SAT Solver

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

The following file contains code and other resources as a starting point for this lab: `sat.zip`

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.

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- a style check (1 point), and
- passing the tests in `test.py` (4 points).

Reminder: Academic Integrity

Please also review the academic-integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

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. 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 6.101 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, Duane, Jonathan, Saman, 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. Duane only eats pickles-flavored cupcakes.
> 5. Years ago, Jonathan and Saman made a pact that, whenever either of them eats cupcakes, they must share with the other one.
> 6. Adam 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, 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 and that will ultimately make the job of *solving* these kinds of problems more straightforward.

```
rule1 = (duane or adam or jonathan or saman 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 duane or not adam)
     and (not duane or not jonathan)
     and (not duane or not saman)
     and (not duane or not tim)
     and (not adam or not jonathan)
     and (not adam or not saman)
     and (not adam or not tim)
     and (not jonathan or not saman)
     and (not jonathan or not tim)
     and (not saman or not tim))
# At most one of the suspects is guilty.  In other words, for any pair of
# suspects, at least one must be NOT guilty (so that we cannot possibly
find
# two or more people guilty).

# Together, rule2 and rule1 guarantee that exactly one suspect is guilty.


rule3 = ((not chocolate or not vanilla or not pickles)
     and (chocolate or vanilla)
     and (chocolate or pickles)
     and (vanilla or pickles))
# Here is our rule that the cupcakes included exactly two of the flavors.
Put
# another way: we can't have all flavors present; and, additionally,
among
# any pair of flavors, at least one was present.


rule4 = ((not duane or pickles)
     and (not duane or not chocolate)
     and (not duane or not vanilla))
# If Duane is guilty, this will evaluate to True only if only pickles-
flavored
# cupcakes were present.  If Duane is not guilty, this will always
evaluate to
# True.  This is our way of encoding the fact that, if Duane is guilty,
only
# pickles-flavored cupcakes must have been present.
```

```
rule5 = (not jonathan or saman) and (not saman or jonathan)
# If Jonathan ate cupcakes without sharing with Saman, the first case
will fail
# to hold.  Likewise for Saman eating without sharing.  Since Jonathan
and Saman
# only eat cupcakes together, this rule excludes the possibility that
only one
# of them ate cupcakes.


rule6 = ((not adam or chocolate)
     and (not adam or vanilla)
     and (not adam or pickles))
# If Adam is the culprit and we left out a flavor, the corresponding case
here
# will fail to hold.  So this rule encodes the restriction that Adam can
only
# be guilty if all three types of cupcakes are present.


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

The piece of code above is a Python program that will tell us whether a given assignment is consistent with the rules we have laid out. For example, if we had set the following variables (representing the hypothesis that Duane was guilty and that only pickles-flavored cupcakes were present):

```
duane = True
adam = False
jonathan = False
saman = False
tim = False

pickles = True
vanilla = False
chocolate = False
```

and then run the code, the `satisfied` variable would be set to `False` (since `rule3` would be `False`), indicating that this assignment did not satisfy the rules we had set out.

If we instead try the following bindings, what is the result of `satisfied`? You can test this by running the code on your own machine!

```
duane = False
adam = False
jonathan = False
saman = False
tim = True

pickles = False
vanilla = True
chocolate = False
```

2.1) Solving the Puzzle

While code like the above could be useful in certain situations, it doesn't help us *solve* the problem (it only helps us check a possible solution). In this lab, we'll look at the problem of Boolean satisfiability: our goal will be, given a description of Boolean variables and constraints on them (like that given above), to find a set of assignments that satisfies all of the given constraints.

2.2) Conjunctive Normal Form

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).

In this form, we say that a *literal* is a variable or the `not` of a variable. Then a *clause* is a multiway `or` of literals, and a CNF *formula* is a multiway `and` of clauses.

It's okay if this representation does not feel completely natural. Some people find this form to be "backwards" from the way they would otherwise think about these constraints. However, forcing our constraints to be in this form can simplify the problem of implementing our solver, compared to other representations we could choose. We'll try in this writeup to help you with the pieces of the lab involving converting expressions to CNF.

2.2.1) Python Representation

When we commit to representing problems in CNF, we can represent:

- a *variable* as a Python string
- a *literal* as a pair (a tuple), containing 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 = [[('duane', True), ('adam', True), ('jonathan', True),
          ('saman', True), ('tim', True)]]

rule2 = [[('duane', False), ('adam', False)],
         [('duane', False), ('jonathan', False)],
         [('duane', False), ('saman', False)],
         [('duane', False), ('tim', False)],
         [('adam', False), ('jonathan', False)],
         [('adam', False), ('saman', False)],
         [('adam', False), ('tim', False)],
         [('jonathan', False), ('saman', False)],
         [('jonathan', False), ('tim', False)],
         [('saman', False), ('tim', False)]]


rule3 = [[('chocolate', False), ('vanilla', False), ('pickles', False)],
         [('chocolate', True), ('vanilla', True)],
         [('chocolate', True), ('pickles', True)],
         [('vanilla', True), ('pickles', True)]]

rule4 = [[('duane', False), ('pickles', True)],
         [('duane', False), ('chocolate', False)],
         [('duane', False), ('vanilla', False)]]

rule5 = [[('jonathan', False), ('saman', True)],
         [('saman', False), ('jonathan', True)]]

rule6 = [[('adam', False), ('chocolate', True)],
         [('adam', False), ('vanilla', True)],
         [('adam', False), ('pickles', True)]]

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

When we have formulated things in this way, the list `rules` contains a formula that encodes all of the constraints we need to satisfy.

2.2.2) Examples

Consider this Boolean formula.

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

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 (which is **not** in CNF).

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

This expression looks innocuous, but translating it into CNF is actually a nontrivial exercise! It turns out, though, that this expression does have a representation in CNF as:

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

or, in our representation, as:

```
[[('a', True), ('c', True)], [('a', True), ('d', False)], [('b', True),
('c', True)], [('b', True), ('d', False)]]
```

Notice that the above expression will be true if `a` and `b` are both true, or if `c` is true and `d` is false.

Now, try your hand at it. Consider the following Boolean Formula (not in CNF):

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

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

3) SAT Solver

A classic tool that works on Boolean formulas is a **satisfiability solver** 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:

```
>>> print(satisfying_assignment(rules))
{'saman': False, 'jonathan': False, 'chocolate': False, 'adam': False,
 'duane': 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 (or `None` if no valid mapping exists).

So, we can see that, in our example above, Tim the Beaver is guilty and has a taste for vanilla and pickles!

It turns out that there are other possible answers that have Tim enjoying other flavors, but it also turns out that 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.

```
>>> print(satisfying_assignment(rules + [[('tim', False)]]))
None
```

3.1) The Naive Approach

There's one straightforward, brute-force way to solve Boolean puzzles: enumerate all possible combination of Boolean assignments for 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 that generates 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(length):
    if length == 0:
        return [[]]
    else:
        out = []
        for v in all_bools(length-1):
            out.append([True] + v)
            out.append([False] + v)
        return out
```

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. This is because there are 2^N length-N Boolean sequences, and that kind of exponential function grows quite quickly as the length N of our mappings grows.

Lots of logic puzzles can 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 this brute-force enumeration of all assignments, thanks to the power of backtracking search.

3.2) A Nicer Approach

Instead of enumerating all assignments, we will ask you to implement a more clever approach for Boolean satisfiability. One such approach is outlined below (with some of the details intentionally omitted):

We start by picking an arbitrary variable x from our formula F. We then construct a related formula F_1, which does not involve x but incorporates all the consequences of setting x to be `True`. We then try to solve F_1. If it produces a successful result, we can combine that result with information about x being `True` to produce our answer to the original problem.

If we could not solve F_1, we should try setting x to be `False` instead. If no solution exists in either of the above cases, then the formula F cannot be satisfied.

3.2.1) Updating Expressions

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)
```

In the context of the `satisfying_assignment` function in the lab, this formula would be formatted as:

```
[[('a', True), ('b', True), ('c', False)], [('c', True), ('d', True)]]
```

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

`(a or b)` or `[[('a', True), ('b', True)]]` in the lab context.

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 when `c` is `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: your `satisfying_assignment` function does not need to return assignments for these nonessential variables.

If we had instead tried setting `c` to `False`, we would update the formula instead as follows:

`d` or `[[('d', True)]]` in the lab context.

How did we get there? Note that, with `c` being `False`, the first clause is already satisfied. The second clause, though, will only be `True` if `d` is `True`.

If we then took the formula containing only `[[('d', True)]]` and set `d` to be `True`, we could update the formula to be a list of how many clauses?

What does the formula that results from setting `c` to `False` and `d` to `True` imply?

- This formula is satisfied regardless of how the other variables are set.
- This formula cannot be satisfied regardless of how the other variables are set.
- Neither of the above.

If we instead took the original formula and set both `c` and `d` to be `False`, the formula could be updated to a new formula containing only `[[]]`. What does this imply?

- ○ This formula is satisfied regardless of how the other variables are set.
- ○ This formula cannot be satisfied regardless of how the other variables are set.
- ○ Neither of the above.

It might be a good idea to implement this process (updating a formula based on a new assignment) as a helper function (which you can test independently), as we will need to perform this operation repeatedly.

3.2.2) Examples

Consider this CNF formula, in the form we use in this lab:

```
[
    [('a', True), ('b', True), ('c', True)],
    [('a', False), ('f', True)],
    [('d', False), ('e', True), ('a', True), ('g', True)],
    [('h', False), ('c', True), ('a', False), ('f', True)],
]
```

Starting from the formula above, what formula results when we set `a` to be `True`?

Which of the following is true for this formula?

--

What formula results if we instead set `a` to be `False`?

Which of the following is true for this formula?

--

What formula results if we instead set `a` to `True` *and* `f` to `True`?

Which of the following is true for this formula?

--

What formula results if we instead set `a` to `True` *and* `f` to `False`?

Which of the following is true for this formula?

--

4) Implementation

Implement the function `satisfying_assignment` as described in `lab.py`. Your function should take as input a CNF formula (in the form described throughout this writeup). It should return a dictionary mapping variable names to Boolean values if there exists such an assignment that satisfies the given formula. If no such assignment exists, it should return `None`.

5) Optimizations

> **Suggested Approach**
>
> While the details below can make a big difference in terms of efficiency for some of the test cases, each new change below adds complexity to your code. We **strongly** encourage you to write and debug the approach described above first, and only to add the features below once you have something that works.

A couple of further optimizations are likely to be necessary in order to pass all of the test cases quickly enough on the server:

- In the procedure described above, if setting the value of x immediately leads to a contradiction, we can immediately discard that possibility (rather than waiting for a later step in the recursive process to notice the contradiction).

- At the start of any call to your procedure, check if the formula contains any length-one clauses ("unit" 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` (because the unit clause alone tells us exactly what the value of `x` must be)!

  Thus, you can begin every call to your function with a loop that **repeatedly** finds unit clauses, if any, and propagates their consequences through the formula until no more unit clauses remain.

  Propagating the effects of one unit clause may reveal further unit clauses, whose later propagations may themselves reveal more unit clauses, and so on. Later assignments may also create unit clauses. For example, setting `x` to `True` would simplify a clause like `[('x', False), ('y', True)]` to `[('y', True)]`, which should force `y` to be `True`.

  This whole process should be completed before chooseing a variable to set to True or False.

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).

6) Scheduling by Reduction

Now that we have a fancy new SAT solver, let's look at applying it to a new problem!

In general, 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 satisfiability is 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 scheduling problem.

In particular, we are interested in the real-life problem of assigning students in a class to different rooms for taking a quiz. For 6.101's quiz, we assigned rooms based on last names; but we could have tried a different streategy instead, asking for room preferences.

In this scenario, each student prefers only some of the rooms, but each room has limited capacity. We want to find a schedule (assignment of students to rooms) that respects all the constraints.

Please implement the function `boolify_scheduling_problem(student_preferences, room_capacities)` as described both below and in `lab.py`:

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

Here's an example call:

```
boolify_scheduling_problem({'Alex': {'basement', 'penthouse'},
                            'Blake': {'kitchen'},
                            'Chris': {'basement', 'kitchen'},
                            'Dana': {'kitchen', 'penthouse',
'basement'}},
                           {'basement': 1,
                            'kitchen': 2,
                            'penthouse': 4})
```

In English, Alex is available for the sessions in the basement and penthouse, Blake is available only for the session in the kitchen, etc. The basement can fit 1 student, the kitchen 2 students, and the penthouse 4 students. In this case, one legal schedule would be Alex in the basement, Blake in the kitchen, Chris in the kitchen, and Dana in the penthouse.

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_room`, where `student` is the name of a student, and where `room` is the name of a room. The variable `student_room` should be `True` if and only if that student is assigned to that room (for example, the variable `Blake_kitchen` should be `True` if Blake is in the kitchen and `False` otherwise).

The CNF clauses you include should enforce exactly the following rules (which are discussed in more detail below):

1. Students are only assigned to rooms included in their preferences.
2. Each student is assigned to exactly one room.

3. No room 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 translation to CNF formulas (which `satisfying_assignment` can then solve).

6.1) Encoding the Rules

Turning these rules into CNF formulas is a tricky task. We'll try to provide some guidance for thinking about each of these rules in the following sections (though if you get stuck, please don't hesitate to ask on `6.101-help`).

Note that each of these rules can be expressed as its own CNF formula, and the AND of these three rules represents the overall formula we need to solve. As such, you may wish to write a helper function to generate the formula for each of these rules and to use those helper functions in `boolify_scheduling_problem`.

The examples below also make nice test cases to add to your lab.py or test.py (simpler than many we will test your code on)! Note also that computing the results for the examples below may also be difficult to do without some additional scratch paper (or at least a bigger text editing area than the little boxes below).

6.1.1) Students Only In Desired Sessions

For each student, we need to guarantee that they are given a room that they selected as one of their preferences. In the example above, for example, we know that Chris must be in the basement or the kitchen, and that Alex must be in the basement or in the penthouse.

In the box below, enter a CNF formula expressing this constraint (students are only assigned to rooms in their preferences) for the example data above (with Alex, Blake, Chris, and Dana). Use variable names of the form described above (e.g., `'Blake_kitchen'` represents Blake being in the kitchen).

Check Yourself:

How could you generate this formula from the arguments given to `boolify_scheduling_problem`?

6.1.2) Each Student In Exactly One Session

This rule is a little bit trickier, but it may help to separate it into two pieces:

- each student must be in at least one room, **and**
- each student must be in at most one room.

We can generate formulae for each of these conditions and combine them to construct the overall formula corresponding to this rule.

In fact, the first bullet (that each student be assigned to at least one room) is redundant with our first condition (that each student be assigned to a room in their preferences).

So let's turn our attention to making sure that each student is assigned to *at most one room*. This one is a bit trickier, particularly since we need to put things in CNF. One flip of perspective that can be helpful is that, if we need each student to be in at most one room, that means that for any pair of rooms, any given student can be in only *one* of them.

For example, one clause in this expression will say that Blake cannot be in both the kitchen and the basement. That is, we cannot have both `Blake_kitchen` and `Blake_basement` be `True` (or, to phrase it a different way, at least one of them must be `False`).

Note that there is a corresponding clause for every other pair of rooms; and each of these clauses has a corresponding clause for Alex (and the other students). **For purposes of this question, include *all* rooms for each student, regardless of their preferences.**

In the box below, enter a CNF formula expressing this constraint (students are in at most one room) for the example data above (with Alex, Blake, Chris, and Dana). Use variable names of the form described above (e.g., `'Blake_kitchen'` represents Blake being in the kitchen).

Check Yourself:

How could you generate this formula from the arguments given to `boolify_scheduling_problem`?

6.1.3) No Oversubscribed Sessions

This last rule is also fairly tricky, and it maybe requires a bit of a shift of perspective to express this constraint in CNF. However, it is similar to the previous rule in some ways.

We can think about this as: if a given room can contain N students, then in every possible group of $N+1$ students, there must be at least one student who is *not* in the given room. For example, since the kitchen holds 2 people, we would need to consider all possible groups of 3 students and make sure that at least one of the students in each group is *not* in that room.

What about the penthouse? It has enough room for everyone, so there is no need even to include it in the constraints (it doesn't constrain our decision in any way)!

For purposes of this question, include *all* groups of students, regardless of their preferences.

In the box below, enter a CNF formula expressing this constraint (no oversubscribed rooms) for the example data above (with Alex, Blake, Chris, and Dana). Use variable names of the form described above (e.g., `'Blake_kitchen'` represents Blake being in the kitchen).

Check Yourself:

How could you generate this formula from the arguments given to `boolify_scheduling_problem`?

7) UI

We have provided a browser UI for this lab, so you can see your code solving scheduling problems in action! When you have implemented both main functions of the lab, you may run `python3 server.py` and then use your web browser to go to `http://localhost:6101/`. Running a test case on the UI involves generating a CNF formula and searching for a satisfying assignment for it, done by calling your functions. Enjoy!

8) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a sat /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

> When this page was loaded, you had not yet made any submissions.
> If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.