170 Wizards CSP Report

We implement modified version of local search which is most similar to “hill climbing” but with a few differences.

First we get a list of wizards and get the total number of violated constraints. We go through each wizard and his restrictions and add them to total.Then while the total number of constraints violated is greater than zero, on every iteration of while loop we sort the list in such a way that the first element in the list is the wizard who has the most of its constraints violated. Then with probability of about 50% we choose a random wizard from sorted list and with probability 50% we choose the wizard who has the most of its constraints violated. After we chose a wizard we apply the greedy logic and insert the wizard in the position that minimizes the total number of constraints violated for entire list. Additionally every time we pick a wizard and move into different position and it does not reduce the number of total violated constraints we increase our counter by 1. When we tried 100 insertions and total number violations stayed the same then we randomly pick a wizard and put him in a random position. If that does not help to avoid being stuck at local minimum and we tried some more times (in our case 500 attempts to reduce number of violations) then we simply reshuffle the list and starting all over again.

solver.py

```
import random
import utils
import datetime
import time
import multiprocessing
from multiprocessing import Pool
import sys
import os
import math


def place_in_best_location(violations, wizard, w, constraint_map):
    """
    Places a wizard in the location causing
    the least amount of constraint violations

    Input:
        violations: Number of constraint violations to beat
        wizard: Wizard we're finding a better place for
        wizards: Current ordering of the wizards
        constraint_map: Wizard names mapped to a list of their constraits

    Output:
        best_cur_violations: Number of violations after the move
    """
    wizards = w[:]

    best_cur_violations = violations
    best_j = wizards.index(wizard)
    wizards.remove(wizard)
    wizards = [wizard] + wizards

    for j in range(len(wizards) - 1):
        temp_violations = utils.check_total_violations(wizards, constraint_map)
        if temp_violations <= best_cur_violations:
            best_cur_violations = temp_violations
            best_j = j
        wizards[j], wizards[j + 1] = wizards[j + 1], wizards[j]
    wizards.pop()
    wizards.insert(best_j, wizard)
    return best_cur_violations, wizards


def place_in_random_location(wizard, wizards, constraint_map):
    random_i = random.randrange(0, len(wizards) - 1)
    random_j = random.randrange(0, len(wizards) - 1)

    wizards[random_i], wizards[random_j] = wizards[random_j], wizards[random_i]
    violations = utils.check_total_violations(wizards, constraint_map)

    return violations, wizards

def solve(wizards, constraints, event, best_so_far_file):
    """
    With a 55%, 45% chance, a random wizard or
    the most constrained wizard is chosen, respectively.
    That wizard is then placed in a position that violates
    the least amount of constraints. If the violations
    have not improved in at least 100 iterations, the chosen
    wizard is placed into a random location. If it has
    not improved in 500 iterations, the ordering is shuffled
    and it begins again.

    Input:
        wizards: Number of constraint violations to beat
        constraints: Constraints from inputfile
        event: Multithreading event, when one core finds
               A solution and event.set() is called, they
               all stop and move on to the next input
        best_so_far_file: name of the file containing the
                          best ordering found so far
    Output:
        wizards: A valid ordering of the wizards
    """
    constraint_ordering = wizards[:]
    constraint_map = utils.get_constraint_map(constraints)
```

```
    violations = utils.check_total_violations(wizards, constraint_map)

    count = 0

    while violations > 0:

        starting_violations = violations

        # Choose a random wizard or the most constrained wizard
        random_or_most_constrained_val = random.randrange(0, 100)
        if random_or_most_constrained_val < 55:
            wizard = random.choice(constraint_ordering)
        else:
            constraint_ordering = utils.sort_wizards(wizards, constraint_map)
            wizard = constraint_ordering[0]

        violations, wizards = place_in_best_location(violations, wizard, wizards, constraint_map)

        if starting_violations == violations:
            count += 1
            if count >= 500:
                count = 0
                random.shuffle(wizards)
                violations = utils.check_total_violations(wizards, constraint_map)

                # print("Stuck at " + str(violations) + " violations")
                # print(wizards)
            elif count >= 100:
                wizard = random.choice(constraint_ordering)
                violations, wizards = place_in_random_location(wizard, wizards, constraint_map)
        else:
            utils.check_best_violations(violations, wizards, best_so_far_file)
            count = 0

    event.set()
    return wizards



def run_inputs(event, input_file, output_file, best_so_far_file):
    print("\nBeginning " + input_file)
    num_wizards, num_constraints, wizards, constraints = utils.read_input(input_file)
    solution = solve(wizards, constraints, event, best_so_far_file)
    print("\nFound Solution")
    print(solution)
    utils.write_output(output_file, solution)


def multi_process(input_file, output_file, best_so_far_file):

    cpus_to_use = multiprocessing.cpu_count()


    p = multiprocessing.Pool(cpus_to_use)
    m = multiprocessing.Manager()
    event = m.Event()

    for _ in range(cpus_to_use):
        p.apply_async(run_inputs, (event, input_file, output_file, best_so_far_file))
    p.close()

    event.wait()
    p.terminate()


def get_phase_2_file_names(num_wizards, file_num):
    input_file = 'phase2_inputs/inputs' + num_wizards + '/input' + num_wizards + '_' + file_num + '.in'
    output_file = 'phase2_inputs/inputs' + num_wizards + '/output' + num_wizards + '_' + file_num + '.out'
    best_so_far_file = 'phase2_inputs/inputs' + num_wizards + '/input' + num_wizards + '_' + file_num + '_best_so_far' + '.in'
    return input_file, output_file, best_so_far_file


def phase_2():
    """
    Runs the program on the phase_2 input files.
    e.g. multi_process("20", 0) runs on input20_0.in
    """

    # Full list of inputs
```

```
    # Solvable
    to_do_list_20 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    to_do_list_35 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    to_do_list_50 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    for file_num in to_do_list_20:
        input_file, output_file, best_so_far_file = get_phase_2_file_names("20", str(file_num))
        multi_process(input_file, output_file, best_so_far_file)

    for file_num in to_do_list_35:
        input_file, output_file, best_so_far_file = get_phase_2_file_names("35", str(file_num))
        multi_process(input_file, output_file, best_so_far_file)

    for file_num in to_do_list_50:
        input_file, output_file, best_so_far_file = get_phase_2_file_names("50", str(file_num))
        multi_process(input_file, output_file, best_so_far_file)


def staff_inputs_all_cores_each_input(to_do_list):

    for n in to_do_list:
        input_file = 'Staff_Inputs/staff_' + str(n) + '.in'
        output_file = 'Staff_Inputs/staff_' + str(n) + '.out'
        best_so_far_file = 'Staff_Inputs/staff_' + str(n) + '_best_so_far' + '.in'
        multi_process(input_file, output_file, best_so_far_file)


def run_staff_inputs_one_per_core(n):
    input_file = 'Staff_Inputs/staff_' + str(n[0]) + '.in'
    output_file = 'Staff_Inputs/staff_' + str(n[0]) + '.out'
    best_so_far_file = 'Staff_Inputs/staff_' + str(n[0]) + '_best_so_far' + '.in'

    run_inputs(n[1], input_file, output_file, best_so_far_file)
    return ("Finished")


def staff_inputs_one_per_core(to_do_list):
    m = multiprocessing.Manager()
    event = m.Event()
    inputs = [(x, event) for x in to_do_list]

    number_processes = multiprocessing.cpu_count()

    with Pool(number_processes) as p:
        reslist = [p.apply_async(run_staff_inputs_one_per_core, (n,)) for n in inputs]
        for result in reslist:
            print(result.get())


def student_inputs():
    input_directory = os.fsencode("all_submissions/inputs")

    output_directory = os.fsencode("all_submissions/outputs")

    output_files = set()

    for file in os.listdir(output_directory):
        filename = os.fsdecode(file)
        filename = filename[:filename.index(".")]
        output_files.add(filename)

    input_files = []
    for file in os.listdir(input_directory):
        filename = os.fsdecode(file)

        filename = filename[:filename.index(".")]

        if filename not in output_files:
            input_files.append(filename)

    for filename in input_files:
        input_file = 'all_submissions/inputs/' + filename + '.in'
        output_file = 'all_submissions/outputs/' + filename + '.out'
        best_so_far_file = 'all_submissions/best_so_far_files/' + filename + '.out'
        multi_process(input_file, output_file, best_so_far_file)


if __name__ == "__main__":

    # phase_2()
```

```
to_do_list = [140, 160, 180, 200, 220, 240, 260, 280, 300, 320, 340, 360, 380, 400]
staff_inputs_all_cores_each_input(to_do_list)
# staff_inputs_one_per_core(to_do_list)
# student_inputs()
```

utils.py

```
"""
utils.py
"""

def check_best_violations(violations, wizards, best_so_far_file):
    """
    Checks the current wizard ordering against a file
    containing the best ordering we've seen so far
    and the number of constraints it violates. Has to
    be in a file because the program is running on
    multiple cores.

    Input:
        violations: Violations the wizard ordering has
        wizards: The wizard ordering
        best_so_far_file: name of the file containing the
                          best ordering found so far
    """
    try:
        with open(best_so_far_file) as f:
            best_violations = int(f.readline().split()[0])
            if violations < best_violations:
                print("Best violations updated: " + str(violations))
                best_list = [str(violations)] + wizards
                utils.write_output(best_so_far_file, best_list)
    except:
        best_list = [str(violations)] + wizards
        write_output(best_so_far_file, best_list)


def check_wizard_violations(ordered_wizards, constraint_map, wizard):
    wizard_index = ordered_wizards.index(wizard)

    if wizard_index == 0 or wizard_index == len(ordered_wizards) - 1:
        return 0

    violations = 0
    prev_wizards = set(ordered_wizards[:wizard_index])
    next_wizards = set(ordered_wizards[wizard_index + 1:])

    if wizard in constraint_map:
        cur_constraints = constraint_map[wizard]
        for constraint in cur_constraints:
            wizard1 = constraint[0]
            wizard2 = constraint[1]

            if wizard1 in prev_wizards and wizard2 in next_wizards:
                violations += 1

            elif wizard2 in prev_wizards and wizard1 in next_wizards:
                violations += 1

    return violations


def sort_wizards(ordered_wizards, constraint_map):
    wizard_tuples = []
    for wizard in ordered_wizards:
        wizard_violations = check_wizard_violations(ordered_wizards, constraint_map, wizard)
        wizard_tuples.append((wizard, wizard_violations))

    wizard_tuples.sort(key=lambda tup: tup[1])
```

```
    sorted_wizards = []
    for wizard_tup in wizard_tuples:
        sorted_wizards.append(wizard_tup[0])
    return sorted_wizards


def get_constraint_map(constraints):
    """
    Returns a mapping of the wizards
    to a list of their constraints
    Input:
        constraints: List of al the constraints

    Output:
        constraint_map: Wizard names mapped to a list of their constraits
    """
    constraint_map = {}
    for constraint in constraints:
        wizard = constraint[2]
        if wizard not in constraint_map:
            constraint_map[wizard] = [constraint[:2]]
        else:
            constraint_map[wizard].append(constraint[:2])
    return constraint_map


def check_total_violations(ordered_wizards, constraint_map):
    """
    Checks how many violations are present
    in the current wizard ordering

    Input:
        ordered_wizards: Current ordering of the wizards
        constraint_map: Wizard names mapped to a list of their constraits

    Output:
        violations: Number of violations
    """
    violations = 0
    prev_wizards = set(ordered_wizards[:1])
    next_wizards = set(ordered_wizards[1:])

    for i in range(1, len(ordered_wizards) - 1):
        cur_wizard = ordered_wizards[i]
        next_wizards.remove(cur_wizard)

        if cur_wizard in constraint_map:
            cur_constraints = constraint_map[cur_wizard]
            for constraint in cur_constraints:
                wizard1 = constraint[0]
                wizard2 = constraint[1]

                if wizard1 in prev_wizards and wizard2 in next_wizards:
                    violations += 1

                elif wizard2 in prev_wizards and wizard1 in next_wizards:
                    violations += 1

        prev_wizards.add(cur_wizard)
    return violations


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))
```