Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [86]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 16

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form. This homework is due at **11:59pm on Thursday, 02 June 2022**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

  `# YOUR CODE HERE`

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  ```
  assert x == 2
  ```

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.

- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

SAT

Let $p_1, p_2, \ldots$ be propositional variables. A SAT problem, represented in conjunctive normal form, consists in a conjunction of disjunctions of propositional variables and their complements, such as

$$(p_1 \vee \bar{p}_2 \vee p_3) \wedge (p_2 \vee p_5) \ .$$

We call each conjuct a *clause*; in the above example, we have two clauses, $c_1 = p_1 \vee \bar{p}_2 \vee p_3$ and $c_2 = p_2 \vee p_5$. The disjuncts in a clause are called *literals*: for example, the first clause $c_1 = p_1 \vee \bar{p}_2 \vee p_3$ contains the literals p_1, $\bar{p}_2$, and p_3.

The satisfiability question is: can we find a truth assignment to the variables that makes the expression true? In the above case, the answer is yes: we can take:

$$p_1 = True, \ p_5 = True$$

and any value for p_2, p_3.

NP completeness of SAT and Sudoku

This is optional material that can be skipped if desired.

In general, determining whether an expression for SAT (in conjunctive normal form) is satisfiable is an NP-complete problem, which means, intuitively, two things:

- **If you guess a solution, you can check it in polynomial time.** The problem being an NP problem means: if we could guess a solution (a truth assignment to the m variables), we would be able to check that the solution is valid in time that is polynomial in n, that is, that is bounded by a polynomial of n. In general, NP is the set of problems that can be solved in nondeterministic polynomial time, and thus, the set of problems that can be solved by making a lucky guess, and checking the guess in polynomial time.
- **The problem is as hard as any other NP problem** (or: the problem is NP-complete). Precisely, if we have another NP problem, we can reduce it in polynomial time to our problem, showing that if we could solve our problem in (deterministic) polynomial time, we could solve all other NP problems in polynomial time too. Unfortunately, it is not known whether NP complete can be solved in deterministic polynomial time.

Thus, NP-complete problems are problems such that, if you chance on the solution, you can check it efficiently, but the only way currently known for solving them consists in searching for a solution, perhaps with the help of heuristics. You will learn about NP completeness in

courses on computational complexity; a marvelous book on the topic, which every competent computer scientist should read, is the one by Garey and Johnson.

SAT is NP-complete; in fact, it is the prototypical NP-complete problem. As we unfortunately do not know how to do lucky guesses, solving it involves search.

Sudoku is actually not NP complete as defined: in fact, it can be solved *in constant time!* For our Sudoku problems, the input size $n = 9^2 = 81$, constant -- and this constant-sized problem can be solved in constant time just by searching through the fixed number of 9^{81} solutions! But 9^{81} is a large number, so our search algorithm is much better than the constant-time brute-force approach. It has been shown that a generalized version of Sudoku with unbounded input size is NP-complete (the author has not checked the proof).

SAT representation

To represent an instance of SAT, we represent literals, clauses, and the overall expression, as follows.

Literals. We represent the literal p_k via the positive integer k, and the literal $\bar{p}_k$ via the negative integer $-k$.

Clauses. We represent a clause via the set of integers representing the clause literals. For instance, we represent the clause $p_1 \vee \bar{p}_3 \vee p_4$ via the set $\{1, -3, 4\}$.

SAT problem. We represent a SAT problem (again, in conjunctive normal form) via the set consisting in the representation of its clauses. For instance, the problem

$$(p_1 \vee \bar{p}_2 \vee p_3) \wedge (p_2 \vee p_5)$$

is represented by the set of sets:

$$\{\{1, -2, 3\}, \{2, 5\}\} \ .$$

There are various operations that we need to do on clauses, and on the overall SAT problem, to solve it. Thus, we encapsulate both clauses, and the SAT problem, in python classes, so we can associate the operations along with the representations.

Clauses

We first define an auxiliary function, which tells us whether a set contains both an integer and its negative. This will be used, for instance, to detect whether a clause contains both a literal and its complement.

In [87]:
```
def has_pos_and_neg(l):
    return len(set(l)) > len({abs(x) for x in l})
```

This is the class representing a clause. Write it in such a way that:

- The list of literals is stored as a frozenset `self.literals`, so that we automatically remove duplicate literals, and so that we can easily define clause equality.
- It can be initialized either with a list of integers, or with a clause. In the first case, the integers are the literals. In the second case, we create a copy of the original clause.
- A clause is true iff it contains both a literal and its complement.
- A clause is false iff it is empty.

In [88]:
```
### Defining Clauses
import copy

class Clause(object):

    def __init__(self, clause):
        """Initializes a clause.  Here, the input clause is either a list or set
        of integers, or is an instance of Clause; in the latter case, a shallow
        copy is made, so that one can modify this clause without modifying the
        original clause.
        Store the list of literals as a frozenset."""
        # YOUR CODE HERE
        if isinstance(clause, Clause):
            self.literals = copy.copy(clause.literals)
        elif isinstance(clause, list) or isinstance(clause, frozenset):
            self.literals = frozenset(clause)
        else:
            self.literals = frozenset()


    def __repr__(self):
        return repr(self.literals)

    def __eq__(self, other):
        return self.literals == other.literals

    def __hash__(self):
        """This will be used to be able to have sets of clauses,
        with clause equality defined on the equality of their literal sets."""
        return hash(self.literals)

    def __len__(self):
        return len(self.literals)

    def __lt__(self, other):
        return len(self.literals) < len(other.literals)

    def __gt__(self, other):
        return len(self.literals) > len(other.literals)

    @property
    def istrue(self):
        """A clause is true if it contains both a predicate and its complement."""
        # YOUR CODE HERE
```

```
        for num in self.literals:
            if num * -1 in self.literals:
                return True
        return False

    @property
    def isfalse(self):
        """A clause is false if and only if it is empty."""
        # YOUR CODE HERE
        return self.literals == frozenset()
```

In [89]:
```
### Here you can test your code.
```

In [90]:
```
try:
    from nose.tools import assert_equal, assert_almost_equal
    from nose.tools import assert_true, assert_false
    from nose.tools import assert_not_equal
except:
    !pip install nose
    from nose.tools import assert_equal, assert_almost_equal
    from nose.tools import assert_true, assert_false
    from nose.tools import assert_not_equal
```

In [91]:
```
# 5 points: Tests for Clause.

c = Clause([2, -3, 4])
d = Clause([-3, 2, 4])
e = Clause(c)
assert_equal(c, d)
assert_equal(c, e)
```

In [92]:
```
# 5 points: More tests for clause

c = Clause([2, 3, 5])
assert_false(c.istrue)
assert_false(c.isfalse)

c = Clause([-2, 5, 2])
assert_true(c.istrue)
assert_false(c.isfalse)

c = Clause([])
assert_false(c.istrue)
assert_true(c.isfalse)
```

In [93]:
```
# 5 points: Tests for clause cloning.

c = Clause([2, -4, 3])
d = Clause(c)
assert_equal(c, d)
```

Truth assignments

We are seeking a truth assignment for the propositional variables that makes the expression true, and so, that makes each clause true.

We represent the truth assignment that assigns True to p_k via the integer k, and the truth assignment that assigns False to p_k via $-k$. Thus, if you have a (positive or negative) literal i, the truth assignment i will make it true.

We represent truth assignments to multiple variables simply as the set of assignments to individual variables. For example, the truth assignment that assigns True to p_1 and False to p_2 will be represented via the set $\{1, -2\}$.

Truth assignments and clause simplification

To solve a SAT instance, we need to search for a truth assignment to its propositional variables that will make all the clauses true. We will search for such a truth assignment by trying to build it one variable at a time. So a basic operation on a clause will be:

> Given a clause, and a truth assignment for one variable, compute the result on the clause.

What is the result on the clause? Consider a clause with representation c (thus, c is a set of integers) and a truth assignment i (recall that i can be positive or negative, depending on whether it assigns True or False to p_i). There are three cases:

- If $i \in c$, then the i literal of c is true, and so is the whole clause. We return True to signify it.
- If $-i \in c$, then the $-i$ literal of c is false, and it cannot help make the clause true. We return the clause $c \setminus \{-i\}$, which corresponds to the remaining ways of making the clause true under assignment i.
- If neither i nor $-i$ is in c, then we return c itself, as c is not affected by the truth assignment i.

Based on the above discussion, implement a *simplify* method for a Clause that, given a truth assignment, returns a simplified clause or True.

In [94]:
```
### Exercise: define simplify

def clause_simplify(self, i):
    """Computes the result simplify the clause according to the
    truth assignment i."""
    # YOUR CODE HERE
    if i in self.literals:
        return True
    elif i * -1 in self.literals:
        return Clause(self.literals.difference({i * -1}))
    else:
        return self
```

```
Clause.simplify = clause_simplify
```

In [95]:
```
### Here you can test your code.
```

Here are some tests to help you verify that your implementation works.

In [96]:
```
# 5 points.

# Let's test our simplification function.  Here is a clause.
c = Clause([1, 2, -3, 4])
# If we assign True to p_1, the whole clause is True.
assert_equal(c.simplify(1), True)

c = Clause([1, 2, -3, 4])
# If we assign False to 1 and True to 3, p_1 and p_3 are not useful
# any more to make the clause true.
assert_equal(c.simplify(-4), Clause([1, 2, -3]))

c = Clause([1, 2, -3, 4])
# Left unchanged.
assert_equal(c.simplify(12), c)
```

In [97]:
```
# 5 points: Additional tests
```

SAT Representation

A SAT instance consists in a set of clauses.

The SAT instance is satisfiable if and only if there is a truth assignment to predicates that satisfies all of its clauses. Therefore:

- If the SAT instance contains no clauses, it is trivially satisfiable.
- If the SAT instance contains an empty clause, it is unsatisfiable, since there is no way to satisfy that clause.

Based on this idea, the initializer method for our SAT class will get a list of clauses as input. It will discard the tautologically true ones (as indicated by the istrue clause method). If there is even a single unsatisfiable clause, then we set the SAT problem to consist of only one unsatisfiable clause, as a shorthand for denoting that the SAT problem cannot be satisfied.

We endow the SAT class with methods isfalse and istrue, that detect SAT problems that are trivially satisfiable by any truth assignment, or trivially unsatisfiable by any truth assignment.

You will need to implement the methods *generate_candidate_assignments*, *apply_assignment*, and *solve*, which together will be used to search for a solution of the SAT instance. These methods are discussed below.

In [98]:
```
class SAT(object):

    def __init__(self, clause_list):
        """clause_list is a list of lists (or better, an iterable of
        iterables), to represent a list or set of clauses."""
        raw_clauses = {Clause(c) for c in clause_list}
        # We do some initial sanity checking.
        # If a clause is empty, then it
        # cannot be satisfied, and the entire problem is False.
        # If a clause is true, it can be dropped.
        self.clauses = set()
        for c in raw_clauses:
            if c.isfalse:
                # Unsatisfiable.
                self.clauses = {c}
                break
            elif c.istrue:
                pass
            else:
                self.clauses.add(c)

    def __repr__(self):
        return repr(self.clauses)

    def __eq__(self, other):
        return self.clauses == other.clauses
```

Now define two functions (that are defined as properties), `istrue` and `isfalse`.

- A SAT instance is true only if it contains no clauses. This because empty clauses are not added to the SAT instance (see above initializer).
- A SAT instance is false if it contains a false clause.

In [99]:
```
### istrue, isfalse, for SAT.

def sat_istrue(self):
    # YOUR CODE HERE
    return self.clauses == set()

def sat_isfalse(self):
    # YOUR CODE HERE
    for i in self.clauses:
        if i.isfalse:
            return True
    return False

SAT.istrue = property(sat_istrue)
SAT.isfalse = property(sat_isfalse)
```

In [100…
```
### Here you can test your code.
```

In [101…
```
### 10 points: tests for .istrue, .isfalse.
```

```
s = SAT([])
assert_true(s.istrue)
assert_false(s.isfalse)

s = SAT([[3, 4, -4], [-2, 4]])
assert_false(s.istrue)
assert_false(s.isfalse)

s = SAT([[3, 4, -4], [-2, 2]])
assert_true(s.istrue)
assert_false(s.isfalse)

s = SAT([[3, -4], []])
assert_false(s.istrue)
assert_true(s.isfalse)
```

generate_candidate_assignments

In order to solve a SAT instance, we proceed with the choice-constraint propagation-recursion setting. Let us build the choice piece first. The idea is this: if we are to make true a clause c, we have to make true at least one of its literals. Thus, we can pick a clause c, and try the truth assignment corresponding to each of its literals in turn: at least one of them should work. Which clause is best to pick? As in the Sudoku case, one with minimal length, so that the probability of one of its literals being true is highest.

Based on this, write a method *generate_candidate_assignments* in the above SAT class, which returns the list or set of literals of one of the clauses of minimal length. These will be the truth assignments we will need to try in turn. Below are some tests that your code should pass.

Note: the solution can (but need not) be written in one line of code.

In [102…
```
### Definition of `generate_candidate_assignments`

def sat_generate_candidate_assignments(self):
    """Generates candidate assignments.
    Picks one of the shortest clauses, and return as candidate assignments
    a list of sets, one for each of the literals of the chosen clause."""
    # YOUR CODE HERE
    return min(self.clauses).literals


SAT.generate_candidate_assignments = sat_generate_candidate_assignments
```

In [103…
```
### Here you can test your code.
s = SAT([[-1,-2,3],[2,-3],[1,-4,2,1]])
s.generate_candidate_assignments()
```

Out[103… `frozenset({-3, 2})`

In [104…
```
### 5 points: Tests for `generate_candidate_assignments`
```

```
s = SAT([[-1,-2,3],[2,-3],[1,-4,2,1]])
assert_equal(set(s.generate_candidate_assignments()), {2, -3})
```

apply_assignment

Once we pick a truth assignment from one of the literals above, we need to propagate its effect to the clauses of the SAT instance.

Write an *apply_assignment* method in the SAT class given above, that takes as input a truth assignment i, and returns a new SAT object, whose clauses are obtained by simplifying the clauses of the current assignment according to i. Clauses that are made true by i (clauses where the *simplify* method returns True) should not be part of the new SAT problem, since they are already satisfied.

We provide below some tests for your code.

Note: the solution can (but need not) be written in two lines of code.

In [105…
```
### Exercise: define `apply_assignment`

def sat_apply_assignment(self, assignment):
    """Applies the assignment to every clause.
    If the result of the simplification is True (the boolean True),
    the clause is discarded. The function returns a SAT problem
    consisting of the simplified, non-True, clauses."""
    # YOUR CODE HERE
    clauses = set()
    for i in self.clauses:
        simple = i.simplify(assignment)
        if type(simple) != bool:
            clauses.add(simple)
    return SAT(clauses)


SAT.apply_assignment = sat_apply_assignment
```

In [106…
```
### Here you can test your code.
```

In [107…
```
### 5 points: Tests for `apply_assignment`

# First, examples in which each clause is simplified and is part of the
# new SAT problem.
s = SAT([[-1, -2, 3], [2, -3], [5, -4, 2, 10]])
t = s.apply_assignment(1)
assert_equal(t, SAT([[-2, 3], [2, -3], [5, -4, 2, 10]]))

s = SAT([[2, 3], [4, 2, -3], [2]])
t = s.apply_assignment(-2)
assert_equal(t, SAT([[3], [4, -3], []]))
```

In [108…
```
### 5 points: More tests for `apply_assignment`
```

```
# Second, an example in which some clauses are made True, and hence removed
# from the new SAT problem.
s = SAT([[-1, -2, 3], [2, -3], [5, -4, 2, 10]])
t = s.apply_assignment(-1)
assert_equal(t, SAT([[2, -3], [5, -4, 2, 10]]))

s = SAT([[2, 3, -4], [-1, -3, 5], [-3]])
t = s.apply_assignment(3)
assert_equal(t, SAT([[-1, 5], []]))
```

solve

The main method for searching for a solution of the SAT instance is the *solve* method. The *solve* method takes no arguments, and should return either False, if the SAT instance is unsatisfiable, or a truth assignment that satisfies it. The satisfying truth assignment should be returned as a set.

The *solve* method uses *generate_candidate_assignments* and *apply_assignment* above.

First, the *solve* method should check whether the SAT instance S is trivially unsatisfiable (and return False) or trivially satisfiable (and return the empty set), using the *istrue* and *isfalse* methods. This takes care of the base cases of the search.

If none of the above applies, *solve* must generate candiate truth assignments, and try them one by one. Each candidate truth assignment, once applied, gives rise to a new SAT problem S'; this new SAT problem can be solved by calling *SAT* recursively. If the new SAT problem S' has no solution, you can move on to the next candidate assignment, if any; if the new SAT problem S' has a solution, the solution can be combined with the candidate truth assignment that gave rise to S', to form a complete solution of the original problem S.

In [132…

```
### Exercise: define `solve`

def sat_solve(self):
    """Solves a SAT instance.
    First, it checks whether the instance is false (in which case
    it returns False) or true (in which case it returns an empty
    assignment).
    If neither of these applies, generates a list of candidate
    assignments, and for each of them, applies them to the current SAT
    instance, generating a new SAT instance, and solves it.
    If the new SAT instance has a solution, merges it with the assignment,
    and returns it.  If it has no solution, tries the next candidate
    assignment.  If no candidate assignment works, returns False, as
    the SAT problem cannot be satisfied."""
    # YOUR CODE HERE
    a = set()
    if self.istrue:
        return a
    if self.isfalse:
        return False
    c = self.generate_candidate_assignments()
```

```
    for candidate in c:
        solved = self.apply_assignment(candidate).solve()
        if type(solved) == set:
            a.add(candidate)
            a.update(solved)
            return a
    return False

SAT.solve = sat_solve
```

To help you verify your code, let us write a method *apply_assignment* that, given a SAT problem, applies an assignment to it, and returns True if the SAT instance is satisfied.

In [133…
```
def sat_verify_assignment(self, assignment):
    assert not has_pos_and_neg(assignment), "The assignment is inconsistent"
    s = self
    for i in assignment:
        s = s.apply_assignment(i)
        if s.istrue:
            return True
        if s.isfalse:
            return False
    return False

SAT.verify_assignment = sat_verify_assignment
```

In [134…
```
### Here you can test your code.
s = SAT([[1, 2], [-2, 2, 3], [-3, -2]])
a = s.solve()
print("Assignment:", a)
s.verify_assignment(a)
```

```
Assignment: {1, -3}
```

Out[134… True

In [135…
```
### 5 points: A solvable problem

s = SAT([[1, 2], [-2, 2, 3], [-3, -2]])
a = s.solve()
print("Assignment:", a)
assert_true(s.verify_assignment(a))
```

```
Assignment: {1, -3}
```

In [136…
```
### 5 points: Another solvable problem.

s = SAT([[1, 2], [-2, 3], [-3, 4], [-4, 5], [8, -1]])
a = s.solve()
print("Assignment:", a)
assert_true(s.verify_assignment(a))
```

```
Assignment: {1, 8, -4, -3, -2}
```

In [137…
```
### 5 points: Yet another solvable problem
```

```
s = SAT([[-1, 2], [-2, 3], [-3, 1]])
a = s.solve()
print("Assignment:", a)
assert_true(s.verify_assignment(a))
```

```
Assignment: {1, 2, 3}
```

In [138…

```
### 5 points: An unsolvable problem

s = SAT([[1], [-1, 2], [-2]])
assert_false(s.solve())
```

In [139…

```
### 5 points: Another unsolvable problem

s = SAT([[-1, 2], [-2, 3], [-3, -1], [1]])
assert_false(s.solve())
```

In [140…

```
### 5 points: Yet another unsolvable problem

s = SAT([[-1, 2], [-2, 3], [-3, -1], [1], [-4, -3, -2]])
assert_false(s.solve())
```

In [123…

Epilogue

The above method for solving SAT instances works!

Is that how it is done in real SAT solvers?

Not quite. What we wrote above is perhaps the simplest SAT solver, not the most efficient. Since SAT solvers are incredibly versatile -- many problems can be encoded into boolean satisfiability -- a very large amount of work has been done to make them faster and better. Indeed, the study of strategies for SAT solvers is a field of computer science in itself, with its own conferences (SAT) and competitions dedicated to it. In particular, the constraint propagation generally uses three procedures in addition to those used above:

Unary clauses correspond to truth assignments

If a clause contains only one literal, that literal must be part of the truth assignment, as it is the only way to satisfy the clause. We note that we include this in an indirect way: when searching, we will preferentially select an unary clause if there is any, and take its literal as the (only) search option.

Binary clauses correspond to implications and can be propagated quickly

If a clause is $p_1 \vee p_2$, it can also be read as $\neg p_1 \rightarrow p_2$, so that if p_1 is assigned False, we know that we need to assign True to p_2. We can represent this via an edge from the literal $\bar{p}_1$

to the literal p_2. Proceeding in this way, by looking at all binary clauses we can compute a graph (and keep it cached) that enables us to efficiently propagate truth assignments to literals.

Our simple SAT solver also does a bit of this. If we assign False to p_1, the binary clause $p_1 \vee p_2$ is simplified in p_2, and next time, p_2 as a unary clause will be likely to be chosen to become part of the truth assignment. Thus, our use of *simplify* and *solve*, along with the heuristic of choosing the shortest clauses in *solve*, go part of the way towards implementing this technique. A cached, precomputed graph however leads to a much faster implementation.

Resolution

Consider two clauses that share a predicate variable, that appears positive (not complemented) in one clause, and complemented in the other:

$$c_1 \ : \ p \vee q_1 \vee \cdots \vee q_n$$
$$c_2 \ : \ \bar{p} \vee r_1 \vee \cdots \vee r_m$$

where $q_1, \ldots, q_n, r_1, \ldots, r_m$ are literals (positive or negative predicate variables). There are two cases:

- either p is False, and $q_1 \vee \cdots \vee q_n$ must be true,
- or p is True, and $r_1 \vee \cdots \vee r_m$ must be true.

In either case, the disjuction

$$c_{12} \ : \ q_1 \vee \cdots \vee q_n \ \vee \ r_1 \vee \cdots \vee r_m$$

must hold.
The process of merging p_1 and p_2 into p_{12} is called *resolution*. Resolution generates *more* clauses: once p_{12} is generated, we cannot discard p_1 or p_2. However, having more clauses can lead to better constraint propagation: generally, what SAT solvers try to do is to cut down on the amont of exploration to be done, that is, they try to cut down the number of truth assignments that must be tried; limiting the number of clauses is a secondary concern. Many heuristics are used to find clauses that are useful candidates for unification.

Clause learning

Many modern SAT solvers, when they encounter a dead end in the exploration and need to backtrack, they summarize the fact that a particular truth assignment cannot be extended to a satisfying one via a *learned clause*. These learned clauses are instrumental in pruning the remaining exploration to be performed.

Precisely, assume that the truth assignment so far is $x_1, \ldots, x_m$, where each x_i, $1 \leq i \leq m$, is a literal. If we find that there is no solution with this truth assignment, we need to

backtrack in the search. We can then add the clause $\neg(x_1 \wedge \cdots \wedge x_m) = \bar{x}_1 \vee \cdots \vee \bar{x}_m$ to remember that the truth assignment was shown to be a dead end.

To learn more

To learn more, you can look at how the MiniSat solver works; MiniSat is a simple yet efficient SAT solver.