Lab 3: Constraint Satisfaction Problems

Submit Assignment

Due Saturday by 11:59pm **Points** 5 **Submitting** a file upload
Available Sep 20 at 12pm - Oct 10 at 11:59pm 21 days

Lab 3: Constraint Satisfaction Problems

To complete Lab 3, download: **lab3.zip**.

Lab 3 is due by **Saturday, October 3rd at 11:59pm EDT**. Once you have completed lab3 and it passes the local tests, submit **only** your lab3.py file here. All of your answers belong in the main file lab3.py.

Contents

A Working Vocabulary

Before beginning, you may want to (re)familiarize yourself with the following terms:

- **variable**: something that can receive an assignment value
- **value**: something that can be assigned
- **domain**: a set of values
- **constraint**: a condition that limits domains of possible values

Part 1: Warm-up

In this lab, you'll write programs that solve constraint satisfaction problems (CSPs). A CSP consists of variables, assignments, and constraints, and is represented by a `ConstraintSatisfactionProblem` object as described in **the API**.

First, we'll get familiarity with CSPs by writing a few helper routines.

- `has_empty_domains(csp)`: returns `True` if the supplied problem has one or more empty domains. Otherwise, returns `False`.
- `check_all_constraints(csp)`: returns `False` if the problem's **assigned values** violate some constraint. Otherwise, returns `True`.

Each function takes in an argument `csp`, which is a **`ConstraintSatisfactionProblem`** instance.

Part 2: Writing a depth-first search solver

Now you can use your helper functions to write the constraint solver:

```
def solve_constraint_dfs(problem) :
```

This is just like depth-first search as implemented in the search lab, but this time the items in the agenda are partially-solved problems instead of paths. Additionally, for this problem, we will also want to track the number of extensions so we can compare the different strategies for constraint propagation. At the end, instead of returning just a solution, you will return a tuple `(solution, num_extensions)`, where

- `solution` is the solution to this problem as a dictionary mapping variables to assigned values (see API for details); or `None` if there is no solution to the problem.
- `num_extensions` is the number of extensions performed during the search. Recall that as before, an extension occurs whenever a problem is **removed** from the agenda for processing.

Here is a rough outline of how to proceed:

1. Initialize your agenda and the extension count.
2. Until the agenda is empty, pop the first problem off the list and increment the extension count.
3. If any variable's domain is empty or if any constraints are violated, the problem is *unsolvable* with the current assignments.

4. If none of the constraints have been violated, check whether the problem has any unassigned variables. If not, you've found a complete solution!
5. However, if the problem has some unassigned variables:
    1. Take the first unassigned variable off the list using `csp.pop_next_unassigned_var()`.
    2. For each value in the variable's domain, create a new problem with that value assigned to the variable, and add it to a list of new problems. Then, add the new problems to the appropriate end of the agenda.
6. Repeat steps 2 through 6 until a solution is found or the agenda is empty.

Benchmarks

So that we can compare the efficiency of different types of constraint-satisfaction algorithms, we'll compute how many extensions (agenda dequeues) each algorithm requires when solving a particular CSP. Our test problem will be the Pokemon problem from 2012 Quiz 2 (http://courses.csail.mit.edu/6.034f/Examinations/2012q2.pdf), pages 2-4.

You can solve the Pokemon problem by calling `solve_constraint_dfs(pokemon_problem)` directly. Note that the Pokemon problem is already defined for you in test_problems.py. To get a copy of it, use the method get_pokemon_problem() in lab3.py.

Please answer the following questions in your lab file:

Question 1

How many extensions does it take to solve the Pokemon problem with just DFS?

Put your answer (as an integer) in for `ANSWER_1`.

Part 3: Forward checking streamlines search by eliminating impossible assignments

One problem with the `solve_constraint_dfs` algorithm is that it explores all possible branches of the tree. We can use a trick called forward checking to avoid exploring branches that cannot possibly lead to a solution: each time we assign a value to a variable, we'll eliminate incompatible or *inconsistent* values from that variable's neighbors.

Finding inconsistent values in neighbors

First, we will write a helper function to eliminate *inconsistent* values from a variable's neighbors' domains:

> Suppose V is a variable with neighbor W. If W's domain contains a value w which violates a constraint with **every value in V's domain**, then the assignment W=w can't be part of the solution we're constructing — we can safely eliminate w from W's domain.

(Note that unlike the `check_all_constraints` function above, `eliminate_from_neighbors` checks all combinations of values, and is not restricted to comparing only variables that have assigned values.)

This function should return an alphabetically sorted list of the neighbors whose domains were reduced (i.e. which had values eliminated from their domain), with each neighbor appearing **at most once** in the list. If no domains were reduced, return an empty list; if a domain is reduced to size 0, quit and immediately return `None`. This method **should** modify the input CSP. Hint: You can remove values from a variable's domain using csp.eliminate(var, val). But don't eliminate values from a variable while iterating over its domain, or Python will get confused!

```
def eliminate_from_neighbors(csp, var) :
```

We strongly suggest working out examples on paper to get a feel for how the forward checker should find inconsistent values.

To reduce the amount of nested for-loops and to make debugging easier, you may find it helpful to write a small helper function that, for example, takes in two variables V and W, and two values v and w in their respective domains, and checks if there are any constraint violations between V=v and W=w.

Depth-first constraint solver with forward checking

Now, we will write our improved CSP solver which uses `eliminate_from_neighbors` above to apply forward checking while searching for variable assignments.

```
def solve_constraint_forward_checking(problem) :
```

The implementation for this function will be very similar to that of `solve_constraint_dfs`, except now the solver must apply forward checking (`eliminate_from_neighbors`) after each assignment, to eliminate incompatible values from the assigned variable's neighbors.

Note that if `eliminate_from_neighbors` eliminates all values from a variable's domain, the problem will be recognized as unsolvable when it is *next* popped off the agenda: do not preemptively remove it from consideration.

Answer the following question in your `lab3.py` file:

Question 2

How many extensions does it take to solve the Pokemon problem with forward checking?

Put your answer (as an integer) in for `ANSWER_2`.

Part 4: Propagation!

Forward checking is a useful tool for checking ahead for inconsistencies and reducing the search space. However, in many situations, it's ideal to prune inconsistent states even faster.

Domain reduction

A far-reaching strategy called *domain reduction* eliminates incompatible values not just between neighbors, but across all variables in the problem. You can apply domain reduction either *before search* (this is what Sudoku players do when they narrow down options before tentatively guessing a value) or *after assigning each variable during search* (as a more powerful variation of forward-checking).

As it turns out, the implementation for both of these are effectively identical:

1. Establish a queue. If using domain reduction *during* search, this queue should initially contain only the variable that was just assigned. If before search (or if no queue is specified), the queue can contain all variables in the problem. (Hint: `csp.get_all_variables()` will make a copy of the variables list.)
2. Until the queue is empty, pop the first variable `var` off the queue.
3. Iterate over that `var`'s neighbors: if some neighbor `n` has values that are incompatible with the constraints between `var` and `n`, remove the incompatible values from `n`'s domain. If you reduce a neighbor's domain, add that neighbor to the queue (unless it's already in the queue).
4. If any variable has an empty domain, quit immediately and return `None`.
5. When the queue is empty, domain reduction has finished. Return a list of all variables that were dequeued, in the order they were removed from the queue. Variables may appear in this list multiple times.

Note that when the queue initially contains only the assigned variable, the first step of propagation is just forward checking of the assigned variable's neighbors. "Propagation" occurs as we add more variables to the queue, checking neighbors of neighbors, etc.

You will now implement `domain_reduction`, which applies forward checking (checking for neighboring values' inconsistencies) with propagation through any domains that are reduced.

Recall that domain reduction utilizes a queue to keep track of the variables whose neighbors should be explored for inconsistent domain values. If you are not explicitly provided a queue from the caller, your queue should start out with all of the problem's variables in it, in their default order.

When doing domain reduction, you should keep track of the order in which variables were dequeued; the function should return this ordered list of variables that were dequeued.

If at any point in the algorithm a domain becomes empty, immediately return `None`.

```
def domain_reduction(csp, queue=None) :
```

This method **should** modify the input CSP.

Hint: You can remove values from a variable's domain using `csp.eliminate(var, val)`. But **don't** eliminate values from a variable while iterating over its domain, or Python will get confused!

Answer the following question in your `lab3.py` file:

Question 3

How many extensions does it take to solve the Pokemon problem with DFS (no forward checking) if you do domain reduction before solving it?

Put your answer (as an integer) in for `ANSWER_3`.

Propagation through reduced domains

Now we'll see how we can take advantage of domain reduction during the search procedure itself.

When used during search, domain reduction makes use of the assignments you've made to progressively reduce the search space. The result is a new, faster, CSP solution method: propagation through reduced domains. After each assignment, propagation through reduced domains uses the domain_reduction subroutine to "propagate" the consequences of the assignment: to neighbors, then to neighbors of neighbors, and so on.

```
def solve_constraint_propagate_reduced_domains(problem) :
```

Note that if `domain_reduction` eliminates all values from a variable's domain, the problem will be recognized as unsolvable when it is *next* popped off the agenda: do not preemptively remove it from consideration.

Debugging hint: be sure to look at the return types of functions that you call!

Answer the following question in your `lab3.py` file:

Question 4

How many extensions does it take to solve the Pokemon problem with forward checking and propagation through reduced domains? (Don't use domain reduction before solving it.)

Put your answer (as an integer) in for `ANSWER_4`.

Part 5A: Generic propagation

The `domain_reduction` procedure is comprehensive, but expensive: it eliminates as many values as possible, but it continually adds more variables to the queue. As a result, it is an effective algorithm to use *before* solving a constraint satisfaction problem, but is often too expensive to call repeatedly during search.

Instead of comprehensively reducing all the domains in a problem, as `domain_reduction` does, you can instead reduce only *some* of the domains. This idea underlies *propagation through singleton domains* — a reduction algorithm which does not detect as many dead ends, but which is significantly faster.

Instead of again patterning our propagation-through-singleton-domains algorithm off of `domain_reduction`, we'll write a fully general propagation algorithm called `propagate` that encapsulates all three checking strategies we've seen: forward checking, propagation through all reduced domains, and propagation through singleton domains.

The function `propagate` will be similar to the propagation algorithms you've already defined. The difference is that it will take an argument `enqueue_condition_fn`, a function that takes a problem and a variable, and outputs whether the variable should be added to the propagation queue.

```
def propagate(enqueue_condition_fn, csp, queue = None) :
```

Propagation through singletons is like propagation through reduced domains, except that variables must pass a test in order to be added to the queue:

> In propagation through singleton domains, you only append a variable to the queue if it has exactly one value left in its domain.

Common misconception: Please note that propagation **never** *assigns* values to variables; it only *eliminates* values. There is a distinction between variables with one value in their domain, and assigned variables: a variable can have one value in its domain without any value being assigned yet.

As a review, propagation eliminates incompatible options from neighbors of variables in the queue. When used during search, the propagation queue initially contains only the just-assigned variable. The three enqueueing conditions we've seen are:

1. *forward checking*: never adds other variables to the queue
2. *propagation through singleton domains*: adds a neighboring variable to the queue if its domain has exactly one value in it
3. *domain reduction / propagation through reduced domains*: adds a neighboring variable to the queue if its domain has been reduced in size

Write functions that represent the enqueueing conditions (predicates) for each of these. Each predicate function below takes in a CSP and the variable in question, returning `True` if that variable should be added to the propagation queue, otherwise `False`.

```
def condition_domain_reduction(csp, var) :

def condition_singleton(csp, var) :

def condition_forward_checking(csp, var) :
```

Part 5B: A generic constraint solver

Now, you can use `propagate` to write a generic constraint solver. Write an algorithm that can solve a problem using any enqueueing strategy. As a special case, if the `enqueue_condition` is `None`, default to ordinary dfs instead --- don't eliminate options from neighbors (don't use any forward checking or propagation) at all.

```
def solve_constraint_generic(problem, enqueue_condition=None) :
```

Answer the following question in your `lab3.py` file:

Question 5

How many extensions does it take to solve the Pokemon problem with forward checking and propagation through singleton domains? (Don't use domain reduction before solving it.)

Put your answer (as an integer) in for `ANSWER_5`.

Part 6: Defining your own constraints

In this section, you will create some constraint functions yourself.

Assuming `m` and `n` are integers, write a function that returns `True` if `m` and `n` are adjacent values (i.e. if they differ by exactly one) and `False` otherwise.

```
def constraint_adjacent(m, n) :
```

Also write one for being non-adjacent.

```
def constraint_not_adjacent(m, n) :
```

The following example shows how you build a constraint object that requires two variables — call them A and B — to be different.

```
example_constraint = Constraint("A","B", constraint_different)
```

Some constraint problems include a constraint that requires all of the variables to be different from one another. It can be tedious to list all of the pairwise constraints by hand, so we won't. Instead, write a function that takes a list of variables and returns a list containing, for each pair of variables, a constraint object requiring the variables to be different from each other. (You can model the constraints on the example above.) Note that for this *particular* constraint (the must-be-different constraint), order does NOT matter, because inequality is a symmetric relation. Hence, in you should only have *one* constraint between each pair of variables (e.g. have a constraint between A and B, **OR** have a constraint between B and A, but not both).

```
def all_different(variables) :
```

Note: You should only use constraint functions that have already been defined. Don't try to create a new constraint function and use it in this function, because our tester will get confused.

API

In this lab, we provide an API for representing and manipulating partial solutions to constraint satisfaction problems.

Constraint Satisfaction Problems

A `ConstraintSatisfactionProblem` is an object representing a partially solved constraint satisfaction problem. Its fields are:

`variables`
A list containing the names of all the variables in the problem, in alphabetical order.

`domains`
A dictionary associating each variable in the problem with its list of remaining values.

`assignments`
A dictionary. Each variable that has already been assigned a value is associated with that value here. When the problem is entirely solved, `assignments` contains the solution.

`unassigned_vars`
An ordered list of all the variables that still need to have a value assigned to them.

`constraints`
A list of the constraints between the variables in the problem. Each constraint is a `Constraint` object.

Note: While you may *read* any of the above variables, you should probably not modify them directly; `instead, you should use the following API methods:`

`get_domain(var)`
Returns the list of values in the variable's domain.

`set_domain(var, domain)`
Sets the domain of the variable to the specified list of values, sorted alphabetically/numerically.

`set_all_domains(domains_dict)`

Sets the `domains` field to the specified dictionary. Does not sort domains.

`get_all_variables()`

Returns a list of all the variables in the problem.

`get_all_constraints()`

Returns a list of all the **constraints** in the problem.

`pop_next_unassigned_var()`

Returns first unassigned variable, or `None` if all variables are assigned. Modifies `unassigned_vars` list.

`set_unassigned_vars_order(unassigned_vars_ordered)`

Given an ordered list of unassigned variables, sets `unassigned_vars`. (By default, the `unassigned_vars` list is initialized in alphabetical order.)

`eliminate(var, val)`

Removes the value from the variable's domain, returning `True` if the value was found in the domain, otherwise `False` if the value wasn't found.

`get_assignment(var)`

If the variable has been assigned a value, retrieve it. Returns `None` if the variable hasn't been assigned yet.

`set_assignment(var, val)`

Sets the assigned value of the variable to `val`, returning a modified copy of the constraint satisfaction problem. Throws an error if `val` is not in the domain of `var`, or if `var` has already been assigned a value. For convenience, also modifies the variable's domain to contain only the assigned value.

`constraints_between(var1, var2)`

Returns a list of all **constraints** between `var1` and `var2`. Arguments that are `None` will match anything: for example, `constraints_between('X',None)` will return all constraints involving `X` and any other variable, and `constraints_between(None, None)` will return all of the constraints in the problem.
Note: For your convenience, the constraints returned will always be altered to match the order of the arguments you passed to this method. For example, `csp.constraints_between(None, 'Y')` will return a list of all constraints involving `'Y'` — and the constraints will be altered so that `'Y'` is their *second* variable (`var2`) in every case.

`get_neighbors(var)`

Returns a list of all the variables that share constraints with the given variable, ordered alphabetically.

`add_constraint(var1, var2, constraint_fn)`

Given two variables and a function to act as a constraint between them, creates a **`Constraint` object** and adds it to the `constraints` list. The function `constraint_fn` must be a binary predicate function: it takes two arguments (a value for the first variable, and a value for the second variable) and returns `True` if the values satisfy the constraint, or `False` otherwise.

`add_constraints(list_of_constraints)`

Add a list of **`Constraint` objects** to the `constraints` list. Useful for when you want to add several constraints to the problem at once, rather than one at a time using `.add_constraint`.

`copy()`

Return a (deep) copy of this constraint satisfaction problem. This method is particularly useful because you will want to make a copy of the CSP every time you assign a value to a variable.

Constraint objects

A `Constraint` is a fairly basic object representing a constraint between two variables. A `Constraint` object has three fields:

`var1`

The first variable associated with this constraint

`var2`

The second variable associated with this constraint

`constraint_fn`

A function that takes in two arguments, returning `True` or `False` depending on whether or not the given constraint is satisfied by the two arguments. For example,

- `constraint_equal(a, b)` is a function requiring that `a` and `b` are equal.
- `constraint_different(a, b)` is a function requiring that `a` and `b` are not equal.

These two functions are already defined in `constraint_api.py`, and can be accessed directly from `lab3.py`.

A `Constraint` object has just one method associated with it:

`check(val1, val2)`

Applies this object's `constraint_fn` to two *values* (**not** variables), returning `True` if the values satisfy the constraint, or `False` otherwise.

Note: Due to certain limitations in our tester, a `Constraint` object constructor must take a **named** `constraint_fn` as an argument, **NOT** a lambda function.

Appendix: Setting up a Constraint Satisfaction Problem

The Pokemon problem from 2012 Quiz 2 (http://courses.csail.mit.edu/6.034f/Examinations/2012q2.pdf) , pages 2-4, is an example of a problem that can be solved using constrained search.

In this section, we will show you how to convert this problem into a `ConstraintSatisfactionProblem` instance using our **constraint satisfaction API**.

To set up a problem, we first establish a new `ConstraintSatisfactionProblem` instance. For the Pokemon problem, there are five variables which we pass an an argument in a list: these are the five "questions" that need to be answered.

```
pokemon_problem = ConstraintSatisfactionProblem(["Q1","Q2","Q3","Q4","Q5"])
```

Here, we specify the values in each variable's domain:

```
pokemon_problem.set_domain("Q1",["A","B","C","D","E"])
pokemon_problem.set_domain("Q2",["A","B","C","D","E"])
pokemon_problem.set_domain("Q3",["A","B","C","D","E"])
pokemon_problem.set_domain("Q4",["A","B","C","D","E"])
pokemon_problem.set_domain("Q5",["A","B","C","D","E"])
```

Next, we set up constraints. Each constraint takes two variable names, and a *named* **binary predicate** (constraint function), not a lambda function:

```
pokemon_problem.add_constraint("Q1","Q4", constraint_different)
pokemon_problem.add_constraint("Q1","Q2", constraint_equal)
pokemon_problem.add_constraint("Q3","Q2", constraint_different)
pokemon_problem.add_constraint("Q3","Q4", constraint_different)
pokemon_problem.add_constraint("Q4","Q5", constraint_equal)
```

By default, the `unassigned_vars` list is initialized in alphabetical order.

To specify the order yourself, you can call `.set_unassigned_vars_order` with an ordered list of the unassigned variables:

```
# How to set the order of unassigned variables (not actually used for the Pokemon problem)
pokemon_problem.set_unassigned_vars_order(["Q4","Q2","Q3","Q1","Q5"])
```

For some problems, efficiently re-ordering the variables can make a large difference in performance.

Note that the Pokemon problem is already defined for you in `test_problems.py`. To get a copy of it, use the method `get_pokemon_problem()` in `lab3.py`.

FAQ

Q: I am getting the right output but the wrong number of evaluations

A: Check that, when reducing domains, you are correctly considering the possibility of having multiple different constraints between two variables. (What does it mean if you have two contradictory constraints between two variables?)

Survey

Please answer these questions at the bottom of your lab file:

- `NAME`: What is your name? (string)
- `COLLABORATORS`: Other than 6.034 staff, whom did you work with on this lab? (string, or empty string if you worked alone)
- `HOW_MANY_HOURS_THIS_LAB_TOOK`: Approximately how many hours did you spend on this lab? (number or string)
- `WHAT_I_FOUND_INTERESTING`: Which parts of this lab, if any, did you find interesting? (string)
- `WHAT_I_FOUND_BORING`: Which parts of this lab, if any, did you find boring or tedious? (string)
- (optional) `SUGGESTIONS`: What specific changes would you recommend, if any, to improve this lab for future years? (string)

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)