Understanding SAT by Implementing a Simple SAT Solver in Python

MAY 13, 2014

Introduction

SAT is short for "satisfiability". Chances are you have heard of it or one of its variants like 3-SAT in passing, especially in discussions of complexity and NP-completeness. In this post, we will go into details of what it is all about, why it is of such importance from both a theoretical and practical perspective, and how to approach solving it by developing a simple Python SAT solver. By the end of this post, we will have a working SAT solver with a command-line interface. The code for it is on GitHub: https://github.com/sahands/simple-sat. Feel free to fork and contribute improvements. Of course, our implementation will not be anywhere close to more complicated SAT solvers implemented in C or C++, such as miniSAT. The focus here is on simplicity since the code is to be an introduction to SAT and SAT solvers.

Sections marked with * are more theoretical and not required for understanding the algorithm we will use. On the other hand, the rest of the introduction section below can be skipped if you already know the problem definition and relevant technical terms.

Non-Technical Definitions & Example

Before we start with the definitions, you might be asking why SAT is written in all capitals if it is not an acronym. Well, great question. SAT happens to fall under what are called *decision problems* in computer science. What that means is that the answer to a particular instance of the problem is either "yes" or "no". Decision problems are often simply identified with the set of inputs for which the answer is "yes", and that set is given a capitalized name. For example, SAT is the set of all satisfiable CNF expressions, and PRIMES is the set of all prime numbers (the decision problem in the latter is that of primality; i.e. given the binary representation of number n, decide if it is a prime or not). To go on a bit of a tangent, this is also the reason that the title of the paper that introduced the AKS primality test ("PRIMES is in P") is not a silly grammar mistake; PRIMES is a set and the paper shows that it is in P, which is the set of decision problems solvable in polynomial-time. This naming style, as far as I know, is mainly due to Garey and Johnson's classic textbook on complexity theory.

So, back to SAT. So far we mentioned that SAT is a decision problem, and something about mysterious sounding "CNF expressions". Now, if you happen to know your Boolean logic and already know all about satisfiability and CNF expressions, then feel free to skip ahead to next section. The rest of this section assumes no prior knowledge of logic. Like many other interesting problems, there are a variety of ways of describing SAT, some more technical and some less. Here I will provide a very non-technical description of the problem that nonetheless is an accurate description.

Assume you are in charge of elections in a society. Elections in this society work as follows: there are n candidates, and any number of them, from 0 (nobody) to n (everybody) can be elected as the result of the elections. Each voter provides a list of candidates they want elected and candidates they want not elected. For example, if we call the candidates A, B and C, then one vote might be "A, B, not C". We say a voter will be *satisfied* with the results of the election if at least one of his/her preferences is met. For example, the voter with the "A, B, not C" vote will be satisfied if either A or B is elected, or if C is not elected. To be clear, that voter will be happy even if nobody is elected (anarchy!) because one of the preferences is "not C" which is met if we do not pick anyone. It's also possible to receive an empty vote. We take this to mean that the voter will not be satisfied regardless of who is elected.

You are given all the votes, and your job is to determine if all the voters can be satisfied or not, and if yes, provide at least one possible pick of candidates that would satisfy everybody.

We assume that each candidate is represented by a unique identifier that will be a string in the input. For the votes, we will write just the string representing candidate x to indicate the voter wants the candidate elected, and $\sim x$ to indicate the voter wants x not elected.

Let's look at an example. Assume the list of votes is given as follows, one per line:

```
 A    B   ~C
 B    C
~B
~A    C
```

Then choosing to elect just candidates A and C but not B will satisfy all the voters. Take a moment to convince yourself that no other choice of candidates (there are a total of $2^3=8$ possibilities) can satisfy everyone. It is easy to see that in general the search-space is of size 2^n where n is the number of candidates.

Technical Terminology

Now that the problem makes sense, let's define the technical vocabulary. First, what we called "candidates" are called *variables*. The variables in the above example are A , B and C . A variable can be assigned true or false. A *literal* is a variable or its negation. For

example A and $\sim A$ are literals. Literals without the $\sim$ are called positive, pure, or unnegated literals. Literals with $\sim$ are called negated literals. A set of literals is called a *clause*. An *assignment* is a mapping of variables to true or false. For example, the assignment that satisfied the clauses in the previous example was given by $A=true$, $B=false$ and $C=true$. A clause is *satisfied* by an assignment if at least one of its unnegated literals is assigned true by the assignment, or one of its negated literals is assigned false in the assignment. It is customary, in logic notation, to separate the literals in a clause using the $\vee$ symbol, read "or". For example, the first clause above is written as $A \vee B \vee \sim C$ in mathematical notation.

So SAT can be summarized as follows: given a list of clauses, determine if there exists an assignment that satisfies all of them simultaneously.

It is also worthy of mention that there is a variation of SAT called 3-SAT with the restriction that each clause consists of at most 3 (distinct) literals. It can be shown with relative ease that SAT is in fact reducible to 3-SAT.

A Simple SAT Solver In Python

Even though SAT is NP-complete and therefore no known polynomial-time algorithm for it is (yet) known, many improvements over the basic backtracking algorithms have been made over the last few decades. However, here we will look at one of the most basic yet relatively efficient algorithms for solving SAT. The encoding and the algorithm are based on Knuth's SAT0W program which you can download from his programs page.

The algorithm is a watch-list based backtracking algorithm. What makes the watch-list based algorithms particularly simple, as we will see, is that very little (practically nothing) needs to be done to "undo" steps taken when we need to backtrack.

Parsing & Encoding The Input

Before we can approach solving a SAT instance, we need to be able to represent the instance in memory. Let's remember that a SAT instance is a set of clauses, and each clause is a set of literals. Finally, a literal is a variable that is either negated or not. Of course, we can just store the instance as a list of clauses, with each clause being a list of strings that are the literals. The problem with this approach is that we will not be able to quickly look up variables, and checking to see if a literal is negated or not, and negating it if not, would be rather slow string operations.

Instead, we will first assign a unique number, starting from 0 and counting up, to each variable as we encounter them, using a dictionary to keep track of the mapping. So variables will be encoded as numbers 0 to $n-1$ where n is the number of variables. Then for an unnegated literal with variable encoded as number x we will encode the literal as $2x$, and the negated one will be $2x+1$. Then a clause will simply be a list of numbers that are the encoded literals, and

Let's look at an example first. For this, let's see how the code that we will look at in a minute behaves:

```
>>> from satinstance import SATInstance
>>> s = SATInstance()
>>> s.parse_and_add_clause('A B ~C')
>>> s.variables
['A', 'B', 'C']
>>> s.variable_table
{'A': 0, 'C': 2, 'B': 1}
>>> s.clauses
[(0, 2, 5)]
```

So as you see, the clause $A \vee B \vee \sim C$ is encoded as the tuple `(0, 2, 5)` since variable A is assigned number 0, and hence literal A is $2 \cdot 0 = 0$. On the other hand, $\sim C$ is encoded as 5 since C is assigned 2 and hence $\sim C$ is encoded as $2 \cdot 2 + 1 = 5$.

Why the funny encoding, you ask? Because it has a few advantages:

- we can keep track of variables by keeping a list of length n, and of literals by keeping a list of length $2n$,
- checking to see if a literal is negated or not is simple: just do a bit-wise AND with 1, that is `x & 1 == 0`,
- looking up the variable in a literal is a matter of dividing by two, which is the same as a bit-wise shift to the right, that is `v = x >> 1`,
- switching a literal from negated to unnegated and back can be done by doing a bit-wise XOR with the number one, that is `negate(x) = x ^ 1`,
- and finally going from a variable to a literal can be done by doing a bit-wise shift to the right (and a bit-wise OR with 1 if negated), that is `x = v << 1` or `x = v << 1 | 1`.

Notice that all of the above can be done using bit-wise operations which are generally very fast to do. And since these operations will be happening an exponential number of times, we will take any performance boost we can get.

With this, we are ready to write the code that takes care of reading an input file and encoding the clauses. Here it is:

```
class SATInstance(object):
    def parse_and_add_clause(self, line):
        clause = []
        for literal in line.split():
            negated = 1 if literal.startswith('~') else 0
            variable = literal[negated:]
            if variable not in self.variable_table:
                self.variable_table[variable] = len(self.variables)
                self.variables.append(variable)
            encoded_literal = self.variable_table[variable] << 1 | negated
            clause.append(encoded_literal)
        self.clauses.append(tuple(set(clause)))

    def __init__(self):
        self.variables = []
        self.variable_table = dict()
        self.clauses = []

    @classmethod
    def from_file(cls, file):
        instance = cls()
        for line in file:
            line = line.strip()
            if len(line) > 0 and not line.startswith('#'):
                instance.parse_and_add_clause(line)
        return instance

    def literal_to_string(self, literal):
        s = '~' if literal & 1 else ''
        return s + self.variables[literal >> 1]

    def clause_to_string(self, clause):
        return ' '.join(self.literal_to_string(l) for l in clause)

    def assignment_to_string(self, assignment, brief=False, starting_with=''):
        literals = []
        for a, v in ((a, v) for a, v in zip(assignment, self.variables)
                     if v.startswith(starting_with)):
            if a == 0 and not brief:
                literals.append('~' + v)
            elif a:
                literals.append(v)
        return ' '.join(literals)
```

As you can see, we also include methods here to decode variables, literals, clauses, and assignments. These are used for outputting logging messages as well as the final solutions.

Keeping Track Of The Assignment

Our algorithm will be a backtracking algorithm, in which we will assign true or false to all the variables, starting from variable 0 and going in order to variable $n-1$. Of course, the basic search space is of size 2^n but by pruning, we will not explore the whole space (usually anyway). The assignment will be kept as a list of length n , with item at index i being None if neither true or false has been assigned variable i , and 0 (false) or 1 (true) otherwise, depending on the assignment. When we backtrack, we set the corresponding item in the assignment list back to None to indicate it is no longer assigned.

Watch-lists

Now that we have the encoding in place, and know how to keep track of the assignment, let's look at the key idea of our algorithm. For each clause to be satisfied, it needs to have at least one of its literals satisfied. As such, we can make each clause *watch* one of its literals, and ensure that the following invariant is maintained throughout our algorithm:

Invariant

All watched literals are either not assigned yet, or they have been assigned true.

We then proceed to assign true or false to variables, starting from 0 to $n-1$. If we successfully assign true or false to every variable while maintaining the above variant, then we have an assignment that satisfies every clause.

To maintain this invariant, any time we assign true or false to a variable, we ensure to update the watch-list accordingly. To do this efficiently, we need to keep a list of clauses that are currently watching a given literal. This is done in the code below using a list of length $2n$ of double-ended queue (`collections.deque`), with each clause initially watching the first literal in it. The function below takes care of this setting up of the watch-list:

```
def setup_watchlist(instance):
    watchlist = [deque() for __ in range(2 * len(instance.variables))]
    for clause in instance.clauses:
        # Make the clause watch its first literal
        watchlist[clause[0]].append(clause)
    return watchlist
```

Why double-ended queues instead of just a list? Short answer is that after experimenting, I found out that double-ended queues provided the best performance.

Back to the algorithm, whenever we assign true to a variable x we must make clauses watching $\sim x$ watch something else. And similarly, whenever we assign false to a variable x we make clauses watching x watch something else. If we can not make a clause watch something, which happens when all the other literals in a clause have already been assigned false, then we know that the current assignment contradicts the clause, and we stop and backtrack. We only need one clause to be contradicted to know not to go any

further. As such, the heart of our algorithm will be where we update the watch-list after an assignment has been made. The Python function below, which is in (`watchlist.py`), implements this part of the algorithm:

```
def update_watchlist(instance,
                     watchlist,
                     false_literal,
                     assignment,
                     verbose):
    """
    Updates the watch list after literal 'false_literal' was just assigned
    False, by making any clause watching false_literal watch something else.
    Returns False it is impossible to do so, meaning a clause is contradicted
    by the current assignment.
    """
    while watchlist[false_literal]:
        clause = watchlist[false_literal][0]
        found_alternative = False
        for alternative in clause:
            v = alternative >> 1
            a = alternative & 1
            if assignment[v] is None or assignment[v] == a ^ 1:
                found_alternative = True
                del watchlist[false_literal][0]
                watchlist[alternative].append(clause)
                break

        if not found_alternative:
            if verbose:
                dump_watchlist(instance, watchlist)
                print('Current assignment: {}'.format(
                      instance.assignment_to_string(assignment)),
                      file=stderr)
                print('Clause {} contradicted.'.format(
                      instance.clause_to_string(clause)),
                      file=stderr)
            return False
    return True
```

So why the watch-list based approach? The main reason is the simplicity it affords us. Since during a backtracking step, assignments only go from 0 or 1 to None, the watch-list does not need to be updated at all to maintain the invariant. This means the backtracking step will simply be changing the assignment of a variable back to None and that's it.

Putting It All Together

We are now ready to put it all together to get a simple recursive algorithm for solving SAT. The steps are simple: try assigning 0 to variable d , update the watch-list, if successful, move on to variable $d+1$. If not successful, try assigning 1 to variable d and update the

watch-list and continue to variable $d+1$. If neither succeed, assign None to variable d and backtrack. Here is the code:

```
def solve(instance, watchlist, assignment, d, verbose):
    """
    Recursively solve SAT by assigning to variables d, d+1, ..., n-1. Assumes
    variables 0, ..., d-1 are assigned so far. A generator for all the
    satisfying assignments is returned.
    """
    if d == len(instance.variables):
        yield assignment
        return

    for a in [0, 1]:
        if verbose:
            print('Trying {} = {}'.format(instance.variables[d], a),
                  file=stderr)
        assignment[d] = a
        if update_watchlist(instance,
                            watchlist,
                            (d << 1) | a,
                            assignment,
                            verbose):
            for a in solve(instance, watchlist, assignment, d + 1, verbose):
                yield a

    assignment[d] = None
```

Making It Iterative *

For fun, let's see if we can implement the above algorithm without recursion. This is in fact how Knuth implements the algorithm. (He seems to dislike recursion, see for example this story on Quora.)

The basic idea here is to manually keep track of the current state of the backtrack tree. When we use recursion, the state is kept implicitly using the stack and which instruction is executing in each of the function calls. In the iterative case, we will store the state using d which is the current depth of the backtrack tree we are currently in, and also the variable we are to assign to currently, and the `state` list which keeps track of which assignments for each variable have been tried so far. Here is the code:

```
def solve(instance, watchlist, assignment, d, verbose):
    """
    Iteratively solve SAT by assigning to variables d, d+1, ..., n-1. Assumes
    variables 0, ..., d-1 are assigned so far. A generator for all the
    satisfying assignments is returned.
    """

    # The state list wil keep track of what values for which variables
    # we have tried so far. A value of 0 means nothing has been tried yet,
    # a value of 1 means False has been tried but not True, 2 means True but
    # not False, and 3 means both have been tried.
    n = len(instance.variables)
    state = [0] * n

    while True:
        if d == n:
            yield assignment
            d -= 1
            continue
        # Let's try assigning a value to v. Here would be the place to insert
        # heuristics of which value to try first.
        tried_something = False
        for a in [0, 1]:
            if (state[d] >> a) & 1 == 0:
                if verbose:
                    print('Trying {} = {}'.format(instance.variables[d], a),
                          file=stderr)
                tried_something = True
                # Set the bit indicating a has been tried for d
                state[d] |= 1 << a
                assignment[d] = a
                if not update_watchlist(instance, watchlist,
                                        d << 1 | a,
                                        assignment,
                                        verbose):
                    assignment[d] = None
                else:
                    d += 1
                    break

        if not tried_something:
            if d == 0:
                # Can't backtrack further. No solutions.
                return
            else:
                # Backtrack
                state[d] = 0
                assignment[d] = None
                d -= 1
```

Theoretical and Practical Significance *

All right, so SAT is a cool problem, sure; possibly even useful. But why is it given so much importance? The short answer is that many other problems, often "difficult" problems, can be reduced to SAT. Let's consider an example first, and then look at Stephen Cook's result that established SAT as the first NP-complete problem, to get a sense of both practical applications of SAT, and its theoretical importance.

Four Colouring *

You might have heard of the "four colour theorem". In simplest terms, it states that the regions in any map can be coloured using at most four colours such that no two neighbouring regions are coloured the same. See the Wikipedia page on it for more details.

This lends itself to a simple decision problem: given a map, is it possible to colour it using 4 or less colours such that no two neighbouring regions are the same colour? The four colour theorem is then true if and only if the answer to this decision problem is always true (provided the input map meets the requirements of a planar graph, a detail we are not too concerned with here). As input, we will take the number of regions n, and assume the regions are labelled using numbers 1 to n, and a list of neighbouring regions of the form $\{i,j\}$ with $i \neq j$, indicating regions i and j are neighbours. Let us use colours red (R), blue (B), green (G), and yellow (Y) to colour the regions. Our variables are going to be R_i, B_i, G_i and Y_i, for $1 \leq i \leq n$, indicating that region i is coloured red, blue, green, or yellow, respectively.

Next, we need to construct the right set of clauses such that if all of them are satisfied, then we have a proper colouring of the map. Specifically, we need every region to be coloured, and we need no two neighbouring regions to be the same colour. First, let us construct the clauses that will make sure every region has one and only one colour assigned to it. For this, we need to make sure only one of R_i, B_i, G_i or Y_i is picked for our assignment at a time. We can express this in terms of K clauses for each region i. First, we add $R_i \vee B_i \vee G_i \vee Y_i$ as a clause, which ensures that region i gets at least one colour assigned to it. Then for pair of colours, say R and B, we add the clause $\sim R_i \vee \sim B_i$ which basically says "not both of R_i and B_i can be picked at the same time", effectively making sure that exactly one colour is assigned to each region. Finally, for any two neighbouring regions, say i and j, and each colour, say R, we add the clause $\sim R_i \vee \sim R_j$ which says not both of i and j can be coloured red.

Let's look at a very simple example. Suppose our map has only two regions, regions 1 and 2 and that they are neighbours. Then our SAT input would be:

```
# Assign at least one colour to region 1
R1 B1 G1 Y1

# But no more than one colour
~R1 ~B1
~R1 ~G1
~R1 ~Y1
~B1 ~G1
~B1 ~Y1
~G1 ~Y1

# Similarly for region 2
R2 B2 G2 Y2
~R2 ~B2
~R2 ~G2
~R2 ~Y2
~B2 ~G2
~B2 ~Y2
~G2 ~Y2

# Make sure regions 1 and 2 are not coloured the same since they are neighbours
~R1 ~R2
~B1 ~B2
~G1 ~G2
~Y1 ~Y2
```

Running this through our SAT solver gives:

```
$ python sat.py --brief --all < tests/colouring/01.in
Y1 G2
Y1 B2
Y1 R2
G1 Y2
G1 B2
G1 R2
B1 Y2
B1 G2
B1 R2
R1 Y2
R1 G2
R1 B2
```

As you can see, there are many possible solutions, since in such a simple case we have a valid colouring as long as we assign a different colour to each region, which can be done in $4 \cdot 3 = 12$ ways, corresponding precisely to the 12 solutions given by our SAT solver.

In the next section, we see that a much broader set of problems can be reduced to SAT.

In general, the decision problem of the above example is known as graph colouring, or GT4 in Garey-Johnson's naming, where given a graph and a number k the decision problem is to determine if a k -colouring for the graph exists. In the above, we had $k=4$. In this more general definition, with n regions, our reduction to SAT involves introducing $k \cdot n$ variables and

$$1+n \cdot \binom{k}{2}+k \cdot e$$

clauses, where e is the number of edges. Since $e=O(n^2)$ (in fact, $e=O(n)$ for planar graphs), the number of variables and clauses in our construction above are polynomials in n and k . Hence we have a polynomial-time reduction to SAT. The significance of this is discussed further in the next section.

NP-Completeness Of SAT *

In previous section we saw how a problem regarding colouring of regions in a map can be reduced to SAT. This can be further generalized to much larger class of problems: any decision problem that can be decided in polynomial time using a non-deterministic Turing machine can be reduced in polynomial time to SAT. This was first proved in Stephen Cook's paper "The Complexity of Theorem-Proving Procedures", which is the paper that introduced the famous P = NP question as well. Let's go over the basic idea in the paper very briefly here. If you are interested in more details, make sure you have a look at the paper, as it is rather short and a pleasure to read.

But before we go into detail, let us take a moment to discuss why it is of such importance. First, nobody has yet come up with an efficient (polynomial time) algorithm to solve SAT in its generality. (SAT with some restrictions, e.g. 2-SAT, can be solved efficiently though.) Showing that a problem can be reduced to SAT means that if we find an efficient algorithm for SAT then we have found an efficient algorithm for that problem as well. For example, if we find a polynomial-time algorithm for SAT then we immediately have a polynomial-time algorithm for the graph colouring problem given above.

Now, the class of decision problems that can be solved in polynomial-time using a non-deterministic Turing machine is known as NP (which stands for Non-deterministic Polynomial). This is a very large class of problems, since Turing machines are one of the most general computational models we have, and even though we are limited to polynomial-time Turing machines, the fact that the Turing machine does not have to be deterministic allows us much more freedom. Some examples of problems that are in NP are:

- all problems in P, e.g. determining if a number is prime or not (PRIMES), and decision versions of shortest path, network flow, etc.,
- integer factorization,

- graph colouring,
- SAT,
- and all NP-complete problems (see here for a rather large list of examples).

A problem is said to be NP-complete if it, in addition to being in NP, also has the property that any other problem in NP can be reduced to it in polynomial-time. Cook's paper proved SAT to be NP-complete. In fact, since that paper introduced the concept of NP-completeness, SAT was the first problem to be proved NP-complete. Since then, many other problems have been shown to be NP-complete, often by showing that SAT (or 3-SAT) can be reduced in polynomial-time to those problems (converse of what we proved earlier for graph colouring).

Now, as promised, let's briefly look at why SAT is NP-complete. For this, we need to know more precisely what a Turing machine is. Unfortunately, this would involve a bit more detail than I want to include in this section. So instead, I am going to show that if a problem can be solved using a finite-state machine (FSM) then it be reduced in polynomial-time to SAT. The case for Turing machines, which are a generalizations of finite-state machines (Turing machines are basically FSM's with the addition of a tape that they can read from and write to), is quite similar, just more complicated. I encourage you to read Cook's original paper for details of the proof with Turing machines.

First, let's define what an FSM is. In simplest terms, an FSM is a program that has a finite number of states, and that when fed an input character, moves to another state (or possibly stays in the same state) based on a fixed set of rules. Also, some states are taken as "accepting" states. Given an input string, we feed the string character by character into the FSM, and if at the end the FSM is in an accepting state, the answer to our decision problem is yes. If not, the answer is no.

The below code shows how an FSM could can be implemented in Python. Note that in this implementation, we are forced to have a *deterministic* FSM. Let's ignore this detail for now though. This particular example implements an FSM that accepts input strings that contain an even number of ones.

```
from __future__ import print_function


def even_ones(s):
    # Two states:
    # - 0 (even number of ones seen so far)
    # - 1 (odd number of ones seen so far)
    rules = {(0, '0'): 0,
             (0, '1'): 1,
             (1, '0'): 1,
             (1, '1'): 0}
    # There are 0 (which is an even number) ones in the empty
    # string so we start with state = 0.
    state = 0
    for c in s:
        state = rules[state, c]
    return state == 0


# Example usage:
s = "001100110"
print('Output for {} = {}'.format(s, even_ones(s)))
```

So the core of an FSM is a list of rules of the form $(S,c) \to T$ which says if the FSM is in state S and receives input character c then it goes to state T . If for any unique pair of (S,c) there is only one rule $(S,c) \to T$ then the FSM is said to be deterministic. This is because the FSM will never need to make a "choice" as to which of the rules to apply. With non-deterministic FSM's, the definition of acceptance needs to be modified a bit: if *any* set of choices of rules would get us to an accepting state given an input then the input is said to be accepted. It is a well-established result in Automata theory that deterministic and non-deterministic FSM's are computationally equally powerful, because any non-deterministic FSM can be translated to an equivalent deterministic one by the "powerset construction" method. The equivalent deterministic FSM might have an exponentially larger number of states compared to the non-deterministic one, however.

It is also well-known that FSM's can solve a class of problems known as "regular" problems. What this means, in very simple terms, is that if you can write a regular expression that would accept the "yes" instances of your decision problem, then you can solve the problem using an FSM. In fact, regular expressions are often implemented using FSM-like structures. The "compile" phase of using regular expression is precisely when the regular expression engine builds the FSM-like structure from your regular expression. (Exercise: Find a regular expression that accepts the above language, namely binary strings with an even number of ones.)

All right, so let's say a decision problem can be solved using an FSM with states numbered 1 to n . For simplicity, let's assume that our input will be binary (character set is $\{0,1\}$).

Suppose the FSM has k rules given by $(S_i, c_i) \rightarrow T_i$, for $1 \leq i \leq k$. And assume the input characters are given by s_1 to s_m. So our input is of length m. Finally, assume that the initial state is 1 and accepting states are a_1 to a_q, where q is the number of accepting states.

Following Cook's footsteps, we will introduce the following variables for our SAT reduction:

- P_t which is true iff $s_t = 1$,
- and Q_t^i which is true iff the FSM is in state i after input character s_t has been fed into the FSM, for $1 \leq i < j \leq n$ and $0 \leq t \leq m$. We will take $t = 0$ to be the starting step, before anything has been fed into the FSM.

With these definitions, we proceed to translate the question of whether the input is accepted by the FSM into an instance of SAT. The goal is to produce a set of clauses that are satisfiable iff the FSM ends in an accepting state given the particular input. The clauses that will accomplish this are:

- P_t for $0 \leq t \leq m$ such that $s_t = 1$ and $\sim P_t$ for all other $1 \leq t \leq m$. These will be the first m clauses, each consisting of a single literal.
- $Q_m^{a_j}$ for $1 \leq j \leq q$. This says that after the last character is fed into the FSM, we want to be in one of the accepting states.
- $\sim Q_t^i \vee \sim Q_t^j$ for any $1 \leq i < j \leq n$ and $1 \leq t \leq m$, which effectively says that the FSM can not be in both states i and j at step t. Collectively, these clauses will ensure that the FSM is not in more than one state at a time.
- $Q_t^1 \vee \ldots \vee Q_t^n$ for $1 \leq t \leq m$. This says that the FSM needs to be in at least one state at any step. Together with the last set of clauses, we ensure that the FSM is in exactly one state at any step.
- $\sim Q_{t-1}^{S_i} \vee P_t \vee Q_t^{T_i}$ for all rules $(S_i, 0) \rightarrow T_i$ and $\sim Q_{t-1}^{S_i} \vee \sim P_t \vee Q_t^{T_i}$ for all rules $(S_i, 1) \rightarrow T_i$, for $1 \leq t \leq m$. These clauses are logically equivalent to "$Q_{t-1}^{S_i}$ and P_t implies $Q_t^{T_i}$ " which is equivalent to $(S_i, 1) \rightarrow T_i$. In other words, they ensure proper transition between states based on the input.
- Finally, we want to start in the initial state so we add the clause Q_0^1.

Let's see this in action for the above FSM which accepts strings with an even number of ones in them. First, we have two states, so $n = 2$. Let's build the SAT instance to handle inputs of length t. Also note that we can leave out the first set of clauses (the P_t and $\sim P_t$ ones), in which case any SAT assignment will give us some accepted input. Which means we can list all the strings accepted by the FSM by looking at all the satisfying assignments of the above set of clauses.

Here is an example for $t = 3$. In this input Q_t^i is written as *Qi-t*. The states are also labelled 0 and 1 instead of 1 to n in the above.

```
# No more than one state at each step
~Q0-0 ~Q1-0
~Q0-1 ~Q1-1
~Q0-2 ~Q1-2
~Q0-3 ~Q1-3

# At least one state in each step
Q0-0 Q1-0
Q0-1 Q1-1
Q0-2 Q1-2
Q0-3 Q1-3

# Add the rules
# (EVEN, 1) -> ODD
~Q0-0  ~P1   Q1-1
~Q0-1  ~P2   Q1-2
~Q0-2  ~P3   Q1-3

# (ODD, 1) -> EVEN
~Q1-0  ~P1   Q0-1
~Q1-1  ~P2   Q0-2
~Q1-2  ~P3   Q0-3

# (EVEN, 0) -> EVEN
~Q0-0   P1   Q0-1
~Q0-1   P2   Q0-2
~Q0-2   P3   Q0-3

# (ODD, 0) -> ODD
~Q1-0   P1   Q1-1
~Q1-1   P2   Q1-2
~Q1-2   P3   Q1-3

# Start in state 0
Q0-0
# End in an accepting state
Q0-3
```

Let's see the output of running a SAT solver on this, and another file for $t{=}3$ and $t{=}4$:

```
$ python sat.py --all --starting_with P --brief < tests/fsm/even-ones-3.in

P1 P3
P1 P2
P2 P3
$ python sat.py --all --starting_with P --brief < tests/fsm/even-ones-4.in

P1 P4
P1 P3
P1 P2 P3 P4
P1 P2
P2 P4
P2 P3
P3 P4
```

As expected, all the possible ways of picking a subset of $\{P1, P2, P3\}$ with an even number of elements in them are listed above, and similarly for $\{P1, P2, P3, P4\}$, although not necessarily in any meaningful order. (Notice that the empty lines are the empty subsets, which also have even numbers of ones.)

Further Reading

Knuth's pre-fascicle of section 7.2.2.2 is available as a gzipped postscript file which goes into much more detail on the subject. Also check out SAT Live which has lots of resources on the subject SAT solvers.

Comments