CS 170 Efficient Algorithms and Intractable Problems
Fall 2017 Prasad Raghavendra and Sanjam Garg **Project**

Andrey Karapetov 3032261519
Cameron Rasmussen 3031736696
Eilam Levitov 3032395458

Algorithm: We used simulated annealing as the final iteration of our solver. The inspiration came from CS188 and was implemented once our bruteforce hill climbing algorithm became inadequte. As we continued to improve our solver we used various tactics to improve its efficiency:

1. Grade the ordering by the number of constraints satisfied
    - We want the accesses to be quick so we assigned each name to an index of the list which will store their positions (all unique positions $\in [0, n-1]$)
    - The constraints will be updated to reference the index of a given name for comparisons
    - Iterate over all constraints to check if they are satisfied, returning total number of the constraints violated
2. When the algorithm exits, we have an answer but not necessarily a solution
    - This allows us to have an early exit with a reasonable number of constraints satisfied without finding an optimal solution
    - We can run this algorithm each time, with each new call acting as a random restart which is expected of simulated annealing
3. When we have found a solution, we need to convert back to the names for the output file
    - Rebuilding the dictionary that gave indices based on names was simple because we create an alphabetized list from the name set
    - Next we save the file and can remove it from the pool of instances to be solved

Almost all original inputs were solved over the course of a couple of days, but some instances required further optimizations. While writing sanity checks for our current solutions, a shot in the dark (adding wizards in order of appearance) found that input 20_3 was already ordered correctly. This caused issues because we start every annealing round with a randomization of the order, so we added a check on import for the given ordering. For the last two instances we struggled with, 50_0 and 50_9, we added a statistical analysis function to give us a better starting points. Finally, for large inputs we simply increased the iterations for each annealing call to climb taller hills. With these additions, we were able to clear the original inputs within the weekend that they were released. The **new** inputs forced us to add multi-threading, which just explored from multiple starting points with parallel annealing calls.

Simulated Annealing: We decided to implement a modified optimization of hill climbing called Simulated Annealing combined with some statistical analisys that helped us improve performance. The basic ideas of the algorithm are as follows:

1. A heuristic function $h(assigment, constraints)$, that takes in an assigment of wizards and the set of constraints and returnes the number of constraints violated. We try to minimize this function, when 0 is the optimal solution.

2. The concept of a temperature T. The value of T determines how much risk are we willing to take when we are choosing a neighboring assigment. When the temperature is high we are more likely to choose some assignment that is worse than the current one. This is done so we can escape from local minimas and converge to a global one. T is going down exponentially with time as specified by a hyper parameter α which was chosen for best performance by trial and error, we chose $\alpha = 0.95$.

3. Acceptance probability function $p(current_h, new_h, T)$, a function that takes in the heuristic value for the current assigment C, the heuristic of the new assigment N and the current tempreture T, and returns the probability of choosing this new assignment over the current one. The function is defined as :

   $$p = e^{\frac{C-N}{T}}$$

   We see if the new assignment is better (lower) the probability is greater than 1, so we will always take it, in reality it is clipped to be at most 1.

4. At each temperature level we would try a number of neigboring assigments, this number is the *iterations* hyper parameter which controlls the number of neigboring assigments we consider at each temperature level. This number is proportional to the number of possible neigboring assignments which is square of the number of wizards, the actual number is

   $$iterations = 0.25 * numWizards^2$$

   The factor was chosen relatively arbitrarily, so we can reduce the amount of work needed.

5. Finally, we pass the simulated annealing function starting points, we set $T = 1$, and for each temperature level check *iterations* number of random neigboring assignments. For each new assignment we calculate the acceptance probability and use it to make a decision whether we take it or skip it. After we are done with the current temperature level we decrease the temperature by a factor of α, so we get $T = \alpha T$.

Statistical Analysis: Phase 1 gave us better understanding of the problem at hand. How do we generate a tough input? We thought giving many constraints will make it time consuming and consequently a 'harder' problem. Eventually we realized that by over-constraining an input file, we actually provide more information than necessary, a fact that we took advantage of. We generated a function that looks at the statistics of the constraints in order to provide the simulated annealing better starting points. The function looks at all of the constraints and counts the number unique pairs of wizards appear. We deducted, from our own experience, that since these pairs are highly correlated that must imply these pairs must be relatively close to each other. We added an empricial threshold value to return only 'strongly' correlated wizards, and generated starting points accordingly.

The Solver: We pass to our simulated annealing algorithm the starting points produced by our statistical analysis function, with potential neighbors proximate to one another, in random orders. As mentioned above, we improved utility by defining a list of good starts, and let a thread pool run on that list, maximizing our chances of finding the global minima of h. We continue to do so untill we find an optimal solution, since coping with NP-COMPLETE problems also requires some level of luck.

References:

Katerina Ellison Geltman: *Simulated Annealing Algorithm*
http://katrinaeg.com/simulated-annealing.html

Simulated Annealing

December 1, 2017

1 Simulated Annealing with Statistical Analysis

Eilam Levitov - 3032395458
Andrey Karapetov - 3032261519
Cameron Rasmussen - 3031736696

```
In [1]: import sys
        import os
        import numpy as np
        import random as rand
        import math
        import matplotlib.pyplot as plt
        from pathlib import Path
        from multiprocessing import Pool
        from functools import partial
```

1.1 File I/O

```
In [2]: # Reads input and aliases names to numbers
        def readInput(filename):
            with open(filename) as f:
                numWizards = int(f.readline())
                numConstraints = int(f.readline())
                constraints = []
                shot_in_the_dark = []
                wizards = set()
                for _ in range(numConstraints):
                    c = f.readline().split()
                    constraints.append(c)
                    for w in c:
                        if w not in wizards:
        #                     sanity check (hello hello 20_3)
                            shot_in_the_dark.append(w)
                        wizards.add(w)
            wizards = list(wizards)
            wizards.sort()
            dictWizards = {}
```

```
    for i in range(len(wizards)):
        dictWizards[wizards[i]] = i

    for i in constraints:
        for j in range(3):
            i[j] = dictWizards[i[j]]
        if int(i[0]) > int(i[1]):
            i[0], i[1] = i[1], i[0]
    return numWizards, numConstraints, list(range(numWizards)), constraints

# Displays input
def displayInput(filename):
    n, c, names, constraints = readInput(filename)
    string = str(n) + "\n"
    string += " ".join(names) + "\n"
    string += str(c) + "\n"
    for i in constraints:
        string += " ".join(i) + "\n"
    print(string)

# Reverts aliasing and generates final solution form
def reinterpret(num, seq):
    with open('./Staff_Inputs/staff_' + str(num) +  '.in') as f:
        numWizards = int(f.readline())
        numConstraints = int(f.readline())
        constraints = []
        wizards = set()
        for _ in range(numConstraints):
            c = f.readline().split()
            constraints.append(c)
            for w in c:
                wizards.add(w)
    wizards = sorted(list(wizards))
    dictWizards = {}
    for i in range(len(wizards)):
        dictWizards[i] = wizards[i]
    newOrder = [0] * len(seq)
    for i in range(len(seq)):
        newOrder[seq[i]] = dictWizards[i]
    writeOutput(num, 0, newOrder, constraints)

# Writes output to a file
def writeOutput(num, idx, order, constraints):
    string = str(len(order))
    string += "\n"
    for i in order:
        string += i + " "
    string += "\n" + str(len(constraints)) + "\n"
```

```
        for i in constraints:
            for j in i:
                string += j + " "
            string += "\n"
        f = open('./outputs/input' + str(num) + '_' + str(idx) + '.out', 'w')
        f.write(string)
        f.close()

    # Calls intance_validator.py with bash to validate our outputs
    def instanceValidator(num, idx):
        file = './outputs/input' + str(num) + '_' + str(idx) + '.out'
        input_size = str(num)
        !python instance_validator.py $file $input_size
```

1.2 Simulated Annealing

```
In [3]: def anneal(wizs, constraints, start, bestAss=None):
            numWiz = len(wizs)
            numCons = len(constraints)
            ass = start.copy()
            conflicts = h(ass, constraints)
            bestScore = numCons
            if bestAss == None:
                bestAss = [start.copy()]
        #     meta parameters, T should be 1, 0.8<alpha<0.99, 100<iterations<1000 should work we
            T = 1.0
            T_min = 0.00001
            alpha = 0.95
            iterations = int(0.15*numWiz**2)
            while conflicts > 0 and T > T_min:
                for i in range(iterations):
        #         Choosing 2 random neighboring wizards to swap
                    wizA, wizB = rand.sample(wizs, 2)
                    x = ass[wizA]
                    y = ass[wizB]
                    ass[wizA] = y
                    ass[wizB] = x
                    newScore = h(ass, constraints)
        #             if solution found, return
                    if newScore == 0:
                        print("0 conflicts - verifying...")
                        reinterpret(numWiz, ass)
                        string = './outputs/input' + str(numWiz) + '_0.out'
                        !python instance_validator.py $string $numWiz
        #                my daddy was a bankrobber (the clash - educate yourselves!)
                        os._exit(2)
                        return ass, None
        #             save permutation with least number of conflicts
```

```
                elif newScore < bestScore:
                    bestScore = newScore
                    bestAss = [ass.copy()]
#                     calculating probability
                ap = acceptance_prob(conflicts, newScore, T)
#               save change
                if ap > rand.random():
                    conflicts = newScore
#               revert change
                else:
                    ass[wizA] = x
                    ass[wizB] = y
#           reduce tempreture
            T *= alpha
        return bestAss[0], bestAss

    # Calculate probabily for given state
    def acceptance_prob(oldCost, newCost, T):
        exponent = (oldCost - newCost)/(1.0*T)
#       if exponent > 0 probability will be >1. just return 1 to prevent crazy big numbers.
        if exponent > 0:
            return 1
        p = math.e**exponent
        return p

    # Check number of constraint conflicts with given permutation
    def h(assigment, constraints):
        conflicts = 0
        for constraint in constraints:
            try:
                t0, t1 = assigment[constraint[0]], assigment[constraint[1]]
            except:
                print(constraint)
                print(len(assigment), assigment)
            a = t0 if (t0 < t1) else t1
            b = t0 + t1 - a
            c = assigment[constraint[2]]
            if a<c and c<b:
                conflicts += 1
        return conflicts
```

1.3 Statistical Analysis

Providing starting point with higher probability of reaching global minimum

```
In [4]: def goodStart(numWiz, wizs, constraints, plot=False, thrsh=1):
#       get potential neighboring pairs
        poteneighbors = potential_neighbors(numWiz, constraints, plot, thrsh)
```

```
    rand.shuffle(poteneighbors)
    andreysAss = []
#    shuffle and generate a starting permutation accordingly
    for i in poteneighbors:
        neighborA, neighborB = i[0], i[1]
        if neighborA not in andreysAss:
            andreysAss.append(neighborA)
        if neighborB not in andreysAss:
            andreysAss.append(neighborB)
    residual = []
    for i in range(numWiz):
        if i not in andreysAss:
            residual.append(i)
    for i in range(numWiz-len(andreysAss)):
        rand.shuffle(residual)
        andreysAss.append(residual.pop())
    return andreysAss

# Analyze statistics of constraicts and find pairs with higher correlation which are pre
def potential_neighbors(input_size, constraints, plot, thrsh):
    stats = np.zeros(input_size*input_size+1)
#    sum unique pair appearances in constraints
    for i in constraints:
        stats[i[0]*input_size+i[1]] += 1
    if plot:
        plt.plot(stats[stats!=0])
        plt.title("appearances")
        plt.show()
        print("greater then threshold neighbor appearance = " + str(len(stats[stats>thrs
        print
#    return pairs with a large number of appearances
    highPairs = [i for i, x in enumerate(stats>thrsh) if x]
    potential_neighbors = []
    for i in highPairs:
        potential_neighbors.append( (i//input_size,i%input_size) )
    return potential_neighbors
```

1.4 Relic department

```
In [5]: #Hill climbing solver
def solve(wizs, constraints):
    numWiz = len(wizs)
    numCons = len(constraints)
    ass = getAssigment(numWiz, wizs)
    conflicts = h(ass, constraints)
    while conflicts > 0:
        best_score = conflicts
        for i in range(numWiz):
```

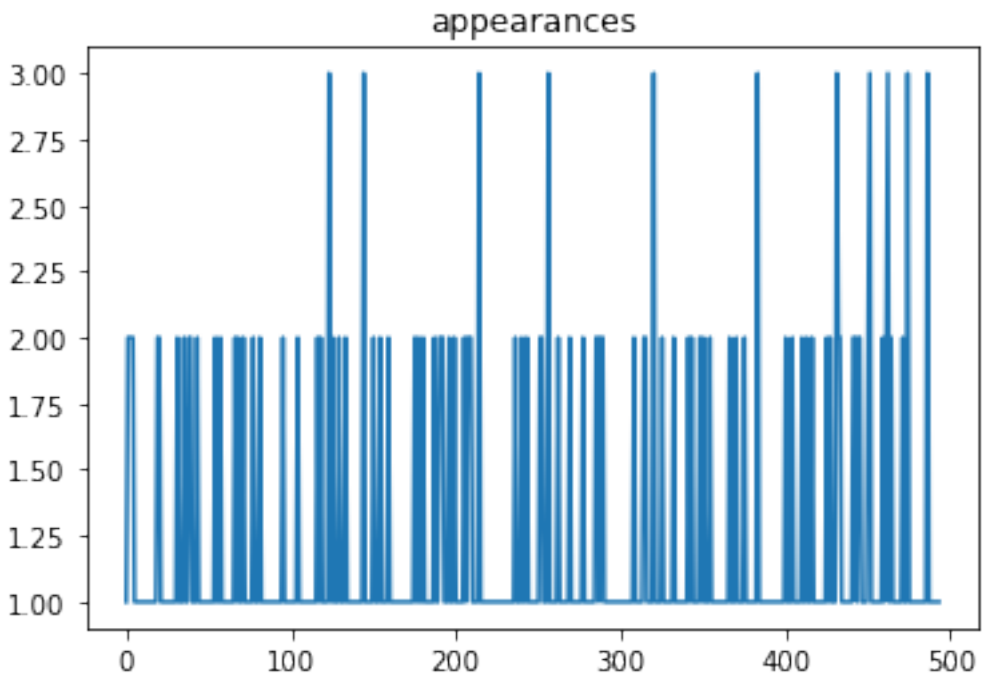
```
                for j in range(i+1,numWiz):
                    x = ass[wizs[i]]
                    y = ass[wizs[j]]
                    ass[wizs[i]] = y
                    ass[wizs[j]] = x

                    newScore = h(ass, constraints)
                    if newScore == 0:
                        return ass
                    if newScore < best_score:
                        best_ass = ass.copy()
                        best_score = newScore

                    ass[wizs[i]] = x
                    ass[wizs[j]] = y
            if best_score >= conflicts:
                ass = getAssigment(numWiz, wizs)
                conflicts = h(ass, constraints)
            else:
                conflicts = best_score
                ass = best_ass.copy()

    # Returns a random assigment
    def getAssigment(numWiz, wizs):
        indecies = list(range(1,numWiz+1))
        rand.shuffle(indecies)
        ass = {wizs[i]:indecies[i] for i in range(len(wizs))}
        return ass
```

1.5 Running the code

Read input and display statistics

```
In [ ]: %%time
    numWiz, numCons, wizs, constraints = readInput('Staff_Inputs/staff_60.in')
    start = wizs.copy()
    rand.shuffle(start)
    print("number of wizards: " + str(numWiz))
    print('initial number of conflicts:', h(start, constraints))
    gs = goodStart(numWiz, wizs, constraints,plot=True)
```

```
number of wizards: 60
initial number of conflicts: 209
```




```
greater then threshold neighbor appearance = 95
CPU times: user 352 ms, sys: 42.7 ms, total: 394 ms
Wall time: 430 ms
```

Run parallelized code

```
In [ ]: %%time
        cores = 1
        partial_aneal = partial(anneal, wizs, constraints)
        p = Pool(cores)
        starts = [goodStart(numWiz, wizs, constraints) for _ in range(1)]
        results = p.map(partial_aneal, starts)
```

```
0 conflicts - verifying...
Success!
```