1 Overview

In this lab, you will continue to sharpen your recursive design skills while gaining more practice with propositional logic – namely conjunctive normal form and satisfiability.

For this assignment, you will implement an algorithm which accepts a proposition written in Conjunctive Normal Form (CNF) and returns true if there exists a satisfying assignment for the proposition or false if the proposition is unsatisfiable (more on what this means below).

This is a very important problem – some might say the most important problem in all of CS! Many real-world problems can cast as CNF satisfiability problems; for example, it is used to check Paris Metro railway system for bugs.

2 Satisfiability Algorithm

Below is an explanation of satisfiability as well as high-level description of the algorithm you must implement.

2.1 Truth Assignments and Satisfiability

Imagine we have a proposition; for example:

\[(p \lor \neg q \lor \neg r)\]

Next, we will apply a truth assignment to this proposition. A truth assignment is a collection of variables and their respective values – (imagine a HashMap where the keys are variables and their values are either true or false).

For example:

\[p \rightarrow \text{False}, \ q \rightarrow \text{True}, \ r \rightarrow \text{False}\]

If we were to apply the above truth assignment to the above proposition, the proposition would evaluate to true, as we’d have (False \lor \neg \text{True} \lor \neg \text{False}) which is ultimately (False \lor \text{False} \lor \text{True}) and thus (True).

Satisfiability is a means of expressing whether it is possible for a given proposition to be evaluated as True. In terms of satisfiability, a given proposition is:

- considered satisfiable if there exists at least one truth assignment under which the proposition evaluates to True
- considered unsatisfiable if there are no possible truth assignments for which the proposition evaluates to True
2.2 Our Algorithm

You will be writing the algorithm to determine satisfiability of a given proposition. Given a proposition, you will search every possible truth assignment to determine if there are any permutation of variable assignments for which the proposition evaluates True.

To better understand this algorithm, let’s define isSatHelper as follows: This function takes a proposition \( \varphi \) and a truth assignment \( T \) and returns True when the proposition can be satisfied using the variable assignments of \( T \).

Note that \( T \) may be an incomplete model, meaning not every variable in \( \varphi \) has an assignment in the model. This function is going to return whether or not \( \varphi \) is satisfiable given the provided variable assignments, and any combination of values for the unassigned variables.

Here are some example inputs and outputs:

- **Input:** \( \varphi := (p \lor q) \land (\neg q) \)
  
  \( M := p \rightarrow True, q \rightarrow False \)

  **Output:** True because \( \varphi \) evaluates to True given the assignments under \( T \)

- **Input:** \( \varphi := (p \lor \neg q \lor \neg r) \)
  
  \( M := p \rightarrow False, q \rightarrow True \)

  **Output:** True because there exists a truth assignment given \( T \)'s partial assignment where \( \varphi \) evaluates to True (if \( r \rightarrow False \))

- **Input:** \( \varphi := (p \lor \neg q \lor \neg r) \)
  
  \( M := p \rightarrow False \)

  **Output:** True for same reasons as above (\( \varphi \) is True if \( q \rightarrow False \) and/or \( r \rightarrow False \))

- **Input:** \( \varphi := (p \lor q) \land (\neg q) \land (\neg r) \)
  
  \( M := p \rightarrow False \)

  **Output:** False. Although \( \varphi \) is satisfiable in general, it is not under this truth assignment, regardless of how the remaining variables are assigned

We can implement isSatHelper using recursion!

The basic idea is this: suppose \( p \) is a variable that appears in the proposition \( \varphi \) but has not yet been assigned a value in the truth assignment \( T \). Let's try setting \( p \) to True and seeing if the proposition is satisfiable.

By assigning \( p \) to True, we have made the problem "smaller": there is one fewer variable to consider. After we assign \( p \) to True, we can make a recursive call. If assigning \( p \) to True does not yield a satisfying assignment, then we set \( p \) to False and see if it’s satisfiable (with another recursive call). If we try setting \( p \) to both True and False, and neither yields a satisfying assignment, then the proposition must be unsatisfiable.

The base case occurs when the model is "complete", meaning every variable in \( \varphi \) has been assigned a value in truth assignment \( T \). In this case, check whether \( \varphi \) evaluates to True under this assignment. If it does, you’ve found a satisfying assignment! If not, this particular truth assignment does not lead to a satisfying result, so return False.

2.3 Pseudocode Implementation

On the next page is a pseudocode implementation of our above recursive algorithm:
def isSatHelper(proposition, model):
    # *** base case ***
    # check the truth assignment: have all variables been assigned?
    if truth assignment has no unassigned variables:
        if proposition evaluates to true given the assignment:
            # you have found a satisfying assignment!
            return True
        else:
            # this particular assignment does not satisfy the proposition
            return False
    # *** recursive case ***
    var = pick one of truth assignment’s unassigned variables

    # try setting var to True, see if this leads a satisfying assignment
    update truth assignment, assigning var to True
    if isSatHelper(proposition, model):
        return True

    # try setting var to False
    update truth assignment, assigning var to False
    if isSatHelper(proposition, model):
        return True

    # looks like we don’t have a satisfying assignment
    update truth assignment, unassign var
    return False  # proposition is unsatisfiable

2.4 Provided Code

You are provided with six starter files (five classes plus the tester file). Your first step should be to familiarize yourself with each of these classes, as well as their attribute and responsibilities.

A high-level description of the (non-tester) classes is provided below:

- **Variable.java**: a single variable of an atomic proposition, such as p
- **Literal.java**: a literal in a boolean proposition. For example, in the proposition \((p \lor \neg q) \land (\neg p \lor r)\), there are four literals: \(p\), \(\neg p\), \(\neg q\), and \(r\).
- **Clause.java**: a single clause in a CNF proposition, such as: \((p \lor \neg q)\)
- **CNFProposition.java**: a proposition in conjunctive normal form (i.e. a conjunction of clauses). Also contains a method to determine satisfiability.
- **TruthAssignment.java**: an assignment of truth values to variables, as described in the Section 2.2.

3 Your Task

Your task is to finish the implementation of the satisfiability checker in the provided code. To do this, you will ultimately implement two functions. Below is the recommended order to complete this task:

1. Review the provided code. Become comfortable with what each class represents and their public methods (you don’t necessarily need to understand how each works).
2. Search the provided code for functions throwing an "implement me" exception and implement them. There are four of these functions in total, though you should ignore isSatisfiableHelper for now.
3. Finally, implement the isSatisfiableHelper method in CNFPropositionHelper based on the provided pseudocode above. This function is the recursive helper to isSatisfiable().
4 Pre-Lab Questions

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

1. For each of the prompts below, fill in the blank with one of the following: [ZERO / AT LEAST ONE / ALL]
   - A CNF Proposition evaluates to True if ____________ clause(s) evaluate to True
   - A CNF Proposition evaluates to False if ____________ clause(s) evaluate to False
   - A disjunct of literals evaluates to True if ____________ literal(s) evaluate to True
   - A disjunct of literals evaluates to False if ____________ literal(s) evaluate to True

2. Suppose we have the variables myClause and myProp which reference Clause and CNFProposition objects representing the following:
   
   $$\text{myClause} = (p \lor \neg q \lor \neg r)$$
   $$\text{myProp} = (p \lor q) \land (\neg q) \land (r \lor \neg p)$$
   
   What would be output by the following? (assume all "implement me!" functions are properly implemented):
   - myProp.getAllVariables.size()?
   - myClause.getNegatedVariables().size()?
   - myClause.getAllVariables().size()?
   - myClause.getAllLiterals().size()?
   - myProp.getClauses().size()?

3. What the result of the following be? (again, assume all functions are properly implemented):

   ```java
   Set<Variable> vars = new HashSet<Variable>();
   Variable a = new Variable("a");
   Variable b = new Variable("b");
   Variable c = new Variable("c");
   vars.add(a);
   vars.add(b);
   vars.add(c);
   TruthAssignment ta = new TruthAssignment(vars);
   m.assign(a, true);
   m.assign(b, true);
   m.assign(a, false);
   System.out.println(m);
   ```

5 Submission

See the top of this document for your lab section’s due date/time. When uploading your submission to Moodle submit only the files listed below:

- Clause.java
- CNFProposition.java
- Lab04Tester.java
- TruthAssignment.java