CS221 Spring 2022: Artificial Intelligence: Principles and Techniques

Homework 6: Scheduling

SUNet ID: jchan7
Name: Jason Chan
Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Problem 0: CSP basics

a. [4 points] Let's create a CSP. Suppose you have n light bulbs, where each light bulb $i = 1, \ldots, n$ is initially off. You also have m buttons which control the lights. For each button $j = 1, \ldots, m$, we know the subset $T_j \subseteq \{1, \ldots, n\}$ 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). If multiple—say Z buttons—controlling the same light bulb are pressed, then that light bulb will be turned on if Z is odd, or it will be turned off if Z is even.

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 m-ary constraints: constraints that can be functions of up to m variables. 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.

Hint: If stuck, take a look at parts (b) and (c) of this problem to see how you could define the constraints using a boolean operator.

What we expect: A clear description of your solution CSP including m variables and their domains, and n constraints.

Your Solution:

(a) Variables are all X_j where j = 1, 2, .. m.

(b) $Domain_j$ is the set of lights assignable to each button $T_j \subseteq \{1, 2, ..., n\}$

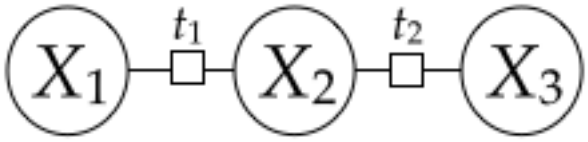
(c) Constraint: For each light, the total number of times it has been assigned to a button needs to be odd and not zero. We can represent that compactly as an indicator function of the product of the modulo of each of those counts. If f = 0 then we have at least one light that will be off which violates the constraint.

- Scope is all X_j where j = 1, 2, .. m. (formulated as m-ary)
- Expression: f is shown below to represent the constraint

$$f = \left[\prod_{i=1}^{n}(t_i \bmod 2) = 1\right] \tag{1}$$

$$\text{where } t_i = \sum_{j=1}^{m} \begin{cases} 1 & t_i \in T_j \\ 0 & otherwise \end{cases} \tag{2}$$

b. [6 points] Now, 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 $t_1(X) = x_1 \oplus x_2$ and $t_2(X) = x_2 \oplus x_3$).

i. What are the consistent assignments for this CSP?

ii. Let's use backtracking search to solve the CSP *without using any heuristics (MCV, LCV, forward checking, AC-3).* The `Backtrack()` algorithm as defined in the lectures is a recursive algorithm, where new instances of `Backtrack()` are called within parent instances of `Backtrack()`.

In this problem, we will ask you to produce the call stack for a specific call to `Backtrack()`. A call stack is just a diagram tracing out every recursive call. For our purposes, for each call to `Backtrack()` you should specify which variable is being assigned, the current domains, and which parent call to `Backtrack()` it's called within. For example, if the order in which we assign variables is X_1, X_2, X_3,

the call stack would be as follows:

$$\{[01],[01],[01]\} \xrightarrow{X_1=0} \{\mathbf{0},[01],[01]\} \xrightarrow{X_2=1} \{\mathbf{0},\mathbf{1},[01]\} \xrightarrow{X_3=0} \{\mathbf{0},\mathbf{1},\mathbf{0}\}$$

$$\xrightarrow{X_1=1} \{\mathbf{1},[01],[01]\} \xrightarrow{X_2=0} \{\mathbf{1},\mathbf{0},[01]\} \xrightarrow{X_3=1} \{\mathbf{1},\mathbf{0},\mathbf{1}\}$$

The notation $\{\mathbf{1},[01],[01]\}$ means that X_1 has been assigned value 1, while X_2 and X_3 are currently unassigned and each have domain $\{0,1\}$. Note that we omit the comma in the domain for easier reading. We also avoid the weight variable for simplicity; the only possible weights for this problem are 0 and 1. In this case, backtrack is called 7 times. Notice that `Backtrack()` is not called when there's an inconsistent partial assignment ($\delta = 0$); for example, we don't call `Backtrack()` on $X_2 = 1$ when X_1 is already set to 1.

Using this call stack, we can produce the list of calls in the order they are explored. For this example where we assign variables in order X_1, X_2, X_3, the list would be

$$\{[01],[01],[01]\},\{0,[01],[01]\},\{0,1,[01]\},\{0,1,0\},\{1,[01],[01]\},\{1,0,[01]\},\{1,0,1\}.$$

Suppose we assign variables in the order X_3, X_1, X_2. Write the list of calls in the order they are explored and draw out the call-stack. How many calls do we make to `Backtrack()`? Why can this number change depending on the ordering?

iii. We often add heuristics like AC-3 to speed up the backtracking search. How many calls to `Backtrack()` from your call stack in the previous question would we skip if we use AC-3? Briefly explain why we skip (or don't skip) calls in this search with AC-3.

What we expect: For i., a list of all the consistent assignments (1 sentence). For ii., a list of calls in order of exploration, a drawing of the call stack, the number of times `Backtrack()` is called, and an explanation for why this number can change based on the order in which you assign variables (1-4 sentences). For **this problem only** you may hand-draw a call stack and paste a picture into the PDF, provided that the drawing is neat and everything is legible. For iii., the number of calls to `Backtrack()` that get skipped along with an explanation for why we skip these calls with AC-3 (1-2 sentences).

Your Solution:

(a) Assignment list 1 (X_1:1, X_2:0, X_3:1). Assignment list 2 (X_1:0, X_2:1, X_3:0)

(b) There are 9 calls in the call stack. The number changes depending on the ordering because it affects the number of adjacent variables with assignments that can be

checked against the factors. In this case, without heuristics, when we assign X_3 and then X_1 we still don't have enough information to check if factors are satisfied until X_2 gets assigned. In the previous case we could check factors earlier hence why it has fewer calls.

$\{[01],[01],[01]\} \xrightarrow{X_3=0} \{[01],[01],\mathbf{0}\} \xrightarrow{X_1=0} \{\mathbf{0},[01],\mathbf{0}\} \xrightarrow{X_2=1} \{\mathbf{0},\mathbf{1},\mathbf{0}\}$

$\xrightarrow{X_1=1} \{\mathbf{1},[01],\mathbf{0}\} \xrightarrow{X_3=1} \{[01],[01],\mathbf{1}\} \xrightarrow{X_1=0} \{\mathbf{0},[10],\mathbf{1}\}$

$\xrightarrow{X_1=1} \{\mathbf{1},[01],\mathbf{1}\} \xrightarrow{X_2=0} \{\mathbf{1,0,1}\}$

(c) If we used AC3 we would skip 2 calls. AC-3 would start with an assignment of $X_3 = 0$, then it would look at its neighbour X_2 and eliminate 0 from its domain, then it would look at X_2's neighbour, X_1 and eliminate 1 from its domain. Next, AC-3 would start with an assignment of $X_3 = 1$, then it would look at its neighbour X_2 and eliminate 1 from its domain, then it would look at X_2's neighbour, X_1 and eliminate 0 from its domain. In total there are 6 calls + 1 to initialise backtrack. In the previous question we had 9 calls total, so AC3 would skip 2 calls.

c. [2 points] Now let's consider a general case: given a factor graph with n variables $X_1, \ldots, X_n$ and $n-1$ binary factors $t_1, \ldots, t_{n-1}$ where $X_i \in \{0, 1\}$ and $t_i(X) = x_i \oplus x_{i+1}$. 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()`.

Problem 1: CSP solving

We'll now pivot towards creating more complicated CSPs, and solving them faster using heuristics. Notice we are already able to solve the CSPs because in `submission.py`, a basic backtracking search is already implemented. For this problem, we will work with *unweighted* CSPs that can only have True/False factors; a factor outputs 1 if a constraint is satisfied and 0 otherwise. The backtracking search operates over partial assignments, and specifies whether or not the current assignment satisfies all relevant constraints. When we assign a value to a new variable X_i, we check that all constraints that depend only on X_i and the previously assigned variables are satisfied. The function `satisfies_constraints()` returns whether or not these new factors are satisfied based on the `unaryFactors` and `binaryFactors`. When `satisfies_constraints()` returns False, any full assignment that extends the new partial assignment cannot satisfy all of the constraints, 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. [5 points] Let's create an unweighted 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 or your code times out on the test cases, 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. You can refer to `create_map_coloring_csp()` and `create_weighted_csp()` in `util.py` as examples of CSP problem implementations. You can try these examples out by running:

   ```
   python run_p1.py
   ```

b. [5 points] 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 `satisfies_constraints()` on $X_j = a$ returns True). 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:

- In `BacktrackingSearch`, if `var` has been assigned a value, you can retrieve it using `assignment[var]`. Otherwise `var` is not in `assignment`.

Problem 2: Course Scheduling

In this problem, you will leverage our CSP solver for 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 Aut2019
  register Win2020
  register Spr2020
  ```

  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 by specifying:

  `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`. Additionally, if A is a prerequisite of B, in order to request B you must either request A or declare you've taken A before.

- Finally, the last operator you can add is `weight`, which adds non-negative weight to each request. To accommodate this, we will work with a standard CSP (as opposed to unweighted, like Problem 1), 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. Notice that our backtracking search already works with normal CSPs; you should simply define factors that output real numbers. For 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()`).

  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:

```
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)
print(cs221.minUnits)
print(cs221.maxUnits)
print(cs221.prereqs)  # the prerequisites
print(cs221.is_offered_in('Aut2018'))
print(cs221.is_offered_in('Win2019'))

# load profile from profile_example.txt
profile = util.Profile(bulletin, 'profile_example.txt')
# see what it's about
profile.print_info()
# iterate over the requests and print out the properties
for request in profile.requests:
    print(request.cids, request.quarters, request.prereqs, request.weight)
```

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 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] 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.

b. [10 points] Let's now add the unit constraints in `add_unit_constraints()`.

   1. 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.
   2. You must take into account the appropriate binary factor between `(request, quarter)` and `(courseId, quarter)` variables.
   3. You must ensure that the sum of units per quarter for your schedule are within the min and max threshold, inclusive. You should use the `create_sum_variable()` function we've implemented for you; pay careful attention to the arguments.

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

   Note: Each grader test only tests the function you are asked to implement. To test your CSP with multiple constraints you can add to the profile text file whichever constraints that you want to add and run `run_p2.py`. Here is an example with `profile2b.txt` as input:

   ```
   python run_p2.py profile2b.txt
   ```

Running this command will print information that may be helpful for debugging, such as profile information, the number of optimal assignments found (along with their weight and the number of times `backtrack()` is called while solving the CSP), one full optimal assignment, and the resulting course schedule.

c. [2 points] Now try to use the course scheduler for any two quarters in the future (or more quarters if you wish, although this might lead to a slower search). Create your own `profile.txt` (take a look at some of the profile text files included in the assignment's main directory for inspiration) and then run the course scheduler:

```
python run_p2.py profile.txt
```

If the courses you wish to schedule are not listed in `courses.json`, feel free to add them in as you please! In addition, feel free to modify course details as well (e.g., you can change the list of quarters that a course is being offered in if it does not match the information on the current year's course calendar). You might want to turn on the appropriate heuristic flags to speed up the computation; in particular, `self.ac3 = True` applies the arc-consistency heuristic that we implement for you, and you can use your own MCV implementation. Does it produce a reasonable course schedule? Please include your `profile.txt` and the best schedule in your writeup (you can just paste it into the pdf that you submit); 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.

What we expect: The profile.txt file (pasted into the pdf with the solutions), the corresponding outputted schedule, and a brief discussion (1-2 sentences) on whether or not it seems reasonable. Please also describe any changes you made to `courses.json` (if applicable; for example, if you added courses).

Your Solution: I modified the json file to add CS230 Deep Learning and updated the availability of the courses to reflect 2022-23.

Unit limit per quarter.
minUnits 3
maxUnits 4

These are the quarters that I need to fill out.
It is assumed that the quarters are sorted in chronological order.
register Spr2022
register Sum2022
register Aut2022
register Win2023

Courses I've already taken

Courses that I'm requesting
request CS221 in Spr2022,Sum2022
request CS229 in Spr2022,Sum2022
request CS230 in Sum2022,Aut2022,Win2023
request CS224N or CS224W in Win2023

The solution looks reasonable. The recommendation fulfils my hard constraint of taking maximum 4 units per quarter because I'm an SCPD student and working full-time. CS221 is recommended first then CS229, CS230 and finally CS224N. CS229 is a prerequisite for CS230, and CS221 is a prerequisite for CS224N. Unsurprisingly, I got warnings about about prerequisites for CS221 but I disregarded them since I've taken their equivalents at an Australian University. Unfortunately, CS224W doesn't make the cut because it isn't offered in Winter so CS224N is recommended instead.

Found 4 optimal assignments with weight 1.0 in 163 operations
First assignment took 81 operations

Here's the best schedule:
Quarter Units Course
Spr2022 3 CS221
Sum2022 3 CS229
Aut2022 4 CS230
Win2023 3 CS224N

Problem 3: Residency Hours Scheduling

a. [2 points] Many uses of constraint satisfaction in real-world scenarios involve assignment of resources to entities, like assigning packages to different trucks to optimize delivery. However, when the agents are people, the issue of fair division arises. In this question, you will consider the ethics of what constraints to remove in a CSP when the CSP is unsatisfiable.

Medical residents are often scheduled to work long shifts with insufficient rest, leading to exhaustion and burnout. This can negatively affect the residents and potentially lead to mistakes that also negatively affect the patients in their care[1]. A hospital could use a constraint-satisfaction approach to try to create a work schedule that respects the "on-call night, day-off, rest period, and total work-hour regulations mandated by the Accreditation Council for Graduate Medical Education, as well as the number of residents needed each hour given the demand (aka number of patients and procedures scheduled)[2]. The constraints are:

1. One day off every 7 days
2. Minimum 8 hour rest period between shifts
3. No more than 80 hours of work per week averaged over a 4 week period
4. At least 14 hours free of clinical work and education after 24 hours of in-house call
5. Number of residents needed each hour

Let's assume for a given hospital that the constraints listed above were collectively **unsatisfiable** given the number of residents assigned to that hospital. However, its formulation as an unsatisfiable CSP depends on other factors remaining fixed, such as

A. The number of total residents
B. The work to be performed by residents as opposed to other staff
C. The budget available to hire residents or other staff

In this case, would you remove one of the numbered constraints 1-5 or advocate that the hospital administration change one of A-C to make the problem solveable? If so, explain which one and give a reason why.

What we expect: In 2-4 sentences, you must explicitly state which numbered constraint or lettered factor you would change and justify your choice with a reason that explains why you chose that one.

[1] https://knowledgeplus.nejm.org/blog/resident-burnout-well-being/
[2] https://www.sciencedirect.com/science/article/abs/pii/S0305054810001024?via%3Dihub

Your Solution: I advocate for the hospital to change letter factor C: *The budget available to hire residents or other staff.* More budget to hire more residents or other staff would have the most impact. More staff in the longer term could allow medical residents to have shorter shifts or longer rest periods between shifts. The drawback of this recommendation is that the hospital could raise prices in an attempt to lift the budget.

I arrived at letter factor C through a process of elimination.

- Eliminated letter factor B because re-allocating tasks to other support staff only shifts the burden.
- Eliminated letter factor A because I presumed that this option requires budget or re-distributing the existing budget, which could negatively impact another part of the hospital. And it's not clear that having more residents would necessarily help - it could be that there aren't enough support staff.
- Eliminated numbered factors 1, 2, 3 because relaxing these will definitely increase medical residents burn out.
- Eliminated numbered factor 4 because 14 hours free clinical work and education serves the common good.
- Eliminated numbered factor 5 because more residents per hour may reduce the intensity per hour but is unlikely to change *work long shifts with insufficient rest.*

In conclusion, letter factor C is what I would advocate.

Submission

Submission is done on Gradescope.

Written: When submitting the written parts, make sure to select **all** the pages that contain part of your answer for that problem, or else you will not get credit. To double check after submission, you can click on each problem link on the right side and it should show the pages that are selected for that problem.

Programming: After you submit, the autograder will take a few minutes to run. Check back after it runs to make sure that your submission succeeded. If your autograder crashes, you will receive a 0 on the programming part of the assignment. Note: the only file to be submitted to Gradescope is `submission.py`.

More details can be found in the Submission section on the course website.