Instructions

Run solver.py in Python 3. For Python 2, remove the parentheses in the print statement at line 97. The only parameter to adjust is the number of iterations at each temperature at line 85. My experience is that 100 iterations works for up to 100 wizards, 1000 iterations works for up to 200 wizards, and 5000 iterations works for up to 400 wizards. These estimates are conservative; 400 wizards can probably be solved with much less than 5000 iterations. Solving 400 wizards takes about 3 minutes on my machine.

Only the Python Standard Library is used. Pass the relative paths to the input and output files as arguments at runtime.

Algorithm

I used simulated annealing, which I had heard about in CS 188. It repeatedly finds a random neighbor of the current state, compares the energies of the current state and neighbor, and then moves to the neighbor if it has lower energy and possibly also if it has higher energy. This prevents it from getting stuck in a local optimum. I based my code off the template at katrinaeg.com/simulated-annealing.html. There are various parameters that must be selected.

Parameters

Energy function - I used the most obvious function, the number of unsatisfied constraints. We want to decrease this number to 0. To improve the runtime, I first extract the indices of all the wizards into a dict before examining each constraint.
Set of neighbors - I defined a neighbor as the result of moving wizard i to location j. For example, if i = 5 and j = 10, wizard 5 goes to location 10, and wizards 6-10 all shift 1 to the left. This was implemented using list slicing. I chose this definition of neighbors because it intuitively makes sense, is easy to implement, and allows me to speed up the energy calculation by only looking at the constraints that include wizard i (all others are unaffected by movement).
Acceptance probability - I went with the one on wikipedia, e^((energy - energy_neighbor) / T)
Initial temperature - I chose 1 because the documentation for scipy.optimize.basinhopping, which I didn't end up using, said it "should be comparable to the separation (in function value) between local minima". My energy function returns an integer, so the difference between local minima should be on the order of 1.
Minimum temperature - I chose .0001 because I found that the energy function was unlikely to decrease further when T reached below .0001.
Temperature decay rate - I chose .99. I did not experiment much with this parameter.
Iterations at each temperature - This is the parameter described above, which I varied from 100 to 5000 for various input sizes.

Other algorithms considered

The other algorithm that I spent the most time on was a greedy algorithm in which I repeatedly chose the move that decreased the number of unsatisfied constraints. Upon reaching a local minimum, I checked if it was better than the best seen local minimum, and if not I reverted to the best one. Then I perturbed the order by randomly swapping some wizards. This strategy worked

for up to 280 wizards, and completely failed for 300 wizards. It was quite slow, taking on the order of 1 second to compute the best move at each position. In retrospect, this is like a worse version of simulated annealing because it doesn't take advantage of the randomness which makes simulated annealing so powerful. Overall, I found that simulated annealing worked surprisingly well, and even the 400 wizard problem was not large enough to reveal any deficiencies.

```
import argparse
import random
import sys
import math

"""
================================================================================
  Complete the following function.
================================================================================
"""

# Input: list of wizard names
# Output: dict mapping wizard names to indices
def get_indices(wizards):
      indices = {}
      for index, wizard in enumerate(wizards):
            indices[wizard] = index
      return indices

# Input: list of constraints, list of wizard names
# Output: list of unsatisfied constraints
def bad_constraints(constraints, wizards):
      indices = get_indices(wizards)
      unsatisfied = []
      for constraint in constraints:
            x0, x1, x2 = indices[constraint[0]], indices[constraint[1]],
indices[constraint[2]]
            if x0 < x2 < x1 or x1 < x2 < x0:
                  unsatisfied.append(constraints)
      return unsatisfied

# Input: current energy, eneregy of neighbor, temperature
# Output: probability of moving to neighbor
# used in simulated annealing, taken from
en.wikipedia.org/wiki/Simulated_annealing
def P(E, E_neighbor, T):
      if E_neighbor < E:
            return 1
      else:
            return math.exp((E - E_neighbor) / T)

# Input: list of wizard names, tuple of (wizard to move, place to move
it)
# Output: list of wizard names after movement
# the neighbors of the current position are defined as the positions
reachable in 1 movement
# a movement only affects the constraints that include the moved wizard,
the order of the other wizards is unchanged
def do_move(wizards, move):
      wizard, location = move
      if wizard > location:
            return wizards[:location] + [wizards[wizard]] +
wizards[location:wizard] + wizards[wizard + 1:]
      else:
            return wizards[:wizard] + wizards[wizard + 1:location + 1] +
[wizards[wizard]] + wizards[location + 1:]

def solve(num_wizards, num_constraints, wizards, constraints):
      """
      Write your algorithm here.
```

```
      Input:
            num_wizards: Number of wizards
            num_constraints: Number of constraints
            wizards: An array of wizard names, in no particular order
            constraints: A 2D-array of constraints, where constraints[0]
may take the form ['A', 'B', 'C']

      Output:
            An array of wizard names in the ordering your algorithm
returns
      """

      # ignore constraints with the same name more than once
      constraints = [constraint for constraint in constraints if
len(constraint) == len(set(constraint))]

      # create a dict mapping a wizard w to the constraints that w
appears in
      relevant_constraints = {}
      for wizard in wizards:
            relevant_constraints[wizard] = [constraint for constraint in
constraints if wizard in constraint]

      # create a list of all possible moves
      moves = []
      for i in range(num_wizards):
            for j in range(num_wizards):
                  if i != j:
                        moves.append((i, j))

      random.shuffle(wizards)

      # simulated annealing, taken from katrinaeg.com/simulated-
annealing.html
      T = 1 # initial temperature
      T_min = 0.0001 # minimum temperature
      while T > T_min:
            for _ in range(5000): # number of iterations at each
temperature
                  random_move = random.choice(moves)
                  neighbor = do_move(wizards, random_move)
                  # anneal on the number of bad constraints
                  relevant = relevant_constraints[wizards[random_move[0]]]
                  E = len(bad_constraints(relevant, wizards))
                  E_neighbor = len(bad_constraints(relevant, neighbor))
                  if P(E, E_neighbor, T) >= random.random():
                        wizards = neighbor
            T *= .99 # temperature decay rate

            # print the current temperature and the number of bad
constraints, used to gauge progress
            num_bad = len(bad_constraints(constraints, wizards))
            print(T, num_bad)
            # return if all constraints are satisfied
            if num_bad == 0:
                  return wizards
            sys.stdout.flush()

      return wizards
```

```
"""
================================================================================
   No need to change any code 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))

if __name__=="__main__":
      parser = argparse.ArgumentParser(description = "Constraint
Solver.")
      parser.add_argument("input_file", type=str, help = "___.in")
      parser.add_argument("output_file", type=str, help = "___.out")
      args = parser.parse_args()

      num_wizards, num_constraints, wizards, constraints =
read_input(args.input_file)
      solution = solve(num_wizards, num_constraints, wizards,
constraints)
      write_output(args.output_file, solution)
```