lab, you will only implement steps 1 and 2 – step 3 will come next week.

Below is a guide to what you need to implement in this lab.
4.1 Finish Helper Functions

You should first spend time perusing the provided code base. Your first task will be to complete all of the methods throwing “implement me” exceptions – you will find these in the Build, NegateProp, and Proposition classes.

Ignore the toSimplified and simplifiedToNNF functions in Build for now.

4.2 To Simplified

Next, you will implement the toSimplified() method in the Build class. This method generates a Proposition that is logically equivalent to the argument, but only contains the logical connectives ∧ and ¬. This method must be implemented recursively.

4.3 Simplified To Negation Normal Form (NNF)

Lastly, you will implement the simplifiedToNNF() method in the Build class. This method accepts a Proposition that is in simplified form (as described above) and converts it to a logically equivalent Proposition in NNF.

A Proposition in NNF only has negation connective (¬) applied to atomic propositions (i.e. AtomicProp), and only contains the logical connectives ∧, ∨, and ¬. This method must be implemented recursively.

5 Pre-Lab Questions

After reading this document and provided code, answer the following before we meet (we will discuss in-lab):

1. Proposition is what type of class? i.e. a(n) ____________ class.
2. Proposition has three children – what are they?
3. What does getRight() return for a BinaryProp object? What about for a NegateProp object?
4. How would you convert the following Propositions to "simplified" forms? hint: reference the table on the last page of this document:
   • q ∨ s
   • ¬(q ∨ s)
   • q ⇒ (p ∨ ¬q)
5. The toNNF() function assumes that the argument Proposition is in "simplified" form – what does it do if it’s not (read the comments!)
6. How would you convert the following Propositions to NNF forms?:
   • ¬(q ∧ s)
   • ¬(q ∧ ¬s)

6 Submission

See the top of this document for the lab’s due date and time. When submitting your code, include only the files listed below:

• Build.java
• NegateProp.java
• Lab06Tester.java
• Proposition.java
7  Tips

Below are some last tips to help you get started:

- Use paper and draw trees! Debug your algorithm by tracing through example Trees that you construct in NormalForms’ main and draw on paper.
- The toSimplified and simplifiedToNNF methods accept a Proposition object, but really this Proposition is the root node of some tree representing a proposition. When thinking about your recursive case(s), again apply the “black-box” method of design.
  In your recursive call(s) you are passing the root node of some (sub-)tree – assuming your call works exactly as intended, what does the returned tree look like?
- Don’t forget about the provided helper functions in Build. Spend some time looking at the main method of Lab06Tester – review and experiment with the different propositions created here.

Finally, the below table is a collection of several propositional equivalences – you will make extensive use of these in this lab, so keep this table handy!

<table>
<thead>
<tr>
<th>DeMorgan’s Laws</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (\neg(p \land q) \equiv \neg p \lor \neg q)</td>
</tr>
<tr>
<td>• (\neg(p \lor q) \equiv \neg p \land \neg q)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition of Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (p \Rightarrow q \equiv \neg p \lor q)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Double Negation Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (\neg(\neg p) \equiv p)</td>
</tr>
</tbody>
</table>