# CS202A Assignment-2 (SAT Solver)

## Contributors

Arnav Gupta (200186)
Hitesh Anand (200449)

## Implementation

The first step is to preprocess the given clauses so that they become easier to work with. For this purpose, we have done the preprocessing so that the final clauses are represented as a list of lists of integers (the coding has been done in Python) where each 'element' list gives a clause and each individual unit in a list gives the literal.

Following this, we are ready to set up the SAT solver. The algorithm used in our case is the **DPLL Algorithm**. This algorithm uses recursive backtracking and uses **Unit Clauses** to speed up the computation. A rough sketch of the algorithm used is given below:

1. Select a variable and assign either True or False to it.
2. Now iterate over all clauses. If the variable (which has just been assigned) is present in the clause, there are two cases:
    2.1 Either it has been assigned the same value as that present in the clause. In that case, the clause has been satisfied, so we would just **remove the clause** and continue.
    2.2 Else, it has been assigned the value opposite to that in the clause. Then, the clause cannot be satisfied by the literal considered, hence we would **remove the literal from the clause** and continue.
3. Now, if there is a unit clause present (a clause wherein only 1 variable has been left unassigned), we would assign the relevant value to it so that the clause beomes true. **This step is called unit propagation and it is the optimisation present in DPLL over brute force.**After this step,recursively move again to step 2.
4. Otherwise, assign another variable some value and perform step 2.
5. If we arrive at a contradiction, then backtrack and reverse the assignment of the most recent variable and proceed again. **This is the recursive step.**

The algorithm halts when either all the clauses have been satisfied, or there is no further backtracking possible (i.e. no solution)

This is the main idea behind our implementation. We also check for pure literals at each iteration (literals only having one sign in the formula) which speeds up the process.

### The code

The functions below have been used in our code :

1. **unitPropagation() -**performs unit propagation on the clauses.
2. **polarityDict() -** Finds out the polarity of each literal in the given formula. If both its negation and assertion is present, we assign the polarity "both".
3. **pureLiterals() -** Finds out the pure literals.
4. **pureElimination() -** Valuates the pure literals and eliminates the clauses containing them.
5. **Consistent() -** Checks the consistency of the present assignment.
6. **removeEmptyClauses() -** Removes all those clauses which have no literals in them.
7. **solve() -** Binds all the above functions together.

**Note**: Pure literals are those literals which either have only their negation or the variables themselves in the given formula. In other words, they with only one polarity throughout the formula.

These functions are explained in greater detail below:

1. **unitPropagation()**

This function takes the given CNF formula as input(in the form of a list of lists). It looks for a unit clause in the remaining formula. On finding a unit clause, it iterates through the other clauses and checks whether the literal in the unit clause is present in the other clauses. If it is present and has the same polarity as in the unit clause, the clause is satisfied and hence removed from the list of clauses. We can also remove all negations of the assignment from other clauses since those clauses can't be satisfied by that literal.

The key idea behind this step is that the only way for a unit clause to be satisfied is for that literal to be assigned corresponding to the polarity in that unit clause.

Pseudocode :

```
for each unit clause {+/- x} in formula:   // x is a number in [1, num_vars]
    remove all non-unit clauses containing +/-x
    remove all instances of -/+ x in every clause
```

2. **polarityDict()**

This function checks the polarity of the literals(i.e. The state of the literal (neg, pos, both, none) in the formula). It returns a dictionary. Initially, the polarity of all literals is set to 'none'. Then we iterate through the literals in the clauses. If at the end it is found that a literal has both its assertion and negation present in the formula, we assign the value 'both' to it. Else, if the formula only has the negation of the current literal, we assign 'neg', and similarly, 'pos'. This function is helpful to find the pure literals in the formula.

3. **pureLiterals()**

It builds upon the polarityDict() function. We iterate through the literals in the dictionary returned from polarityDict(). If its dictionary value is 'pos' or 'neg' we can conclude that it is a pure literal, and add the (literal, polarity) pair to a list. This list is then returned from the function.

4. **pureElimination()**

Each clause where a pure literal appears can be satisfied by assigning the pure literal the value corresponding to its polarity (i.e. True if it is 'pos' and False for 'neg'). In this way, we can append the literal as a unit clause and remove all those clauses from the formula where it appears. The idea behind this is that the satisfiability of the formula is not disturbed since we can't satisfy any clause by assigning the pure literal to be the opposite of its polarity. It is an example of a 'greedy' approach.
In this step, the pure literals are eliminated.

Pseudocode:

```
for each  variable x:
    if x is a pure literal(appears with same polarity everywhere in the formula):
        remove all clauses containing +/- x
        add a unit clause {+/-x}
```

5. **Consistent()**

First, we check whether the formula contains two unit clauses of form [+x] and [-x]. If these exist simultaneously in our formula, then we arrive at a contradiction. Hence, we return the string "unsat", and the integer 0 signifying that the formula is unsatisfiable.
Other conditions where the formula is inconsistent are:-

- The formula contains a non-unit clause
- The number of clauses is not equal to the total number of literals
- The formula does not contain one of the unit clauses representing the assertion/negation of a literal.

When one of the above three conditions is encountered, we return the string "continue" telling the model to continue the process.
If none of the conditions mentioned above are present, then the formula is said to be consistent and we return the string "sat" and integer 1 signifying that the formula is consistent.

6. **removeEmptyClauses()**

This function removes the empty clauses from the formula. Empty clauses might be present as a result of the unitPropagation and pureElimination steps. Removing the empty clauses is essential for the correct checking of the consistency of the formula.

7. **solve()**

This function combines all the above functions. It takes the list of clauses(formula) as the input and returns a string and the model depending upon whether the model is satisfiable or unsatisfiable.

Pseudocode:

```
solve(formula):
    formula = unitPropagation(formula)
    polarity, counts = polarityDict(formula)
    pureLits = pureLiterals(polarity)
    formula = pureElimination(pureLits, formula)
    formula = removeEmptyClauses(formula)

    if formula represents an empty list:
        return "unsat"

    if formula is consistent set of unit clauses containing all variables:
        return "sat", formula

    x := pick the literal with 0 count(if any) or else pick the max occuring literal

    if(solve(formula + [x]) returns "sat")
        return solve(formula + [x])
    else
        return solve(formula + [-x])
```

## Assumptions

The input file must be a .cnf file and the path to the input file must be clearly specified while running the satSolver.py file.

## Limitations

Since SAT solving is an exponential time algorithm, the execution time increases exponentially with the increase in the number of clauses and literals.