# CS 170
# Fall 2017
# Project Write-Up

**Approach**

We used a heuristic local-search algorithm with some random elements to solve this problem. The basic local-search strategy was to start at some random ordering of the wizards and then check which flipping of the wizards would result in the largest reduction of violated constraints. If we got stuck in a local minimum then we would "kick" the ordering by making a random change in an effort to leave the local minimum. The general procedure is as follows:
Repeat until all constraints satisfied:

1. Iterate through a sample of potential two-wizard swaps, and check how many constraints are violated if that swap was to be performed. Because checking all $\binom{n}{2}$ swaps is expensive for higher $n$, we limit the algorithm to checking no more than $99^2$ potential wizard swaps by randomly preselecting a pool of swap-eligible wizards.
2. Store the 9 swaps that would result in the largest decrease of violated constraints.
3. If any of those 9 swaps leads to an decrease, execute it.
   If not, select 1 arbitrary violated constraint and swap the wizards involved to satisfy it. (this is the kick)

The values (99, 9, 1) were hand-picked. For optimal results on different inputs changes may be needed.

**Optimizations**

· Instead of checking every single constraint every iteration, we instead stored which wizards were involved in which constraints. Upon making a swap we only update the constraints that the wizards in the swap were involved in.
· Following Mudit's advice on Piazza we implemented our algorithm to be able to be terminated early and write the best found ordering so far (not necessarily the one currently being tested) to the output file.
· We found that usually our algorithm reached an ordering with few conflicts quickly but would bounce around at low conflict orderings for a while. So a technique we used was to terminate and restart the algorithm a few times and save different approximate solutions. Then we could run the algorithm starting from these orderings instead of a random ordering and would sometimes get lucky that the approximate solution was close to an optimal solution.
· This restarting from an ordering also allowed for the algorithm to be interrupted, modified and then run again without losing any of the work it did from before the interruption. We could also tweak the values from earlier.
· We ran our code in PyPy, an alternative implementation of Python with a Just-In-Time compiler which gave a speedup over standard Python.

**How to Run Algorithm**

Requires a PyPy installation to run at full speed. This can be obtained from BitBucket.(Note we ran this is on linux)

```
wget https://bitbucket.org/pypy/pypy/downloads/pypy3-v5.9.0-linux64.tar.bz2
tar xf pypy3-v5.9.0-linux64.tar.bz2
```

Install PyPy into the same directory as the solver, and replace all `python3` commands with `./pypy3-v5.9.0-linux64/bin/pypy3`.
Our algorithm has two required arguments and a third optional one. The first is the input file that defines the constraints. The second is the output file where the best ordering at the time of termination is written to. The third optional argument is the starting ordering. It is invoked with `-s` before it. If this argument is left blank, the algorithm starts from a random ordering. Here's an example run

```
python3 solver2_optimized.py Staff_Inputs/staff_60.in outputs/staff_60.out
```

This runs the algorithm on the input file `staff_60.in` from the folder `Staff_Inputs` and writes to the file `staff_60.out` in the folder `outputs`. Say we terminate this run early and want to restart from where we left off. We can run

```
python3 solver2_optimized.py Staff_Inputs/staff_60.in outputs/staff_60.out -s outputs/staff_60.out
```

In practice we were a little uncomfortable reading from and writing to the same file so we usually re-named our approximations e.g. `staff_60_approx.out` so a more typical run from an approximation would look like

```
python3 solver2_optimized.py Staff_Inputs/staff_60.in outputs/staff_60.out -s outputs/staff_60_approx.out
```

```
import argparse
import random
import heapq
import itertools
from itertools import combinations

# for use when keyboard interrupt.
best_state = (float("inf"), None)

def solve(num_wizards, num_constraints, wizards, constraints):
    global best_state

    def is_conflict(constraint):
        wi, wj, wk = constraint
        i, j, k = wizard_index[wi], wizard_index[wj], wizard_index[wk]
        return (k > i and k < j) or (k < i and k > j)

    def swap(i, j):
        """Swaps wizard i with wizard j in the ordering
        and updates conflicts accordingly."""
        nonlocal conflicts

        wizard_index[wizards[i]], wizard_index[wizards[j]] = wizard_index[wizards[j]],
            wizard_index[wizards[i]]

        wizards[i], wizards[j] = wizards[j], wizards[i]

        # for each constraint that involves wizard i or wizard j
        for constraint in set.union(wiz_to_constraints[wizards[i]], wiz_to_constraints[wizards[j]]):
            if constraint_states[constraint] and not is_conflict(constraint):
                # if constraint was violated but is not anymore
                conflicts -= 1
                constraint_states[constraint] = False # set state to false
            if not constraint_states[constraint] and is_conflict(constraint):
                # if constraint was not violated but now it is
                conflicts += 1
                constraint_states[constraint] = True

    def kick(strength=1):
        for i in range(strength):
            conflicts = [constraint for constraint in constraints if is_conflict(constraint)]
            for wa, wb, wc in [random.choice(conflicts)]:
                # in a b c swap either a c or b c
                a, b, c = wizard_index[wa], wizard_index[wb], wizard_index[wc]
                if random.random() < 0.5: swap(a, c)
                else: swap(b, c)
        return


    def successors(num_conflicts):
        """Generator for the successor states.
        Does not actually return successors to save on space.
        Returns a sequence of swaps."""
        swap_cap = 99

        i_s = random.sample(range(num_wizards), min(swap_cap, num_wizards))
        j_s = random.sample(range(num_wizards), min(swap_cap, num_wizards))
        for i, j in itertools.product(i_s, j_s):
```

```
            yield ((i, j),)


    wizard_index = {wizard: i for i, wizard in enumerate(wizards)}

    constraint_states = {constraint: is_conflict(constraint) for constraint in constraints}

    wiz_to_constraints = {w: set() for w in wizards}
    # map from wizard name to set of constraints it's involved in
    for constraint in constraints:
        for w in constraint:
            wiz_to_constraints[w].add(constraint)

    conflicts = sum(c for c in constraint_states.values())

    for _ in itertools.count():

        print("=============")
        print("iteration", _)
        print("conflicts", conflicts)

        if 0 == conflicts:
            return wizards
        if conflicts <= best_state[0]:
            best_state = (conflicts, wizards.copy())

        n = 9 if conflicts >= 10 else 29
        least_conflicts = [(float("-inf"), None)] * n

        for swaps in successors(conflicts):
            for i, j in swaps: swap(i, j)

            new_conflicts = conflicts

            # terminate early
            if 0 == conflicts:
                best_state = (0, wizards.copy())
                return wizards

            # undo changes
            for i, j in reversed(swaps): swap(i, j)

            heapq.heappushpop(least_conflicts, (-new_conflicts, swaps))

        if all(-conflicts == t[0] for t in least_conflicts):
            # if stagnant then kick
            kick(strength=1)
            print("kicked at conflicts", conflicts)
        else:
            # commit to swaps
            _, swaps = random.choice(least_conflicts)
            for i, j in swaps: swap(i, j)
    return wizards

def read_input(filename):
    with open(filename) as f:
        num_wizards = int(f.readline())
        # wizards = f.readline().split()
        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(set(wizards))
    random.shuffle(wizards)
    constraints = [tuple(constraint) for constraint in constraints]
    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))

#return order of the wizards from the order given in filename.out
#state is a list of strings where each string corresponds to the name of the wizard
#the position of the string in the list corresponds to the position of the wizard in the ordering
def read_start_state(filename):
    with open(filename) as f:
        state = f.readline().split() #separates by whitespace
    return state


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")
    parser.add_argument("-s", "--start_state_file", type=str, help = "___.out") #optional
    #need to type -s before argument in command line
    args = parser.parse_args()

    num_wizards, num_constraints, wizards, constraints = read_input(args.input_file)

    if (args.start_state_file != None):
        wizards = read_start_state(args.start_state_file)

    try:
        solution = solve(num_wizards, num_constraints, wizards, constraints)
    except KeyboardInterrupt:
        print("instance halted early.")
        solution = best_state[1]

    write_output(args.output_file, solution)
    print("best state below.")
    print(best_state)
    print("solution:", solution)
```