COMP3070 – Coursework 2 Sample Guidance 4th November, 2025.

Python code

```
!pip install ortools

!pip install z3-solver
from z3 import*

from time import time as currenttime
from ortools.sat.python import cp_model
import re
import numpy

# helper function provided in Moodle
def transform_output(d):
    crlf = '\r\n'
    s = []
    s = ''.join(kk + crlf for kk in d['sol'])
    s = d['sat']+crlf+s+d['mul_sol']
    s = crlf + s + crlf+str(d['exe_time']) if 'exe_time' in d else s
    return s
class VarArraySolutionPrinter(cp_model.CpSolverSolutionCallback):
    """Print intermediate solutions."""

    def __init__(self, variables, limit):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__variables = variables
        self.__solution_count = 0
        self.__solution_limit = limit

    def on_solution_callback(self):
        self.__solution_count += 1

        for v in self.__variables:
            for user in range(len(v)):
                if self.Value(v[user]) == 1:
                    print(f'[{v[user]}]', end=' ')

        print()

        if self.__solution_count >= self.__solution_limit:
            print(f'\nStop searching after {self.__solution_limit} solutions found')
            self.StopSearch()

    def solution_count(self):
        return self.__solution_count


class Instance:
    def __init__(self):
        self.number_of_steps = 0
```

```
  self.number_of_users = 0
self.number_of_constraints = 0
self.auth = []          # index: the user || element: steps authorised to the user
self.SOD = []         # the list of pairs of steps that must be assigned to the different users
self.BOD = []         # the list of pairs of steps that must be assigned to the same user
self.at_most_k = []     # the list of pairs of k and steps
self.one_team = []      # the list of pairs of steps and teams

def read_file(filename):
    def read_attribute(name):
        line = f.readline()
        match = re.match(f'{name}:\\s*(\\d+)$', line)
        if match:
            return int(match.group(1))
        else:
            raise Exception("Could not parse line {line}; expected the {name} attribute")

    instance = Instance()

    with open(filename) as f:
        instance.number_of_steps = read_attribute("#Steps")
        instance.number_of_users = read_attribute("#Users")
        instance.number_of_constraints = read_attribute("#Constraints")

        # initialise instance.auth with empty lists as elements
        instance.auth = [[] for u in range(instance.number_of_users)]

        for i in range(instance.number_of_constraints):
            l = f.readline()

            # 1st Constraint: Authorisations
            m = re.match(r"Authorisations u(\d+)(?: s\d+)*", l)
            if m:
                user_id = int(m.group(1))
                steps = [-1]  # for users that are not authorised to perform any steps eg. Authorisations u1
                for m in re.finditer(r's(\d+)', l):
                    if -1 in steps:
                        steps.remove(-1)  # if user has specified steps, then only store the steps authorised
                    steps.append(int(m.group(1)) - 1)  # -1 cuz list index starts from 0
                instance.auth[user_id - 1].extend(steps)
                continue

            # 2nd Constraint: Separation-of-duty
            m = re.match(r'Separation-of-duty s(\d+) s(\d+)', l)
            if m:
                steps = (int(m.group(1)) - 1, int(m.group(2)) - 1)
                instance.SOD.append(steps)
                continue
            # 3rd Constraint: Binding-of-duty
            m = re.match(r'Binding-of-duty s(\d+) s(\d+)', l)
            if m:
                steps = (int(m.group(1)) - 1, int(m.group(2)) - 1)
                instance.BOD.append(steps)
                continue

            # 4th Constraint: At-most-k
            m = re.match(r'At-most-k (\d+) (s\d+)(?: (s\d+))*', l)
            if m:
                k = int(m.group(1))
                steps = []
                for m in re.finditer(r's(\d+)', l):
                    steps.append(int(m.group(1)) - 1)
                instance.at_most_k.append((k, steps))
                continue

            # 5th Constraint: One-team constraint
            m = re.match(r'One-team\s+(s\d+)(?: s\d+)* (\((u\d+)*\))*', l)
            if m:
                steps = []
                for m in re.finditer(r's(\d+)', l):
                    steps.append(int(m.group(1)) - 1)

                teams = []
                for m in re.finditer(r'\((u\d+\s*)+\)', l):
                    team = []
                    for users in re.finditer(r'u(\d+)', m.group(0)):
                        team.append(int(users.group(1)) - 1)
                    teams.append(team)

                instance.one_team.append((steps, teams))

                continue

            else:
                raise Exception(f'Failed to parse this line: {l}')  #

    return instance


def Solver(instance, filename, **kwargs):
    '''
    :param filename:
    The constraint path
    :param kwargs:
    As you wish, you may supply extra arguments using the kwargs
    :return:
    A dict.
    '''
    # Printing (accessing or output) values from test instances
    print("======================================================")
    print(f'\tFile: {filename}')
    print(f'\tNumber of Steps: {instance.number_of_steps}')
    print(f'\tNumber of Users: {instance.number_of_users}')
    print(f'\tNumber of Constraints: {instance.number_of_constraints}')
    print(f'\tAuthorisations: {instance.auth}')
    print(f'\tSeparation-of-duty: {instance.SOD}')
    print(f'\tBinding-of-duty: {instance.BOD}')
    print(f'\tAt-most-k: {instance.at_most_k}')
    print(f'\tOne-team: {instance.one_team}')
    print("======================================================")

    ''' Start of Solver '''
    model = cp_model.CpModel()

    user_assignment = [[model.NewBoolVar(f's{s + 1}: u{u + 1}') for u in
range(instance.number_of_users)]
                       for s in range(instance.number_of_steps)]

    # each step is assigned to exactly one user
    for step in range(instance.number_of_steps):
        model.AddExactlyOne(user_assignment[step][user] for user in
range(instance.number_of_users))
    '''
    ---------- Authorisations constraint: ------------------------------------------
    if user u is not authorised to step s, then step s cannot be assigned to user u
    '''
    # apply constraint ONLY to users with specified authorisations
    for user in range(instance.number_of_users):
        if instance.auth[user]:
            for step in range(instance.number_of_steps):
                if step not in instance.auth[user]:
                    model.Add(user_assignment[step][user] == 0)
    '''
    ---------- Separation-of-duty constraint: ------------------------------------------
    separated steps must be assigned to the different user
    '''
    for (separated_step1, separated_step2) in instance.SOD:
        for user in range(instance.number_of_users):
            model.Add(user_assignment[separated_step2][user] ==
0).OnlyEnforceIf(user_assignment[separated_step1][user])

    '''
    ---------- Binding-of-duty constraint: ------------------------------------------
    bound steps cannot be assigned to the same user
    '''
    for (bound_step1, bound_step2) in instance.BOD:
        for user in range(instance.number_of_users):
            model.Add(user_assignment[bound_step2][user] ==
1).OnlyEnforceIf(user_assignment[bound_step1][user])

    '''
    ---------- At-most-k constraint: -----------------------------------------
    number of users assigned to the group of steps must not be greater than k
    '''
    # takes long time to solve when number of users is large
    # '''
    for (k, steps) in instance.at_most_k:
        # print(f'{k} --- {steps}')
        user_assignment_flag = [model.NewBoolVar(f'at-most-k_u{u}') for u in
range(instance.number_of_users)]
        for user in range(instance.number_of_users):
            for step in steps:
                model.Add(user_assignment_flag[user] ==
1).OnlyEnforceIf(user_assignment[step][user])
            model.Add(sum(user_assignment[step][user] for step in steps) >=
user_assignment_flag[user])
        model.Add(sum(user_assignment_flag) <= k)
    # '''
# much longer time in large instances (eg. Examples 16 and 17) because more variables
and more constraints are declared

    '''
    ---------- One-team constraint: -----------------------------------------
    steps can only be assigned to team with flag = 1,
    steps cannot be assigned to users with no team
    '''
    for (steps, teams) in instance.one_team:
        team_flag = [model.NewBoolVar(f'team{t}') for t in range(len(teams))]
        model.AddExactlyOne(team_flag)  # only one team can be chosen
        for team_index in range(len(teams)):
            for step in steps:
                for user in teams[team_index]:
                    model.Add(user_assignment[step][user] ==
0).OnlyEnforceIf(team_flag[team_index].Not())

        # steps cannot be assigned to users that is not listed in teams too
        users_in_teams = list(numpy.concatenate(teams).flat)
        for step in steps:
            for user in range(instance.number_of_users):
                if user not in users_in_teams:
                    model.Add(user_assignment[step][user] == 0)

    ''' End of Solver '''

    starttime = float(currenttime() * 1000)
    solver = cp_model.CpSolver()
    solution_printer = VarArraySolutionPrinter(user_assignment, 1000)
    solver.parameters.enumerate_all_solutions = True
    # solver.parameters.max_time_in_seconds = 10.0
    status = solver.Solve(model, solution_printer)
    endtime = float(currenttime() * 1000)

    d = dict(
        sat='unsat',
        sol='',
        mul_sol='',
        exe_time=str(endtime - starttime) + 'ms'
    )
    if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
        d['sat'] = 'sat'
        dsol = []

        for s in range(instance.number_of_steps):
            for u in range(instance.number_of_users):
                solver_value = solver.Value(user_assignment[s][u])
                if solver_value:
                    dsol.append(f's{s + 1}: u{u + 1}')

        d['sol'] = dsol

        print('\nStatus: %s' % solver.StatusName(status))
        print('Number of solutions found: %i' % solution_printer.solution_count())
        if solution_printer.solution_count() > 1:
            d['mul_sol'] = f'other solutions exist, {solution_printer.solution_count()} solutions found'
        else:
            d['mul_sol'] = "this is the only solution"
        return d
    else:
        print('\nStatus: %s' % solver.StatusName(status))
        return d

if __name__ == '__main__':
    dpath = 'instances/example14.txt'
    inst = read_file(dpath)
    d = Solver(inst, dpath, silent=False)
    s = transform_output(d)
    print(s)
```

------------- End of **CW2** Sample Guidance -------------