This assignment is meant to enhance your understanding of constraint satisfaction problems (CSP) and the techniques for solving them.

**Concepts**

* *Constraint satisfaction with depth first search*. This is the same procedure as the DFS we used for searching. In this case, the nodes are partially solved constraint problems, and the branches are variable assignments.
* *Constraint propagation*. The process of checking possible variable assignments using existing constraints. Equivalent to arc consistency checking.
* *Domain reduction*. This is a process for eliminating impossible values from a variable’s domain, reducing the space we have to search. For a particular variable *v*, we can check to see if one if its neighbors *n* has a value *nval* that will violate some constraint for every values in *v*’s domain. If so, we can eliminate *nval* from *n*’s domain. We can then check *n*’s neighbors to see if their domains have been reduced as well. If we perform this check at every step during the search, we may greatly reduce the search space. This approach is called propagation through domain reduction, or through reduced domains.
* *Singleton propagation*. Unfortunately, propagation through reduced domains is too expensive to be practical in most problems. There is, however, a technique that detects fewer dead ends, but is significantly faster. In *propagation through singleton domains*, we only check nodes if we have reduced their domain to exactly one value.
* *Forward checking*. This technique is even more restrictive than singleton propagation. In forward checking, we only check *v*’s neighbors, not the neighbors’ neighbors.
* Forward checking + singleton propagation. A combination of the two techniques (described below).

To work on this assignment, you will need to get csp\_lab.zip from Canvas.

**CSP solver**

In this portion of the assignment, you will complete the implementation of a general constraint satisfaction problem solver.

We have provided a basic CSP implementation in csp.py. The implementation has the depth-first-search already completed. It even has a basic built in constraint checker. So it will produce the search trees of the kind for DFS w/ back tracking with basic constraint checking. However, it doesn't do forward checking or forward checking + singleton propagation.

So your job is to complete:

forward\_checking(state):

and

forward\_checking\_prop\_singleton(state):

in the file csp\_lab.py. Here state is an instance of CSPState, an object that keeps track of the current variable assignments and domains. These functions are called by the Search algorithm at every node in the search tree. These functions should return False at points at which the Domain Reduction Algorithm would backtrack, and True otherwise (i.e. continue extending).

Here is the (unrefined) pseudocode for the two algorithms.

**Forward Checking**

1. Let X be the variable currently being assigned.

2. Let x be the value being assigned to X.

3. Find all the binary constraints that are associated with X.

4. For each constraint:

1. Let Y be the variable connected to X by that binary constraint.

2. For each variable value y in Y's domain

1. If constraint checking fails for X=x and Y=y

1. Remove y from Y's domain

2. If the domain of Y is reduced down to the empty set, then the entire check fails: return False.

5. If all constraints passed declare success, return True

If you get a state with no current variable assignment (at the Root of the search tree) then you should just return True, since forward checking could only be applied when there is some variable assignment.

**Forward Checking with Propagation through Singletons**

1. Run forward checking, fail if forward checking fails.

2. Find variables with domains of size 1.

3. Create a queue of singleton variables.

4. While singleton queue is not empty

1. Pop off the first singleton variable X (add X to list of visited singletons)

2. Find all the binary constraints that singleton X is associated with.

3. For each constraint therein:

1. Let Y be the variable connected to X by that binary constraint:

2. For each value of y in Y's domain:

1. If constraint check fails for X = (X's singleton value) and Y = y:

1. Remove y from Y's domain

2. If the domain of Y is reduced down to the empty set, then the entire check fails, return False.

4. Check to see if domain reduction produced any new and unvisited singletons; if so, add them to the queue.

5. return True.

**API**

These are some useful functions defined in csp.py that you should use in your code to implement the above algorithms:

**CSPState**: representation of one of the many possible search states in the CSP problem.

get\_current\_variable() - gets the Variable instance being currently assigned. Returns None if we are in the root state, when there are no variable assignments yet.

get\_constraints\_by\_name(variable\_name) - retrieves all the BinaryConstraint objects associated with variable\_name.

get\_variable\_by\_name(variable\_name) - retrieves the Variable object associated with variable\_name.

get\_all\_variables() - gets the list of all Variable objects in this CSP problem.

**Variable**: representation of a variable in these problems.

get\_name() - returns the name of this variable.

get\_assigned\_value() - returns the **assigned** value of this variable. **Returns None if is\_assigned() returns False, that is if the variable hasn't been assigned yet.**

is\_assigned() - returns True if we've made an assignment for this variable.

get\_domain() - returns a copy of the list of the current domain of this variable. Use this to iterate over values of Y.

**You might want to consider using this method to get the singular value of a variable with domain size reduced to 1.**

reduce\_domain(value) - remove value from this variable's domain.

domain\_size() - returns the size of this variable's domain

**BinaryConstraint**: a binary constraint on variable i, j: i -> j.

get\_variable\_i\_name() - name of the i variable

get\_variable\_j\_name() - name of the j variable

check(state, value\_i=value, value\_j=value) - checks the binary constraint for a given CSP state, with variable i set by value i, and variable j set by value j. Returns False if the constraint fails. Raises an exception if value\_i or value\_j are not set or cannot be inferred from state.

**NOTE**: in this implementation of CSPs, constraints are symmetrical; a constraint object exists for each "direction" of a constraint, so you can check for the presence of a constraint by substituting for i and/or j in the most convenient fashion for you.

Here is how you might use the API to get the value of a variable currently being assigned.

var = state.get\_current\_variable()

value = None

if var is not None: # we are not in the root state

value = var.get\_assigned\_value()

# Here value is the value of the variable current being assigned.

Here is how you might use the API to get the singular value from a singleton variable:

if singleton\_var.domain\_size() == 1

value = singleton\_var.get\_domain()[0]

**Testing**

Your code will be tested using moose\_csp.py, an implementation of the seating problem (described below) involving a Moose, Palin, McCain, Obama, Biden and You -- in terms of the framework as defined in csp.py.

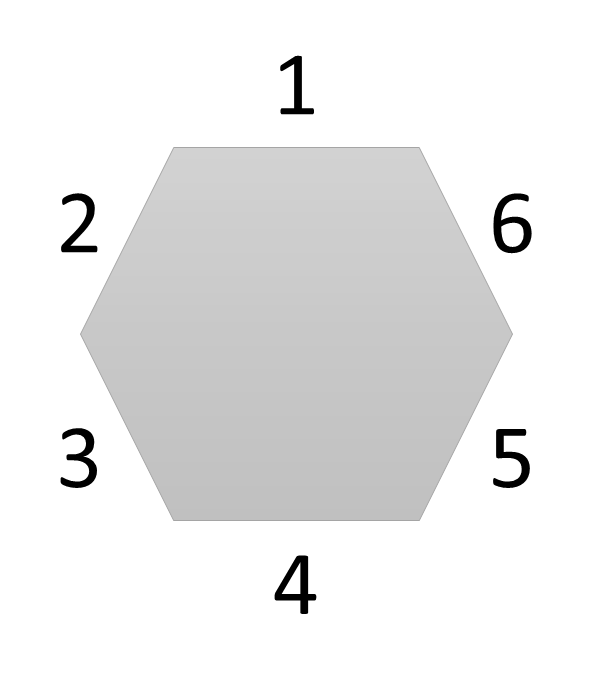
Running:

python moose\_csp.py dfs

will return the search tree for DFS with constraint checking. When you have finished your implementation, running python moose\_csp.py fc or python moose\_csp.py fcps should return the correct search trees under forward checking and forward checking with singleton propagation.

**Problem: Dinner with a Moose**

You just bought a 6-sided table (because it looks like a benzene ring) and want to hold a dinner party. You invite your 4 best friends: McCain, Obama, Biden and Palin. Luckily a moose wanders by and also accepts your invitation. Counting yourself, you have 6 guests for seats labeled 1-6.



Your guests have seven seating demands:

1. Palin wants to sit next to McCain
2. Biden wants to sit next to Obama
3. Neither McCain nor Palin will sit next to Obama or Biden
4. Neither Obama nor Biden will sit next to McCain or Palin
5. The moose is afraid to sit next to Palin
6. No two people can sit in the same seat, and no one can sit in 2 seats.
7. McCain insists on sitting in seat 1

**Part A**

You realize there are 2 ways to represent this problem as a constraint problem. For each below, run the domain reduction algorithm (by hand) and continue to propagate through domains reduced to one value. That is, cross out all the impossible values in each domain without using any search.

Variables: You, Moose, McCain, Palin, Obama, Biden

Domains: Seats 1-6

Constraints: I-VII

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| You: | 1 | 2 | 3 | 4 | 5 | 6 |
| Moose: | 1 | 2 | 3 | 4 | 5 | 6 |
| McCain: | 1 | 2 | 3 | 4 | 5 | 6 |
| Palin: | 1 | 2 | 3 | 4 | 5 | 6 |
| Obama: | 1 | 2 | 3 | 4 | 5 | 6 |
| Biden: | 1 | 2 | 3 | 4 | 5 | 6 |

Variables: Seats 1-6

Domains: You, Moose, McCain, Palin, Obama, Biden

Constraints: I-VII

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1: | You | Moose | McCain | Palin | Obama | Biden |
| 2: | You | Moose | McCain | Palin | Obama | Biden |
| 3: | You | Moose | McCain | Palin | Obama | Biden |
| 4: | You | Moose | McCain | Palin | Obama | Biden |
| 5: | You | Moose | McCain | Palin | Obama | Biden |
| 6: | You | Moose | McCain | Palin | Obama | Biden |

**Part B**

Implement the forward\_checking and forward\_checking\_prop\_singleton functions as outlined above. Run python tester.py and include the results here. Place the contents of the code directory, including your modified csp\_lab.py file, in a csp\_lab folder on your shared drive.

**Part C**

Obama and Biden hear that you've got a Nintendo Wii, and now both of them want to sit next to you. Keeping all the other constraints the same, describe what happens to your dinner party.

**Fun with CSPs**

There are two other solved CSP problems in the directory that you can test and play around with.

Running:

python map\_coloring\_csp.py [dfs|fc|fcps]

Should return the expected search trees for the following B,Y,R, state coloring problem:

You are asked to color a map of some states and regions of the United States according to how pollsters think they will vote in the next election. You created the following constraint network, which contains:

* Nodes (which are the variables representing states that you will color)
* Edges (which represent the constraints between variables)
* Domains (the possible colors for each state or region, also listed in each node)

You also know that Massachusetts will vote Blue, and that Texas will vote Red. For this reason, you have restricted the value of those domains to just B and R, respectively. If you’ve determined the vote of a particular state, no surrounding state may be the opposite color. Whenever possible, you want to color surrounding states the same color, but if that is not possible, you decide to color them yellow, which acts as a buffer between the blue and red states. However, yellow states cannot share a border. The constraint used is NOT THE SAME AS IN THE MAP COLORING PROBLEM. Instead, the edge between states (or regions) indicates:

• R-B, B-R, and Y-Y pairs are not allowed

• R-R, B-B, R-Y, B-Y are allowed by the constraint New England {R, B, Y} Great Lakes {R, B, Y} New York {R, B} Texas {R} South {R, B} Florida {R, B}

Running

python sudoku\_csp.py [dfs|fc|fcps] should solve a 9-square Sudoku puzzle. Caution: This may take a very long time using dfs, and will likely error out before completion.

**EXTRA CREDIT**

As extra credit, try to follow the code in moose\_csp.py or map\_coloring\_csp.py, and implement a problem() function that returns a CSP instance for a problem of your own choosing.

You may do one of the problems from lectures, or you may implement something that you find useful or interesting, ideas include: scheduling classes, seating guests for a wedding or dinner party (to maximize harmony), solving crypt-arithmetic puzzles, or crossword puzzles.

You may also try to extend csp.py. For instance, you can add ability to find an optimal solution rather than just a constraint-satisfying solution (i.e. replace DFS with one of the optimal searches we've learned).

When you've succeeded in implementing such a problem or extension, send your working code to me. Your reward: either an automatic 5-day extension on any future assignment, or a 20 point bonus. Or if your grade is already perfect, praise and admiration of your classmates.