# Lab 5: SAT Solver

The questions below are due on Monday October 16, 2017; 10:00:00 PM.

---

**You are not logged in.**

If you are a current student, please Log In (https://6009.csail.mit.edu/fall17/labs/lab5?loginaction=redirect) for full access to the web site.
Note that this link will take you to an external site (`https://oidc.mit.edu`) to authenticate, and then you will be redirected back to this page.

# Table of Contents


- 1) Preparation (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_1)
- 2) Introduction (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_2)
- 3) Implementing Your SAT Solver (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_3)
- 4) Scheduling by Reduction (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_4)
- 5) Code Submission (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_5)
- 6) Checkoff (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_6)
    - 6.1) Grade (https://6009.csail.mit.edu/fall17/labs/lab5#catsoop_section_6_1)


# 1) Preparation

This lab assumes you have Python 3.5 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab5.zip` (https://6009.csail.mit.edu/fall17/lab_distribution.zip?path=%5B%22fall17%22%2C+%22labs%22%2C+%22lab5%22%5D)

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file. Submissions for the lab are due on Monday, 16 October. You may submit portions of the lab late (see the grading page (https://6009.csail.mit.edu/fall17/grading) for more details), but the last day to submit this lab will be Friday, 20 October.

This lab is worth a total of 4 points. Your score for the lab is based on:

- answering the questions on this page (0.3 points)
- passing the test cases from `test.py` under the time limit (1.7 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (2 points).

For this lab, you will only receive credit for a test case if it runs to completion in under 30 seconds on the server.

Please also review the collaboration policy (https://6009.csail.mit.edu/fall17/collaboration) before continuing.

# 2) Introduction

From recreational mathematics to standardized tests, one popular problem genre is *logic puzzles*, where some space of possible choices is described using a list of rules. A solution to the puzzle is a choice (often the one unique choice) that obeys the rules. Similarly to the way we ruined Sudoku forever in lecture, if nothing else, this lab should give you the tools to make short work of any of those puzzles, assuming you have your trusty Python interpreter.

Here's an example of the kind of logic puzzle we have in mind.

The 6.009 staff were pleased to learn that grateful alumni had donated cupcakes for last week's staff meeting. Unfortunately, the cupcakes were gone when the staff showed up for the meeting! Who ate the cupcakes?

1. The suspects are Adam C., Adam H., Chris, Duane, and Tim the Beaver.
2. Whichever suspect ate any of the cupcakes must have eaten *all* of them.
3. The cupcakes included exactly two of the flavors chocolate, vanilla, and pickles.
4. Adam C. only eats pickles-flavored cupcakes.
5. Years ago, Chris and Duane made a pact that, whenever either of them eats cupcakes, he must share with the other one.
6. Adam H. feels strongly about flavor fairness and will only eat cupcakes if he can include at least 3 different flavors.

Let's translate the problem into **Boolean logic (https://en.wikipedia.org/wiki/Boolean_algebra)**, where we have a set of variables, each of which takes the value True or False. We write the rules as conditions over these variables. Here is each rule as a Python expression over Boolean variables. We include one variable for the guilt of each suspect, plus one variable for each potential flavor of cupcake.

In reading these rules, note that Python `not` binds more tightly than `or`, so that `not p or q` is the same as `(not p) or q`. It's also fine not to follow every last detail, as this rule set is just presented as one example of a general idea! You may also find that some of our encoding choices don't match what you would come up with, such that our choices lead to longer or less comprehensible rules. We are actually intentionally forcing ourselves to adhere to a restricted format that we will explain shortly.

```
rule1 = adamC or adamH or chris or duane or tim
# At least one of them must have committed the crime!  Here, one of these
# variables being True represents that person having committed the crime.

rule2 = (not adamC or not adamH) and (not adamC or not chris) \
  and (not adamC or not duane) and (not adamC or not tim) \
  and (not adamH or not chris) and (not adamH or not duane) \
  and (not adamH or not tim) and (not chris or not duane) \
  and (not chris or not tim) and (not duane or not tim)
# Only one of the suspects is guilty.  In other words, for any pair of suspects,
# at least one must be NOT guilty.  That means we can't find a pair of guilty
# suspects.

rule3 = (not chocolate or not vanilla or not pickles) \
  and (chocolate or vanilla) and (chocolate or pickles) and (vanilla or pickles)
# It can't be that all flavors were present, and among any pair of flavors, at
# least one was present.

rule4 = (not adamC or pickles) and (not adamC or not chocolate) \
  and (not adamC or not vanilla)
# If Adam C. is not guilty, this will be true.  Otherwise, it will be true if
# only pickles-flavored cupcakes were present.

rule5 = (not chris or duane) and (not duane or chris)
# If Chris ate cupcakes without sharing with Duane, the first case will fail to
# hold.  Likewise for Duane eating without sharing.

rule6 = (not adamH or chocolate) and (not adamH or vanilla) \
  and (not adamH or pickles)
# If Adam H. is the culprit and we left out a flavor, the corresponding case
# here will fail to hold.

rules = rule1 and rule2 and rule3 and rule4 and rule5 and rule6
```

For some practice coming up with Boolean formulas, consider this **truth table**, explaining which values of Boolean variables `a`, `b`, and `c` lead to which values of a formula. We write T for `True` and F for `False`.

| a | b | c | **Answer** |
|---|---|---|---|
| F | F | F | F |
| T | F | F | F |
| F | T | F | F |
| T | T | F | F |
| F | F | T | F |
| T | F | T | F |
| F | T | T | F |
| T | T | T | T |

Write a Python formula over `a`, `b`, and `c` that matches the truth table. (The format should be as in the puzzle encoding above, e.g. `adam or not (chris and duane)`, and your formula need not follow the "CNF" restriction introduced below).

In encoding the puzzle, we followed a very regular structure in our Boolean formulas, one important enough to have a common name: **conjunctive normal form (CNF) (https://en.wikipedia.org/wiki/Conjunctive_normal_form)**. We say that a *literal* is a variable or the `not` of a

variable. Then a *clause* is a multi-way `or` of literals, and a CNF formula is a multi-way `and` of clauses.

When we commit to this restrictive format, we can represent:

- a *variable* as a Python string
- a *literal* as a pair of a variable and a Boolean value ( `False` if `not` appears in this literal, `True` otherwise)
- a *clause* as a list of literals
- a *formula* as a list of clauses

For example, our puzzle from above can be encoded as follows, where again it is OK not to read through every last detail.

```
rule1 = [[('adamC', True), ('adamH', True), ('chris', True),
          ('duane', True), ('tim', True)]]
rule2 = [[('adamC', False), ('adamH', False)],
         [('adamC', False), ('chris', False)],
         [('adamC', False), ('duane', False)],
         [('adamC', False), ('tim', False)],
         [('adamH', False), ('chris', False)],
         [('adamH', False), ('duane', False)],
         [('adamH', False), ('tim', False)],
         [('chris', False), ('duane', False)],
         [('chris', False), ('tim', False)],
         [('duane', False), ('tim', False)]]
rule3 = [[('chocolate', False), ('vanilla', False), ('pickles', False)],
         [('chocolate', True), ('vanilla', True)],
         [('chocolate', True), ('pickles', True)],
         [('vanilla', True), ('pickles', True)]]
rule4 = [[('adamC', False), ('pickles', True)],
         [('adamC', False), ('chocolate', False)],
         [('adamC', False), ('vanilla', False)]]
rule5 = [[('chris', False), ('duane', True)],
         [('duane', False), ('chris', True)]]
rule6 = [[('adamH', False), ('chocolate', True)],
         [('adamH', False), ('vanilla', True)],
         [('adamH', False), ('pickles', True)]]
rules = rule1 + rule2 + rule3 + rule4 + rule5 + rule6
```

Consider this Boolean formula.

```
(c or b) and a and a and not a and (c or a)
```

Write an equivalent CNF formula (as a Python literal), in the format used in the lab (e.g., `[[('a', True), ('b', False)], [('c', True)]]` ).

Now, consider this Boolean formula.

```
((b or c) and not a) or (a and b and (a or c))
```

Write an equivalent CNF formula (as a Python literal), in the format used in the lab (e.g., `[[('a', True), ('b', False)], [('c', True)]]` ).

A classic tool that works on Boolean formulas is a **satisfiability solver (https://en.wikipedia.org/wiki/Boolean_satisfiability_problem#Algorithms_for_solving_SAT)**, or SAT solver. Given a formula, either the solver finds Boolean variable values that make the formula true, or the solver indicates that no solution exists. In this lab, you will write a SAT solver that can solve puzzles like ours, as in:

```
>>> satisfying_assignment(rules)
{'duane': False, 'chris': False, 'chocolate': False, 'adamH': False,
 'adamC': False, 'pickles': True, 'tim': True, 'vanilla': True}
```

The return value of `satisfying_assignment` is a dictionary mapping variables to the Boolean values that have been inferred for them. So, we see that Tim the Beaver is guilty and has a taste for vanilla and pickles! Actually, other legal answers would have had Tim enjoying other flavors, too. However, Tim is the uniquely determined culprit. How do we know? The SAT solver fails to find an assignment when we add an additional rule proclaiming Tim's innocence.

```
>>> satisfying_assignment(rules + [[('tim', False)]])
# Nothing printed (corresponds to answer of None)
```

There's one very easy way to solve Boolean puzzles: enumerate all possible Boolean assignments to the variables. Evaluate the rules on each assignment, returning the first assignment that works. Unfortunately, this process can take prohibitively long to run! For a concrete illustration of why, consider this Python code to generate all sequences of Booleans of a certain length. When we have *N* Boolean variables in our puzzle, the possible assignments can be represented as length-*N* sequences of Booleans.

```
def all_bools_generator(length):
    if length == 0:
        yield []
    else:
        for v in all_bools_generator(length-1):
            yield [True] + v
            yield [False] + v
def all_bools(length):
    return list(all_bools_generator(length))
```

Here's an example output.

```
>>> all_bools(3)
[[True, True, True], [False, True, True], [True, False, True],
 [False, False, True], [True, True, False], [False, True, False],
 [True, False, False], [False, False, False]]
```

We could get more ambitious and try to generate longer sequences.

```
>>> len(all_bools(3))
8
>>> len(all_bools(4))
16
>>> len(all_bools(5))
32
>>> len(all_bools(6))
64
>>> len(all_bools(20))
1048576
>>> len(all_bools(25))
# Python runs for long enough that we give up!
```

It's actually quite expensive even to run through all Boolean sequences of nontrivial lengths, let alone to test each sequence against the rules. The reason is that there are $2^N$ length-$N$ Boolean sequences, and that kind of exponential function grows quite quickly in $N$.

If we hope to be world logic-puzzle champions, we'll need to be ready for puzzles that lead to hundreds of Boolean variables. Are we out of luck if we want Python to do all the work? Worry not! In this lab, you will implement a SAT solver that uses a much smarter algorithm than *brute-force* enumeration of all assignments, thanks to the magic of backtracking search.

# 3) Implementing Your SAT Solver

In this assignment, we ask you to implement the **Davis–Putnam–Logemann–Loveland (DPLL) algorithm (https://en.wikipedia.org/wiki/DPLL_algorithm)** for Boolean satisfiability. Instead of enumerating all assignments, we will repeatedly *pick a variable to split on*. That is, we consider the case where the variable is `True` and the case where it is `False`, with each corresponding to a recursive call. It would be good to convince yourself that, since in the recursive calls we never need to revisit variables that we already split on, such a recursion must terminate eventually, giving exactly the right answer. Why this approach is an improvement on brute-force enumeration is less obvious, but we'll get to that shortly.

Here's a first cut at the algorithm.

To solve satisfiability of formula *F*:

1. If *F* contains no more variables, then return an empty assignment.
2. Otherwise, let *x* be the first variable that appears in *F*.
3. Consider the case where *x* is `True`.
    1. Create *FT*, a version of *F* that incorporates all the consequences of fixing *x* as `True`. It will necessarily be the case that *x* no longer appears in *FT*.
    2. Actually, in trying to produce *FT*, we might realize a contradiction, if some clause actually **requires** *x* to be `False`. If so, skip ahead to trying *x* = `False`.
    3. Run the algorithm recursively on *FT*. If it finds an assignment, return that assignment with *x* = `True` added. Otherwise, on to the next case.
4. Consider the case where *x* is `False`.
    1. Create *FF*, a version of *F* that incorporates all the consequences of fixing *x* as `False`. It will necessarily be the case that *x* no longer appears in *FF*.
    2. Actually, in trying to produce *FF*, we might realize a contradiction, if some clause actually **requires** *x* to be `True`. If so, return `None` as the answer for `F`, too.
    3. Otherwise, run the algorithm recursively on *FF*. If it finds an assignment, return that assignment with *x* = `False` added. Otherwise, return `None`.

A key operation here is updating a formula to model the effect of a variable assignment. As an example, consider this starting formula.

```
(a or b or not c) and (c or d)
```

If we learn `c = True`, then the formula should be updated as follows.

```
(a or b)
```

We removed `not c` from the first clause, because we now know conclusively that that literal is `False`. Conversely, we can remove the second clause, because with `c` true, it is assured that the clause will be satisfied. Note a key effect of this operation: *variable `d` has disappeared from the formula, so we no longer need to consider values for `d`*. In general, this approach often saves us from an exponential enumeration of variable values, because we learn that, in some branches of the search space, some variables are actually irrelevant to the problem. This pruning will show up in the assignments that your SAT solver returns: you are allowed (but not required) to omit variables that turn out to be irrelevant.

For completeness, here is how we update the formula when we learn `c = False`.

```
d
```

Note that, with this formula, it is rather obvious which variable to set next and what Boolean value to give it! That brings us to the next topic (after an interlude of two questions to check your understanding).

Consider this CNF formula.

```
(c or c or b) and (not a or b or not c)
```

Write a CNF formula (in normal Python expression syntax, e.g. `(a or b) and (c or not d)`) for the case where `a = True`, without using variable `a`. Write `None` if this assignment falsifies the formula, and leave the blank empty if this assignment renders the formula true.

Write a CNF formula (in normal Python expression syntax, e.g. `(a or b) and (c or not d)`) for the case where `a = False`, without using variable `a`. Write `None` if this assignment falsifies the formula, and leave the blank empty if this assignment renders the formula true.

A further optimization to this procedure will be needed, to pass all of the test cases quickly enough. The one we require you to implement is **unit propagation**, a heuristic for choosing variables to split on. Namely, at the start of any call to your procedure, check if the formula contains any length-one clauses. If such a clause `[(x, b)]` exists, then we may set `x` to Boolean value `b`, just as we do in the `True` and `False` cases of the outline above. However, we know that, if this setting leads to failure, there is no need to backtrack and also try `x = not b`! The reason is that the unit clause alone tells us exactly what the value of `x` must be.

So, for your unit-propagation optimization, begin the function with a loop that repeatedly finds unit clauses, if any, and propagates their consequences through the formula. One unit propagation may reveal further unit clauses, whose later propagations may themselves reveal more unit clauses, and so on.

Implementing function `satisfying_assignment` in this way should allow your code to pass the first half of our test cases in time. You are free to add additional optimizations beyond what we laid out above, or even make broader changes to the algorithm, so long as you avoid "hard coding" for rather specific SAT problems (except for base cases like empty formulas).

# 4) Scheduling by Reduction

It's possible to write a new implementation of backtracking search for each new problem we encounter, but another strategy is to *reduce* a new problem to one that we already know how to solve well. Boolean formulas are a popular target for reductions, because a lot of effort has gone into building fast SAT solvers. In this last part of the lab, you will implement a reduction to SAT from a simple scheduling problem.

In particular, we are interested in the real-life problem of assigning 6.009 students to different sessions for taking a quiz. Each student is available for only some of the sessions, but each session has limited capacity. We want to find a schedule (assignment of students to sessions) that respects all the constraints.

Please implement function `boolify_scheduling_problem(student_preferences, session_capacities)`.

- Argument `student_preferences` is a dictionary mapping a student name (string) to a set of session names (strings) when that student is available.
- Argument `session_capacities` is a dictionary mapping each session name to a positive integer for how many students can fit in that session.
- The function returns a CNF formula encoding the schedule problem, as we explain next.

Here's an example call:

```
boolify_scheduling_problem({'Alice': {'basement', 'penthouse'},
                            'Bob':   {'kitchen', 'penthouse'}},
                           {'basement':  1,
                            'kitchen':   2,
                            'penthouse': 3})
```

In English, Alice is available for the sessions in the basement and penthouse, and Bob is available for the sessions in the kitchen and penthouse. The basement can fit 1 student, the kitchen 2 students, and the penthouse 3 students. One legal schedule would be Alice in the basement and Bob in the kitchen. Your job is to translate such inputs into CNF formulas, such that your SAT solver can then find a legal schedule (or confirm that none exists).

The CNF formula you output should mention only Boolean variables named like `student_session`, where `student` is the name of a student, and where `session` is the name of a session. (We will only call your code on student and session names that are free of underscore characters, so there is no issue with ambiguity in naming Boolean variables.) The variable `student_session` should be `True` if and only if that student is assigned to that session. The CNF clauses you include should enforce exactly the following rules:

1. Each student is assigned to exactly one session.
2. Students are only assigned to sessions included in their preferences.
3. No session has more assigned students than it can fit.

Our requirement for this part of the lab is that your code *should not* solve the optimization problem. Rather, you should implement a "dumb" translation to CNF formulas. One concrete manifestation of that rule is that *your translation should consider students and sessions completely separately*, effectively generating a separate CNF formula for each and combining them with `and`. The formula for some student preferences should be usable accurately with the formula for any session capacities (though the latter depends on the names of available students, but not their preferences). During checkoff, our staff will try to make sure you've followed this rule and not merely, say, implemented a backtracking search directly for the scheduling problem, to output trivial CNF problems that directly encode answers.

If you're feeling stuck on how to encode these rules in CNF, you can consult our hints (https://6009.csail.mit.edu/fall17/labs/lab5/hints). The following sequence of questions might also function as a hint for how to implement these constraints as CNF clauses.

Write a CNF Python formula (e.g. `(a or b) and (c or not d)`) over `a`, `b`, and `c` expressing that exactly 2 of those variables are `True`.

Write a CNF Python formula (e.g. `(a or b) and (c or not d)`) over `a`, `b`, `c`, and `d` expressing that exactly 3 of those variables are `True`.

Write a CNF Python formula (e.g. `(a or b) and (c or not d)`) over `a`, `b`, `c`, and `d` expressing that at most 3 of those variables are `True`.

Write a CNF Python formula (e.g. `(a or b) and (c or not d)`) over `a`, `b`, `c`, and `d` expressing that at most 2 of those variables are `True`.

As usual, we have provided a browser UI for this lab. When you have implemented both main functions of the lab, run `python3 server.py` for an interface to load scheduling test cases and see the results of scheduling. As usual, visit `http://localhost:8000/` to load the UI. Running a test case involves generating a CNF formula and searching for a satisfying assignment for it, each done by calling one of your functions.

# 5) Code Submission

Select File `No file selected`

# 6) Checkoff

Once you are finished with the code, please come to a tutorial, lab session, or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular, be prepared to discuss:

- Your implementation of `satisfying_assignment`, including any helper functions.
- In English, the general recipe you came up with for translating scheduling problems into CNF formulas.
- Your code for `boolify_scheduling_problem` to implement that recipe.

## 6.1) Grade

*You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.*

```
\
/   /\__/\
\__=(  o_O )=
(_________)
 |_ |_ |_ |_
```