Sudoku and Propositional Logic

Jake Fleming - 10/24/2023

Introduction

This project outlines an import aspect of artificial intelligence known as propositional logic. Mathematics plays an important role in this form of logic by having many laws for how terms such as “and”, “or”, “not”, etc. interact with each other in a clause. In order to represent a sudoku, we examine .cnf files, standing for conjunctive normal form. For these problems, each clause is simplified down to just a chain of “or” statements connecting several variables that should either be true or false. Using two generalized algorthims known as GSAT and WalkSAT, we can take a random boolean assignment for every variable and eventually reach a goal where every clause is true. GSAT will be able to solve only a few of these puzzles, but WalkSAT can solve them all with enough time. Once I have shown how WalkSAT can solve these sudokus, I will also demonstrate it on the 4-coloring map problem on the map of the United States.

Implementation

Generalized Initialization

For my SAT class, I have initialized it to solve any given satisfaction problem (in cnf) by assigning each variable to a positive integer starting at 1. In our .cnf files, I have let “-” represent “not”, meaning we want it to be false. In this case, we will let our generalized clauses be filled with positive and negative integers, and each clause is true if and only if at least one positive variable maps to true or negative variable maps to false.

With this all noted, the following code demonstrates how I read the .cnf file and put the variables into the dictionaries. The three dictionaries below all have important uses. New_variables allows us to check if we have already mapped a given variable to a positive integer yet when creating the new variables. Old_variabels allows us to rewrite the solution file using the original notation. Finally, variable_values maps each new variable to either true or false. The model list is a list of lists that will hold each generated clause.

```
def __init__(self, filename):
    self.model = []
    self.new_variables = {}     # map old to new
    self.old_variables = {}     # map new to old
    self.variable_values = {}   # map new to boolean
```

Then, we loop through each line in the file. If the variable does not have the designated “-”, we check if it is in self.new_variables yet. If not, we add it to new and old and clause. Otherwise, we only add it to clause.

```
if var not in self.new_variables:
    self.new_variables[var] = new_var
    self.old_variables[new_var] = var
    clause.append(new_var)
    new_var += 1
else:
    clause.append(self.new_variables[var])
```

Otherwise, if the variable is negatived, we must do the same thing, but making sure to add it to the dictionaries without the hyphen and add it to the clause with a negative value.

```
neg_variable = var[1:]
if neg_variable not in self.new_variables:
    self.new_variables[neg_variable] = new_var
    self.old_variables[new_var] = neg_variable
    new_var += 1
clause.append((-1) * self.new_variables[neg_variable])
```

Now, we have a generalized model that we can run our SAT algortihms on.

GSAT

GSAT is a simple algorithm that takes parameters h and max_iterations, where h is a probability of whether we should flip a random variables boolean value, or run the algorithm and flip the best variable. Following the given pseudocode, here is what I have produced (all quotes come from Devin Balkcom):

1 - "Choose a random assignment (a model)"

```
def random_assignment(self):
    for var in self.old_variables:
        self.variable_values[var] = bool(random.getrandbits(1))
```

2 - "If the assignment satisfies all the clauses, stop."

```
def goal_test(self):
    for clause in self.model:
        clause_value = False
        for variable in clause:
            # if positive and true
            if variable > 0 and self.variable_values[variable]:
                clause_value = True
            # if negative and false
            elif variable < 0 and not self.variable_values[(-1) * variable]:
                clause_value = True
        if not clause_value:
            return False

    return True
```

3 - "Pick a number between 0 and 1. If the number is greater than some threshold h, choose a variable uniformly at random and flip it; go back to step 2."

```
number = random.uniform(0, 1)
if number > h:
    random_variable = random.choice(list(self.variable_values.keys()))
    self.variable_values[random_variable] = not self.variable_values[random_variable]
```

4 - "Otherwise, for each variable, score how many clauses would be satisfied if the variable value were flipped."

```
def score(self):
    count = 0
    for clause in self.model:
        clause_value = False
        for variable in clause:
            if variable > 0 and self.variable_values[variable]:
                clause_value = True
            elif variable < 0 and not self.variable_values[(-1) * variable]:
                clause_value = True
        if clause_value:
            count += 1

    return count
```

5 - "Uniformly at random choose one of the variables with the highest score. (There may be many tied variables.) Flip that variable. Go back to step 2."

```
def flip_highest_variable(self, candidate):
    successor_score = {}
    highest_score = 0
```

```
    for var in candidate:
        self.variable_values[var] = not self.variable_values[var]
        score = self.score()
        successor_score[var] = score

        if score > highest_score:
            highest_score = score

        self.variable_values[var] = not self.variable_values[var]

    best_variables = []
    for successor in successor_score:
        if successor_score[successor] == highest_score:
            best_variables.append(successor)

    random_var = random.choice(best_variables)
    self.variable_values[random_var] = not self.variable_values[random_var]
```

With all these helper functions, we reach a simple code for GSAT:

```
def gsat(self, h, max_iterations):
    self.random_assignment()
    iterations = 0

    while not self.goal_test() and iterations < max_iterations:
        iterations += 1
        number = random.uniform(0, 1)

        if number > h:
            # flip random variable
            random_variable = random.choice(list(self.variable_values.keys()))
            self.variable_values[random_variable] = not self.variable_values[random_variable]
        else:
            # flip best variable otherwise
            self.flip_highest_variable(self.variable_values)

    print(iterations)
    return True
```

The reason we implement this probability is because the flip_highest_variable function can take some time when running over a large problem. Because of this, it can actually save some time to flip variables at random with the chance of making a good choice. It works almost like a trial and error kind of approach. The proability I have found to be fastest is to run flip_highest_variable 70% of the time.

Below is the result of running GSAT on "rows.cnf", a problem where we are only making sure each row has one of each integer 1-9. After 717 iterations and 188 seconds, it finally finds a solution.

```
717
188.0181142920046
8 9 7 | 2 3 4 | 5 1 6
1 2 6 | 9 7 3 | 5 4 8
5 8 4 | 7 6 3 | 9 2 1
---------------------
2 4 7 | 9 5 6 | 1 3 8
9 2 1 | 3 7 4 | 8 5 6
```

```
1 2 3 | 5 4 8 | 9 7 6
--------------------
6 1 3 | 5 4 7 | 9 8 2
5 7 9 | 2 6 4 | 8 1 3
6 8 9 | 3 2 7 | 1 4 5
```

WalkSAT

WalkSAT operates almost exactly the same as GSAT, but works much faster becasue it only looks through the variabes of a single unsatisfied clause for flip_highest_variable instead of all the variables. Below is the function I use to get a list of unassigned clauses. It also works very similar to score and goal test, just checking for true and positive or false and negative. If a clause is false, we add it to the list.

```
def unsatisfied_clauses(self):
    unsatisfied = []
    for clause in self.model:
        clause_value = False
        for variable in clause:
            if variable > 0 and self.variable_values[variable]:
                clause_value = True
            elif variable < 0 and not self.variable_values[(-1) * variable]:
                clause_value = True
        if not clause_value:
            unsatisfied.append(clause)

    return unsatisfied
```

We see in testing walksat that it operates significantly faster as expected. While GSAT only really solves "rows.cnf" (without me getting impatient) WalkSAT will solve even a real sudoku puzzle such as "puzzle1.cnf".

Below is the result I get for running walksat on "rows.cnf". It took only less that a second, and found a solution in less iterations. This demonstrates the power of the unsatisfied clause check.

```
671
0.8980674589984119
3 6 8 | 1 5 4 | 9 7 2
2 4 6 | 7 3 9 | 5 8 1
1 4 8 | 7 9 2 | 5 6 3
--------------------
9 7 4 | 6 5 8 | 2 3 1
5 4 8 | 2 7 9 | 1 6 3
4 7 2 | 9 6 8 | 1 3 5
--------------------
6 9 3 | 8 5 7 | 2 4 1
1 7 5 | 4 2 9 | 6 8 3
2 5 3 | 7 1 8 | 9 4 6
```

Furthermore, here is the result my walksat finds for puzzle1 which I have also included unsolved. It figures this problem (which only begins with 8 starting values) in just over a minute, using 38567 iterations. I hvae noticed that the time it takes to solve the problem relies heavily on the random seed used to generate the initiale variable booleans. For some seeds, the algorithm caps at the max iterations which I have set to 100,000 and returns a puzzle that is almost complete. In this case though, I have found a seed that solves puzzle one in just over a third of the time.

```
5 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 9 0 | 0 0 0
0 0 0 | 0 0 0 | 0 6 0
```

```
--------------------
8 0 0 | 0 6 0 | 0 0 0
0 0 0 | 8 0 0 | 0 0 0
7 0 0 | 0 0 0 | 0 0 0
--------------------
0 0 0 | 0 0 0 | 2 0 0
0 0 0 | 0 1 0 | 0 0 5
0 0 0 | 0 0 0 | 0 0 0

38567
75.95288699999219
5 2 3 | 6 7 1 | 9 8 4
4 6 7 | 2 9 8 | 3 5 1
1 8 9 | 3 5 4 | 7 6 2
--------------------
8 3 2 | 1 6 7 | 5 4 9
6 9 5 | 8 4 3 | 1 2 7
7 1 4 | 9 2 5 | 8 3 6
--------------------
9 4 1 | 5 8 6 | 2 7 3
3 7 8 | 4 1 2 | 6 9 5
2 5 6 | 7 3 9 | 4 1 8
```

Extensions Beyond Basic Requirements

Simulated Annealing

Another form of a random walk algorithm is the simulated annealing algorithm. This simulates the process that occurs as a metal is heated and cooled to remove defects. The pseudocode for this algorithms as follows:

(1) Randomly assign all the variables

(2) Loop until goal is found or min temperature is reached

(3) Flip a random variable and calculate score

(4) If the score improves, accept the assignment

(5) For some random number between 0 and 1, accept the assignment if it is less to an exponential that decreases as temperature decreases (exp(delta_score / current_temp))

(6) Decrement temperature by cooling rate

This algorithm is quite differnet than our other two algorithms but works in a very interesting way. When testing it on "rows.cnf" it works much better than GSAT but worse than Walk_SAT. When running it on puzzle1 at a min_temp of 1e-100, it gets close to finding a solution, but doesn't get it perfect.

Comparing Simulated Annealing to WalkSAT, simulated annealing is better at escaping local minima whereas WalkSAT has trouble with this. WalkSAT performs slightly better but Simulated Annealing works on a larger range of problems. Simulated Annealing is nice because we can adjust things like temperature and cooling rate and min temperature, whereas Walksat only has probability h and max iterations. Finally, WalkSAT does a very targeted search whereas Simulated Annealing is more broad at first and narrows as temperature decreases.

Map Coloring Problem

Just like the previous project, these constraint satisfaction problems all follow the same basic principle. To implement this problem, I create a .cnf file given a neighbors map that I can then feed to my SAT algorithm

to compute the solution. The following code writes the .cnf file for me. First, we must add all the clauses that tell us that each state must have at least one color (where the integers 1-4 represent our 4 colors).

```
for region in neighbors:
    for color in range(1, 5):  # we use four coloring theorem
        file.write(f"{region}_{color} ")
    file.write("\n")
```

Then, we ensure all states have no more than one color.

```
for region in neighbors:
    for color1 in range(1, 5):
        for color2 in range(color1 + 1, 5):
            file.write(f"-{region}_{color1} -{region}_{color2}\n")
```

Finally, we add the constraint that no neighbors can have the same color.

```
for region, region_neighbors in neighbors.items():
    for neighbor in region_neighbors:
        for color in range(1, 5):
            file.write(f"-{region}_{color} -{neighbor}_{color}\n")
```

When running this over the map of the united states, I made a simple writing algorithm that prints out the result from the .sol file that the SAT created. Here is the result I find.

204 Iterations - AL green AK blue AZ green AR green CA red CO red CT yellow DE blue FL blue GA red HI green ID red IL blue IN red IA yellow KS yellow KY yellow LA blue ME green MD yellow MA red MI yellow MN blue MS red MO red MT blue NE blue NV yellow NH yellow NJ yellow NM yellow NY blue NC yellow ND green OH blue OK blue OR green PA red RI blue SC blue SD red TN blue TX red UT blue VT green VA red WA blue WV green WI red WY yellow

This proves that my SAT class works on the general satisfaction problem using its several dictionaries and new variable mappings.

Conclusion

Conjunctive Normal Form is an important concept to understand when solving problems like these. Once a clause can be written in a series of just “or” statements. It then becomes incredibly to loop through boolean values and run algorithms until a solution is found. Propositional logic is useful in all kinds of math and computer science, but especially here in artificial intelligence as agents gather information to make decisions.