XCS221 Assignment 5 — Course Scheduling

Due Sunday, June 19 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at `http://xcs221-scpd.slack.com/`
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at `https://communitystandards.stanford.edu/policies-and-guidance/honor-code`.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

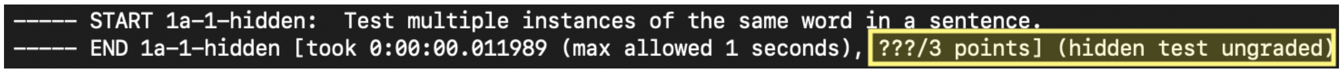
Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

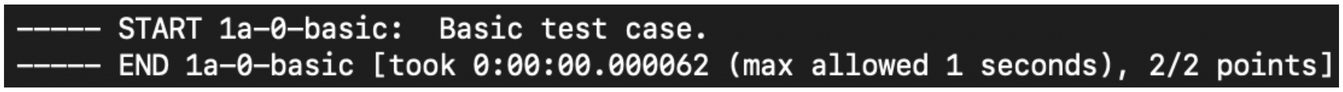
All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:



```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

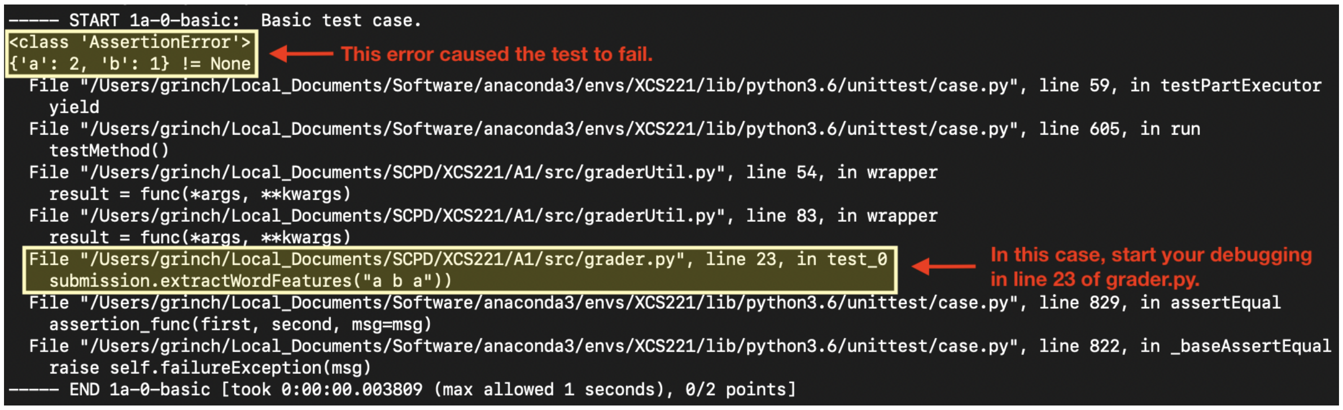
Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:



```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

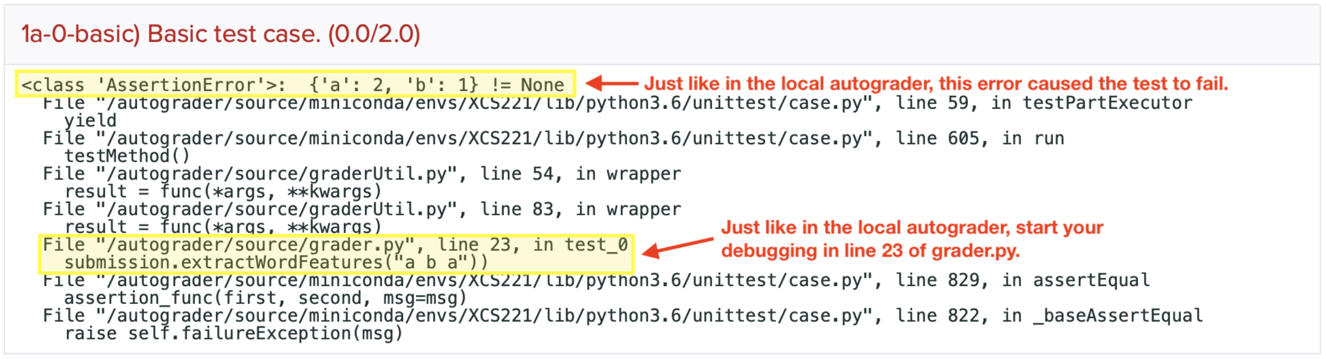


```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

This error caused the test to fail.
In this case, start your debugging in line 23 of grader.py.

Remote Execution

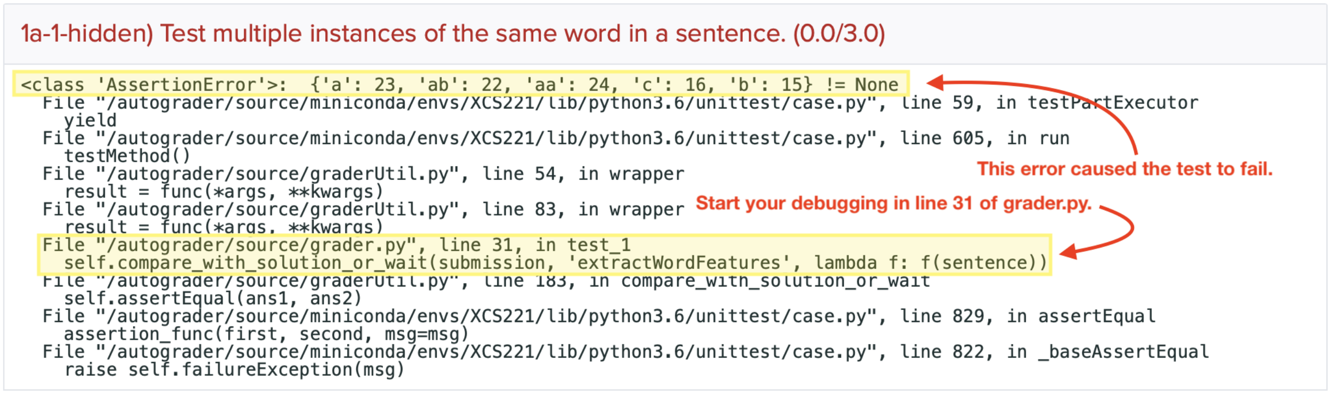
Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.



1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.
Just like in the local autograder, start your debugging in line 23 of grader.py.

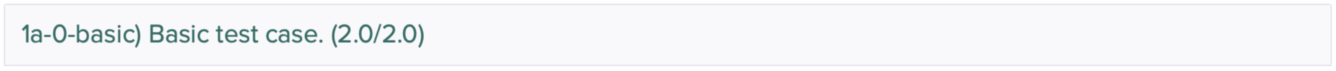


1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

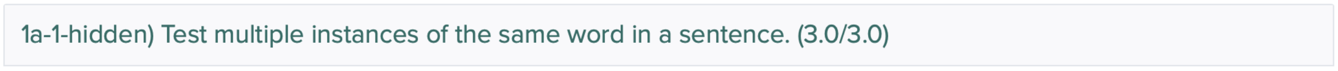
```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.
Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.



1a-0-basic) Basic test case. (2.0/2.0)



1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Dear Professional Students,

This assignment was originally created for matriculated Stanford students who regularly face the complicated task of balancing course schedules from semester to semester. We at the Stanford Center for Professional Development have chosen to retain the original content of this assignment to remain aligned with the graduate version of this course, CS221 (Artificial Intelligence: Principles and Techniques).

We encourage you to think about and discuss on Slack the broad applicability of CSPs beyond course scheduling. For example, manufacturing assembly lines, surgerical operating rooms, and sustainable farming all present important and very complex problems that can be tackled effectively using Constraint Satisfaction. It is incredibly useful!

Good luck and thanks for the hard work so far!

Sincerely,
The AI Course Development Team

Introduction

What courses should you take in a given quarter? Answering this question requires balancing your interests, satisfying prerequisite chains, graduation requirements, availability of courses; this can be a complex tedious process. In this assignment, you will write a program that does automatic course scheduling for you based on your preferences and constraints. The program will cast the course scheduling problem (CSP) as a constraint satisfaction problem (CSP) and then use backtracking search to solve that CSP to give you your optimal course schedule.

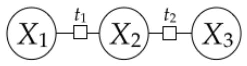
You will first get yourself familiar with the basics of CSPs in Problem 0. In Problem 1, you will implement two of the three heuristics you learned from the lectures that will make CSP solving much faster. In problem 2, you will add a helper function to reduce n-ary factors to unary and binary factors. Lastly, in Problem 3, you will create the course scheduling CSP and solve it using the code from previous parts.

0. **CSP Basics**

(a) **[3 points (Written)]** Let's create a CSP. Suppose you have n light bulbs, where each light bulb $i = 1, \dots, n$ is initially off. You also have m buttons which control the lights. For each button $j = 1, \dots, m$, we know the subset $T_j \subseteq \{1, \dots, n\}$ of light bulbs that it controls. When button j is pressed, it toggles the state of each light bulb in T_j (For example, if $3 \in T_j$ and light bulb 3 is off, then after the button is pressed, light bulb 3 will be on, and vice versa).

Your goal is to turn on all the light bulbs by pressing a subset of the buttons. Construct a CSP to solve this problem. Your CSP should have m variables and n constraints. *For this problem only*, you can use n-ary constraints. Describe your CSP precisely and concisely. You need to specify the variables with their domain, and the constraints with their scope and expression. Make sure to include T_j in your answer.

(b) **[3 points (Written)]** Let's consider a simple CSP with 3 variables and 2 binary factors:



where $X_1, X_2, X_3 \in \{0, 1\}$ and t_1, t_2 are XOR functions (that is $t_1(X) = x_1 \bigoplus x_2$ and $t_2(X) = x_2 \bigoplus x_3$).

i. How many consistent assignments are there for this CSP?

ii. To see why variable ordering is important, let's use backtracking search to solve the CSP *without using any heuristics (MCV, LCV, AC-3) or lookahead.* How many times will `backtrack()` be called to get all consistent assignments if we use the fixed ordering X_1, X_3, X_2? Draw the call stack for `backtrack()`.

(You should use the Backtrack algorithm from the slides. The initial arguments are $x = \emptyset$, $w = 1$, and the original Domain.) In the code, this will be `BacktrackingSearch.numOperations`.

iii. To see why lookahead can be useful, let's do it again with the ordering X_1, X_3, X_2 and AC-3. How many times will Backtrack be called to get all consistent assignments? Draw the call stack for `backtrack()`.

(c) **[2 points (Coding)]** Now let's consider a general case: given a factor graph with n variables $X_1, ..., X_n$ and $n - 1$ binary factors $t_1, ..., t_{n-1}$ where $X_i \in \{0, 1\}$ and $t_i(X) = x_i \bigoplus x_{i+1}$. Note that the CSP has a chain structure. Implement `create_chain_csp()` by creating a generic chain CSP with XOR as factors.

Note: We've provided you with a CSP implementation in `util.py` *which supports unary and binary factors. For now, you don't need to understand the implementation, but please read the comments and get yourself familiar with the CSP interface. For this problem, you'll need to use* `CSP.add_variable()` *and* `CSP.add_binary_factor()`*.*

1. CSP Solving

So far, we've only worked with unweighted CSPs, where $f_j(x) \in \{0, 1\}$. In this problem, we will work with weighted CSPs, which associates a weight for each assignment x based on the product of m factor functions $f_1, \ldots, f_m$:

$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

where each factor $f_j(x) \geq 0$. Our goal is to find the assignment(s) x with the **highest** weight. As in problem 0, we will assume that each factor is either a unary factor (depends on exactly one variable) or a binary factor (depends on exactly two variables).

For weighted CSP construction, you can refer to the CSP examples we have provided in `util.py` for guidance (`create_map_coloring_csp()` and `create_weighted_csp()`). You can try these examples out by running

```
python run_p1.py
```

Notice we are already able to solve the CSPs because in `submission.py`, a basic backtracking search is already implemented. Recall that backtracking search operates over partial assignments and associates each partial assignment with a weight, which is the product of all the factors that depend only on the assigned variables. When we assign a value to a new variable X_i, we multiply in all the factors that depend only on X_i and the previously assigned variables. The function `get_delta_weight()` returns the contribution of these new factors based on the `unaryFactors` and `binaryFactors`. An important case is when `get_delta_weight()` returns 0. In this case, any full assignment that extends the new partial assignment will also be zero, so *there is no need to search further with that new partial assignment.*

Take a look at `BacktrackingSearch.reset_results()` to see the other fields which are set as a result of solving the weighted CSP. You should read `submission.BacktrackingSearch` carefully to make sure that you understand how the backtracking search is working on the CSP.

(a) [**6 points (Coding)**] Let's create a CSP to solve the n-queens problem: Given an $n \times n$ board, we'd like to place n queens on this board such that no two queens are on the same row, column, or diagonal. Implement `create_nqueens_csp()` by **adding n variables** and some number of binary factors. Note that the solver collects some basic statistics on the performance of the algorithm. You should take advantage of these statistics for debugging and analysis. You should get 92 (optimal) assignments for $n = 8$ with exactly 2057 operations (number of calls to `backtrack()`).

Hint: If you get a larger number of operations, make sure your CSP is minimal. Try to define the variables such that the size of domain is $O(n)$.

Note: Please implement the domain of variables as 'list' type in Python (again, you may refer to `create_map_coloring_csp()` and `create_weighted_csp()` in `util.py` as examples of CSP problem implementations), so you can compare the number of operations with our suggestions as a way of debugging.

(b) [**6 points (Coding)**] You might notice that our search algorithm explores quite a large number of states even for the 8×8 board. Let's see if we can do better. One heuristic we discussed in class is using most constrained variable (MCV): To choose an unassigned variable, pick the X_j that has the fewest number of values a which are consistent with the current partial assignment (a for which `get_delta_weight()` on $X_j = a$ returns a non-zero value).

Implement this heuristic in `get_unassigned_variable()` under the condition `self.mcv = True`. It should take you exactly 1361 operations to find all optimal assignments for 8 queens CSP — that's 30% fewer!

Some useful fields:

- `csp.unaryFactors[var][val]` gives the unary factor value.
- `csp.binaryFactors[var1][var2][val1][val2]` gives the binary factor value. Here, `var1` and `var2` are variables and `val1` and `val2` are their corresponding values.
- In `BacktrackingSearch`, if `var` has been assigned a value, you can retrieve it using `assignment[var]`. Otherwise `var` is not in `assignment`.

(c) [**11 points (Coding)**] The previous heuristics looked only at the local effects of a variable or value. Let's now implement arc consistency (AC-3) that we discussed in lecture. After we set variable X_j to value a, we remove the values b of all neighboring variables X_k that could cause arc-inconsistencies. If X_k's domain has changed, we use X_k's domain to remove values from the domains of its neighboring variables. This is repeated until no domain can be updated. Note that this may significantly reduce your branching factor, although at some cost. In `backtrack()` we've implemented code which copies and restores domains for you. Your job is to fill in `arc_consistency_check()`.

You should make sure that your existing MCV implementation is compatible with your AC-3 algorithm as we will be using all three heuristics together during grading.

With AC-3 enabled, it should take you 769 operations only to find all optimal assignments to 8 queens CSP — That is almost 45% fewer even compared with MCV!

Take a deep breath! This part requires time and effort to implement — be patient.

Hint 1: documentation for `CSP.add_unary_factor()` and `CSP.add_binary_factor()` can be helpful.

Hint 2: although AC-3 works recursively, you may implement it iteratively. Using a queue might be a good idea. `li.pop(0)` removes and returns the first element for a python list `li`.

2. **Handling n-ary Factors**

So far, our CSP solver only handles unary and binary factors, but for course scheduling (and really any non-trivial application), we would like to define factors that involve more than two variables. It would be nice if we could have a general way of reducing n-ary constraint to unary and binary constraints. In this problem, we will do exactly that for two types of n-ary constraints.

Suppose we have boolean variables X_1, X_2, X_3, where X_i represents whether the i-th course is taken. Suppose we want to enforce the constraint that $Y = X_1 \vee X_2 \vee X_3$, that is, Y is a boolean representing whether at least one course has been taken. For reference, in `util.py`, the function `get_or_variable()` does such a reduction. It takes in a list of variables and a target value, and returns a boolean variable with domain `[True, False]` whose value is constrained to the condition of having at least one of the variables assigned to the target value. For example, we would call `get_or_variable()` with arguments $(X_1, X_2, X_3, \text{True})$, which would return a new (auxiliary) variable X_4, and then add another constraint $[X_4 = \text{True}]$.

The second type of n-ary factors are constraints on the sum over n variables. You are going to implement reduction of this type but let's first look at a simpler problem to get started:

(a) **[3 points (Written)]** Suppose we have a CSP with three variables X_1, X_2, X_3 with the same domain $\{0, 1, 2\}$ and a ternary constraint $[X_1 + X_2 + X_3 \leq K]$. How can we reduce this CSP to one with only unary and/or binary constraints? Explain what auxiliary variables we need to introduce, what their domains are, what unary/binary factors you'll add, and why your scheme works. Add a graph if you think that'll better explain your scheme.

Hint: draw inspiration from the example of enforcing $[X_i = 1$ for exactly one $i]$ which is in the first CSP lecture.

(b) **[7 points (Coding)]** Now let's do the general case in code: implement `get_sum_variable()`, which takes in a sequence of non-negative integer-valued variables and returns a variable whose value is constrained to equal the sum of the variables. You will need to access the domains of the variables passed in, which you can assume contain only non-negative integers. The parameter `maxSum` is the maximum sum possible of all the variables. You can use this information to decide the proper domains for your auxiliary variables.

How can this function be useful? Suppose we wanted to enforce the constraint $[X_1 + X_2 + X_3 \leq K]$. We would call `get_sum_variable()` on (X_1, X_2, X_3) to get some auxiliary variable Y, and then add the constraint $[Y \leq K]$. Note: You don't have to implement the $\leq$ constraint for this part.

3. **Course Scheduling**

In this problem, we will apply your weighted CSP solver to the problem of course scheduling. We have scraped a subset of courses that are offered from Stanford's Bulletin. For each course in this dataset, we have information on which quarters it is offered, the prerequisites (which may not be fully accurate due to ambiguity in the listing), and the range of units allowed. You can take a look at all the courses in `courses.json`. Please refer to `util.Course` and `util.CourseBulletin` for more information.

To specify a desired course plan, you would need to provide a *profile* which specifies your constraints and preferences for courses. A profile is specified in a text file (see `profile*.txt` for examples). The profile file has four sections:

- The first section specifies a fixed minimum and maximum (inclusive) number of units you need to take for each quarter. For example:

```
minUnits 0
maxUnits 3
```

- In the second section, you `register` for the quarters that you want to take your courses in. For example,

```
register Aut2018
register Win2019
register Spr2019
```

  would sign you up for this academic year. The quarters need not be contiguous, but they must follow the exact format `XxxYYYY` where `Xxx` is one of `Aut, Win, Spr, Sum` and `YYYY` is the year.

- The third section specifies the list of courses that you've taken in the past and elsewhere using the `taken` keyword. For example, if you're in CS221, this is probably what you would put:

```
taken CS103
taken CS106B
taken CS107
taken CS109
```

- The last section is a list of courses that you would like to take during the registered quarters, specified using `request`. For example, two basic requests would look like this:

```
request CS224N
request CS229
```

  Not every request must be fulfilled, and indeed, due to the additional constraints described below, it is possible that not all of them can actually be fulfilled.

Constrained requests. To allow for more flexibility in your preferences, we allow some freedom to customize the requests:

- You can request to take exclusively one of several courses but not sure which one, then specify:

```
request CS229 or CS229A or CS229T
```

  Note that these courses do not necessarily have to be offered in the same quarter. The final schedule can have at most one of these three courses. **Each course can only be requested at most once.**

- If you want to take a course in one of a specified set of quarters, use the `in` modifier. For example, if you want to take one of CS221 or CS229 in either Aut2018 **or** Sum2019, do:

```
request CS221 or CS229 in Aut2018,Sum2019
```

  If you do not specify any quarters, then the course can be taken in any quarter.

- Another operator you can apply is `after`, which specifies that a course must be taken after another one. For example, if you want to choose one of CS221 or CS229 and take it after both CS109 **and** CS161, add:

```
request CS221 or CS229 after CS109,CS161
```

  Note that this implies that if you take CS221 or CS229, then you must take both CS109 and CS161. In this case, we say that CS109 and CS161 are `prereqs` of this request. (Note that there's **no space** after the comma.)

  If you request course A and B (separately), and A is an official prerequisite of B based on the `CourseBulletin`, we will automatically add A as a prerequisite for B; that is, typing `request B` is equivalent to `request B after A`. Note that if B is a prerequisite of A, to request A, you must either request B or declare you've taken B before.

- Finally, the last operator you can add is `weight`, which adds non-negative weight to each request. All requests have a default weight value of 1. Requests with higher weight should be preferred by your CSP solver. Note that you can combine all of the aforementioned operators into one as follows (again, no space after comma):

  ```
  request CS221 or CS229 in Win2018,Win2019 after CS131 weight 5
  ```

Each `request` line in your profile is represented in code as an instance of the `Request` class (see `util.py`). For example, the request above would have the following fields:

- `cids` (course IDs that you're choosing one of) with value `['CS221', 'CS229']`
- `quarters` (that you're allowed to take the courses) with value `['Win2018', 'Win2019']`
- `prereqs` (course IDs that you must take before) with value `['CS131']`
- `weight` (preference) with value `5.0`

It's important to note that a request does not have to be fulfilled, *but if it is*, the constraints specified by the various operators `after,in` must also be satisfied.

You shall not worry about parsing the profiles because we have done all the parsing of the bulletin and profile for you, so all you need to work with is the collection of `Request` objects in `Profile` and `CourseBulletin` to know when courses are offered and the number of units of courses.

Well, that's a lot of information! Let's open a python shell and see them in action:

```
(XCS221) $ python
Python 3.6.9
Type "help", "copyright", "credits" or "license" for more information.
>>> import util
>>> # load bulletin
...
>>> bulletin = util.CourseBulletin('courses.json')
>>> # retrieve information of CS221
...
>>> cs221 = bulletin.courses['CS221']
>>> print(cs221)
(look at various properties of the course)
>>> print(cs221.cid)
CS221
>>> print(cs221.minUnits)
3
>>> print(cs221.maxUnits)
4
>>> print(cs221.prereqs)  # the prerequisites
['CS107', 'CS103', 'CS106X', 'CS106B']
>>> print(cs221.is_offered_in('Aut2018'))
True
>>> print(cs221.is_offered_in('Win2019'))
False
>>> # load profile from profile_example.txt
...
>>> profile = util.Profile(bulletin, 'profile_example.txt')
>>> # see what it's about
...
>>> profile.print_info()
Units: 3-9
Quarter: ['Aut2017', 'Win2018']
Taken: {'CS229'}
Requests:
  Request{['CS228'] ['Aut2017'] [] 1}
  Request{['CS229T'] [] ['CS228'] 2.0}
>>> # iterate over the requests and print out the properties
...
>>> for request in profile.requests:
...   print(request.cids, request.quarters, request.prereqs, request.weight)
...
['CS228'] ['Aut2017'] [] 1
['CS229T'] [] ['CS228'] 2.0
>>> exit()
```

Solving the CSP.

Your task is to take a profile and bulletin and construct a CSP. We have started you off with code in `SchedulingCSPConstructor` that constructs the core variables of the CSP as well as some basic constraints. The variables are all pairs of requests and registered quarters `(request, quarter)`, and the value of such a variable is one of the course IDs in that Request or `None`, which indicates none of the courses should be taken in that quarter. We will add auxiliary variables later. We have also implemented some basic constraints: `add_bulletin_constraints()`, which enforces that a course can only be taken if it's offered in that quarter (according to the bulletin), and `add_norepeating_constraints()`, which constrains that no course can be taken more than once.

You should take a look at `add_bulletin_constraints()` and `add_norepeating_constraints()` to get a basic understanding how the CSP for scheduling is represented. Nevertheless, we'll highlight some important details to make it easier for you to implement:

- The existing variables are tuples of `(request, quarter)` where `request` is a `Request` object (like the one shown above) and `quarter` is a `str` representing a quarter (e.g. `'Aut2018'`). For detail please look at `SchedulingCSPConstructor.add_variables()`.
- The domain of each variable `(request, quarter)` is the course IDs of the request **plus** `None` (e.g. `['CS221', 'CS229', None]`). When the value `cid` is `None`, this means no course is scheduled for this request. **Always remember to check if `cid` is `None`.**
- The domain for `quarter` is all possible quarters (`self.profile.quarters`, e.g. `['Win2016', 'Win2017']`).
- Given a course ID `cid`, you can get the corresponding `Course` object by `self.bulletin.courses[cid]`.

(a) [**6 points (Coding)**] Implement the function `add_quarter_constraints()` in `submission.py`. This is when your profile specifies which quarter(s) you want your requested courses to be taken in. This is not saying that one of the courses must be taken, *but if it is*, then it must be taken in any one of the specified quarters. Also note that this constraint will apply to all courses in that request.

We have written a `verify_schedule()` function in `grader.py` that determines if your schedule satisfies all of the given constraints. Note that since we are not dealing with units yet, it will print `None` for the number of units of each course. For profile3a.txt, you should find 3 optimal assignments with weight 1.0.

(b) [**7 points (Coding)**] Let's now add the unit constraints in `add_unit_constraints()`. (1) You must ensure that the sum of units per quarter for your schedule are within the min and max threshold inclusive. You should use `get_sum_variable()`. (2) In order for our solution extractor to obtain the number of units, for every course, you must add a variable `(courseId, quarter)` to the CSP taking on a value equal to the number of units being taken for that course during that quarter. When the course is not taken during that quarter, the unit should be 0.

NOTE: Each grader test only tests the function you are asked to implement. To test your CSP with multiple constraints you can use `run_p3.py` and add whichever constraints that you want to add. For profile3b.txt, you should find 15 optimal assignments with weight 1.0.

Hint: If your code times out, your `maxSum` passed to `get_sum_variable()` might be too large.

(c) [**1 point (Written)**] Now try to use the course scheduler for the winter and spring quarters (and next year if applicable). Create your own `profile.txt` and then run the course scheduler:

```
python run_p3.py profile.txt
```

You might want to turn on the appropriate heuristic flags to speed up the computation. Does it produce a reasonable course schedule? Please include your `profile.txt` and the best schedule in your writeup; we're curious how it worked out for you! Please include your schedule and the profile in the PDF, otherwise you will not receive credit.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

0.a

0.b

2.a

3.c