# Prop Variable Conversion

The conversion of propositional formulae into internal data structures is done with the help of an external library I found on github. The packages are included in the project file under **aima.core.logic**. I only included packages that were needed for my uses in this project, the entire library itself is extensive.

The library is extremely useful, and I used it to convert propositional input into Conjunctive Normal Form (CNF), which is the format used in the ALLSAT solver. The ALLSAT solver then converts CNF input into a set of States. There are several objects that are used throughout this conversion process.

1. Propositional formulae are introduced to the system as a **String**.
2. The **String** is converted into **Clauses** and **Literals** by the parser, which I convert into **FormulaSet** and **Formula** objects, which are my system’s internal representation of **Clause** and **Literal**.
3. The ALLSAT algorithm is run on the **FormulaSet** object, which returns a set of satisfying assignments represented by a **BeliefState** object.

This is the point where revision takes place. **BeliefState** objects are the primary object used in revision.

There are two methods that handle the conversion process, they are in the **InputTranslation** class. The two method headers are:

Since CNF form is just used internally by the ALLSAT solver, this function is private

private static FormulaSet propToCNFForm(String formula, Set<Character> vocab)

public static BeliefState convertPropInput(String text, Set<Character> chars)

The convertPropInput method is used externally throughout the project to convert formulae. This is the method encapsulates the entire process that was described above. This method is used whenever propositional format is inputted by the user.

# All Satisfying Assignments (ALLSAT)

An ALLSAT solver is used to find all satisfying assignments to a propositional formula. This process is used throughout this project to convert user input into states which can be used in belief revision. The solver uses the DPLL algorithm find a satisfying assignment and then adds a blocking clause to continue to find all other possible assignments.

This solver requires input to be in conjunctive normal form (CNF). CNF formulae are made up of literals and clauses. Literals are propositional variables converted into integers. Clauses are disjunctions of literals (Successive or’s between literals). This format is called conjunctive normal form because a set of clauses is conjunctive, meaning an and between each clause. The following example converts ‘human readable CNF’ into proper CNF form.

**Human readable:**

[p or q or r] and

[!p or q] and

[p or !r or s]

**CNF format**

[1 2 3]

[-1 2]

[1 -3 4]

This type of format is essential to the DPLL algorithm. CNF allows a large set of clauses to be reduced quickly through elimination. In simple terms the algorithm looks like this:

**Algorithm SAT\_DPLL (FormulaSet set)**

If any clause empty

Return false

If set consistent

Return true

set = unit propagation(set)

set = pure literal assign(set)

Int chosen literal = choose literal(set)

Return SAT\_DPLL((reduceset(set, +chosen\_literal))

or SAT\_DPLL(reduceset(set. – chosen\_literal))

**Done**

**Choosing a Literal (Reduce Set):**

When a literal is chosen by the algorithm clauses containing this literal are removed form the set, and literals containing the opposite polarity are removed from clauses. Since a chosen literal is assumed to be true, clauses containing this literal no longer need to be considered. Opposite polarity literals are no longer considered because they have been set to false.

Eg. Formula Set Before choosing 2:

[1,3,-2]

[1,2,5]

[3,4,5]

Formula Set After choosing 2:

[1,3]

[3,4,5]

**Consistency**:

A formula set is consistent if **ALL** propositional variables are of only one polarity in the set.

Eg. Consistent

[1,2,3]

[2,3,5]

Eg. Inconsistent

[1, 2, 3]

[-1, 3, 4]

**Unit Propagation**:

Unit propagation is a key step in this algorithm because it can significantly reduce the search space before a recursive call is made. Any clause that only contains one literal is a unit and must be true for the entire formula set to be true. These unit clauses along with any other clause containing that literal are removed. Additionally, opposite polarity literals are removed from clauses. Unit propagation continues until no unit clauses exist in the set.

Eg. Before Unit Propagation:

[1,3,4]

[3]

[2,-3, 4]

[1,5,-2]

After Unit Propagation:

[2,4]

[1,5,-2]

**Pure Literal Assignment:**

A literal is ‘pure’ if it has only one polarity in the formula set. Pure literal assignment removes all clauses containing this literal from the set.

Before removing 2:

[1,2,3,-4]

[-1,2,4]

[1,5]

After:

[1,5]

The distinction between a SAT solver and an ALLSAT solver is the returning of all satisfiable states. The main addition made to a SAT solver to return all states is some sort of function to ensure unique solutions are found every iteration of the algorithm. In the case of this SAT solver, running it multiple times will always yield the same result. Since this is the case, a blocking clause must be added to prevent past solutions to be found.

A blocking clause is a clause added to the original formula that prevents a solution from being found.

Say this solution was just found: [1,-2,3,-4,5]

A blocking clause is the polar opposite of the solution: [-1,2-3,4,-5]

With the addition of this blocking clause, the next iteration of the DPLL algorithm can no longer return the first solution as satisfiable and must return another solution if it exists.

**Algorithm ALLSAT\_DPLL(FormulaSet set)**

While (SAT\_DPLL finds solution in set)

add solution

add blocking clause to formulaset

Done while

Return solutions

**Done**

# Trust Graph

The trust graph is represented by a **DistanceMap** object and managed by a **DistanceState** object. The **DistanceMap** itself is A HashMap of HashMaps. This allows me to store a distance value together with a pair of states.

HashMap map = HashMap<State, HashMap<State, Double>>();

The **DistanceMap** object is essentially a data object, the only methods defined by this object are getters and setters. The **DistanceState** object manages all the complicated logic that requires **DistanceMap** updates.

**Set Map Member:**

The **setMapMember** method is used to set a value inside the **DistanceMap** object safely, without violating any of the constraints. It checks triangle inequality, and if the proposed value would violate the system, handle the constraint. Once the value is determined to be safe, the value will be set. The way that inequality is handled is described in the Triangle Inequality Responses section.

**Algorithm setMapMember(State s1, State s2, double currentval, double newval, TriangleInequalityResponse tr)**

Invalid state = check triangle inequality

If (there are invalid states)

Get new\_val from calcDistanceValueTriIneq

Set distance for s1 and s2 to new val

**Add Report:**

The **addReport** method adds a **Report** object to the current **DistanceMap** object, which produces a new **DistanceMap**. The new map will depend on the values of the report. A true report will have values of greater size than before, a false report will update values to have less weight than before.

The driver function behind report addition is the **modByReport** method, which iterates through all combinations of Satisfied states and unsatisfied states, based on the report formula, and modifies the distance value for each of those state combinations.

**Algorithm modByReport(BeliefState satstate, BeliefState unsatstate, TrustOperation to, TriangleInequalitlyResponse tr)**

Foreach(satstate)

Foreach(unsatstate)

Get current\_value

Generate new\_value

setMapMember(satstate, unsatstate, new value, to, tr);

**Check Triangle Inequality:**

Checking Triangle Inequality is done whenever a new value is being added to the trust graph. The method must handle each case differently when a value is being increased or decreased. This method checks the distance value of each endpoint state to all intermediate states. If the new value would cause triangle inequality for an intermediate state, that state is added to a **BeliefState** which is returned at the end of the function.

Increasing a value requires that we check the other two edges and ensure that our new value is less than or equal to the sum of the other two edges.

6

2

3

In this example. Introducing the value 6 to the triangle would make this triangle invalid. The value of this edge must be <= 2 + 3. 5 is the max allowed value for this triangle.

The situation changes slightly when the value proposed is smaller than the original value in the graph. Now we must look at the adjacent edges for invalidity rather than the direct edge itself.

1

2

4

Say we are again changing the hypotenuse edge in this example. We want to reduce the edge value to a 1. This invalidates the edge with length 4, because now 4 is less than 1+2, which makes the triangle invalid. The minimum allowable value for the edge we are changing is a 2, because 2+2<=4;

**Algorithm checkTriangleInequality(State s, State u, double proposed\_val)**

Foreach(intermediate state t)

if (proposed val > current val)

if proposed val > sum of adjacent edges (s,t) + (t,u)

add to invalid set

else //proposed val is decreasing in value

if edge a is > proposed val + b

add to invalid set

if edge b is > proposed val + a

add to invalid set

return invalid set

**Calculate Distance Triangle Inequality:**

This method is executed when a value would trigger triangle inequality. The purpose of this method is to respond to this constraint and produce a value that satisfies triangle inequality in the system.

# Triangle Inequality Responses