Introduction

The projects for this class assume you use Python 3.6.

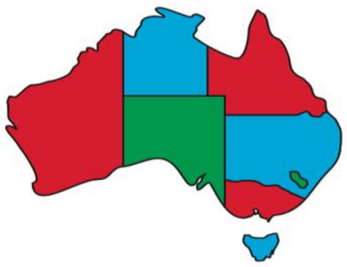
Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's judgements -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, section, and the discussion forum are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Solutions should be send to your instructor: basa.matei@gmail.com or nistor.grozavu@gmail.com

Lab 3: CSP Problem (due April 16 at 6:pm)



How can we color **with** no adjacent colors **this** Australia map?

Table of Contents

- Introduction
- Before You Begin
- Q1: Recursive Backtracking
- Q2: Maintaining Arc Consistency
- Q3: Preprocessing
- Submission

Introduction

A constraint satisfaction problem (CSP) is a problem specified such that a solution is an assignment of values to variables that is valid given constraints on the assignment and the variables' domains. CSPs are very powerful because a single unchanging set of algorithms can be used to solve any problem specified as a CSP, and many problems can be intuitively specified as CSPs. In this project you will be implementing a CSP solver and improving it with heuristic and inference algorithms.

For this project, your are provided with an autograder as well as many test cases that specify CSP problems. These test cases are specified within the csps directory. Functions are provided in Testing to parse these files into the objects your algorithms will work with. The files follow a simple format you can understand by inspection, so if your code does not work looking at the problems themselves and manually checking your logic is the best debugging strategy.

Files you will edit

BinaryCSP.py Your entire CSP implementation will be within this file

Files you will not edit

Testing.py Helper functions for invoking CSP problems **autograder.pyc** Acustom autograder to check your code with

Before You Begin: A Note on Structure

All of the necessary structure for assignments and csp problems is provided for you in BinaryCSP.py. While you do not need to implement these structures, it is important to understand how they work.

Almost every function you will be implementing will take in a *Constraint Satisfaction Problem* and an *Assignment*. The *Constraint SatisfactionProblem* object serves only as a representation of the problem, and it not intended to be changed. It holds three things: a dictionary from variables to their domains (*varDomains*), a list of binary constraints(*binaryConstraints*), and a list of unary constraints (*unaryConstraints*). An *Assignment* is constructed from a *ConstraintSatisfactionProblem* and is intended to be updated as you search for the solution. It holds a dictionary from variables to their domains (*varDomains*) and a dictionary from variables to their assigned values (*assignedValues*). Notice that the *varDomains* in *Assignment*is meant to be updated, while the *varDomains* in *ConstraintSatisfactionProblem* should be left alone.

A new assignment should never be created. All changes to the assignment through the recursive backtracking and the inference methods that you will be implementing are designed to be reversible. This prevents the need to create multiple assignment objects, which becomes very spaceconsuming.

The constraints in the csp are represented by two classes: *BinaryConstraint* and *UnaryConstraint*. Both of these store the variables affected and have an *isSatisfied* function that takes in the value(s) and returns False if the constraint is broken. You will only be working with binary constraints, as *eliminateUnaryConstraints* has been implemented of for you. Two useful methods for binary constraints include *affects*, which takes in a variable and returns True if the constraint has any impact on the variable, and *otherVariable*, which takes in one variable of the binaryConstraint and returns the other variable affected.

Question 1: Recursive Backtracking

In this question you will be creating the basic recursive backtracking framework for solving a constraint satisfaction problem. First implement the function *consistent*. This function indicates whether a given value would be possible to assign to a variable without violating any of its constraints. You only need to consider the constraints in csp.binaryConstraints that affect this variable and have the other affected variable already assigned.

Once this is done implement *recursiveBacktracking*. This function is designed to take in a problem definition and a partial assignment. When finished, the assignment should either be a complete solution to the CSP or indicate failure. Note that for now your implementation will not use any inferences (what forward checking finds), as this will be implemented later.

Remark: The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown below.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure

return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment,csp) returns a solution, or failure

if assignment is complete then return assignment
var ←SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)
for each value in ORDER-DOMAIN-VALUES(var,assignment,csp) do

if value is consistent with assignment according to CONSTRAINTS[csp] then

        add {var = value} to assignment
        result ← RECURSIVE-BACKTRACKING(assignment, csp)
        if result/= failure then return result

        remove {var = value} from assignment

return failure
```

To test and debug, use both the autograder and the functions in Testing.py. Be aware that individual questions and tests can be set for grading to the autograder with q and t respectively.

Some Notes on Functions

About   the Variable Selection

While the recursive backtracking method eventually finds a solution for a constraint satisfaction problem, this basic solution will take a very long time for larger problems. Fortunately, there are heuristics that can be used to make it faster. One place to include heuristics is in selecting which variable to consider next.

About Forward Checking

While heuristics help to determine what to attempt next, there are times when a particular search path is doomed to fail long before all of the values have been tried. Inferences are a way to identify impossible assignments early on by looking at how a new value assignment affects other variables. Each inference made by an algorithm involves one possible value being removed from one variable. It should be noted that when these inferences are made they must be kept track of so that they can later be reversed if a particular assignment fails.

This is implemented in *forwardChecking*. This is a very basic inference making function. When a value is assigned, all variables connected to the assigned variable by a binary constraint are considered. If any value in those variables is inconsistent with that constraint and the newly assigned value, then the inconsistent value is removed.

About Maintaining Arc Consistency

There are other methods for making inferences than can detect inconsistencies earlier than forward checking. One of these is the Maintaining Arc Consistency algorithm, or the MAC algorithm with the pseudocode given below:

```
function REMOVE-INCONSISTENT-VALUES($X_i$, $X_j$) returns true iff we remove a value removed ← false
for each x in DOMAIN[$X_i$] do

if no value y in DOMAIN[$X_j$] allows (x,y) to satisfy the constraint between $X_i$ and $X_j$ then

delete x from DOMAIN[$X_i$]; removed ← true

return removed
```

This is implemented in *maintainArcConsistency*. The MAC algorithm starts off very similar to forward checking in that it removes inconsistent values from variables connected to the newly assigned variable. The different is that is uses a queue to propagate these changes to other related variables. The revise function is implemented. This helper function for MAC remove values from var domain if constraint cannot be satisfied. Note that Each inference should take the form of (variable, value) where the value is being removed from the domain of variable. This format is important so that the inferences can be reversed if they result in a conflicting partial assignment. If the algorithm reveals an inconsistency, any inferences made should be reversed before ending the function.

Question 2: Preprocessing

Another step to making a constraint satisfaction solver more efficient is to perform preprocessing. This can eliminate impossible values before the recursive backtracking even starts. One method to do this is to use the AC3 algorithm. You could use again the revise helper function.

```
function AC-3(csp) returns the CSP, possibly with reduced domains inputs: csp, a binary CSP with variables
{$X_1, X_2, \ldots, X_n$} local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    ($X_i$, $X_j$)←REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES($X_i$, $X_j$) then
        for each $X_k$ in NEIGHBORS[$X_i$] do add ($X_k$, $X_i$) to queue
```

Implement *AC3*. This algorithm is almost identical to MAC in that it propagates inferences in order to remove as many values as possible. The major difference is that instead of starting with one variable, AC3 begins with all variables already in the queue. Thus, reusing MAC is a fast way to implement this AC3. It also does not need to track the inferences that are made, because if the assignment fails at any point then there is no prior state to back up to. This means that there is no solution to the CSP.

Submission

Send to your instructor your modified **BinaryCSP.py** before the due date.