Project 2: Constraint Satisfaction Problems (CSPs)

Due: 11:59pm, Jun.21, 2024

Introduction

A constraint satisfaction problem (CSP) is a problem whose 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.

The goal of this project is to help you better understand CSPs and how they are solved. In this project you will be implementing a CSP solver and improving it with heuristic and inference algorithms.

Starter Files

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. The tree diagram of the starter system is as follows.

```
.
+-- BinaryCSP.py
+-- Interface.py
+-- StudentAutograder.py
+-- Testing.py
+-- csps
|   +-- csp7.assignment
|   +-- csp7.csp
|   +-- ...
+-- test_cases
    +-- backtracking3solvable1_1.solution
    +-- backtracking3solvable1_1.test
    +-- ...
```

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.

Data Structure

All the necessary interface structures for assignments and CSP problems are provided for you in `Interface.py`, and every function you will be implementing are marked with `TODO` 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`
- 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`)
- 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`)
- 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`
- `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 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.

Questions

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`

- 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: 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, and one place to include heuristics is in selecting which variable to consider next.

Implement `minimumRemainingValueHeuristic`.

- This follows the minimum remaining value heuristic 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: Value Ordering

Another way to use heuristics 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, 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 `StudentAutograder.py` 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 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. They are stored as a set of tuples (variable, value).

First, update `recursiveBacktracking` to deal with the parameter `inferenceMethod`.

Then, 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.**

Question 5: 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 (MAC) algorithm, which is an adaptation of the AC-3 algorithm. The differences between MAC and AC-3 are as follows.

- While AC-3 is called as a preprocessing step, which explains why all arcs are 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 unassigned variables X_j that are neighbors of X_i.

First, implement `revise`, which is a helper function that is responsible for determining inconsistent values in a variable.

Then implement `maintainArcConsistency`.

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

Question 6: 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 `AC3`.

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 it to online judge before the due date. Note you just need to compress one file in your .zip submission, `BinaryCSP.py`. Please do not include any other given files.