CS 2051: Honors Discrete Mathematics
Spring 2023 Homework 3 Supplement

Sarthak Mohanty

Overview

The field of discrete mathematics is impossibly vast. One could even imagine it as a museum, filled to the brim with centuries worth of ideas, theorems, and algorithms. In this supplement, I'll act as your tour guide through this museum, and introduce some of the most impactful problems in recent history, such as the **Boolean Satisfiability Problem** and the **Four Color Theorem**. As you journey through these exhibits, you'll also get the chance to curate and contribute something of your own. By utilizing code, you'll be able to add a distinctive and innovative perspective to these concepts.

Part 1: Inference (10 pts)

As you've seen on your homework this week, rules of inference are difficult to get right. It can often take a lot of time and energy to check if a conclusion is valid. Luckily, by exercising our programming skills, we can expedite this procedure.

In this part, you'll implement the following function:

`infer(inference_rules, conclusion)`: This function takes in a list of inference rules and a conclusion (both in their string representations), and returns `True` iff the conclusion can be proved using the provided rules. For example, testing it on the familiar "modus ponens" rule yields:

```
>>> infer(["p |implies| q", "p"], "q")
True
```

Since you've been acquainted with manipulating propositions of this nature from the last supplement, this method should (hopefully!) not take too long.

Part 2: Satisfiability (10 pts)

We continue our exploration of propositional logic with the **boolean satisfiability problem**, also known as `SAT`. Let's begin our explanation with the following story:

You are in charge of elections for the Georgia Tech Advisory Board. Elections for the board 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. Is it possible to choose some combination of people to elect to the board such that everyone is satisfied?

Let's formalize this problem using logical notation. A *literal* is either a variable or its negation. Examples include p, $\neg q_0$, and r_{10}. Next, define a *clause* as a disjunction of literals. The clause containing the previous three literals would be $(p \vee \neg q_0 \vee r_{10})$. We call a proposition *satisfiable* if there exists a model under which the proposition evaluates to `True`. Finally, a proposition is in *conjunctive normal form* (or CNF) if it is expressed as a conjunction of clauses.

Finally, the SAT problem asks "given a proposition in CNF, is it possible to create a satisfiable model?"

Brute Force Solver

One solution to the problem is simple: generate all possible models for the proposition, and evaluate the function over each of them. This has all been done for you in the function `brute_force_SAT_solver`. However, generating all the models is computationally expensive, since for n variables there exist 2^n possible models. Another way to say this is that the runtime of this function is *exponential*.

WalkSAT Solver

For Part 3, we will need a more efficient SAT solver. Luckily, we have many to choose from. In this supplement, we adopt the `WalkSAT` algorithm, developed by Christos Papadimitriou.[1] It's an example of a *randomized* algorithm, an algorithm that uses some degree of randomness as part of its logic. The pseudocode for the algorithm is provided below:

Algorithm 1 WALKSAT($prop, p, maxFlips$)

m = randomly chosen assignment to all variables in $prop$
for $flip = 1$ **to** $maxflips$ **do**
  if m satisfies $prop$ **then return** m
  c = randomly selected unsatisfied clause in $prop$
  with probability p flip a randomly selected variable in c
  otherwise flip a variable in c which will result in the fewest previously satisfied clauses to be unsatisfied
return `None`

In this part, you'll implement the following function:

`walkSAT_solver`: This function takes in a proposition in conjunctive normal form and outputs a satisfying assignment (or `None` if no such assignment exists). Implement this algorithm according to the specifications outlined in Algorithm 1. The randomization of the algorithm guarantees a solution will eventually be found, but if the algorithm takes too long, you should prematurely end the algorithm.

A few tips:

- Use the `random.choice()` method to pick a random element from a list.
- Probability can be emulated using the statement `random.random() < p`.

Part 3: Reductions (10 pts)

We've learned about the `SAT` problem, but what's the purpose? The short answer is that many other "difficult" problems can be represented as an equivalent `SAT` problem. We'll demonstrate such a representation below.

[1] Papadimitriou is also the coauthor of the textbook *Algorithms* used in CS 3510.

The 4COLOR Problem

You might have heard of the "four color theorem". In simplest terms, it states that the regions in any map can be colored using at most four colors such that no two neighbouring regions are coloured the same. Even though we know every map can be colored this way, it remains a problem to find this coloring.

Let's formalize this problem as follows: As input, we will take the number of regions n, and assume the regions are labeled using numbers 1 to n, and a list of neighboring regions of the form i, j with $i \neq j$, indicating regions i and j are neighbors. Let us use colors red (R), blue (B), green (G), and yellow (Y) to color 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 colored red, blue, green, or yellow, respectively.

4COLOR to SAT

While we don't know how to solve this problem, we do know how to solve the `SAT` problem. By converting the `4COLOR` problem to an equivalent `SAT` problem, we can run our `SAT` solver to generate a solution for the `4COLOR` problem. The conversion is as follows: for each region,

1. First, add $(r_i \vee b_i \vee g_i \vee y_i)$ as a clause, ensuring that region i gets at least one color assigned to it.
2. Next, for every pair of colors, say r and b, add the clause $(\neg r_i \vee \neg b_i)$, effectively making sure that exactly one color is assigned to each region.
3. Finally, for any two neighboring regions, say i and j, and each color, say r, add the clause $(\neg r_i \vee \neg r_j)$. This represents the fact that regions i and j cannot both be colored red.

In this part, you'll implement the following function:

> `fourColoringtoSAT(neighbors)`: This problem takes in `neighbors` (a representation of the `4COLOR` problem) and returns an equivalent proposition constructed in CNF using the method outlined above. Make sure to use the variable names `ri`, `bi`, `gi`, and `yi` to indicate the i-th region will be colored red, blue, green, and yellow respectively.

Putting It All Together

Finally, it's time to see your hard work come to fruition. In the file `fourColor.py`, we've created some methods to generate Voronoi diagrams, an example of which is shown in Figure (1). If `walkSAT_solver` is implemented correctly, running the main function in the file should generate a successful coloring.

However, when we set `num_points` higher than ≈ 30, our `walkSAT` solver times out, as the coloring problem becomes too large to solve. If you want to generate graphs with more regions, change the `solver` parameter in the `colormap` call in the main function from `walkSAT_solver` to `pySAT_solver`. `pysat_solver` is a much more powerful solver, and can handle up to 5000 points with ease (for example, the map generated in Figure (2) contains 500 points, and was generated in seconds). This solver is also handy if you want to test the `fourColoringtoSAT` function without having completed `walkSAT_solver`.

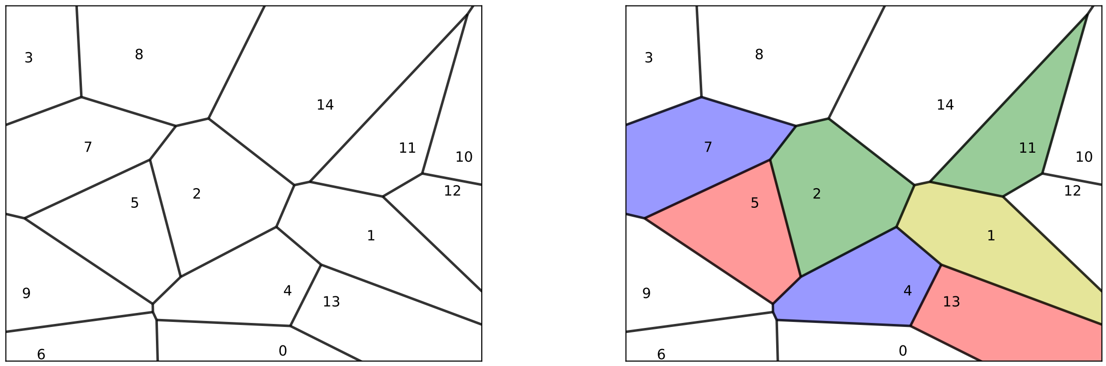

Figure 1: A map coloring generated using the `walkSAT` solver.

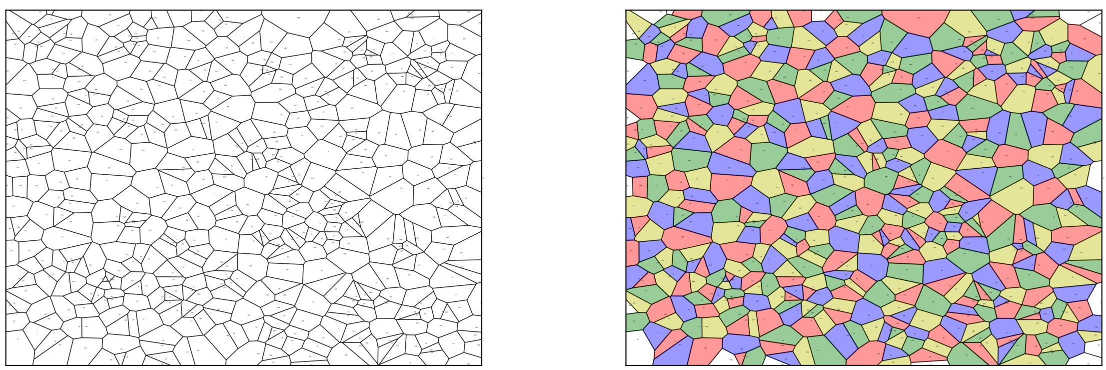

Figure 2: A much larger map coloring generated using the `pySAT` solver.

The Four Color Theorem. The Four Color Theorem states that every (planar) map can be colored using only four colors or less such that no two adjacent regions have the same color. The theorem, first posited in 1852, proved to be a challenging puzzle even for the likes of Augustus De Morgan. (yes, that De Morgan!)

Countless others tried their hand at a proof, but none were successful until 1976, when two mathematicians at the University of Illinois, Kenneth Appel and Wolfgang Haken announced their computer-assisted proof. It was the first major theorem to be proven with extensive computer assistance, and was initially met with doubt and concern.

It took many years for the proof to gain widespread acceptance, and even now mathematicians approach computer-assisted proofs with trepidation. In fact, some claim that "proofs" should only be accepted if they are proved by people, not machines, and any work performed by machines should be regarded only as "calculation." Nevertheless, is of the author's personal opinion that computer-assisted proofs are a definitive step in the right direction.

1 Submission Instructions

After you fill the appropriate functions, submit the following files to Gradescope and make sure you pass all test cases:

- `inference.py`
- `SAT.py`
- `fourCOLOR.py`

Notes

- The autograder may not reflect your final grade on the assignment. We reserve the right to run additional tests during grading.
- Do not import additional packages, as your submission may not pass the test cases.
- If you think you may encounter dependency/versioning issues by running code on your local machine, try completing the assignment on Google Colab.