CS170 Project 1 Writeup

Grayson Chao, David Fair, and Gustavo de Leon

November 30, 2017

1 Algorithm

1.1 Approach

Our group initially approached this phase from three different angles: Using backtracking, simulated annealing, and solving a system of XOR variables modulo 2. The backtracking approach ended up taking too long to find a solution for some of the larger staff files and solving the system of XOR variables required the creation of a number of boolean variables $\in O(W!)$. The approach that we eventually settled on was simulated annealing as it seemed to be the only approach that could find a valid ordering under reasonable time and space constraints. In our formulation, a state is an ordering of wizards, and the cost of a state is the number of constraints it breaks. Additionally, we ran into an odd edge case where annealing broke down: When one wizard appears in all constraints. It turns out that our prior attempt at backtracking can solve these cases efficiently (listed below as 'solver.py'). So we augmented our code to check for this condition and run backtracking instead if it was found.

1.2 Simulated Annealing (annealer.py)

1.2.1 State and neighborhood representation

A state is represented by a map of wizard names to their positions in the ordering. This is so that accessing the position of a wizard, and changing it, can be performed in constant time.

Given a state S, the neighborhood of adjacent states to S are all states S' that can be generated by the following function:

1. Take S, and randomly choose one constraint, c, that S does not satisfy.
2. Randomly choose two wizards a and b, where a is mentioned in c. b can be any wizard not equal to a.
3. Let $S' = S$ (copying value rather than reference), let $S'[a] = S[b]$, and let $S'[b] = S[a]$.
4. Return S'.

This corresponds to a simple rule: the neighbors of S are states like S, but where one wizard in a currently-broken constraint is swapped with any other wizard.

1.2.2 Performance

Our initial strategy, which simply swapped any two random wizards, had trouble finding good moves at low temperatures and low cost, so it got stuck in the last 20-40 constraints of large problems. However, we observed that swapping two wizards can affect only constraints mentioning those wizards. Therefore, at low cost, it's likely that most neighbors will be worse. At low energy, this just translates to no forward progress. To fix this, we ensured that at least one of the wizards swapped was always in a broken constraint, and thus could potentially be part of an improving change.

Our other major performance gain was realized by switching to Pypy, an alternative implementation of Python that relies on just-in-time compilation. We saw about a 200% speed increase from doing this, and if you are going to run our code, we recommend that you install Pypy as well (https://pypy.org).

1.3 Backtracking (solver.py)

1.3.1 Formulation

We initially realized the problem could be reduced to SAT. Each of the constraints $C = (W_1, W_2, W_3)$ is equivalent to an equality on variables $W_{1,3}$ and $W2, 3$ where:

$$W_{1,3}: \text{ is true iff } W_1 < W_3 \text{ in ordering}$$

$$W_{3,2}: \text{ is true iff } W_3 < W_2 \text{ in ordering}$$

$$C = (W_1, W_2, W_3) \equiv W_{1,3} = W_{2,3}$$

Because the ordering must form a valid list, there are also two additional types of constraints to satisfy.

Variables are asymmetric:

$$W_{a,b} \equiv \neg W_{b,a}$$

Relationships between wizards are transitive:

$$W_{a,b} \wedge W_{b,c} \implies W_{a,c}$$

1.3.2 Performance

Our solver solves up to `staff_60.in` very fast, but slows down rapidly around `staff_80.in`. In the end, we got stuck around `staff_120.in` with the backtracking solver. The problem is that the number of transitive relationships to be checked between variables grows exponentially with the problem size.

Given m constraints, every constraint has $\binom{m}{2}$, or $O(m^2)$ transitive relationships. At 2000 constraints in `staff_200.in`, that's 4 million constraints to check per assignment. In retrospect, we could have tried limiting the duplicate work done by the transitive constraint checks. Instead of iterating over all possible transitive relationships and checking them, it would be better to intelligently generate a list of only the transitive constraints that could be affected by the assignment. But by the time we got to the point of realizing this we'd already solved all the problems with simulated annealing.

2 Dependencies

We used the library `simanneal` to perform simulated annealing. The code can be found at `https://github.com/perrygeo/simanneal`, but in order to make the code runnable without the installation of additional packages, we've included it with appropriate licensing in our code submission. `simanneal` implements a popular form of simulated annealing which uses an acceptance probability function similar to the one used in the Metropolis-Hastings algorithm for obtaining clean random samples from an irregular distribution. This is the same acceptance probability function used in the CS170 textbook on page 292 (print version) - where in the pseudocode it mentions "replace s by s' with probability $e^{-\Delta/T}$."

3 Code

The code of `simanneal` has been reproduced according to the terms of its license. The license can be found in section 3.3.

3.1 annealer.py

Our work. `move()` generates a neighbor of the current state, and `energy()` gives the cost of the current state.

```
#!/usr/bin/env python

from simanneal import Annealer
import argparse
import random
import sys
import time

class WizardAnnealer(Annealer):
    """ Let him who doubts my power come forth, and be annealed.
    """
    def __init__(self, initial_state, constraints, wizards):
        super(WizardAnnealer, self).__init__(initial_state)
        self.wizards = wizards
        self.constraints = constraints

    def move(self):
        """ Swap a random name in an unsatisfied constraint, and a random other name
        """
        broken = self.get_broken(self.state)
        if broken:
            a = random.choice(broken[random.randint(0, len(broken) - 1)])
            b = a
            while b == a:
                b = self.wizards[random.randint(0, len(self.state) - 1)]
        else:
            a = self.wizards[random.randint(0, len(self.state) / 2)]
            b = self.wizards[random.randint(len(self.state) / 2, len(self.state) - 1)]
        self.state[a], self.state[b] = self.state[b], self.state[a]

    def energy(self):
        """ # of constraints violated.
        """
        e = len(self.get_broken(self.state))
        if e == 0:
            answer = list(sorted(self.state.keys(), key=lambda x: self.state[x]))
```

```
            print " ".join(answer)
            sys.exit(0)
        return e

    def get_broken(self, state):
        """ Get the constraints broken by STATE.
        """
        broken = []
        for constraint in self.constraints:
            a, b, c = constraint
            if (state[a] < state[c]) != (state[b] < state[c]):
                broken.append(constraint)
        return broken

def check_violates(constraint, o):
    """ Return true if O, an ordering, violates CONSTRAINT.
    """
    left, right, subject = constraint
    if o.index(left) < o.index(subject) and o.index(right) < o.index(subject
    ):
        print "%s satisfied: %s > %s and %s" % \
            (str(constraint), subject, left, right)
    elif o.index(left) > o.index(subject) and o.index(right) > o.index(
    subject):
        print "%s satisfied: %s < %s and %s" % \
            (str(constraint), subject, left, right)
    else:
        print "VIOLATED", constraint

def anneal(args, input_file):
    num_wizards = int(input_file.readline())
    num_constraints = int(input_file.readline())
    constraints = []
    wizards = set()
    for line in input_file:
        left, right, middle = line.split()
        constraints.append((left, right, middle))
        wizards.update((left, right, middle))
    state = {w: idx for idx, w in enumerate(wizards)}
    annealer = WizardAnnealer(state, constraints, list(wizards))
    schedule = {
        ’tmax’: 90.0, # Adjust till init temp ˜= 98% for best results
        ’tmin’: 0.001,
        ’steps’: 4 * (10 ** 6),
        ’updates’: 1000,
        ’copy_strategy’: ’method’
    }
    # schedule = annealer.auto(minutes=2)
```

```
    annealer.set_schedule(schedule)
    state, cost = annealer.anneal()
    return state, cost, constraints

def main():
    parser = argparse.ArgumentParser(description='Solve CS170 project 1 wizard files.')
    parser.add_argument('input_file', help='Name of the input file to solve.')
    parser.add_argument(
        '--show_proof',
        '-p',
        action='store_true',
        help='Tell me why each constraint is solved')
    args = parser.parse_args()
    # Read the input file in, parsing out the unique wizard names and constraints.
    with open(args.input_file, 'r') as infile:
        state, cost, constraints = anneal(args, infile)
        answer = list(sorted(state.keys(), key=lambda x: state[x]))
        if args.show_proof:
            for c in constraints:
                check_violates(c, answer)
        #print answer
        #print "Cost:", cost
        print " ".join(answer)

if __name__ == '__main__':
    main()
```

3.2 simanneal/anneal.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals
import abc
import copy
import datetime
import math
import pickle
import random
import signal
import sys
import time


def round_figures(x, n):
    """Returns x rounded to n significant figures."""
    return round(x, int(n - math.ceil(math.log10(abs(x)))))


def time_string(seconds):
    """Returns time in seconds as a string formatted HHHH:MM:SS."""
    s = int(round(seconds)) # round to nearest second
    h, s = divmod(s, 3600) # get hours and remainder
    m, s = divmod(s, 60) # split remainder into minutes and seconds
    return '%4i:%02i:%02i' % (h, m, s)


class Annealer(object):

    """Performs simulated annealing by calling functions to calculate
    energy and make moves on a state. The temperature schedule for
    annealing may be provided manually or estimated automatically.
    """

    __metaclass__ = abc.ABCMeta

    # defaults
    Tmax = 25000.0
    Tmin = 2.5
    steps = 50000
    updates = 100
    copy_strategy = 'deepcopy'
    user_exit = False
    save_state_on_exit = False

```

```
    # placeholders
    best_state = None
    best_energy = None
    start = None

    def __init__(self, initial_state=None, load_state=None):
        if initial_state is not None:
            self.state = self.copy_state(initial_state)
        elif load_state:
            self.state = self.load_state(load_state)
        else:
            raise ValueError('No valid values supplied for neither \
            initial_state nor load_state')

        signal.signal(signal.SIGINT, self.set_user_exit)

    def save_state(self, fname=None):
        """Saves state to pickle"""
        if not fname:
            date = datetime.datetime.now().strftime("%Y-%m-%dT%Hh%Mm%Ss")
            fname = date + "_energy_" + str(self.energy()) + ".state"
        with open(fname, "wb") as fh:
            pickle.dump(self.state, fh)

    def load_state(self, fname=None):
        """Loads state from pickle"""
        with open(fname, 'rb') as fh:
            self.state = pickle.load(fh)

    @abc.abstractmethod
    def move(self):
        """Create a state change"""
        pass

    @abc.abstractmethod
    def energy(self):
        """Calculate state's energy"""
        pass

    def set_user_exit(self, signum, frame):
        """Raises the user_exit flag, further iterations are stopped
        """
        self.user_exit = True

    def set_schedule(self, schedule):
        """Takes the output from `auto` and sets the attributes
        """
        self.Tmax = schedule['tmax']
```

```
        self.Tmin = schedule['tmin']
        self.steps = int(schedule['steps'])
        self.updates = int(schedule['updates'])

    def copy_state(self, state):
        """Returns an exact copy of the provided state
        Implemented according to self.copy_strategy, one of

        * deepcopy : use copy.deepcopy (slow but reliable)
        * slice: use list slices (faster but only works if state is list-like)
        * method: use the state's copy() method
        """
        if self.copy_strategy == 'deepcopy':
            return copy.deepcopy(state)
        elif self.copy_strategy == 'slice':
            return state[:]
        elif self.copy_strategy == 'method':
            return state.copy()
        else:
            raise RuntimeError('No implementation found for ' +
                               'the self.copy_strategy "%s"' %
                               self.copy_strategy)

    def update(self, *args, **kwargs):
        """Wrapper for internal update.

        If you override the self.update method,
        you can chose to call the self.default_update method
        from your own Annealer.
        """
        self.default_update(*args, **kwargs)

    def default_update(self, step, T, E, acceptance, improvement):
        """Default update, outputs to stderr.

        Prints the current temperature, energy, acceptance rate,
        improvement rate, elapsed time, and remaining time.

        The acceptance rate indicates the percentage of moves since the last
        update that were accepted by the Metropolis algorithm. It includes
        moves that decreased the energy, moves that left the energy
        unchanged, and moves that increased the energy yet were reached by
        thermal excitation.

        The improvement rate indicates the percentage of moves since the
        last update that strictly decreased the energy. At high
        temperatures it will include both moves that improved the overall
```

```
        state and moves that simply undid previously accepted moves that
        increased the energy by thermal excititation. At low temperatures
        it will tend toward zero as the moves that can decrease the energy
        are exhausted and moves that would increase the energy are no longer
        thermally accessible."""

        elapsed = time.time() - self.start
        if step == 0:
            print(
            ' Temperature Energy Accept Improve Elapsed Remaining',
                  file=sys.stderr)
            print('\r%12.5f %12.2f %s ' %
                  (T, E, time_string(elapsed)), file=sys.stderr, end="\r")
            sys.stderr.flush()
        else:
            remain = (self.steps - step) * (elapsed / step)
            print('\r%12.5f %12.2f %7.2f%% %7.2f%% %s %s\r' %
                  (T, E, 100.0 * acceptance, 100.0 * improvement,
                   time_string(elapsed), time_string(remain)), file=sys.stderr, end="\r")
            sys.stderr.flush()

    def anneal(self):
        """Minimizes the energy of a system by simulated annealing.

        Parameters
        state : an initial arrangement of the system

        Returns
        (state, energy): the best state and energy found.
        """
        step = 0
        self.start = time.time()

        # Precompute factor for exponential cooling from Tmax to Tmin
        if self.Tmin <= 0.0:
            raise Exception('Exponential cooling requires a minimum "\
                "temperature greater than zero.')
        Tfactor = -math.log(self.Tmax / self.Tmin)

        # Note initial state
        T = self.Tmax
        E = self.energy()
        prevState = self.copy_state(self.state)
        prevEnergy = E
        self.best_state = self.copy_state(self.state)
        self.best_energy = E
        trials, accepts, improves = 0, 0, 0
```

```
        if self.updates > 0:
            updateWavelength = self.steps / self.updates
            self.update(step, T, E, None, None)

        # Attempt moves to new states
        while step < self.steps and not self.user_exit:
            step += 1
            T = self.Tmax * math.exp(Tfactor * step / self.steps)
            self.move()
            E = self.energy()
            dE = E - prevEnergy
            trials += 1
            if dE > 0.0 and math.exp(-dE / T) < random.random():
                # Restore previous state
                self.state = self.copy_state(prevState)
                E = prevEnergy
            else:
                # Accept new state and compare to best state
                accepts += 1
                if dE < 0.0:
                    improves += 1
                prevState = self.copy_state(self.state)
                prevEnergy = E
                if E < self.best_energy:
                    self.best_state = self.copy_state(self.state)
                    self.best_energy = E
            if self.updates > 1:
                if (step // updateWavelength) > ((step - 1) // updateWavelength):
                    self.update(
                        step, T, E, accepts / trials, improves / trials)
                    trials, accepts, improves = 0, 0, 0

        self.state = self.copy_state(self.best_state)
        if self.save_state_on_exit:
            self.save_state()

        # Return best state and energy
        return self.best_state, self.best_energy

    def auto(self, minutes, steps=2000):
        """Explores the annealing landscape and
        estimates optimal temperature settings.

        Returns a dictionary suitable for the ‘set_schedule‘ method.
        """

        def run(T, steps):
```

```
            """Anneals a system at constant temperature and returns the state,
            energy, rate of acceptance, and rate of improvement."""
            E = self.energy()
            prevState = self.copy_state(self.state)
            prevEnergy = E
            accepts, improves = 0, 0
            for _ in range(steps):
                self.move()
                E = self.energy()
                dE = E - prevEnergy
                if dE > 0.0 and math.exp(-dE / T) < random.random():
                    self.state = self.copy_state(prevState)
                    E = prevEnergy
                else:
                    accepts += 1
                    if dE < 0.0:
                        improves += 1
                    prevState = self.copy_state(self.state)
                    prevEnergy = E
            return E, float(accepts) / steps, float(improves) / steps

        step = 0
        self.start = time.time()

        # Attempting automatic simulated anneal...
        # Find an initial guess for temperature
        T = 0.0
        E = self.energy()
        self.update(step, T, E, None, None)
        while T == 0.0:
            step += 1
            self.move()
            T = abs(self.energy() - E)

        # Search for Tmax - a temperature that gives 98% acceptance
        E, acceptance, improvement = run(T, steps)

        step += steps
        while acceptance > 0.98:
            T = round_figures(T / 1.5, 2)
            E, acceptance, improvement = run(T, steps)
            step += steps
            self.update(step, T, E, acceptance, improvement)
        while acceptance < 0.98:
            T = round_figures(T * 1.5, 2)
            E, acceptance, improvement = run(T, steps)
            step += steps
```

```
            self.update(step, T, E, acceptance, improvement)
        Tmax = T

        # Search for Tmin - a temperature that gives 0% improvement
        while improvement > 0.0:
            T = round_figures(T / 1.5, 2)
            E, acceptance, improvement = run(T, steps)
            step += steps
            self.update(step, T, E, acceptance, improvement)
        Tmin = T

        # Calculate anneal duration
        elapsed = time.time() - self.start
        duration = round_figures(int(60.0 * minutes * step / elapsed), 2)

        # Don’t perform anneal, just return params
        return {’tmax’: Tmax, ’tmin’: Tmin, ’steps’: duration, ’updates’: self.updates}
```

3.3 License of simeanneal/anneal.py

3.4 solver.py

```
#!/usr/bin/env python
""" Solver for CS170 project 1.
"""

import argparse
import collections
import functools
import itertools
import random
import sys

```

```
DEBUG = False

sys.setrecursionlimit(99999)

class Contradiction(Exception):
    pass

class Solution(object):
    def __init__(self, wizards, assignments={}, constraints={}, equalities=None):
        self.assignments = assignments.copy()
        self.constraints = list(constraints)
        self.equalities = collections.defaultdict(set)
        self.to_set = []
        if equalities is None:
            for constraint in constraints:
                left, right, middle = constraint
                self.equalities[(left, middle)].add((right, middle))
                self.equalities[(right, middle)].add((left, middle))
        else:
            self.equalities = equalities.copy()

        self.wizards = wizards

    def constraints_broken(self):
        """ Return a list of the constraints broken by this solution.
        """
        broken = []
        o = self.answer()
        for constraint in self.constraints:
            left, right, subject = constraint
            if o.index(left) < o.index(subject) and o.index(right) < o.index(subject):
                if DEBUG:
                    print "SATISFIED %s: %s > %s and %s" % (
                        str(constraint), subject, left, right)
            elif o.index(left) > o.index(subject) and o.index(right) > o.index(subject):
                if DEBUG:
                    print "SATISFIED %s: %s < %s and %s" % (
                        str(constraint), subject, left, right)
            else:
                if DEBUG: print "VIOLATED:", constraint
                broken.append(constraint)
        return broken

    def reduce(self):
        while self.to_set:
```

```
            pair, value = self.to_set.pop()
            self._set_truth(pair, value)

    def set_truth(self, pair, value):
        self.to_set.append((pair, value))

    def _set_truth(self, pair, value):
        """ Given PAIR = (a, b) and VALUE, a boolean, sets "a < b = VALUE" and propagates
        implications.
        """
        a, b = pair
        if (a, b) in self.assignments:
            if self.assignments[pair] == value:
                return
            else:
                raise Contradiction()
        self.assignments[(a, b)] = value

        self.enforce_eq(pair, value)
        self.enforce_asym(pair, value)
        self.enforce_transitivity(pair, value)

    def enforce_eq(self, pair, value):
        """ Enforce equality (e.g. for constraint (A, B, C), the value of A>C = the value of B>C)
        """
        for eq in self.equalities[pair]:
            self._set_truth(eq, value)

    def enforce_asym(self, pair, value):
        """ Enforce the rule that if a < b, then b > a
        """
        a, b = pair
        self._set_truth((b, a), not value)

    def enforce_transitivity(self, pair, value):
        """ Enforce the rule that a < b and b < c implies a < c
        """
        a, b = pair
        for c in self.wizards:
            if (b, c) in self.assignments and self.assignments[(b, c)] == value:
                self.set_truth((a, c), value)

    def copy(self):
        """ Return a copy of this solution
        """
```

```
        return Solution(self.wizards, self.assignments, self.constraints, self.equalities)

    def answer(self):
        """ Return an ordering of wizards (whose names are in O) representation of this solution.
        """
        return sorted(
            self.wizards,
            key=functools.cmp_to_key(
                lambda a, b: -1 if (a, b) in self.assignments and self.assignments[(a, b)] else 1
            ))

    def cost(self):
        """ Returns the number of constraints violated by this solution.
        """
        return len(self.constraints_broken())


class Solver(object):
    def __init__(self, input_file, is_solution = False):
        self.assignments = {}
        self.constraints = []
        self.equalities = collections.defaultdict(list)
        self.wizards = set()
        self.num_wizards = int(input_file.readline())
        if is_solution:
            input_file.readline()
        self.num_constraints = int(input_file.readline())
        # Iterating here doesn't include the first two lines
        for line in input_file:
            left, right, middle = line.split()
            self.constraints.append((left, right, middle))
            self.equalities[(left, middle)].append((right, middle))
            self.equalities[(right, middle)].append((left, middle))
            self.wizards.update(line.split())

    def solve(self, output_file=None):
        """ Solve the puzzle.
        """
        self.tried = 0
        pairs = [(a, b) for a in self.wizards for b in self.wizards if a != b]

        print "Starting: %d todo" % len(pairs)
        s = Solution(self.wizards, self.assignments, self.constraints)
        s = self._solve(s, pairs)
```

```
        print '%d/%d violated' % (s.cost(), len(s.constraints))
        print s.answer()
        print 'Tried a total of %d states' % self.tried
        if output_file is not None:
            with open(output_file, 'w') as outfile:
                outfile.write(" ".join(s.answer()))

    def _solve(self, solution, todo):
        """ Solve helper. Maintains a todo list (kinda like a fringe from 188) of variables
        that need to be set.

        For each of those, we go through and try set it to True or False, propagating the
        implications of doing so (see set_truth for that). If we get to a point where neither
        True nor False produces a valid partial solution, then we definitely need to backtrack.
        """

        self.tried += 1
        done = []
        while todo and todo[-1] in solution.assignments:
            done.append(todo.pop())
        if not todo:
            return solution

        pair = todo.pop()
        for value in [True, False]:
            s = solution.copy()
            try:
                s.set_truth(pair, value)
                s.reduce()
                return self._solve(s, todo)
            except Contradiction:
                # Raised when directly setting this value produces a contradiction
                continue

        # If we get to this point, then PAIR can't be set to true or false given the existing
        # partial solution passed in as an argument to this call.
        todo.append(pair)
        todo.extend(done)
        raise Contradiction()

def main():
```

```
    parser = argparse.ArgumentParser(description='Solve CS170 project 1 wizard files.')
    parser.add_argument('input_file', help='Name of the input file to solve.')
    parser.add_argument('--is-solution', '-i', action='store_true',
        help='Set to true to parse solution files (W20.txt etc)')
    parser.add_argument('--output', '-o', help='Output file to store Gradescope-ready answers in')
    args = parser.parse_args()

    import time
    start = time.time()
    # Read the input file in, parsing out the unique wizard names and constraints.
    with open(args.input_file, 'r') as infile:
        solver = Solver(infile, args.is_solution)
        solver.solve(args.output)

    print "%f sec elapsed" % (time.time() - start)

    # print "%d Wizards: %s" % (num_wizards, wizards)
    # print "%d Constraints: %s" % (num_constraints, equalities)


if __name__ == '__main__':
    main()
```

3.5 Discarded Approach Using Modular Arithmetic

1. For a given constraint (W_1, W_2, W_3), create two variables:

$$W_{1,3}: \text{ indicating } W_1 < W_3 \text{ in ordering}$$

$$W_{3,2}: \text{ indicating } W_3 < W_2 \text{ in ordering}$$

2. $W_{1,3} \oplus W_{3,2}$ then indicates:

$$[(W_1 < W_3) \wedge (W_2 < W_3)] \vee [(W_3 < W_1) \wedge (W_3 < W_2)]$$

3. Create an equivalent expression using modular arithmetic:

$$W_{1,3} + W_{3,2} \equiv 1 \mod 2$$

4. For every constraint, create such an equation and construct a matrix to house this system of equations where the columns represent each variable:

$$\text{(a)}\left\{\underbrace{\begin{bmatrix} \overbrace{\begin{matrix} 1 & 0 & 1 & \dots & 0 & 0 & 0 \end{matrix}}^{\text{(b)}} \\ \begin{matrix} 0 & 0 & 1 & \dots & 0 & 0 & 0 \end{matrix} \\ \begin{matrix} \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & \dots & 0 & 1 & 1 \end{matrix} \\ \begin{matrix} 0 & 0 & 1 & \dots & 0 & 0 & 1 \end{matrix} \\ \begin{matrix} 0 & 0 & 1 & \dots & 1 & 0 & 1 \end{matrix} \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} W_{1,2} \\ W_{1,3} \\ \vdots \\ \vdots \\ W_{i,j} \\ W_{j,k} \end{bmatrix}}_{\vec{x}} = \underbrace{\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \\ 0 \\ 0 \end{bmatrix}}_{\vec{b}}\right\}\text{(c)}$$

Note that:

(a) Rows of A (coefficients) always sum to 2.

(b) Columns of A (coefficients) always sum to the number of times a variable appears in an equation brought about by a constraint.

(c) Entries of $\vec{b}$ (solutions to modular equations) are always either 1 or 0.

During the construction of the matrix, ensure that if the variable $W_{1,3}$ is going to be created that $W_{3,1}$ does not already exist. In cases where it already exists, we are going to use the logical negation of the preexisting variable. This will require that our equation change to the following form:

$$W_{3,1} + W_{i,j} \equiv 0 \mod 2$$

Repeat the process above for variables which share a transitive relationship. Transitive variables will be of the form:

$$W_{i,j,..k}$$

Which is representative of a sequence of inequalities being true. This is the step that has the potential to lead to a factorial number of boolean variables given a large set of constraints and was ultimately the reason we discarded this approach.

It is highly likely that we will not be dealing with a square matrix after this process.

To arrive at a solution we then solve $R * x = Q^T * b$ where Q and R were attained during the decomposition step, x is our vector of XOR variables, and b is our vector of solutions to our equations modulo 2.

Completing the last step was made possible using "SageMath", a mathematics software that has packages for the QR decomposition and solving of linear systems over finite fields (modulo 2 in this instance).