ICS 381: Principles of AI – 242
PA 2 – Programming Instructions

General Helpful Tips:

- Do not copy others' work. You can discuss general approaches with students, but do not share specific coding solutions.

- **NOTE1:** For this homework, we will be using itertools so import it.
- **NOTE2:** For this homework, we will be using copy so import it as well.
- **NOTE3:** For this homework, we will be using numpy and chess packages.
- **NOTE4:** Be sure to implement the functions as specified in this document. In addition, you can implement any extra helper functions as you see fit with whatever names you like.

- Submit the required files only: [**csp_scheduler.py, games.py**]

Autograder Note: the autograder on gradescope will take around **100 seconds** to complete.

Implementing CSP Problems

csp_scheduler.py: Implement class for SchedulerCSP

On blackboard you are given **backtracking.py** which contains backtracking search and ac3 implementation. So, in this programming assignment, you will just implement a CSP problem.

Consider a course scheduling problem that we will model as a CSP in **csp_scheduler.py**. There are N courses for which a professor, location, and start-time need to be assigned. For locations, we will have a dictionary of capacity info. For courses, we will have a dictionary for course info where the keys are the course names and values is a list of course info properties: list of preferred professors, number of students, duration, and list of courses that must come before in the schedule.

`loc_info_dict` is location info dictionary with format `{locationname: capacity}`

`course_info_dict` is course info dictionary with format
`{coursename: [list-barred-professors, student-count, duration, list-after-courses] }`

For our CSP modelling we will have:

- Variables: the courses $C_1, C_2, \dots, C_N$
- Domains: courses are assigned a tuple value $C_i = (prof, loc, start\_time)$ indicating professor, location, and start-time. Domain of these three is determined by constructor arguments.
- Constraints:
  - No two courses C_i and C_j can be assigned the same professor at same overlapping time. Also, no two courses can be assigned same location at same overlapping time. You can construct the Boolean logic for this using the 3-tuple value in C_i and C_j, and using the `duration` course info. Notice that this constraint makes the binary constraint graph fully-connected, we check during backtracking.
  - For any C_i that has non-empty `list-after-courses`, it must be assigned a time block that **ends before** the **start** of courses in `list-after-courses`. These are binary constraints that we check during backtracking.
  - For any C_i it must be assigned a location with capacity ≥ `student-count`. Notice that this is a unary constraint, which we will enforce on the domains in the constructor before backtracking search.
  - For any C_i it must be assigned a professor that is **not** from `list-barred-professors`. Notice that this is a unary constraint, which we will enforce on the domains in the constructor before backtracking search.

<table>
<tr><th colspan="2">SchedulerCSP Constructor</th></tr>
<tr><th>Constructor arguments</th><th>Constructor body</th></tr>
<tr><td><code>courses</code>,
<code>professors</code>,
<code>loc_info_dict</code>,
<code>course_info_dict</code>,
<code>time_slots</code></td><td>
<code>courses</code> is a python list of the N course names as strings.

<code>professors</code> is a python list of the M professor names as strings.

<code>loc_info_dict</code> is location info dictionary with format <code>{locationname: capacity}</code>

<code>course_info_dict</code> is course info dictionary with format
<code>{coursename: [list-barred-professors, student-count, duration, list-after-courses] }</code>

<code>time_slots</code> is a python list of available time slots $[0, 1, 2, \dots, maxT]$. A course $start\_time$ is assigned one of these values.

Add the above arguments to self.loc_info_dict, self. course_info_dict, self.time_slots

Then setup the following:
Set <code>self.variables</code> to <code>courses.</code>

<code>self.domains</code> is a dictionary of variable domain values for the N courses. The key should be the variable course-name and value is domain in form of list of tuples $[(prof, loc, start\_time), \dots]$. For each C_i you want to restrict its domain as follow:
• $prof$ should be restricted to be <code>professors</code> not-in <code>list-barred-professors</code> for C_i
• loc should be restricted to be locations for which capacity ≥ <code>student-count</code>
• $start\_time$ domain is the same as <code>time_slots</code>.
• Finally, now you can construct the variable domain as a list of all possible 3-tuples (cartesian product) of the above three. You can achieve this using itertools in python (see this link)

<code>self.adjacency</code> is a dictionary for constraint-graph. A keys is variable, and value is list of neighbor variables. For this problem, every variable is connected to all other variables. Just be sure not to add the same variable to its own adjacency list.
</td></tr>
</table>

<table>
<tr><th colspan="4"><code>SchedulerCSP Functions</code></th></tr>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Implementation hints/clarifications</th></tr>
<tr><td><code>constraint_consistent</code></td><td><code>var1, val1, var2, val2</code></td><td>Returns true if var1 and var2 do not violate a constraint.

val1 and val2 are the tuple assignment values. $(prof, loc, start\_time)$</td><td>Notice for this problem, all variables are connected to each other with a constraint. Some may also have the extra before-constraint.

Check the following:
• Check1: val1 and val2 should not be assigned same prof at same overlapping time, nor same loc at same overlapping time. Be careful with the Boolean logic for this. See text below table.
• Check2: If var2 is in var1's after-list, then check that var2's start-time comes strictly after var1's end-time. Also, check vice-versa if var1 is in var2's after-list. Note the code for this can be done in 4-5 lines. Also, it is not necessary that this check needs to be done; some variables have empty after-list.
• Based on the above two checks, return true\false accordingly.</td></tr>
</table>

Check1: It is easier to code this in following order:

- check-overlap. Can be done using following Boolean
  not(((start_time1 < start_time2) and (end_time1 <= start_time2)) or ((start_time2 < start_time1) and (end_time2 <= start_time1)))
  Where end_time of course is its start_time + duration.
- check-same-prof-same-overlap: same prof **and** overlapping-time.
- check-same-loc-same-overlap: same loc **and** overlapping-time.
- Check1 is then given by the following Boolean
  `not(check_same_prof_same_overlap or check_same_loc_same_overlap)`

`check_partial_assignment`	`assignment`	Returns true if the partial assignment is consistent.	`assignment` is a dictionary where the key is a variable and the value is a value. If assignment is None, then return False. Check for each variable in assignment that their **assigned neighbors** do not violate constraints. Use constraint_consistent as helper. Be sure to just check **assigned neighbors**. Ignore **unassigned neighbors**.
`is_goal`	`assignment`	Returns true if assignment is **complete** and **consistent**. Otherwise, false.	`assignment` is a dictionary where the key is a variable and the value is a value. If assignment is None, then return False. **Hint**: First check if assignment is consistent. Make use of check_partial_assignment. Then check if it is complete.

Test your code on test_scheduler.py

You can test your code by running test_scheduler.py. My implementation output can be found in test_scheduler.out