Project 1 - 3SAT generator

Due June 24, 2016

1 Background

SAT stands for SATisfiability for a given logical expression. That is given a logical formula such as $x \wedge y$ are there logical values (TRUE or FALSE) that can be assigned to x and y to make the whole expression true?

In this case it is clearly the case, simply make x =TRUE and y =TRUE, but for more complicated expressions it is not always obvious:

$$((\neg x_1 \wedge x_2) \vee x_3) \wedge (\neg x_2 \wedge x_3) \wedge (\neg x_3 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee (\neg x_2 \wedge x_3))$$

The goal of a SAT solver is to take an expression as above and determine if some values for x_i make the whole expression TRUE. Allowing arbitrary logical expression is very annoying, because of the mixed $\wedge$'s and $\vee$'s. To fix this there is a standard set. It can be shown that any logical expression can be written in Conjunctive Normal Form, this process may involve adding extra "dummy" values, but will always have the same truth value.

Conjunctive Normal Form (CNF) requires that the expression be separated into clauses, where each variable is $\vee$'d together and every clause is $\wedge$'d together. This is complicated to explain but easy to understand through examples. The above equation is NOT in CNF however the following is, since it is a collection of clauses (in which variables are only $\vee$'d together) and each clause is $\wedge$'d together:

$$(\neg x_1 \vee x_3 \vee x_4 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4)$$

SAT is a very difficult problem to solve very quickly (in general) it is part of a class of problems known as NP-Complete.

We will be working with 3SAT, which is a "simplification" – technically it is the same problem, but that is above the topics of this course. The "simplification" for 3SAT is that each clause must have EXACTLY 3 variables. So far in this handout we have NOT seen any 3SAT problems. An example of one is:

$$(\neg x_5 \vee \neg x_2 \vee x_1) \wedge (x_2 \vee x_5 \vee \neg x_6) \wedge (\neg x_8 \vee x_1 \vee x_3) \wedge (x_9 \vee \neg x_7 \vee x_4)$$

You'll notice it is perfectly valid to reuse variables between clauses, in fact if you didn't then this would be very easy to solve.

2 Project

For this project you will be writing a 3SAT generator, that can also test given truth values. That is the goal of this program is to create a general 3SAT problem with at most 10 variables and 20 clauses. Your program will then print this logical expression so the user can see it (you will use $\sim$ as opposed to $\neg$) and then ask the user for truth values for each of the variables. Then it will test whether those truth values make the whole expression true or not. If they do make the expression true, you should inform the user. If they do not, you again inform the user AND print the clause which is invalid.

3 Expectations

Each run of the program should *randomly* choose the number of possible variables (between 3 and 10 variables is allowed) it should also *randomly* choose the number of clauses (between 1 and 20 is allowed). Within a clause, a variable should not be repeated, ever. That is $(x_1 \vee x_1 \vee x_2)$ is not a valid clause, neither is $(x_2 \vee \neg x_2 \vee x_3)$. But of course variables can be repeated between other clauses.

After choosing the number of variables and clauses, each clause should be generated *randomly*. You may represent this generation anyway that you like. The standard way to represent a clause is to use a `list` (in Python) or an `int[]` in `C++`. For example: $(x_1 \vee x_2 \vee x_3)$ can be represented as `[1,2,3]`, and $(\neg x_2 \vee x_1 \vee \neg x_5)$ can be represented as `[-2,1,-5]`, that is negative numbers represent a negated variable.

One way to randomly generate clauses is for each clause randomly choose a number between 1 and the number of variables. This selects your x_k, then you should also randomly decide whether or not to negate this variable, you can do this as a 50/50 chance.

After *randomly* generating all of the clauses, your program should print this clause to the screen. Since you cannot directly print $\neg$ you will use $\sim$ as an alternative. Similarly subscripts are not possible, so you will just place numbers next to an x. You will print something like this:

$$(\sim x5 \vee \sim x2 \vee x1) \wedge (x2 \vee x5 \vee \sim x6) \wedge (\sim x8 \vee x1 \vee x3) \wedge (x9 \vee \sim x7 \vee x4)$$

That is you MUST print parentheses and $\vee$ and $\wedge$, use v and ˆ for this (it will, ironically, look better printed by `C++` or Python).

After printing the expression you will ask the user for T/F values for each variable. You MUST ensure input is correct! If I enter: s, (instead of T/F) then you should ask again until I successfully choose T/F. However you should allow upper OR lower case, that is valid inputs are T,t,F,f. Any other inputs are to be ignored and you should re-ask.

After asking for user input for EVERY variable, you should evaluate whether the values that were given do or do not satisfy the generated expression. Then you should inform the user whether they were correct or not, you may use any message you like to inform the user. But if they were incorrect you MUST

also print the clause that is evaluated to FALSE (and ONLY that clause). It is possible that many clauses fail, you should print the FIRST clause which fails.

4   Deliverables

You need to turn your program into Moodle. You may structure your program however you desire. Personally I would write this as either 1 Python file or 3 `C++` files (one `.h` file, one `.cpp` file which contains the implementations of my functions, and one `.cpp` file which contains my `main` method), but as stated do as you please. Moodle will allow up to 4 file submissions, if for some reason you feel that you need more, please talk to me first, you may be doing too much work. You may use any well known language to write this.

Personally, I found this to be easier to write in Python than `C++`, but both were very doable. Before beginning to write code you should decide on a language (obviously) and write down a basic structure. You will find it much easier to write the code if you write the general structure first. Decide what functions you want, what you want them to do, then write the code.

5   Examples

The following are examples of my versions of this program running, so that you can see how the output should be, and how I prompt the user and force correct input.


```
> ./Project1.py
(~x6 v ~x4 v ~x1) ^ (~x1 v x2 v x7) ^ (~x2 v ~x3 v ~x1) ^ (~x1 v ~x3 v x2) ^ (~x4 v x2 v x7)
Please enter a value (T/F) for variable x1: a
Please enter a value (T/F) for variable x1: v
Please enter a value (T/F) for variable x1: d
Please enter a value (T/F) for variable x1: F
Please enter a value (T/F) for variable x2: t
Please enter a value (T/F) for variable x3: F
Please enter a value (T/F) for variable x4: T
Please enter a value (T/F) for variable x5: T
Please enter a value (T/F) for variable x6: F
Please enter a value (T/F) for variable x7: T
This assignment is correct!
```

Figure 1: First Example

This example is run with my Python solution, I have made my code executable, you *should* do this, but it is not required, if you aren't sure how to do this, you should ask, this is something that you should learn.

In this example we can see I repeatedly ask for input for x_1 since the initial inputs are incorrect. Also worthy of note is that I randomly chose to use 7 variables but x_5 does not appear in the randomly generated expression. This is perfectly fine. This will happen with some non-zero probability given based on the number of clauses there will be. However you can see that while x_5 does not appear in the expression, I still ask for a truth value for it. Again this is

fine. If you wish to be more clever and only ask for values that appear in the expression, feel free, but you need not.

Figure 2: Second Example

This example is my `C++` solution. The picture is smaller (sorry), due to the very long expression (many clauses). This is fine you do not need to add extra newlines to make the output look better, you can but you don't need to. If you do add artificial newlines the output should look pretty, that is don't put a newline in the middle of a clause, do it in between clauses.

In this example we can see that even in the middle variables I (the user) type something wrong and I (the programmer) still ask for them to correct it.

Figure 3: Third Example

Another example of the `C++` solution.

6 Final thoughts

Start this project immediately. This project is not conceptually difficult (I hope!) but there are a lot of details involved. You should start immediately to run into any issue as soon as possible. If you do run into issues you should be well aware of the help you can receive. You may work with other students as much as you would like. HOWEVER you must write your own code. You may talk with others about the structure of the program. If there is a specific part of the program you do not how to code you may ask others for examples, but at the end of the day you must write all of your own code.

You may of course ask me any questions you want and I will be very helpful. This is expected to be:

1. interesting – the 3SAT problem is a very interesting problem.

2. difficult – handling the details of this program can be very difficult to keep straight.

3. illuminating – hopefully this helps you play with logical expressions and gain a deeper understanding. Also determining how to interact with logical expressions and strings in a program.

4