PA 4 - Constraint Satisfaction

Julien Blanchet | CS 276 F20 (Artificial Intelligence) | Prof. Li.

October 18, 2020

Description

This submission offers a python implementation capable of solving Constraint Satisfaction Problems. I will describe the design of this work component by component:

Generic CSP Model & Solver

This component, coded in `CSPProblem.py` and `CSPSolver.py`, implements the model and solving algorithm for a generic constraint satisfaction problem. These take an object oriented approach–the `CSPProblem` and `CSPSolver` classes can be configured by initialization variables and offer various instance methods to expose their functionality.

I made a few design decisions the the `CSPProblem` model. First and foremost, the model represents the domains of each variable as a finite set of possible values - thus, a continuous range-representation of a domain not possible. Another choice is the type system used - rather than introduce new classes to represent binary constraints, variable domains, etc; I chose to utilize a feature new to python 3.8 - type annotations. By creating type aliases for `VariableIndex`, `BinaryConstraint`, and `VariableDomain`, as well as a generic type for `VariableValue`, this feature allows me to present a clear API while minimizing unnecessary additional abstraction - under the hood, the code is using native python types (tuples, sets, and lists).

Another neat feature of this implementation is with it's printing / string-making functionality. One difficulty with debugging CSP problems arises from genericism of the code - it can be tricky to quickly make the conversion between the generic representation used by the solver and the original problem values / states. By accepting functions to convert variables indices and values to strings, the model can provide a useful string representation for any input.

`CSPSolver` is basically a wrapper around a backtracing CSP solver the provides a few configuration settings upon initialization. Any conbination of variable-selection heuristic, domain-ordering heuristic, and inference-technique can be chosen - supported options are:

Variable Selection	Ordering Heuristic	Inference
Randomly selected, minimum remaining values (MRV), highest degree	Random ordering, least constraining value (LCV)	None, forward checking, ac3 (maintains arc consistency)

In addition, the solver assists with testing by keeping track of search statistics and performing debug logging.

Map Coloring

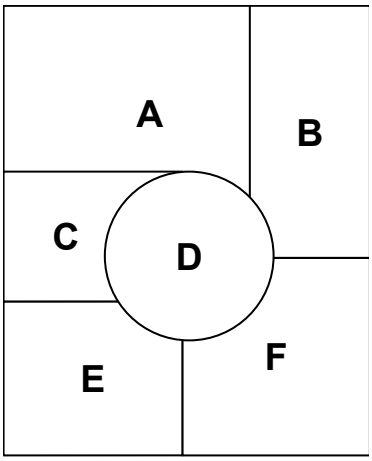
It was fairly straightforward to convert a map-coloring problem to a constraint satisfaction problem for use by the generic solver. `map_coloring.py` impements this process, and takes in tuple pairs of bordering countries (or states / provinces, etc.). The color of each country is its own variable, and the domains are all initialized to the possible color values (as proven elsewhere, no more than 4 colors are ever needed, but some maps can be colored with less and this implementation allows creation of CSP problems for those circumstances).

Circuit-Board Layout Problem

Converting the circuit board layout problem to a representation compatible with my generic CSP solver was slighly more tricky - mostly because I had started with the assumption that all variable values would be integers. To accomodate the circuit layout problem, I rewrote the solver to accept generic values for the variable domains and values - so long as it's a hashable and equatable type, it'll work. This allowed me to represent the component locations (x, y) as a single variable rather than two separate ones, and allowed me to encode the "cannot overlap" constraint as a binary constraint rather than a 4-way constraint without the use of hidden values. The string representation of `CSPProblem` was in need of improvement for this problem, so I added a `print_layout_csp` method that prints a nice ASCII representation of the board.

Evaluation

As in my previous assignments, my tests are implemented in separate test files anre are designed to be executed with `pytest`. For `map_coloring`, I validated the functionality of the model with few examples, including the actual national borders of South America, a simple chain, the example from the textbook, and the contribed example shown below. In all cases, the algorithm successfully finds a valid coloring, except for when I ask it to generate a 3-color coloring for the contrived example, which correctly fails to find a solution (as no such coloring is possible). I did some performance analysis of the impact of heuristics on the map coloring problem, but found that the problem sizes were insufficiently large

to capture meaningful results. 

The algorithm also successfully solves circuit layout problems. In addition to testing critical methods of my implementation (that the constraint generation ouputs the correct / expected pairs of non-overlapping positions given two pieces and that the inital domains of components contains all the expected possible locations), I also took care to validate the null case (when a placement isn't possible). The circuit-layout problems were larger and more challenging for the solver as compared to the map-coloring problems, so I chose to do my performance analysis on these instead.

Solver	Attempted Assignments	Solving Time
Backtrace only	258,784	70.06 secs
MRV	321	0.08 secs
MRV, forward checking	623	0.41 secs
Degree	4,838	1.14 secs
Random select, LCV order	21,577	15.57 secs
Random sel & order, forward checking	250,187	148.54 secs
Random sel & order, ac3	47,323	28 secs
MRV, LCV, Forward Checking	87	0.83 secs
MRV, LCV, AC3	87	0.77 secs

A few observations here:

- The results that used random variable selection (as opposed to MRV or *degree* heuristics) exhibited a large variation in performance across several trials. Sometimes they'd complete within a second and sometimes would take a few minutes.
- The $Select-Variable$ heuristic seemed to have the most dramatic impact on performance, with MRV being faster than *degree*. In most cases, we'd expect MRV to rely on forward checking (or a more powerful inference technique) to prune the domain first - however, I believe MRV was still

effective by itself in this case because it led the solver to attempt to place the larger components first.

- The combination of both heuristics and inference yielded the best and most consistent performance. Over many trials, these always attempted exactly 87 assignments and completed in between 0.7s and 0.83s. In these cases, there was little difference between ac3 and forward checking - if anything, forward checking was slightly slower.

Discussion

1. Assuming a 0-indexed board, the domain of a variable corresponding to a component of width w and height h on a circuit board of width n and height m will be all the possible combinations of the following x and y ranges:

$$x \in [0 \dots n - w]$$

$$y \in [0 \dots m - h]$$

   This means there will be a total of $(n - w) \cdot (m - y)$ values in the domain.

2. For the following components: $a : 3 \times 2$, $b : 5 \times 2$ on a 10×3 board, the constraint that enforces that these components may not overlap would contain a set of 48 possible combinations, described by the table below.

a locations	b locations
(0, 0), (0, 1)	(3, 0), (3, 1), (4, 0), (4, 1), (5, 0), (5, 1)
(1, 0), (1, 1)	(4, 0), (4, 1), (5, 0), (5, 1)
(2, 0), (2, 1)	(5, 0), (5, 1)
(6, 0), (6, 1)	(0, 0), (0, 1)
(7, 0), (7, 1)	(0, 0), (0, 1), (1, 0), (1, 1)
(8, 0), (8, 1)	(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)

3. The code for converting the *cannot overlap* constraint into integer values for use by the solver is written in `circuit_layout.py`. The code simply iterates through all combinations of locations in the domains of each of the variables and filters by those that would not cause an overlap. The key segment of code is the following filter function:

```
def not_overlapping(locations: Tuple[Location, Location]) -> bool:
    i_loc, j_loc = locations
    ix, iy = i_loc
    jx, jy = j_loc

    return ix + i_width  <= jx or jx + j_width  <= ix or \
           iy + i_height <= jy or jy + j_height <= iy
```

   The condition is based of the following observation: for two pieces not

to be overlapping, they must be offset in the x direction and/or the y direction.