Project 4: Constraint Satisfaction Problems (CSPs)

Deadline: 2019-11-22 23:59

November 6, 2019

Abstract

The goal of this project is to help you better understand CSPs and how they are solved.

Introduction

A constraint satisfaction problem (CSP) is a problem whose a solution is an assignment of values to variables that respects the constraints on the assignment and the variables domains. CSPs are very powerful because a single fixed 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 to Edit and Submit: You will submit `BinaryCSP.py` where your entire CSP implementation will reside. You should submit this file (and only this one) with your code and comments to online judge.

Evaluation: We provide you an autograder for you to pre-test. Run

```
python student_autograder.py -t ./test_cases/example.test
```

We have provided you some test cases and the solutions using a simple CSP and a more complicated Sudoku one. They are inside the `test_case` folder. You need to replace `example.test` in the above command by the name of any of the test files.
Your code will be autograded for technical correctness. 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 and the discussion forum on Piazza 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.

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 `ConstraintSatisfactionProblem` and an `Assignment`. The `ConstraintSatisfactionProblem` object serves only as a representation of the problem, and it is 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 space-consuming.
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 (4 points): 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` (see Slide 25 of CSP I). Ignore for now the parameter `inferenceMethod`.
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.

Question 2 (2 points): Variable Selection

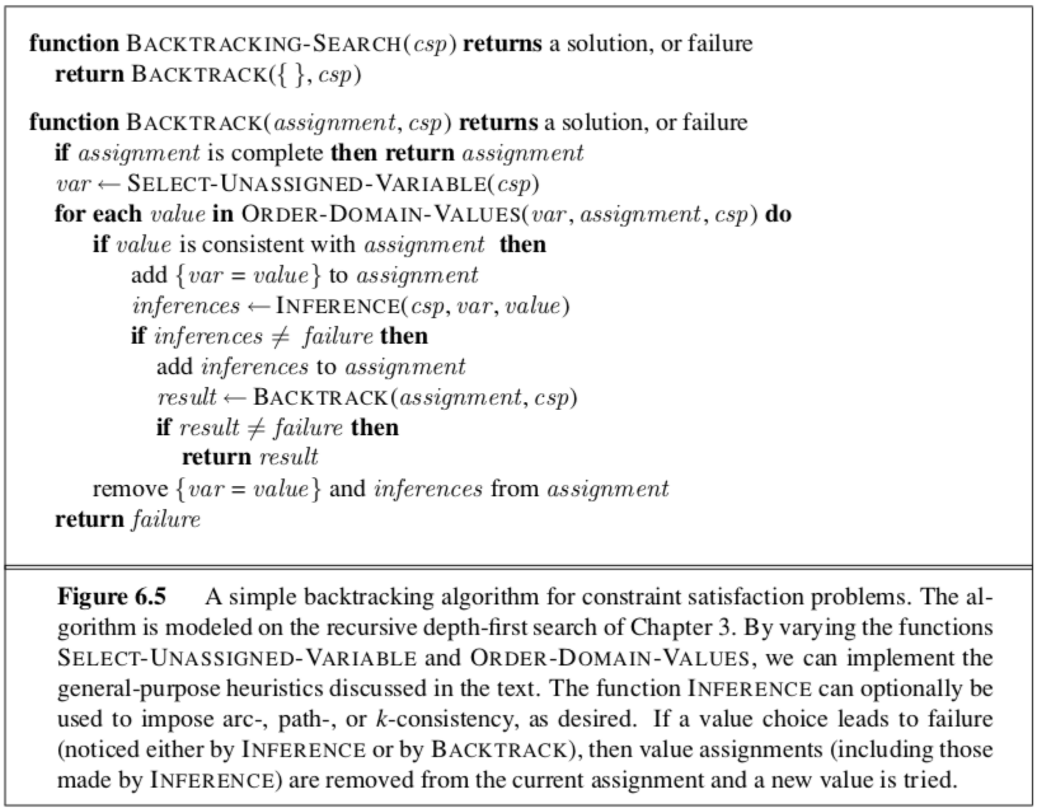
While the recursive backtracking method eventually finds a solution for a constraint satisfaction problem, this basic solution will take a very long time for larger problems. Fortunately, there are heuristics that can be used to make it faster. One place to include heuristics is in selecting which variable to consider next. Implement `minimumRemainingValueHeuristic`. This follows the minimum remaining value heuristic (see Slide 40 of CSP I) to select a variable with the fewest options left and uses the degree heuristic to break ties. The degree heuristic chooses the variable that is involved in the largest number of constraints on other unassigned variables.

Question 3 (2 points): Value Ordering

Another way to use heuristics is to optimize a constraint satisfaction problem solver is to attempt values in a different order. Implement `leastConstrainingValuesHeuristic`. This takes in a variable and determines the order in which the possible values should be attempted according to the least constraining values heuristic (see Slide 41 of CSP I), which prefers values that eliminate the fewest possibilities from other variables.
Your code should be able to solve small CSPs. To test and debug, use both the autograder and the functions in `Testing.py`.

Question 4 (4 points): 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 (see Slides 28-36 of CSP I) are a way to identify impossible assignments early on by looking at how a new value assignment affects other variables.
The pseudocode of `recursiveBacktracking` that integrates inferences is given in Figure 1. 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. They are stored as a set of tuples (variable, value). Update `recursiveBacktracking` to deal with the parameter `inferenceMethod`.
Implement `forwardChecking`. This is a very basic `inferencemaking` 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.
You can test again your code by passing `forwardChecking` to `inferenceMethod`. It should be faster than the previous version in Question 3.



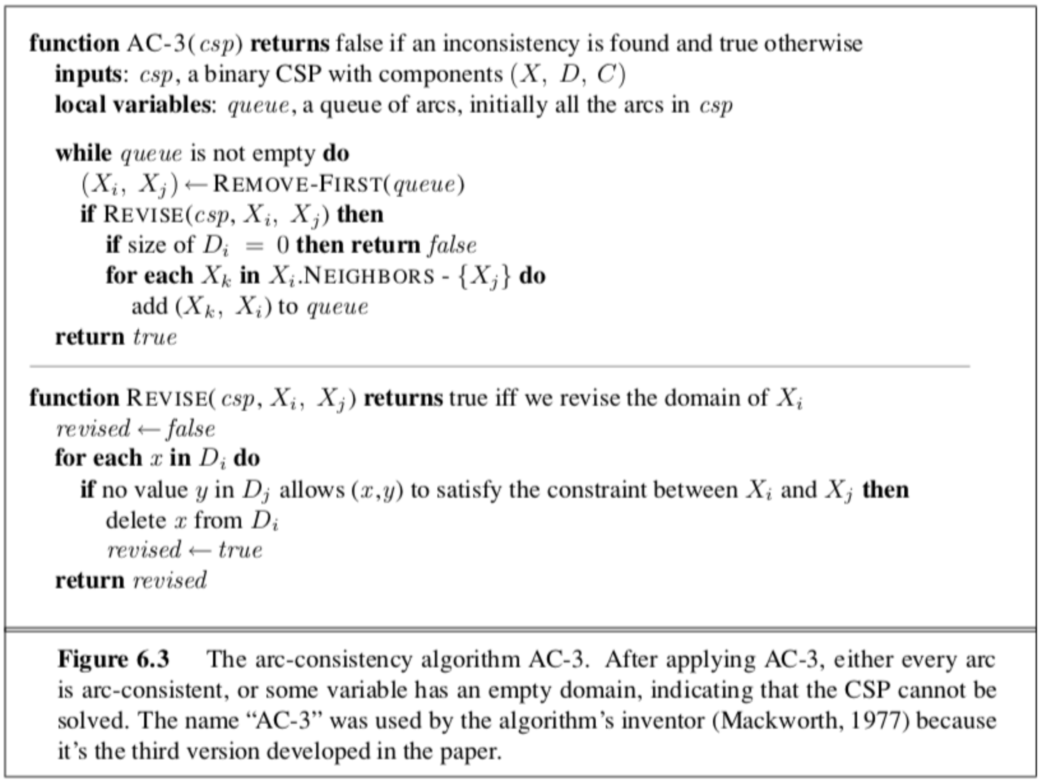
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
          add inferences to assignment
          result ← BACKTRACK(assignment, csp)
          if result ≠ failure then
            return result
     remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

Figure 1: Pseudo-code for recursive backtracking.

Question 5 (4 points): 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. It works like the AC-3 algorithm (see Slide 34 of CSP I). We recall in Figure 2 a version of AC-3 where inferences are saved. The difference between MAC and AC-3 are as follows. While AC-3 is called as a preprocessing step (which explains why all arcs inserted in the queue), MAC is called during the search. After a variable X_i is assigned a value, MAC performs the same operations as AC-3, but starts with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i.
First implement `revise`. This is a helper function that is responsible for determining inconsistent values in a variable. Then implement
`maintainArcConsistency`. The MAC algorithm starts off very similar to



```
function AC-3($csp$) returns false if an inconsistency is found and true otherwise
  inputs: $csp$, a binary CSP with components $(X, D, C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in $csp$

  while $queue$ is not empty do
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE($csp, X_i, X_j$) then
      if size of $D_i = 0$ then return $false$
      for each $X_k$ in $X_i$.NEIGHBORS - $\{X_j\}$ do
        add $(X_k, X_i)$ to $queue$
  return $true$

function REVISE( $csp, X_i, X_j$) returns true iff we revise the domain of $X_i$
  $revised \leftarrow false$
  for each $x$ in $D_i$ do
    if no value $y$ in $D_j$ allows $(x,y)$ to satisfy the constraint between $X_i$ and $X_j$ then
      delete $x$ from $D_i$
      $revised \leftarrow true$
  return $revised$
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

Figure 2: Pseudo-code for AC-3.

forward checking in that it removes inconsistent values from variables connected to the newly assigned variable. The difference is that is uses a queue to propagate these changes to other related variables.

Question 6 (2 points): 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 AC-3 algorithm. Implement AC-3. Note it 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

Submit `BinaryCSP.py` in a zip file to online judge, see the announcement later.