PA-4 Constraint Satisfaction Problems: Maps and Circuit Boards

Sudharsan Balasubramani Credit: Discussed ideas with James Fleming

Introduction

The purpose of this programming assignment was to develop a general purpose constraint satisfaction problem solving algorithm and apply it to different CSPs. The solving algorithm we will be using is a backtracking algorithm very similar to how DFS works. Take the map coloring problem with Australia. The idea is it will give one of the regions a color, say red. Then it will assign the next region a color, say red again. Backtracking will realize that this region cannot be red because the one next to it is red and so will "backtrack" and choose another color. This is the premise of the backtracking solver and we implement it with heuristics to help it along the way.

Description

General CSP

The GeneralCSP class is a class that houses the general constraint satisfaction problem. This class is later inherited by other classes that hold more specific constraint satisfaction problems, such as MapColoring and CircuitBoard. The three main components of a CSP are variables, the domains associated with those variables and the constraints associated with those variables. My GeneralCSP class takes in a list of variables, a dictionary of associated domains and a list of tuples that represent implicit constraints -- that is, if 0 were Vermont and 1 were New Hampshire and we were doing Map Coloring, then (0, 1) would mean 0 and 1 could not be the same color. These three parameters are taken in as private self variables. The variables are taken in normally but the domains are actually taken in as copies and are stored in not just self.domains, but also self.working_domains. This will be explained more later on, but essentially, they are stored as deep copies of dictionaries so that when they are

manipulated, they don't affect further testing. The reason for working domains and domains being a copy of the same domains var is so that if working domains changes, then domains remains the same. Again, this will become more evident with backtracking algorithm. The constraints are taken into self.implicit_constraints and then in the subclass are converted into explicit constraints. Other constructor variables include Booleans for each of the heuristics. The rest of the class consists of functions to help with those heuristics, should the be used. get_degrees aids the degree heuristic by using the implicit constraints to find the degree of each node in a constraint graph. get_neighbors similarly makes a neighbor dictionary to use for the LCV heuristic by using implicit constraints. generate_arcs and get_arcs help the inferencing arc consistency algorithm and max_domain_len() simply returns the length of the maximum domain to use for the minimum remaining values heuristic.

MapColoringCSP and CircuitBoardCSP

These two problems are very different CSPs but can be represented very similarly, allowing for generality in the CSP solver algorithm. The MapColoringCSP inherits the GeneralCSP class and pretty much has the same constructor. It actually uses the superclass constructor and only adds one more, self.constraints, generating the explicit constraints using the implicit ones provided. The generate_explicit_constraints creates a dictionary of constraints. Then, for every pair of implicit constraints, it generates all the possible values they can be. For instance, in the Australia problem, if 0 were WA and 1 were NT, then (0, 1) would be implicit, and then (r,g), (r,b), (b,g) and so on would be explicit. However, the issue is then what about (1, 0) -- the issue is they would have the same set of tuples associated in the dictionary, which would be a waste of space. So, my immediate thought was, why not use the set (0, 1) as a key, but then found out that keys cannot be mutable, which sets are. Finally, I decided to make it so the key always has the lower number before the larger. That way it will only be stored in that way. Then, it computes all the possible colors two can take on, stores in list and assigns that list to that key, returns that dictionary. In the CirucitBoardCSP, the explicit constraint generation is done quite similarly. Again, the key is constructed such that the lower number comes before the larger. Then similarly, possible coordinates for two bottom left coordinates, based on the width and height of a component, are generated. For instance, say the current domains for component a is (0,1), (0,0) ... and the current domain for component b is (0,0), (3,0), ... The generate_explicit_constraints would go through all possible left coordinates and store the ones that would work for both. In this case, the dictionary entry would be like {(0,1): [((0,0), (3,0))...] where the entries are tuples of tuples. For both, the print_assignment functions

takes in an assignment of values determined by backtracking (discussed next) and any variable/value dictionaries to return an ASCII representation of the solution!

CSP Solver

CSP.py houses a Library of functions, namely the backtracking algorithm, that works on any CSP passed into it. backtracking_search simply takes in a CSP and calls the recursive backtracking algorithm, providing it an empty dictionary so that it acts as an accumulator. Backtrack first checks if the assignment is complete with is_complete_assignment(), which simply checks whether the length of the assignment is the same length as the variables. This also acts as a goal test because it would never reach a complete assignment without being correct due to consistency checks. If assignment is not complete, then we select an unassigned variable with select_unassigned_variable. This function takes in a CSP and the current assignment and returns the next variable to assign. Now, without heuristics, this just returns the first variable in the list of csp.variables that is not in the assignment already. If MRV is on, then the algorithm loops through the variable list and looks for the variable that has the minimum remaining values, or smallest domain, that is not in the assignment already. If the degree heuristic is on, the algorithm similarly loops through and finds the variable with the highest degree that is not already in the assignment. Once a variable to be assigned has been found, we go through all values that the variable takes on and determine whether it would fit the assignment. The way that these values are ordered to be tried are decided in order_domain_values. Normally, without heuristics, the order doesn't matter. However, with the LCV heuristic, the algorithm runs a list.sort where the sort function is based on how many neighbors the current assignment on that variable will affect. Once we determine the value, we check if that value is consistent with the current assignment. That is, is_consistent checks if the current assignment is within all of the explicit constraints. If so, we add to the assignment and then change the **working_domains** of that variable to just that value. The reason we change the working_domains is because if this value doesn't work, we need to rewrite the domain to what it was previously, and the only way we do that is with the copy stored in domains. If arc consistency is turned on, we run through AC-3 and find shrink our working domains based on the current assignment. If any domains shrink to the size of 1 and that associated var isn't already in the assignment, then we add those inferences. We then backtrack and continue. In the case that the result of the backtrack is nothing, we remove that assignment as well as those inferences that may have been added. If inferencing was not on, that we simply just recursively backtrack after assigning the variable to a value and if it doesn't work, we remove that assignment.

Evaluation

I believe my implemented algorithms actually work really well. The backtracking solver was able to solve any problem I threw at it and worked extremely well with varying CSPs because of the generality involved with using numbers. I am very proud of how this turned out. Some testing also surprised me as well, but once I thought about the results, it made sense. For the MapColoringCSP, I first did a small test with 4 states, Vermont, New Hampshire, Maine, and Massachusetts. Everything was represented as numbers and dictionaries were provided to parse when printing out the assignment. With no heuristics at all, backtracking solved it with 7 consistency checks. I measured my runtime by incrementing each time the backtracking checked if we were consistent, representing how many times we had to "backtrack". That test was purely for proof of concept. Once I determined it worked, we then moved onto Australia, also represented by numbers. Without heuristics, we got assignment: WA:red NT:green Q:red NSW:green V:red SA:blue T:red with 11 consistency checks. The same runtime actually occurred with the MRV heuristic. This makes sense because MRV was not really doing anything because every domain was the same length, so there was no variable with minimum remaining variables. This heuristic only works with forward checking and arc-consistency. With the degree heuristic alone, we moved up to 15 checks. This was surprising, but made sense; we were assigning a value to the variable that would constrain a bunch of neighbors but if we choose the wrong value then everything has to be reversed. With LCV, the way we ordered the domain values, our checks dropped to 7, meaning we were picking the right value more often. MRV/Degree + LCV yielded the same results. Then trying just arc consistency drastically decreased the number of checks to just 3 and any combination of heuristics for this problem could not improve it, just spat back different solutions. This is because as soon as 2 colors were picked for 2 variable, then arc consistency got rid of all the other possibilities that would conflict with the assignment.

The CircuitBoardCSP worked similarly. Pure backtracking, no heuristics, performed 23 consistency checks and spat back the same assignment seen in the description.

['e', 'e', 'e', 'e', 'e', 'e', 'e', '.', 'c', 'c']

['a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'c', 'c']

['a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'c', 'c']

With the MRV heuristic, though nothing is done here without arc consistency, because of the way it is implemented, it returns the variable that is last in the csp.variables and because of that, by chance, the number of consistency checks went down to 19 with another solution because we started somewhere else. The degree heuristic once again drastically increased the consistency checks to 172 and this is again most likely due to the fact that we were choosing the wrong value for the vars that would affect the most. Similarly, the LCV reordering by itself made it slower to 254 checks. Once again, this makes sense because we were choosing values that would not affect the others that much. But this would mean that LCV with Degree would do wonders; indeed LCV with Degree showed just 13 checks, showing that we were picking the var that would affect the most and then value that wouldn't constrain them. MRV with LCV showed 26 checks, which was slightly more than no heuristics at all. This also makes sense because the heuristics are opposing each other. Once again, just Arc Consistency on its own did amazing, with only 4 checks and then combining that with MRV and LCV decreased that to 3, which was amazing to see.

Discussion

Map Coloring Discussion

Map Coloring Testing was fun! I tested on the VT, NH, ME, and MA to start off with proof of concept and then tested the Australia problem with no heuristics, varying heuristics, and then a combination of heuristics and inferencing and saw vast improvements and occasionally deterioration from without heuristics at all. Overall, it appeared that inferencing made a real impact on all of the tests.

Circuit Board Discussion

- Take the Circuit Board CSP. Say there were a variable with width w and height h on a circuit board of width n and height m. How did I determine the domain of this board such that the component completely fit on the board. Well, it was pretty simple. Let say we start the domain off at bottom left coordinate (0, 0). From there, the component can move to the right only enough so that it doesn't fall of the board. This would mean x + w < n, or the x coordinate plus its width was always less than the actual width of the board. Similarly, for y, we had to make sure the y

coordinate plus the height wouldn't surpass m. Both of these together would give the domain of the given component.

- Now consider components a and b from the problem description on a 10 x 3 board. Based on the constraints, the following are the possible left corner pairs:

[((0, 0), (3, 0)), ((0, 0), (4, 0)), ((0, 0), (5, 0)), ((0, 0), (3, 1)), ((0, 0), (4, 1)), ((0, 0), (5, 1)), ((1, 0), (4, 0)), ((1, 0), (5, 0)), ((1, 0), (4, 1)), ((1, 0), (5, 1)), ((2, 0), (5, 0)), ((2, 0), (5, 1)), ((5, 0), (0, 0)), ((6, 0), (0, 0)), ((6, 0), (1, 0)), ((6, 0), (1, 1)), ((7, 0), (0, 0)), ((7, 0), (1, 0)), ((7, 0), (2, 0)), ((7, 0), (1, 1)), ((7, 0), (2, 1)), ((0, 1), (3, 0)), ((0, 1), (4, 0)), ((0, 1), (5, 0)), ((0, 1), (3, 1)), ((0, 1), (4, 1)), ((0, 1), (5, 1)), ((1, 1), (4, 0)), ((1, 1), (5, 0)), ((1, 1), (4, 1)), ((1, 1), (5, 1)), ((2, 1), (5, 0)), ((2, 1), (5, 1)), ((5, 1), (0, 0)), ((6, 1), (0, 0)), ((6, 1), (1, 0)), ((6, 1), (1, 1)), ((7, 1), (0, 0)), ((7, 1), (1, 0)), ((7, 1), (2, 0)), ((7, 1), (1, 1)), ((7, 1), (2, 1))]

- My code has all inputs as numbers. I decided it would be easier to represent a CSP as numbers and then convert the numerical solution back into the specific problem later on. To that end, I write variable and value dictionaries that help the CSP uses to parse the assignment into solutions that we can appreciate with our eyes 😊