CS170 Project Report (Fall 2017)

Micah Carroll, Orkun Duman, Kenneth Lai

December 1, 2017

Contents

1 Main Idea.

We propose a reduction of the problem into a 3-Satisfiability(3-SAT) and solved it using simulated annealing. Suppose that we have n wizards in the input files with random values denoted $w_1, \ldots, w_n$. Then we produce $\binom{n}{2}$ literals $X_{i,j}$ where

$$X_{i,j} \Rightarrow w_i < w_j$$

$$\neg X_{i,j} \Rightarrow w_i > w_j$$

Then given the input files of relative ordering of wizards, we can generate our satisfiability clauses. In particular, each line w_i, w_j, w_k can be translated as $(w_k < w_i \vee w_k > w_j) \wedge (w_k < w_j \vee w_k > w_i)$. Equivalently,

$$(X_{k,i} \vee X_{j,k}) \wedge (X_{k,j} \vee X_{i,k})$$

Where we naturally define $X_{i,j} = \neg X_{j,i}$ if one of them is not defined. Also, we add extra condition of transitivity for comparison, since $w_i < w_j \wedge w_j < w_k \Rightarrow w_i < w_k$ and $w_i > w_j \wedge w_j > w_k \Rightarrow w_i > w_k$. In our literals, this means we have a clauses

$$(\neg X_{i,j} \vee \neg X_{j,k} \vee X_{i,k}) \wedge (X_{i,j} \vee X_{j,k} \vee \neg X_{i,k})$$

for each triplet. We aggregate all the clause mentioned above to define a 3SAT problem. For the smaller inputs (up to around 100) we simply solve it with a python inplemented 3SAT library (pycosat). For the inputs with larger input, we run a local search with simulated annealing: assigning literals and optimize over the number of constraints violated. With well chosen parameters we reached global optimum for all our problems. Then we use assignments of $X_{i,j}$ that satisfy all our constraints to recreate a valid ordering of the wizards.

2 Libraries.

2.1 pycosat

We used an python implemented 3SAT solver initially to solve inputs with sizes up to 100. To install, run `pip install pycosat` on the terminal.

2.2 simanneal

`simanneal` is a multipupose simmulated annealing library. We used it to do local search to solve large input sizes. To install, run `pip install simanneal` on the terminal.

3 Instructions.

Run the solver by:

```
python3 solver.py input.in output.out
```

If you want to print verbose debug messages add the `-d` flag:

```
python3 solver.py input.in output.out -d
```

You can also solve all files in a directory in order and pipe the outputs to another directory:
`python3 solver.py phase2/inputs phase2/outputs`

4 Code.

The files we wrote include:

1. `SAT3.py`
2. `solver.py`
3. `code_generator.py` (used to generate random test inputs)
4. `randomized3SAT.py` (deprecated)
5. `solver2.py` (deprecated)

4.1 SAT3.py

includes class files for solvers

```
import pycosat

class SAT3(object):
    def __init__(self, clauses, variables):
        self.clauses = clauses
        self.variables = variables

class LocalSearch(SAT3):
    def solve(self, depth=1000):
        for i in range(depth):
            self.switch_one()

    def switch_one(self):
        for clause in self.clauses:
            if not clause.satisfied:
                best_change = 0
                best_variable = None
                for var in clause.variables:
                    change = self.try_switch(var)
                    print(change)
                    if change > best_change:
                        best_change = change
                        best_variable = var

                if best_variable:
```

```
                    print("Switching {} with {} change".format(best_variable,
                        best_change))
                    best_variable.switch()
                    break

        def try_switch(self, variable):
            current = self.num_satisfied()
            variable.switch()
            with_switch = self.num_satisfied()
            variable.switch()
            return with_switch - current

        def num_satisfied(self):
            return sum([clause.satisfied for clause in self.clauses])

    class PycosatSolver(SAT3):

        def solve(self):
            pycosat_clauses = self.convert_to_pycosat(self.variables.encoder)
            pycosat_solution = pycosat.solve(pycosat_clauses)
            self.set_variables_from_pycosat(pycosat_solution, self.variables.
                decoder)

        def convert_to_pycosat(self, encoder):
            pycosat_instance = []
            for clause in self.clauses:
                pycosat_clause = []
                for item in zip(clause.variables, clause.truth_values):
                    truth_value = item[1]
                    encoded_variable = encoder[item[0]]
                    if not truth_value:
                        encoded_variable *= -1
                    pycosat_clause.append(encoded_variable)
                pycosat_instance.append(pycosat_clause)
            return pycosat_instance

        def set_variables_from_pycosat(self, pycosat_solution, decoder):
            for var in pycosat_solution:
                if var < 0:
                    decoder[-var].state = 0
                else:
                    decoder[var].state = 1
```

4.2 solver.py

Include various structs and the main solver method

```
import os
import re
import time
import math
import random
import pycosat
import argparse
import itertools
import simanneal
import output_validator

DEBUG = False

class Variable:
    def __init__(self, wizard1, wizard2):
        self.wizard1 = min(wizard1, wizard2)
        self.wizard2 = max(wizard1, wizard2)

    def __eq__(self, other):
        return self.wizard1 == other.wizard1 and self.wizard2 == other.wizard2

    def __hash__(self):
        return hash(self.wizard1 + self.wizard2)

    def __repr__(self):
        return "Variable: {} BEFORE {}".format(self.wizard1, self.wizard2)

class VariableList:
    def __init__(self, wizards):
        combinations = list((itertools.combinations(wizards, 2)))
        variables = [Variable(wizard1, wizard2) for wizard1, wizard2 in
            combinations]
        self.encoder = {}
        for i in range(len(variables)):
            self.encoder[variables[i]] = i + 1

    def encode_variable(self, variable):
        return self.encoder[variable]

    def __len__(self):
        return len(self.encoder)

    def __repr__(self):
```

```
            result = "Variable List <size: {}>".format(len(self.encoder))
            for variable in self.encoder:
                result += "\n{} -> {}".format(self.encoder[variable], str(variable
                    ))
            return result

    class Literal:
        def __init__(self, variable, value):
            self.variable = variable
            self.value = value

        def to_pycosat(self, variable_list):
            multiplier = 1 if self.value else -1
            return variable_list.encode_variable(self.variable) * multiplier

        def __repr__(self):
            assigned = "TRUE" if self.value else "FALSE"
            return "Literal: <{}> set to {}".format(str(self.variable), assigned)

    class Constraint:
        def __init__(self, triplet):
            assert(len(triplet) == 3)
            self.bound1 = triplet[0]
            self.bound2 = triplet[1]
            self.middle = triplet[2]

        # For each constraint "middle not between bound1 and bound2"
        # return a caluse of the form:
        # (bound1 < middle OR bound2 > middle) AND (bound2 < middle OR bound1 >
            middle).
        def to_clause(self, variable_list):
            a1 = Literal(Variable(self.bound1, self.middle), self.bound1 < self.
                middle)
            a2 = Literal(Variable(self.bound2, self.middle), self.bound2 > self.
                middle)
            b1 = Literal(Variable(self.bound2, self.middle), self.bound2 < self.
                middle)
            b2 = Literal(Variable(self.bound1, self.middle), self.bound1 > self.
                middle)
            return [Clause([a1, a2]), Clause([b1, b2])]

    class Clause(object):
        def __init__(self, literals):
            self.literals = literals

        def to_pycosat(self, variable_list):
```

```
            return [literal.to_pycosat(variable_list) for literal in self.literals
                ]

        def __repr__(self):
            return "Clause:" + " OR ".join([str(literal) for literal in self.
                literals])

    def solve(num_wizards, num_constraints, wizards, constraints, data = None):
        # Pre-processing.
        processing_start = time.time()
        if DEBUG:
            print("Input with {} wizards and {} constraints.".format(len(wizards),
                 len(constraints)))
        random.shuffle(wizards)

        # Generate all possible variables of 2 wizards each.
        variables = VariableList(wizards)
        assert(len(variables) == len(wizards) * (len(wizards) - 1) / 2)
        if DEBUG:
            print(variables)

        # Convert constraints to SAT clauses.
        clauses = []
        for triplet in constraints:
            constraint = Constraint(triplet)
            clause_pair = constraint.to_clause(variables)
            clauses.extend(clause_pair)
        assert(len(clauses) == len(constraints) * 2)

        # Add 3-Term clauses to prevent loops.
        sorted_wizards = sorted(wizards)
        for i in range(len(sorted_wizards)):
            for j in range(i + 1, len(sorted_wizards)):
                for k in range(j + 1, len(sorted_wizards)):
                    w1, w2, w3 = sorted_wizards[i], sorted_wizards[j],
                        sorted_wizards[k]
                    a1 = Literal(Variable(w1, w2), False)
                    a2 = Literal(Variable(w1, w3), True)
                    a3 = Literal(Variable(w2, w3), False)
                    clauses.append(Clause([a1, a2, a3]))
                    b1 = Literal(Variable(w1, w2), True)
                    b2 = Literal(Variable(w1, w3), False)
                    b3 = Literal(Variable(w2, w3), True)
                    clauses.append(Clause([b1, b2, b3]))
        assert(len(clauses) == len(constraints) * 2 + len(list(itertools.
            combinations(range(len(wizards)), 3))) * 2)
```

```
        if DEBUG:
            print("Clauses (first 100):\n", clauses[:100])

        # Solve using pycosat.
        pycosat_input = [clause.to_pycosat(variables) for clause in clauses]
        algorithm_start = time.time()
        print("Calling Pycosat to solve the problem.")
        pycosat_output = pycosat.solve(pycosat_input)
        print("Pycosat returned an assignment.")
        algorithm_duration = round(time.time() - algorithm_start, 2)

        # Decode pycosat solution.
        assignments = {}
        for assignment in pycosat_output:
            assignments[abs(assignment)] = True if assignment >= 0 else False
        if DEBUG:
            print("Pycosat Assignments:\n", pycosat_output)

        # Find a valid ordering of wizards.
        solution = []
        for wizard in wizards:
            target_index = 0
            for i in range(len(solution)):
                variable = Variable(solution[i], wizard)
                value = assignments[variables.encode_variable(variable)]
                if variable.wizard1 != solution[i]:
                    value = not value
                if value:
                    target_index = i + 1
            solution.insert(target_index, wizard)
        if DEBUG:
            print("Solution:", solution)

        # Completion info.
        processing_duration = round(time.time() - processing_start, 2) -
            algorithm_duration
        print("Solver complete. Algorithm took {} seconds. Processing took {}
            seconds.".format(algorithm_duration, processing_duration))
        return solution

    class WizardSolver(simanneal.Annealer):
        Tmax = 60       # Max (starting) temperature (over-written below)
        Tmin = 0.01        # Min (ending) temperature
        steps = 1000000  # Number of iterations
        updates = steps / 100   # Number of updates (by default an update prints
            to stdout)
```

```
        def __init__(self, wizards, constraints):
            self.constraints = constraints
            #self.Tmax = 10 + math.sqrt(len(constraints))
            super(WizardSolver, self).__init__(wizards)

        def move(self):
            a = random.randint(0, len(self.state) - 1)
            b = random.randint(0, len(self.state) - 1)
            self.state[a], self.state[b] = self.state[b], self.state[a]

        def energy(self):
            output_ordering_set = set(self.state)
            output_ordering_map = {k: v for v, k in enumerate(self.state)}
            not_satisfied = 0
            for constraint in self.constraints:
                c = constraint # Creating an alias for easy reference
                m = output_ordering_map # Creating an alias for easy reference
                wiz_a = m[c[0]]
                wiz_b = m[c[1]]
                wiz_mid = m[c[2]]
                if (wiz_a < wiz_mid < wiz_b) or (wiz_b < wiz_mid < wiz_a):
                    not_satisfied += 1
            return not_satisfied

    def anneal(num_wizards, num_constraints, wizards, constraints, data = None):
        # Pre-processing.
        algorithm_start = time.time()
        if DEBUG:
            print("Input with {} wizards and {} constraints.".format(len(wizards),
                 len(constraints)))
        random.shuffle(wizards)

        # Start simulated annealing.
        start_state = data if data is not None else wizards
        solver = WizardSolver(start_state, constraints)
        print("Starting with ordering where {} constrains are violated.".format(
            solver.energy()))
        if solver.energy() < 100:
            solver.Tmax = 1
        if solver.energy() < 25:
            solver.Tmax = 0.5
        if solver.energy() < 10:
            solver.Tmax = 0.01
        solution, num_constraints_failed = solver.anneal()
```

```
    # Completion info.
    algorithm_duration = round(time.time() - algorithm_start, 2)
    print("\nSolver complete. Algorithm took {} seconds.".format(
        algorithm_duration))
    return solution

"""
==============================================================================
   Input parsing happens below this line.
==============================================================================
"""

def read_input(filename):
    with open(filename) as f:
        num_wizards = int(f.readline())
        num_constraints = int(f.readline())
        constraints = []
        wizards = set()
        for _ in range(num_constraints):
            c = f.readline().split()
            constraints.append(c)
            for w in c:
                wizards.add(w)
    wizards = list(wizards)
    return num_wizards, num_constraints, wizards, constraints

def write_output(filename, solution):
    with open(filename, "w") as f:
        for wizard in solution:
            f.write("{0} ".format(wizard))

def atoi(text):
    return int(text) if text.isdigit() else text

def natural_keys(text):
    '''
    alist.sort(key=natural_keys) sorts in human order
    http://nedbatchelder.com/blog/200712/human_sorting.html
    (See Toothy's implementation in the comments)
    '''
    return [atoi(c) for c in re.split('(\d+)', text)]

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description = "Constraint Solver.")
    parser.add_argument(
        "input",
```

```
        type=str,
        help = "provide a file/folder where inputs should be read from")
    parser.add_argument(
        "output",
        type=str,
        help = "provide a file/folder where outputs should be saved")
    parser.add_argument(
        "--anneal", "-a",
        dest="anneal",
        action="store_true",
        help="use simulated annealing instead of 3SAT")
    parser.add_argument(
        "--start",
        dest="start",
        help="use the wizard ordering in this file as the starting state")
    parser.add_argument(
        "--debug", "-d",
        dest="debug",
        action="store_true",
        help="print out verbose debug messages")
    args = parser.parse_args()

    if args.debug:
        DEBUG = True

    start_state = None
    if args.start:
        with open(args.start, "r") as start_file:
            start_state = start_file.readline().split()

    inputs = [args.input]
    if os.path.isdir(args.input):
        inputs = [os.path.join(args.input, f) for f in os.listdir(args.input)
            if os.path.isfile(os.path.join(args.input, f)) and f.endswith(".in
            ")]
        inputs.sort(key=natural_keys)

    for input_file in inputs:
        if not os.path.isdir(args.output):
            output_file = args.output
        else:
            output_file = os.path.join(args.output, os.path.split(input_file)
                [1].replace(".in", ".out").replace("input", "output"))
        if len(inputs) > 1 and os.path.isfile(output_file):
            try:
```

```
            if len(output_validator.processInput(input_file, output_file)
                [2]) == 0:
                print("File {} already has a valid solution in {},
                    skipping.".format(input_file, output_file))
                continue
        except:
            pass
    num_wizards, num_constraints, wizards, constraints = read_input(
        input_file)
    print("Solving file: {} ({} wizards, {} constraints)".format(
        input_file, num_wizards, num_constraints))
    f = anneal if args.anneal else solve
    solution = f(num_wizards, num_constraints, wizards, constraints,
        start_state)
    write_output(output_file, solution)
    constraints_satisfied, num_constraints, constraints_failed =
        output_validator.processInput(input_file, output_file)
    if constraints_satisfied == num_constraints:
        print("Solution file {} verified!".format(output_file))
    else:
        print("Solution file {} did not satisfy {}/{} constraints.".format
            (output_file, len(constraints_failed), num_constraints))
```