# Implementation Notes

## Class Structure

**Structure will likely change in some form**

* **RankingState:**
* **BeliefRevision:**

**Trust Revision**

* **State**: Represented as a string of 0’s and 1’s, with length equal to the propositional vocabulary
* **BeliefState**: A collection of State
* **DistanceState**: Manages the distances between all possible state combinations.
* **Report**: A formula and reported value which indicates the accuracy of the formula

**ALLSAT Solver**

* **DPLL:** Class implementation of a SAT and ALLSAT solver using the DPLL algorithm and blocking clauses.
* **Formula:** System representation of a Conjunctive Normal Form(CNF) clause.
* **FormulaSet:** System representation of a set of CNF clauses.

## Diagram

## Class/Concept Implementation

### Trust Revision

#### Report

A Report object is used by a DistanceState object to modify distances between States. The implementation of the Report class is simple. This object is intended to store a String formula and an Integer ‘boolean’ value. The only function it has, other than getters and setters, is to convert its formula member variable into a BeliefState object.

**Member Variables:**

1. String formula
2. Int report value (0 or 1)

Ex. Report object

**String** = “a&b”

**int** = 0

**Functions:**

**convertFormToStates(Set characters vocab)**

return a BeliefState representation of the formula member variable.

#### DistanceState

A DistanceState object stores the distance between two states. When an object is created, all possible states for a given vocabulary are generated and stored as a BeliefState. Next, all possible combinations of these states are stored in the Hashmap data member with a default distance value.

**Member Variables:**

1. Set of characters as vocab
2. BeliefState containing all possible states, given the vocabulary
3. Hashmap of state combinations storing the distance between them.

Ex. DistanceState object:

**Set** = {A,B}

**BeliefState** = {00,01,10,11}

**HashMap** = <state1, HashMap<state2, distance>>

[<00 , <01, 2>, <10, 1>, <11, 1>>

<01 -> <10, 3>, <11, 1>>

<10 -> <11, 4>>]

**Functions:**

**getDistance(State s1, State s2)**

returns the distance based on both states

**setDistance(State s1, State s2, double distance)**

sets the distance based on both states

**checkTriangleInequality(BeliefState possiblestates, State s, State u, double val, List errors)**

for each intermediate state t

if val violates triangle inequality between (s,t) and (t,u)

add to errors

return true

return false

**modByReport(BeliefState s1, BeliefState s2, double val)**

for each combination of states in s1 and s2

add val to existing val

**modByReport(BeliefState s1, double val)**

for each combination of states in s1

add val to existing val

**addReport(Report r)**

convert report to states

separate states into two categories

* + States that satisfy formula
  + States that do not satisfy formula

If r val == 0

modByReport(satset, unsatset, 1)

if r val == 1

modByReport(satset, -1)

modByReport(unsatset, -1)

return errors if any

### ALLSAT Solver

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 (CNF FormulaSet)**

If any clause empty

Return false

If formula set consistent

Return true

Cnf formula = unit propagation(formula)

Cnf formula = pure literal assign(formula)

Int chosen literal = choose literal(formula)

Return DPLL((reduceset(formulaset, +chosen\_literal))

or DPLL(reduceset(formulaset. – 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 no longer need to be considered as well, because now they are 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 a propositional variable only has 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**

Hopefully, the concept of SAT and ALLSAT using DPLL and blocking clauses is clear at this point. Next I’ll go into the implementation and representation of these concepts as objects and classes in the system.

#### Formula

A **formula** is a class that represents a **clause** in the system. A formula is simply a list of **literals**, which are represented as integers. Apart from getters and setters one method is important in the Formula class, the isSatisfiedBy method. This is a simple method that checks the list to see if it contains a certain literal, returning a Boolean with the result.

**Member Variables:**

1. List of Integers

Eg. [1,-2,4]

**Functions:**

**isSatisfiedBy(int literal)**

returns true if literal found in list

returns false if not

//should be replaced with copy constructor

//and remove method

**addAllBut(int val)**

adds every value in this objects list to another Formula EXCEPT val

returns a new formula

#### FormulaSet

The FormulaSet class contains a list of Formula objects. This is the type of object that is used by the DPLL algorithm to find satisfying assignments for a propositional formula. There are several critical methods defined in the FormulaSet class.

A simple method but critical, when eliminating clauses I choose to add formulae to a new formula set rather than remove, mainly because multiple clauses can be removed during the elimination process and removing while iterating can be messy.

**Member Variables:**

1. List of Formula objects
2. Number of propositional variables

Eg.

List = [1,-2,3], [-1,4,5], [2,4]

Vars = 5

**Functions:**

**Void addFormula(Formula f)**

adds f to the list

This method takes a specific input array double the size of the variable count for the propositional formula. So, if a formula has 4 variables, this array is length 8. One index for a variable and a specific polarity. This is a unique count array that is returned by **countLiteralsUnique()**.

**Boolean isConsistent(int[] var\_counts)**

returns false if count array contains positive count for both polarity of variable (eg, -1, 1)

returns true else

Important method for the DPLL algorithm. An empty clause indicates failure for this iteration of variable choices. Backtracking must be done.

**Boolean hasEmptyClause()**

returns true if Formula object is empty

false if not

This method counts all the literals in the FormulaSet. There is a distinction between positive and negative integers, they are counted separately. This method returns an integer array with the count values.

**int countLiteralsUnique()**

This method counts all the literals in the FormulaSet, but no distinction is made between positive and negative literals. An integer array is returned containing the counts.

**int countLiterals()**

#### DPLL

This function is a main driver function in the **runDPLL** function. Eliminating formula’s is used during unit propagation and when selecting literals to before a recursive runDPLL call is made.

**Member Variables:**

No variables, static class

**Functions:**

**FormulaSet eliminateFormulas(int literal, FormulaSet set)**

FormulaSet newset;

Foreach formula in set

Check if formula is NOT satisfied by literal

If clause is satisfied by opposite polarity literal

Remove that literal

endif

Add clause to new set

endif

Return newset

The method responsible for unit propagation. Unit clauses must be found before this function is run. The method returns a new set after eliminating all found unit clauses.

**FormulaSet runUnitProp(Set of unit values, FormulaSet set)**

For each unit value

Newset = eliminateFormulas(unit\_value, set)

Return newset

This method removes clauses containing pure literals from the formula set.

**pureListAssign(int[] unique count, FormulaSet set)**

purelits = findpurelitals(unique count)

for each formula in set

if formula does not contains a pure literal

add to set

return newset

This function is used to find all satisfying assignments. After an assignment is found by the DPLL algorithm we need to ensure that that solution is not found again when the algorithm is run. A simple way to accomplish this is by adding a blocking clause to the original formula set.

**Formula generateBlockingClause(int[] solution)**

For each value in solution

Add negation of value to formula

Return formula

This method takes the raw output from the DPLL algorithm and converts it into a **State** object. State and BeliefState objects are used all throughout the project in areas such as Belief Revision.

The input of this function looks something like this [1,2,-3,4,5]. It would be turned into something like this. [1,1,0,1,1]. If some literal had no impact on the solution (did not matter if it was 0 or 1), it will be represented by a 0 in the DPLL solution. So the DPLL output will look like this [1,2,0,-4,5]. This means we need to generate two resulting **State** objects from this result. [1,2,3,-4,5] and [1,2,-3,-4,5].

The implementation of this function is not completely straightforward, especially when you consider there can be multiple zeroes in the output array from the DPLL algorithm. This essentially turns into a combination problem where several states are generated equal to 2^number of zeroes found in the output array.

Making assignments to solution arrays with multiple zeroes requires combinations of values 2^zeroes. What I do is generate an array with assignments and then create states based on that assignment until all combinations are exhausted. It looks something like this.

Solution[] = [1,0,0,4,-5]

Assignments array = [00,01,10,11]

The first state we create corresponds to the first item in the assignments array, meaning that both 0s in the solution array are assigned the value 0 in the output state. The second state would assign 0 to the first 0 found in solution and assign 1 to the second zero found in solution. This continues for the rest of the items in the assignments array.

**State solutionToBelief(solution array, array zero assignments)**

For all integers in solution

If positive

Add 1

Else

Add 0

If 0

Handle zero assignment

Return state

Driver function for the DPLL algorithm. The runDPLL method functions similarly to the algorithm mentioned at the beginning of the section, with minor adjustments. There are some repeating aspects, and it is not completely clean/perfect in it’s implementation.

**Boolean runDPLL(FormulaSet set)**

This is the driver method for single SAT DPLL. This method returns a Boolean result indicating whether the formula has a satisfying assignment or not.

**satDPLL(dimacs input, number variables)**

create initial formulaset(input)

return runDPLL(formulaset)

The distinction between a SAT and an ALLSAT is that the ALLSAT must find all satisfying states for a given formula. The method at its core is simple. Continue to look for solutions if the DPLL algorithm returns answers. The blocking clause added to the original FormulaSet ensures that unique solutions are found each DPLL iteration. Return a BeliefState representing all satisfying assignments to the Formula.

**BeliefState allSatDpllBlock(FormulaSet set)**

While(DPLL solution found)

Convert solution to beliefstate

Add blocking clause to formula set

End while

Return BeliefState