Constraint Satisfaction Problem

Note that not all code can be provided in this report. Also, some code snippets have places where code is omitted, as, in these cases, the omitted material is not as relevant to the accompanying explanations. Sometimes code goes past the page breaks, but this, unfortunately, cannot be helped for a particularly lengthy method.

1 Introduction

Constraint satisfaction problems (CSPs) are problems having components that depend on each other in a very particular way. These constrain the the way these variables can act, something we have seen before in our robot-motion planning lab before. Unlike robot-motion, however, we are not aiming to cover the entire problem space, but, quite to the contrary, remove as much of it as we can from the picture. By doing so, we do not have to call upon brute-force tactics more than we need to. We have several techniques (MRV, LCV, MAC3) that aim to either guide the algorithm as a heuristic or prune away fruitless paths.

One CSP is the map-coloring problem. In this problem, we attempt to color each region of a map a different color than its neighboring regions. Interestingly enough, four distinct colors will always suffice when dealing with such planar graphs (all maps), a statement proved by Appel and Haken in 1976. Another problem is the circuit-board problem, where we are taking rectangular blocks and trying to fit them into a board with a certain width and height. Both problems, as we will see, are very similar in their underlying solutions. The only differences in the code for either problem is the pre-processing that we do to make each problem fit into our generic CSP solver.

2 General Structure of CSP and Solver

Each problem will have its own subclass of the Constraint class, a subclass of the ConstraintSatisfaction-Problem class, and a driver class.

2.1 Constraints

These are the structures that are going to provide structure to our problem. That is, they will determine whether we will eliminate the entire problem space, resulting in no solution, or too little of it, resulting in lengthy run-time. They are the backbone of the problem. The actual constraint consists of the relevant subset of the variables involved in a particular relationship (e.g. neighboring countries) and the statement that must be fulfilled in order for the constraint to be satisfied.

```
1 public abstract class Constraint {
2         ArrayList<Integer> vars = new ArrayList<Integer>();
3
```

```
4         // Depends a lot on the type of constraint
5         public abstract boolean isSatisfied(int[] assigned);
6
7         public abstract HashMap<Integer, ArrayList<Integer>> trimDomain(int[] assigned);
8
9         public ArrayList<Integer> getVars() {
10                return vars;
11        }
12  }
```

In the immediate, the constraint relates these aforementioned variables, but there is a bigger, emergent property. We also implement a trimDomain method in each constraint, which shrinks the domain of one variable in the constraint based on the other. The constraints, as they are implemented, allow for n-ary constraints. We only concern ourselves with binary ones, though. By having variables adhere to one constraint, this limits the wiggle room they have to adhere to others. The rest of the program uses logic and heuristics to quickly limit this wiggle room.

Time and time again, we will find it necessary to access the constraints that only involve particular variables. To accommodate this, we use memoization to remember the sets of constraints associated with each set of variables that we process.

2.2 Constraint Satisfaction Problem

2.2.1 Overview

As stated before, we are trying to quickly limit all the different possibilities. Before we do that, however, we must discuss exactly how we the basic backtracking algorithm works. In essence, we are trying all the combinations of variables and whether all the constraints are met. There is a lot of subtlety here, though. For instance, the constraints often do not require us to have a fully assigned state to check for satisfaction (here state refers to the variables and their values). As long as a constraint has assigned values for the relevant variables, we can evaluate it. This allows us to prevent going down a path that already has a contradiction.

2.2.2 Minimum-Remaining Values

We have two choices on how we approach choosing the variable to examine next. We can either just pick the first unassigned variable (which we denote with value 0) or we can make use of the minimum-remaining values (MRV) heuristic (pick the variable with the smallest domain). In the case that several variables have the same number of values in their domain, we tiebreak based on which has the least number of constraints. We perform this in getUnassignedVar:

```
1   int getUnassignedVar() {
2         int minDomainSize = Integer.MAX_VALUE, size;
3         HashMap<Integer, ArrayList<Integer>> freq = new HashMap<>();
4         //Find vars with smallest domains
5         for (int i = 0; i < assignment.length; i++) {
6                 ArrayList<Integer> domain = domains.get(i);
7
8                 if (assignment[i] == 0) {//assignment to zero => no value assigned
```

```
                    if (!MRV)//just get the first unassigned value if no MRV
                        return i;
                    if (domain.size() < minDomainSize) {
                        minDomainSize = domain.size();
                        if (freq.containsKey(minDomainSize)) {
                            freq.get(minDomainSize).add(i);
                        } else {
                            ArrayList<Integer> start = new ArrayList<>();
                            start.add(i);

                            freq.put(domain.size(), start);
                        }
                    }
                }
            }
            //Tiebreak based on number of constraints
            if (minDomainSize != Integer.MAX_VALUE) {
                ArrayList<Integer> minDomain = freq.get(minDomainSize);
                int minConstraints = Integer.MAX_VALUE, minIndex = 0;

                for (int i : minDomain) {
                    HashSet<Integer> extendedDomain = getAssigned(); //currently assigned vars
                    extendedDomain.add(i);

                    if ((size = getConstraints(extendedDomain).size()) < minConstraints) {
                        minConstraints = size;
                        minIndex = i;
                    }
                }
                return minIndex;
            } else {
                return -1;
            }
        }
```

2.2.3 Least-Constraining Value

Once we have picked a variable to work on, another optimization we can make is to follow the least-constraining value (LCV) heuristic. In it, we sort all the values in the domain based on how many variable-value pairings each eliminates. We want to pursue the one that eliminates the least, as it will be the likeliest to take us to a valid solution. We could alternatively just pick the values in the order they already are in. The latter is obviously much faster, whereas the implementation of the former involves sorting based on the mentioned criteria, making it $\Omega(nlog(n))$. The former, however, provides the benefit that it potentially prunes entire branches of the problem space (possibly saving exponential amounts of time).

```
ArrayList<Integer> sortedLCVs(int var) {
    ArrayList<Integer> domain = domains.get(var);
    //Comparator implemented such that values with
    //less constraints show up earlier when sorted
    Collections.sort(domain, new Comparator<Integer>() {...});

```

```
7         return domain;
8 }
```

2.2.4 Maintained Arc Consistency

Finally, inference via maintained arc consistency (MAC-3) allows us yet another way to make our backtrack method even more efficient. It is different from the previous two, however, in that it is not a heuristic. It is something of a pre-processing that shrinks domains of all the variables following the assignment of some variable to a value. The idea is that, by assigning a value to this original variable, there may be some other variable with a value that relies on the original variable taking some other value to work. In this case, the other variable should remove this vestigial value from the domain. This removal may spur a similar process for all the variables that have a constraint with this variable. After effecting all the changes, eventually we reach a state where the domains are non-empty and can no longer be cut down, or there is a domain that is empty. In the former, we would proceed with our backtrack, but in the latter, it is clear that the assignment made prior to MAC-3 gave rise to an inconsistent system of variables. In this case, we should not backtrack, but rather find some other assignment. One note is that, although the CSP solver works with n-ary constraints, the MAC-3 algorithm only works when we are using binary ones.

```
1  boolean MAC3(int var, HashMap<Integer, ArrayList<Integer>> removed) {
2          ArrayList<Integer> queue = new ArrayList<>();
3          queue.add(var);
4          HashSet<Constraint> varConstraints;
5
6          while (!queue.isEmpty()) {
7                  int curr = queue.remove(0);
8                  // get neighbors through constraints
9                  HashSet<Integer> currList = new HashSet<>();
10                 currList.add(curr);
11
12                 varConstraints = getConstraints(currList);
13                 // Each constraint including the current var
14                 for (Constraint c : varConstraints) {
15                         ArrayList<Integer> vars = c.getVars();
16                         int unassigned = -1;
17
18                         // get the unassigned var
19                         for (int i : vars) {
20                                 if (i != curr && assignment[i] == 0)
21                                         unassigned = i;
22                         }
23                         boolean change = false;
24
25                         if (unassigned != -1) {
26                                 ArrayList<Integer> domain = domains.get(unassigned);
27
28                                 for (int j : new ArrayList<Integer>(domain)) {
29                                         assignment[unassigned] = j;
30                                         assigned.add(unassigned);
31
32                                         if (!c.isSatisfied(assignment)) {//Need to add back to queue
```

```
33                                          change = true;
34                                          removed.get(unassigned).add(j);
35                                          domain.remove(j);
36                                  }
37                          }
38
39                          assignment[unassigned] = 0;
40                          assigned.remove(unassigned);
41
42                          if (domain.isEmpty())//arc inconsistent
43                                  return false;
44                  }
45
46                  if (change)
47                          queue.add(unassigned);
48          }
49      }
50
51      return true;
52  }
```

3 Map Coloring Problem

3.1 AdjRegionConstraint

This binary constraint takes in two variable indices and is satisfied whenever the two variables take on different values for their colors. We trim the domain in trimDomain by simply not allowing one color's domain to be the color if the other. This only happens if one has a value assigned and other does not.

3.2 MapColoringDriver

The driver constructs exactly one constraint for every two neighboring regions. We set up the default domains to be all the available colors for each variable. As previously noted, only a maximum of four would be necessary. In the main method, the accompanying strings of colors are set, the neighbors are decided, and the problem solution is attempted.

3.3 Australia

Just as we desire, we receive consistent results:

WA: Red
NT: Yellow
SA: Purple
Q: Red
NSW: Yellow
V: Red
T: Red

4 Circuit-Board Problem

4.1 AdjRectConstraint

This constraint is a little more complex than the previous one. We pass not only each rectangular block's dimensions and index, but also the dimension of the allowed space. In this problem, our domain values are integers representing each square in the space. From left to right, bottom to top, we label each square as an integer, in increasing order. These integers encompass both the x and y coordinate of the block, making it easy to use the CSP solver we made that takes in sets of integers, not sets of tuples.

```
1 public boolean isSatisfied(int[] assigned) {
2         Rectangle rectA = new Rectangle((assigned[var2] − 1) % width, (assigned[var2]) / width, rect2[0], rect2[1]);
3         Rectangle rectB = new Rectangle((assigned[var1] − 1) % width, (assigned[var1]) / width, rect1[0], rect1[1]);
4
5         return !rectB.intersects(rectA);
6 }
```

In this constraint, trimDomain will find the rectangle that has been assigned a value, if either have been, and then make sure that the other rectangle, if unassigned, does not overlap with the first. We represent a rectangle in space by using such an integer to mark the top-left corner of the rectangle, and then using the width and height to figure out the rest. We make use of the Rectangle class to check for constraint satisfaction (true if no overlaps between the two rectangles). Notice that, because of rectangular symmetry, we do not need to worry about the results' being reflected across some horizontal or vertical line interfering with the solution. The answer will be the same.

```
1 public HashMap<Integer, ArrayList<Integer>> trimDomain(int[] assigned) {
2         HashMap<Integer, ArrayList<Integer>> trimmed = new HashMap<>();
3         ArrayList<Integer> restrictions = new ArrayList<>();
4         if (assigned[var1] != 0 && assigned[var2] == 0) {//remove domain values already used by var1 from var2
5                 for (int i = (assigned[var1] − 1) / width; i < (assigned[var1] − 1) / width + rect1[0]; i++)
6                         for (int j = (assigned[var1] − 1) % width; j < (assigned[var1] − 1) % width + rect1[1]; j
                              ++)
```

```
7                                   restrictions.add(i * width + j);
8               trimmed.put(var2, restrictions);
9       }
10
11      if (assigned[var2] != 0 && assigned[var1] == 0) {//vice-versa
12              for (int i = (assigned[var2] - 1) / width; i < (assigned[var2] - 1) / width + rect2[0]; i++)
13                      for (int j = (assigned[var2] - 1) % width; j < (assigned[var2] - 1) % width + rect2[1]; j
                            ++)
14                              restrictions.add(i * width + j);
15              trimmed.put(var1, restrictions);
16      }
17      return trimmed;
18  }
```

4.2 CircuitBoardDriver

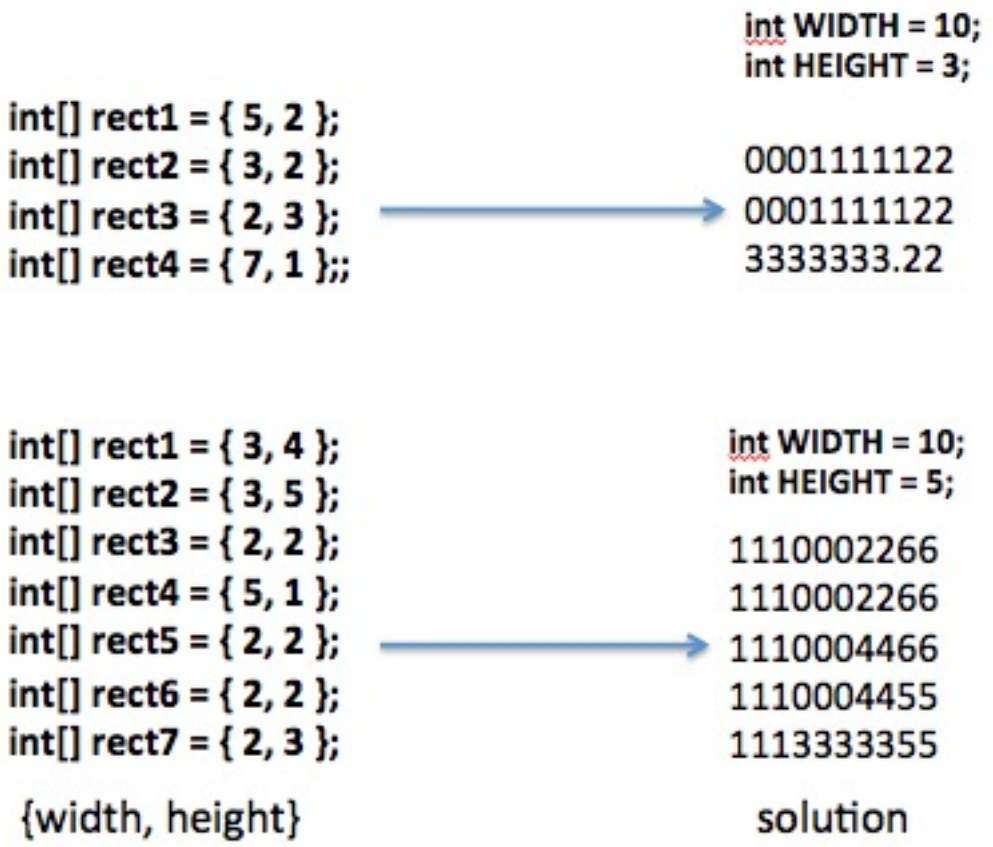
Much like the MapColoringDriver, we create a binary constraint for every two rectangles. We also populate each rectangle's domain to include every integer except those where the rectangle would be outside the frame. We use integers to represent a square where a certain rectangle covers in the solution, though this will result in awkward formatting if there are more than 9 rectangles.

```
1   public ArrayList<Constraint> getConstraints() {
2       ArrayList<Constraint> constraints = new ArrayList<>();
3
4       for (int i = 0; i < rectangles.size(); i++)
5               for (int j = 0; j < i; j++)
6                       constraints.add(new AdjRectConstraint(width, height, i, rectangles.get(i)[0], rectangles.get(i)
                            [1], j,
7                                       rectangles.get(j)[0], rectangles.get(j)[1]));
8       return constraints;
9   }
```

```
1   public HashMap<Integer, ArrayList<Integer>> getDomains() {
2       HashMap<Integer, ArrayList<Integer>> domains = new HashMap<>();
3
4       for (int i = 0; i < rectangles.size(); i++) {
5               domains.put(i, new ArrayList<Integer>());
6               for (int j = 0; j <= height - rectangles.get(i)[1]; j++)
7                       for (int k = 0; k <= width - rectangles.get(i)[0]; k++)
8                               domains.get(i).add(j * width + k + 1);
9       }
10      return domains;
11  }
```

4.3 Example Runs

The first example was the one on the class site.



4.4 Testing

We looked at the second of the previous two circuitboard setups and ran an analysis. We basically time backtrack for each of the following settings (which will likely be similar for the map coloring problem as well, considering these are both just binary constraint problems).

Circuit-Board Problem on Setup 2. Time averaged over 10,000 iterations.			
Time (s)	MRV	LCV	MAC3
0.3497			
0.8326	✓		
0.7898		✓	
0.5054			✓
1.9205	✓	✓	
1.1388	✓		✓
0.9522		✓	✓
2.2141	✓	✓	✓

We see that the "optimizations" do not seem to be helping us all that much. We should keep in mind, though, that, being heuristics, MRV and LCV may only show truly significant results when there are more variables. With too few variables, the processing might take up much more time than the time that is saved. Similarly, perhaps checking for arc consistency does more harm than good in this case by taking more time than actually just backtracking. This may also become more beneficial when the number of variables increase, when the amount of pruning is more visible.

5 Concluding Thoughts

We have successfully built here a CSP solver for n-ary constraint problems. By adding this generality, we are able to construct and solve problems with multiple variable constraints (i.e. find solutions $(x, y, z)|x^2 + y^2 = 100$ and $x + y = 14$). It is useful to note, however, that all n-ary constraints can be re-written with binary ones, so this is more of a feature than anything else.