CS76 PA4, Fall 2021, Kunal Jha

Description of Algorithms

I began by making general classes for constraints. The constraints class store a list of variables, and had a general method which determined if all of the variables within the constraint satisfied some criteria.

I then made a CSP class which took in a list of variables, a domain (which was a dictionary), and booleans **MRV** and **DH** which were by default set to False. These simply toggled whether the MRV or DH heuristics would be used later. My CSP class also instantiated a dictionary of constraints for each variable. From there, I made a simple methods which added a constraint to the constraint dictionary given a constraint object. It iterated through each variable in the list of variables provided by the constraint class and appended the constraint object to the list of constraints for each variable in the constraint dictionary. The other simple method I made was isConsistent, which iterated through all of the variables in the constraint dictionary and checked to see whether each constraint was satisfied for each variable given an assignment.

The backtrack algorithm was a recursive algorithm which took an assignment (initialized to be an empty dictionary) and a boolean **LCV** which indicated whether or not the LCV heuristic would be used. If the assignment was the length of the number of variables, the assignment was returned. Otherwise, the initial variable was selected using either the unassigned variables, the MRV heuristic, or DH heuristics. If the LCV parameter was false, each value of the domain was iterated through, and the program checked whether this value led to a consistent assignment. If it was, the backtrack was run recursively on the copied assignment and the result was returned if it was not equal to None. If the LCV parameter was true, the value of the selected variable was assigned to be the result of the LCV method, and a similar procedure was executed as in the case of the LCV parameter being false (backtracking and returning the result of the local assignment if it was consistent and not empty). If the program never returned the result for either of these scenarios, it returned None.

The MRV algorithm simply iterated through all variables that weren't assigned and returned the one whose domain had the smallest length. The DH algorithm simply iterated through all variables that weren't assigned and returned the one who had the largest number of constraints. The LCV algorithm iterated through every possible value in the domain for a variable. It then kept a counter for how many values for neighbors of the variable satisfied the constraint between neighbors and the variable. After doing this process, the method returned a list of values for a variable sorted by their maximum number of satisfiable neighbors to minimum (which is equivalent to the minimum amount of variables removed).

The AC3 algorithm instantiated a queue with all of the arc constraints. It then continued to loop through the queue until it was empty. It popped an arc constraint and ran a Revise method. If after running the Revise method, the length of the domain for the main variable popped was empty False was returned. Otherwise, all arcs connected to the main variable popperd were iterated through and appended to the queue if they were not equal to the arc currently being explored. The Revise method iterated through all values of the domain of two variables, xi and xj. If, for each value in xi, there was not a value in the domain of xj which satisfied the constraint between xi and xj, that value was removed and True was returned. Otherwise, False was returned.

Since we made a general class for Constraints and based our CSP class around that, the only step for the Map Coloring and Circuit problems needed was to make the Map Constraint and Circuit Constraint classes. The Map Constraint class took in a list of two countries as variables. The isSatisfied method returned True if either one of the regions was not assigned or if their assignments were different. The Circuit Constraint class took two pieces of the board as inputs as variables. If either one of the pieces were not assigned True was returned. True was also returned if both pieces did not overlap, which was determined by comparing the starting locations of both pieces and their widths/heights.

I then made driver functions for both problems, and added constraints to a CSP object, ran the backTrack method, then printed the solution if one existed. For the sake on focusing on the main algorithms, I did not describe how I made the ASCII art for the Circuit Board Problem.

Evaluation

My algorithms in general work as expected. For both the map problems and the circuit driver problems, if a solution existed one was found. Details of their actual results are below.

The degree heuristic itself takes O(n) runtime, where n is the number of variables. This is because it iterates through all of the variables (O(n) time), looks up the constraint list (O(1) time), and compares the length of the list to an integer (O(1) time). It had O(1) memory since it only took in an initial list as a parameter and did not create any continguous blocks of memory during the course of its search. Since MRV did a similar process, it also had an O(n) runtime and O(1) space complexity. LCV had a runtime of O(n(v+c) + nlogn), where n is the number of values in the domain of a variable, c is the number of variables in a single constraint, and v is the number of values for a constrained variable. It had a space complexity of O(n), since it only had to record the number of neighbors unconstrained for each value in the domain of a variable. AC3 has a worst case runtime of O(ed^3), where e is the number of arcs, and d is the size of the domain for the largest domain. It had a space complexity of O(e).

Discussion

My map coloring problem found a solution for every heuristic. When only MRV was enabled, the solution was:

```
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'green'}
```

When only DH was enabled, the solution was

```
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'V': 'blue', 'WA': 'blue', 'T': 'red'}
```

This indicates that DH and MRV are proposing different orders for which variables should be explored first. By starting with a different variable first, each variable will get a different assignment (as seen by 'SA', 'WA', 'Q', 'V', and 'T' being different colors depending on the heuristic). Enabling AC3 with DH gives us the same solution as standard DH, as did AC3 with MRV (seen below).

```
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'green'}
```

Purely running AC3 without anything else enabled gives us

```
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'green'}
```

Running AC3 with LCV and DH gives us the same solution as only DH. In my testing, LCV did not actually result in a different solution than purely running DH or MRV, but it may have influenced how fast a solution was arrived at.

The domain in the circuit board problem for each block is every coordinate (x, y) of width w and height h in a board of size m by n such that 0 <= x <= m - w + 1, and 0 <= y <= n - h + 1. This ensures the piece will fit on the board regardless of what coordinate in the domain it is assigned to.

The constraint for two pieces is

```
if self.piece1 not in assignment or self.piece2 not in assignment:
            return True

x1 = (piece1.xcoord, piece1.xcoord + piece1.width - 1)
x2 = (piece2.xcoord, piece2.xcoord + piece2.width - 1)
y1 = (piece1.ycoord, piece1.ycoord + piece1.height - 1)
y2 = (piece2.ycoord, piece2.ycoord + piece2.height - 1)
```

```
if x2[0] >= x1[0] and x2[0] <= x1[1] and y2[0] >= y1[0] and y2[0] <= y1[1]: # piece 2 overlaps piece 1
    return False
if x1[0] >= x2[0] and x1[0] <= x2[1] and y1[0] >= y2[0] and y1[0] <= y2[1]: # piece 1 overlaps piece 2
    return False

return True
```

This is essentially saying piece 1 can't overlap with piece 2 and piece 2 can't overlap with piece 1, which constrains the locations of each piece. The explicit possible legal coordinates for pieces a and b on a 10x3 board specified in the problem are

Piece 1 Coordinate	List of Piece 2 Coordinates which satisfy constraint
(0,0) or (0,1)	[(5,0),(5,1),(6,0),(6,1),(7,0),(7,1)]
(1,0) or (1,1)	[(6,0),(6,1),(7,0),(7,1)]
(2,0) or (2,1)	[(7,0),(7,1)]
(3,0) or (3,1)	[(0,0),(0,1)]
(4,0) or (4,1)	[(0,0),(0,1),(1,0),(1,1)]
(5,0) or (5,1)	[(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)]

My code used tuples as inputs into the constraints, and each tuple represented the width and height of a piece. The isSatisfied method of my constraint relied on the coordinate assignment of a piece in conjunction with the width and height of a piece. However, when I created my domain for each piece I preselected values which would fit on the board, which means I wouldn't have to determine a piece fitting in the boundary while running CSP. Thus, everything was in an integer format when I inputted pieces into the CSP solver, so it did not have to convert values to solve the circuit problem. In the Map case, the CSP solver did not have to convert strings to integers either, as my constraints compared strings to strings, which was enabled since it took in a generic type as a parameter. By using generics, I was able to avoid having to convert strings or integers in either the Map problem or Circuit problem, so the only question became how to create the domain.