Problem Set 4

To work on this problem set, you will need to get the code.

This lab has two parts. The first part is on CSPs and the second part is on learning algorithms, specifically KNN and decision trees.

Constraint Satisfaction Problems

In this portion of Lab 4, you are to complete the implementation of a general constraint satisfaction problem solver. You'll test it on problems we've worked out by hand in class.

We have provided you 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 `lab4.py`. Here `state` is an instance of `CSPState` an object that keep 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).

As a hint, 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. For each neighbor variable, Y, connected to X by a binary constraint.
        1. 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 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 single 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. For each neighbor variable, Y, connected to X by a binary constraint:
            1. 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 our 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

For unit testing, we have provided `moose_csp.py`, an implementation of the seating problem 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.

Similarly

Running:

```
python map_coloring_csp.py [dfs|fc|fcps]
```

Should return the expected search trees for the B,Y,R, state coloring problem from the 2nd Quiz in 2006.

There are also other fun solved CSP problems in the directory that you can test and play around with. You can submit your own unique solution to an interesting CSP problem to get extra credit!

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 past quizzes: the 2009 Time Traveler scheduling problem, the 2010 Jigsaw puzzle question or the phoneme-syllabification problem (from tutorial). Alternately, 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, the 8-queens problem, 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). Or you can add support for multi-variable constraints, and make the code solve the Max-flow problem from the 2006 final.

When you've succeeded in implementing such a problem or extension, send your working code to the TAs. Your reward: either a 1-to-3-day extension (depending on difficulty) on one of the previous or future labs, possibly erasing any late penalties. Or if your lab grade is already perfect, praise and recognition from the 6.034 staff.

Learning

Now for something completely different. Learning!

Classifying Congress

During Obama's visit to MIT, you got a chance to impress him with your analytical thinking. Now, he has hired you to do some political modeling for him. He seems to surround himself with smart people that way.

He takes a moment out of his busy day to explain what you need to do. "I need a better way to tell which of my plans are going to be supported by Congress," he explains. "Do you think we can get a model of Democrats and Republicans in Congress, and which votes separate them the most?"

"Yes, we can!" You answer.

The Data

You acquire the data on how everyone in the previous Senate and House of Representatives voted on every issue. (These data are available in machine-readable form via `voteview.com`. We've included it in the lab directory, in the files beginning with `H110` and `S110`.)

`data_reader.py` contains functions for reading data in this format.

`read_congress_data("FILENAME.ord")` reads a specially-formatted file that gives information about each Congressperson and the votes they cast. It returns a list of dictionaries, one for each member of Congress, including the following items:

- 'name': The name of the Congressperson.
- 'state': The state they represent.
- 'party': The party that they were elected under.
- 'votes': The votes that they cast, as a list of numbers. 1 represents a "yea" vote, -1 represents "nay", and 0 represents either that they abstained, were absent, or were not a member of Congress at the time.

To make sense of the votes, you will also need information about what they were voting on. This is provided by `read_vote_data("FILENAME.csv")`, which returns a list of votes in the same order that they appear in the Congresspeople's entries. Each vote is represented a dictionary of information, which you can convert into a readable string by running `vote_info(vote)`.

The lab file reads in the provided data, storing them in the variables `senate_people`, `senate_votes`, `house_people`, and `house_votes`.

Nearest Neighbors

You decide to start by making a nearest-neighbors classifier that can tell Democrats apart from Republicans in the Senate.

We've provided a `nearest_neighbors` function that classifies data based on training data and a distance function. In particular, this is a third-order function:

- First, call `nearest_neighbors(distance, k)`, with `distance` being the distance function you wish to use and `k` being the number of neighbors to check. This returns a *classifier factory*.
- A classifier factory is a function that makes classifiers. You call it with some training data as an argument, and it returns a classifier.
- Finally, you call the classifier with a data point (here, a Congressperson) and it returns the classification as a string.

Much of this is handled by the `evaluate(factory, group1, group2)` function, which you can use to test the effectiveness of a classification strategy. You give it a classifier factory (as defined above) and two sets of data. It will train a classifier on one data set and test the results against the other, and then it will switch them and test again.

Given a list of data such as `senate_people`, you can divide it arbitrarily into two groups using the `crosscheck_groups(data)` function.

One way to measure the "distance" between Congresspeople is with the *Hamming distance*: the number of entries that differ. This function is provided as `hamming_distance`.

An example of putting this all together is provided in the lab code:

```
senate_group1, senate_group2 = crosscheck_groups(senate_people)
evaluate(nearest_neighbors(edit_distance, 1), senate_group1, senate_group2,
verbose=1)
```

Examine the results of this evaluation. In addition to the problems caused by independents, it's classifying Senator Johnson from South Dakota as a Republican instead of a Democrat, mainly because he missed a lot of votes while he was being treated for cancer. This is a problem with the distance function -- when one Senator votes yes and another is absent, that is less of a "disagreement" than when one votes yes and the other votes no.

You should address this. Euclidean distance is a reasonable measure for the distance between lists of discrete numeric features, and is the alternative to Hamming distance that you decide to try. Recall that the formula for Euclidean distance is:

[(x1 - y1)^2 + (x2 - y2)^2 + ... + (xn - yn)^2] ^ (1/2)

- Make a distance function called `euclidean_distance` that treats the votes as high-dimensional vectors, and returns the Euclidean distance between them.

When you evaluate using `euclidean_distance`, you should get better results, except that some people are being classified as Independents. Given that there are only 2 Independents in the Senate, you want to avoid classifying someone as an Independent just because they vote similarly to one of them.

- Make a simple change to the parameters of `nearest_neighbors` that accomplishes this, and call the classifier factory it outputs `my_classifier`.

ID Trees

So far you've classified Democrats and Republicans, but you haven't created a model of which votes distinguish them. You want to make a classifier that explains the distinctions it makes, so you decide to use an ID-tree classifier.

`idtree_maker(votes, disorder_metric)` is a third-order function similar to `nearest_neighbors`. You initialize it by giving it a list of vote information (such as `senate_votes` or `house_votes`) and a function for calculating the disorder of two classes. It returns a classifier factory that will produce instances of the `CongressIDTree` class, defined in `classify.py`, to distinguish legislators based on their votes.

The possible decision boundaries used by `CongressIDTree` are, for each vote:

- Did this legislator vote YES on this vote, or not?
- Did this legislator vote NO on this vote, or not?

(These are different because it is possible for a legislator to abstain or be absent.)

You can also use `CongressIDTree` directly to make an ID tree over the entire data set.

If you `print` a `CongressIDTree`, then you get a text representation of the tree. Each level of the ID tree shows the minimum disorder it found, the criterion that gives this minimum disorder, and (marked with a +) the decision it makes for legislators who match the criterion, and (marked with a -) the decision for legislators who don't. The decisions are either a party name or another ID tree. An example is shown in the section below.

An ID tree for the entire Senate

You start by making an ID tree for the entire Senate. This doesn't leave you anything to test it on, but it will show you the votes that distinguish Republicans from Democrats the most quickly overall. You run this (which you can uncomment in your lab file):

```
print CongressIDTree(senate_people, senate_votes, homogeneous_disorder)
```

The ID tree you get here is:

```
Disorder: -49
Yes on S.Con.Res. 21: Kyl Amdt. No. 583; To reform the death tax by setting
the
exemption at $5 million per estate, indexed for inflation, and the top death
tax rate at no more than 35% beginning in 2010; to avoid subjecting an
estimated 119,200 families, family businesses, and family farms to the death
tax each and every year; to promote continued economic growth and job
creation;
and to make the enhanced teacher deduction permanent.:
+ Republican
- Disorder: -44
```

```
  Yes on H.R. 1585: Feingold Amdt. No. 2924; To safely redeploy United States
  troops from Iraq.:
  + Democrat
  - Disorder: -3
    No on H.R. 1495: Coburn Amdt. No. 1089; To prioritize Federal spending to
    ensure the needs of Louisiana residents who lost their homes as a result
of
    Hurricane Katrina and Rita are met before spending money to design or
    construct a nonessential visitors center.:
    + Democrat
    - Disorder: -2
      Yes on S.Res. 19: S. Res. 19; A resolution honoring President Gerald
      Rudolph Ford.:
      + Disorder: -4
        Yes on H.R. 6: Motion to Waive C.B.A. re: Inhofe Amdt. No. 1666; To
        ensure agricultural equity with respect to the renewable fuels
standard.:
        + Democrat
        - Independent
      - Republican
```

Some things that you can observe from these results are:

- Senators like to write bills with very long-winded titles that make political points.
- The key issue that most clearly divided Democrats and Republicans was the issue that Democrats call the "estate tax" and Republicans call the "death tax", with 49 Republicans voting to reform it.
- The next key issue involved 44 Democrats voting to redeploy troops from Iraq.
- The issues below that serve only to peel off homogenous groups of 2 to 4 people.

Implementing a better disorder metric

You should be able to reduce the depth and complexity of the tree, by changing the disorder metric from the one that looks for the largest homogeneous group to the information-theoretical metric described in lecture.

You can find this formula on page 429 of the textbook.

- Write the `information_disorder(group1, group2)` function to replace `homogeneous_disorder`. This function takes in the lists of classifications that fall on each side of the decision boundary, and returns the information-theoretical disorder.

Example:

```
information_disorder(["Democrat", "Democrat", "Democrat"], ["Republican",
"Republican"])
  => 0.0
information_disorder(["Democrat", "Republican"], ["Republican", "Democrat"])
  => 1.0
```

Once this is written, you can try making a new CongressIDTree with it. (if you're having trouble, keep in mind you should return a float or similar)

Evaluating over the House of Representatives

Now, you decide to evaluate how well ID trees do in the wild, weird world of the House of Representatives.

You can try running an ID tree on the entire House and all of its votes. It's disappointing. The 110th House began with a vote on the rules of order, where everyone present voted along straight party lines. It's not a very informative result to observe that Democrats think Democrats should make the rules and Republicans think Republicans should make the rules.

Anyway, since your task was to make a tool for classifying the newly-elected Congress, you'd like it to work after a relatively small number of votes. We've provided a function, `limited_house_classifier`, which evaluates an ID tree classifier that uses only the most recent *N* votes in the House of Representatives. You just need to find a good value of *N*.

- Using `limited_house_classifier`, find a good number *N_1* of votes to take into account, so that the resulting ID trees classify at least 430 Congresspeople correctly. How many training examples (previous votes) does it take to predict at least 90 senators correctly? What about 95? **To pass the online tests**, you will need to find close to the minimum such values for N_1, N_2, and N_3. Keep guessing to find close to the minimum that will pass the offline tests. Do the values surprise you? Is the house more unpredictable than the senate, or is it just bigger?

- Which is better at predicting the senate, 200 training samples, or 2000? Why?

The total number of Congresspeople in the evaluation may change, as people who didn't vote in the last *N* votes (perhaps because they're not in office anymore) aren't included.

Survey

Please answer these questions at the bottom of your `ps4.py` file:

- How many hours did this problem set take?
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

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

Errata

In the code in the section for finding N_2 such that using N_2 votes classifies at least 90 senators correctly, lab4.py creates senator_classified using house_people and house_votes. Our fault.

Please just take it that you're to find the n required to classify 90 house members. It will still be instructive to find the smallest required value for the senate, but please leave the variable names and tests as they are.

6.034 Artificial Intelligence

Fall 2010