Constraint Satisfaction

James Jia

November 7, 2016

1 Introduction

In this report, I will discuss my approach to creating a backtracking algorithm that can solve multiple constraint satisfaction problems. The backtracking class will take a constraint satisfaction problem and try different domain values for the variables until it finds a solution that satisfies all the constraints. At its most basic level, the backtracking algorithm is very similar to a depth first search through the possible variable assignments. However, I also added in an inferencing component to reduce the number of tried domain values, as the inferencing algorithm will be able to rule out certain domain values as variables are assigned. Additionally, I implemented an MRV for the domain values to better order the variables that the backtracker will test out. Specifically, it will first look at variables with the lowest number of possible remaining domain values. The output from the backtracker can be obtained by running Driver.java - this will print out the solution for both the Map Coloring problem as well as the Circuit Board problem.

2 Constraint

One of the first things I did in parallel with the development of the framework for the CSP problems was to set up a constraint map. This is a map that maps variable pairs to disallowed values. The reason that this map only considers variable pairs is that for the circuitboard problem and the map problem, we only deal with binary constraints. The Constraint.java class is shown below.

```
1
2
3  //WRAPS CONSTRAINT MAP
4  public class Constraint {
5
6    //CONSTRAINT MAP
7    public static HashMap<ArrayList<Integer>, ArrayList<ArrayList<Integer>>> constraints = new HashMap<
       ArrayList<Integer>, ArrayList<ArrayList<Integer>>>();
8
9    //TEST IF BINARY CONSTRAINTS ARE SATISFIED
10   public boolean isSatisfied(int xi, int xj, int iv, int jv){
11
12
13     ArrayList<Integer> varPair = new ArrayList<Integer>();
14     varPair.add(xi);
15     varPair.add(xj);
16
17
18     ArrayList<Integer> valPair = new ArrayList<Integer>();
19     valPair.add(iv);
20     valPair.add(jv);
21
22
23     ArrayList<Integer> flipVarPair = new ArrayList<Integer>();
24     flipVarPair.add(xj);
25     flipVarPair.add(xi);
26
27     ArrayList<Integer> flipValPair = new ArrayList<Integer>();
28     flipValPair.add(jv);
29     flipValPair.add(iv);
30
31     //IF THE KEYS ARE IN THE TABLE
32     if(constraints.get(varPair)!=null){
33
34       //IF THE VALUES CORRESPOND TO THE KEYS
35       if(constraints.get(varPair).contains(valPair)){
36         return false;
37       }
```

```
38
39        if(constraints.get(flipVarPair).contains(flipValPair)){
40          return false;
41        }
42      }
43
44      return true;
45    }
```

Here, the constraint class has a hashmap that includes variable pairs as its key set and disallowed domain values for each variable pair as its value set. The variable pair keys are stored as integer arraylists and the disallowed values are stored as an arraylist of integer arraylists. This class's isSatisfied() method returns true if the given variable pair and its values were found inside the constraint table. If they were found, this means the variable pair is set as disallowed values, and isSatisfied will return false to signal that the constraints have been violated by the particular variable pair's domain value assignments. One thing to point out is that it not only considers the variable pair in the order it was passed in, but also in reverse because they are equivalent and should have the same constraints.

3 Map Coloring Problem

In order to conduct backtracking, I also established a framework for the CSPs that I would use my backtracker on. The first problem that I considered was the Map Coloring Problem. For this, I used a map of the provinces of Australia, which included 7 unique states. Each state would have three possible domain values, but the colors were constrained such that each state could not have the same color as a bordering state. The constructor of MapCSP does the bulk of the work in terms of initializing the problem's variables, domains, a variable list that associates each variable with its respective domain, its adjacency list, and its constraint. The constructor is shown below.

```
1
2  public class MapCSP implements GeneralCSP{
3
4    public String[] countries = { "WESTERN AUSTRALIA", "SOUTH AUSTRALIA", "NORTHERN TERRITORY",
5                    "QUEENSLAND", "NEW SOUTH WALES", "VICTORIA", "TASMANIA" };
6
7    public int[] variables = { 0, 1, 2, 3, 4, 5, 6 };
8
9    public String[] colors = { "RED", "GREEN", "BLUE" };
10   int[] domains = {0,1,2};
11
12   public List<ArrayList<Integer>> variableList = new ArrayList<ArrayList<Integer>>();
13   public HashMap<Integer, int[]> adjacency = new HashMap<Integer, int[]>();
14   Constraint constraint = new Constraint();
15
16   //INITIALIZE VARIABLE LIST, ADJACENCY LIST, AND CONSTRAINT MAP
17   public MapCSP() {
18
19     //MAKE VARIABLELIST - INITIALIZE WITH ALL DOMAINS
20     for(int i=0; i<variables.length; i++){
21       ArrayList<Integer> temp = new ArrayList<Integer>();
22       for(int j = 0; j<domains.length; j++){
23         temp.add(domains[j]);
24       }
25       variableList.add(temp);
26     }
27
28     //MAKE ADJACENCY LIST
29     int[] WAneighbors = {2,1};
30     adjacency.put(0, WAneighbors);
31     int[] SAneighbors = {0,2,3,4,5};
32     adjacency.put(1, SAneighbors);
33     int[] NTneighbors = {0,1,3};
34     adjacency.put(2, NTneighbors);
35     int[] Qneighbors = {2,1,4};
36     adjacency.put(3, Qneighbors);
37     int[] NSWneighbors = {3,1,5};
38     adjacency.put(4, NSWneighbors);
39     int[] Vneighbors = {1,4};
40     adjacency.put(5, Vneighbors);
41     int[] Tneighbors = {};
42     adjacency.put(6, Tneighbors);
43
44
45     //SET CONSTRAINTS
```

```
46      HashMap<ArrayList<Integer>, ArrayList<ArrayList<Integer>>> consMap = new HashMap<ArrayList<Integer>,
        ArrayList<ArrayList<Integer>>>();
47
48      //loop through all pairs of variables
49      for (int i: variables){
50
51        for (int j: variables){
52
53          //if neighbors
54          if(isAdjacent(adjacency.get(i),j)){
55
56            ArrayList<Integer> varPair = new ArrayList<Integer>();
57            varPair.add(i);
58            varPair.add(j);
59
60            //neighbors can't be same value - put invalid combinations into the constraint map
61            ArrayList<ArrayList<Integer>> badVals = new ArrayList<ArrayList<Integer>>();
62
63            ArrayList<Integer> cons1 = new ArrayList<Integer>();
64            cons1.add(0);
65            cons1.add(0);
66
67            ArrayList<Integer> cons2 = new ArrayList<Integer>();
68            cons2.add(1);
69            cons2.add(1);
70
71            ArrayList<Integer> cons3 = new ArrayList<Integer>();
72            cons3.add(2);
73            cons3.add(2);
74
75            badVals.add(cons1);
76            badVals.add(cons2);
77            badVals.add(cons3);
78
79            consMap.put(varPair,  badVals);
80          }
81        }
82      }
83
84      //set the constraint map in the constraint
85      constraint.constraints = (consMap);
86    }
```

The constructor first initializes the variable list, which is an arraylist of integer arraylists that stores each variable's respective domain. For example, the domain of WA would be the arraylist found at index 0 of the variable list, since WA is the first country to appear in the country list. Initially, the domains are all 1,2,3, since no domains have been assigned yet and no domains have been ruled out. Additionally, the constructor will initialize the adjacency list that stores each province's neighbors. Finally, the constructor will populate the constraints in the constraint table of the MapCSP's constraint object. It fills it with all the variable pairs and disallowed values for those pairs. Specifically, it will put in all pairs of neighboring variables and put in values where they both have the same value, as neighboring states cannot have the same value.

4 Circuit Board Problem

The circuit board problem had a very similar implementation as the map coloring problem. However, because the domains were x,y coordinates, I created a coordinateMap that would assign each value on the 10x3 board a single value. This way, the domains in the variable list could just contain a list of integers, rather than a list of tuples or a list of integer lists. The constructor that initializes the circuit board problem is shown below.

```
1
2  public class CircuitCSP implements GeneralCSP{
3
4    public String[] shapes = { "A", "B", "C", "E"};
5    public int[] variables = { 0, 1, 2, 3};
6
7    Constraint constraint = new Constraint();
8
9    public List<ArrayList<Integer>> variableList = new ArrayList<ArrayList<Integer>>();
10   public HashMap<Integer, int[]> adjacency = new HashMap<Integer, int[]>();
11   public ArrayList<int[]> dimensions = new ArrayList<int[]>();
12   public HashMap<Integer, int[]> coordinateMap = new HashMap<Integer, int[]>();
13
```

```
14  int height=3;
15  int width=10;
16
17  public CircuitCSP() {
18
19    //INITIALIZE DOMAINS AS ALL POSSIBLE DOMAIN KEYS FOR COORD MAP
20    int coordKey = 0;
21    for (int y=0; y<height; y++) {
22      for (int x=0; x<width; x++) {
23        int[] coordinate = {x,y};
24        coordinateMap.put(coordKey, coordinate);
25        coordKey++;
26      }
27    }
28
29    //MAKE DIMENSIONS LIST - ADDS SHAPES IN ORDER
30    int[] aDim = {3,2};
31    dimensions.add(aDim);
32    int[] bDim = {5,2};
33    dimensions.add(bDim);
34    int[] cDim = {2,3};
35    dimensions.add(cDim);
36    int[] eDim = {7,1};
37    dimensions.add(eDim);
38
39    //MAKE VARIABLE LIST
40    for (int i : variables) {
41      ArrayList<Integer> var = new ArrayList<Integer>();
42      int[] dim = dimensions.get(i);
43      int domKey = 0;
44
45      //LOOP THROUGH ALL XY COMBINATIONS
46      for (int y=0; y<height; y++) {
47        for (int x=0; x<width; x++) {
48
49          //IF WITHIN 10X3 BOUNDS, ADD INTO DOM
50          if ((x+dim[0]<=width) && (y+dim[1]<=height)){
51              var.add(domKey);
52          }
53          domKey++;
54        }
55      }
56    variableList.add(var);
57    }
58
59    //MAKE ADJACENCY
60    int[] Aneighbors = {1,2,3};
61    adjacency.put(0, Aneighbors);
62    int[] Bneighbors = {0,2,3};
63    adjacency.put(1, Bneighbors);
64    int[] Cneighbors = {0,1,3};
65    adjacency.put(2, Cneighbors);
66    int[] Eneighbors = {0,1,2};
67    adjacency.put(3, Eneighbors);
68
69    //SET CONSTRAINTS - REFERENCED http://www.geeksforgeeks.org/find-two-rectangles-overlap/
70    HashMap<ArrayList<Integer>, ArrayList<ArrayList<Integer>>> consMap
71    = new HashMap<ArrayList<Integer>, ArrayList<ArrayList<Integer>>>();
72
73    //LOOP THROUGH ALL ADJACENT VARIABLE COMBINATIONS
74    for(int i: variables){
75      for(int j: variables){
76
77        if(isAdjacent(adjacency.get(i),j)){
78
79          //DIMENSIONS
80          int[] dim1 = dimensions.get(i);
81          int[] dim2 = dimensions.get(j);
82
83          //MAKE KEYS
84          ArrayList<Integer> varPair = new ArrayList<Integer>();
85          varPair.add(i);
86          varPair.add(j);
87
88          //LOOP THROUGH DOMAINS
89          ArrayList<ArrayList<Integer>> badValues = new ArrayList<ArrayList<Integer>>();
```

```
90            for (int x : variableList.get(i)) {
91              for (int y : variableList.get(j)) {
92
93                //GET ACTUAL XY COORDS FROM HASHMAP
94                int[] coord1 = coordinateMap.get(x);
95                int[] coord2 = coordinateMap.get(y);
96
97                //BOTTOM LEFT1
98                int[] bottomleft1 = {coord1[0], coord1[1]};
99
100               //TOP RIGHT1
101               int[] topright1 = {coord1[0] + dim1[0] - 1, coord1[1] + dim1[1] - 1};
102
103               //BOTTOM LEFT2
104               int[] bottomleft2 = {coord2[0], coord2[1]};
105
106               //TOP RIGHT2
107               int[] topright2 = {coord2[0] + dim2[0] - 1, coord2[1] + dim2[1] - 1};
108
109               //RECTANGLE OVERLAP CHECK
110               if(!( (bottomleft1[0] > topright2[0] || bottomleft1[1]>topright2[1]) ||
111                   (topright1[0]<bottomleft2[0] || topright1[1]<bottomleft2[1]) )){
112
113                 //IF DOESN'T PASS CHECK, ADD IN DISALLOWED VALUES
114                 ArrayList<Integer> temp = new ArrayList<Integer>();
115                 temp.add(x);
116                 temp.add(y);
117                 badValues.add(temp);
118               }
119             }
120           }
121           consMap.put(varPair, badValues);
122         }
123       }
124     }
125     constraint.constraints = consMap;
126   }
```

Here, the variable list is the same as the map coloring problem, except that it contains a key that maps to an actual x,y coordinate found in the coordMap. Its adjacency list is initialized such that every shape is neighbors with every other shape, since none of the shapes can overlap with each other. Finally, the domains inside the variable list are initialized as any value such that the bottom left corner of the shape can occupy that location and fit inside the confines of the 10x3 grid, as specified in the instructions.

The constraint for the circuit board is rather interesting, as it is essentially testing whether two rectangular objects intersect with each other. A simple way to do this is to consider two opposite corners of both shapes being considered, as specified on this source: http://www.geeksforgeeks.org/find-two-rectangles-overlap/. The example in the link demonstrates a condition for rectangles overlapping with the top left corner and the bottom right corner of both rectangles, but it can also be done with the bottom left corner and the top right corner of both rectangls. I chose the latter, as we are working with the bottom left corner of the shapes inside the variable list.

Discussion Questions:
1. In your write-up, describe 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. Make sure the component fits completely on the board.

The domain for the x-value of its lower left corner would be all values between 0 and ((Width of board) - (Width of object) +1). The domain for the y-value of its lower left corner would be all values between 0 and ((Height of board) - (Height of object) +1).

2.Consider components a and b above, on a 10x3 board. In your write-up, write the constraint that enforces the fact that the two components may not overlap.

if ((bottomleft1.x greater than topright2.x OR bottomleft.y greater than topright2.y) OR
(topright1.x less than bottomleft2.x OR topright1.y less than bottomleft2.y))

The values here refer to corners; for example, bottomleft1 would refer to the bottom left corner of the first rectangle (a). bottomleft1.x refers to that corner's x value.

3. Describe how your code converts constraints, etc, to integer values for use by the generic CSP solver.

While the shapes are strings, they are actually represented as integers ranging from 0-3. Similarly, the domain values are x,y coordinates, but they are represented by a single integer value key. This key can be put into the coordMap to get the actual x,y coordinate that the key represents.

5 Backtracker

Finally, we have the pices to create a backtracker to solve the CSP. The code for this was implemented inside the ConstraintSatisfactionProblem.java class. The backtracker is essentially a depth first search that will try out various assignments for the variable list that it takes from the CSP. It will store assignments in a one dimensional int array called assignment. For each assignment, it will test if that assignment is consistent with the rest of the existing assignments, and then run an inference based upon that assignment. If it passes both the assignment test and the inference test, it will recursively call backtrack again and search further down the tree. If it fails the consistency check or the inference test, then it will undo the assignment and also revert any changes it made to the variable list's set of domains. If no value is found at a given depth, then it will return null back up to the previous call. The code for backtracker is shown below.

```
1
2  public int[] backtracker() {
3
4      //ALL ASSIGNED
5      if (isComplete()){
6        return assignment;
7      }
8
9      //COPY VARIABLE LIST FOR RESET LATER
10     List<ArrayList<Integer>> copy = new ArrayList<ArrayList<Integer>>();
11     for(int i=0; i<vList.size(); i++){
12       copy.add(vList.get(i));
13     }
14
15     //GET UNASSIGNED VARIABLE USING MRV HEURISTIC
16     int var = getUnassignedMRV();
17     //int var = getFirstUnassigned();
18
19     //TRY EACH DOMAIN VALUE
20     for(int i=0; i<vList.get(var).size(); i++){
21
22       //INCREMENT # OF TRIED ASSIGNMENTS
23       nodes++;
24
25       //ASSIGN
26       int value = vList.get(var).get(i);
27       assignment[var]=value;
28
29       //IF CONSISTENT
30       if(isConsistent(var)){
31
32         //UPDATE DOMAIN WITH ASSIGNMENT
33         vList.get(var).clear();
34         vList.get(var).add(value);
35
36         //INFERENCE
37         boolean inferences = inference(var);
38
39         //IF PASS INFERENCE
40         if (inferences) {
41           int[] result = backtracker();
42           if (result != null)
43             return result;
44         }
45
46       }
47
48       //UNDO ASSIGNMENT AND DOM CHANGES WHEN BACKTRACKING
49       assignment[var] = UNASSIGNED;
50       vList = copy;
51     }
52
53     //NOTHING FOUND
```

```
    return null;
  }
```

6 Inference

In order to reduce the number of variables assignments that backtracker tests, I also implemented an inferencing algorithm. This is contained in the inference() method, which also uses revise(). These are shown below.

```

  //INFERENCE USING MAC3
  public boolean inference(int var) {

    //GET UNASSIGNED NEIGHBORS
    List<Integer> uaNeighbors = new ArrayList<Integer>();
    for (int neighbor : aList.get(var)) {
      if (assignment[neighbor] == UNASSIGNED)
        uaNeighbors.add(neighbor);
    }

    //QUEUE OF ARCS
    Queue<int[]> exploreQueue = new LinkedList<int[]>();

    //MAKE ARCS
    for (int uaNeighbor : uaNeighbors) {
      int[] tempArc = {uaNeighbor, var};
      exploreQueue.add(tempArc);
    }

    //ITERATE THROUGH ARCS
    while (!exploreQueue.isEmpty()) {

      //PULL FIRST ARC AND TEST FOR REVISE
      int[] arc = exploreQueue.poll();
      if (revise(arc[0], arc[1])) {

        //IF NO DOMAINS LEFT, FAILS INFERENCE
        if (vList.get(arc[0]).size()==0){
          return false;
        }

        //OTHERWISE, CONSIDER REVISED VARIABLE'S NEIGHBORS
        else{
          int[] newNeighbors = aList.get(arc[0]);
          for (int newNeighbor : newNeighbors) {

            //DON'T REPEAT THE PREVIOUS ARC
            if (newNeighbor != arc[1]){
              int[] newArc = {newNeighbor, arc[0]};
              exploreQueue.add(newArc);
            }
          }
        }
      }
    }
    return true;

  }

  public boolean revise(int xi, int xj) {

    //INCREMENT NUMBER OF REVISIONS
    revisions++;

    boolean revise = false;

    //GET XI DOMAINS
    ArrayList<Integer> xiDomains = new ArrayList<Integer>();
    for(int i: vList.get(xi)){
      xiDomains.add(i);
    }

    //LOOP THROUGH XI'S DOMAINS
    for (int di : xiDomains) {
```

```
66
67        boolean pass = false;
68
69        //GET XJ DOMAINS
70        ArrayList<Integer> xjDomains = new ArrayList<Integer>();
71        for(int j: vList.get(xj)){
72          xjDomains.add(j);
73        }
74
75        //LOOP THROUGH XJ'S DOMAINS
76        for (int dj : xjDomains) {
77
78          //TEST CONSTRAINT - IF ANY DOM SATISFIES CONSTRAINT, THEN PASS
79          if (problem.getConstraint().isSatisfied(xi, xj, di, dj)){
80            pass = true;
81            break;
82          }
83        }
84
85        //IF DID NOT PASS CONSTRAINT TEST
86        if (!pass) {
87
88          //REMOVE DOM VALUE FROM XI'S DOM
89          vList.get(xi).remove(vList.get(xi).indexOf(di));
90          revise = true;
91        }
92      }
93      return revise;
94    }
```

Here, the inference implementation follows the pseudocode inside the Russell and Norvig textbook. Essentially, for a given variable assignment, the inferencer will eliminate domain values for its neighbors that are incompatible with the variable's assignment. After that, if there were any changes to a neighbor's domain, then it will then look at that neighbor's neighbors and propogate the constraint checking further until. Ultimately, the inferencer will return true or false and trim down the set of allowable domains for the remaining unassigned values so that the backtracker has less work to do trying out assignments.

7 MRV Heuristic

Another way to further improve the backtracker was to implement an MRV heuristic. This stands for minimum remaining values, and this means that the backtracker will want to first look at variables with few remaining domain values before looking at variables that still have large domains. This was implemented in getUnassignedMRV() and it is shown below.

```
1
2   //MRV HEURISTIC - SELECTS UNASSIGNED WITH SMALLEST DOMAIN
3   public int getUnassignedMRV() {
4
5     //INITIALIZE BEST VALUE AS VERY LARGE NUMBER
6     int best = posInfinity;
7     int var = UNASSIGNED;
8
9     //LOOP THROUGH ASSIGNMENTS
10    for (int i = 0; i<assignment.length; i++) {
11
12      //FOR EACH UNASSIGNED VALUE
13      if (assignment[i] == UNASSIGNED){
14
15        //IF BETTER THAN PREVIOUS BEST VALUE, KEEP
16        if (vList.get(i).size() < best) {
17          best = vList.get(i).size();
18          var = i;
19        }
20      }
21    }
22
23    if(var == UNASSIGNED){
24      System.out.println("all assigned");
25    }
26
27    return var;
28  }
```

Here, instead of returning the first unassigned variable it sees or a random unassigned variable, getUnassignedMRV will return the variable that has the fewest remaining domain values. This allows the backtracker to order its variable selection more efficiently.

8 Output

The output for the solutions of the CSPs can be generated by running the driver.java class. Some of the output is shown below:
MAP PROBLEM

State: WESTERN AUSTRALIA Color: RED
State: SOUTH AUSTRALIA Color: GREEN
State: NORTHERN TERRITORY Color: BLUE
State: QUEENSLAND Color: RED
State: NEW SOUTH WALES Color: BLUE
State: VICTORIA Color: RED
State: TASMANIA Color: RED
node count: 7
revision count: 27

CIRCUIT PROBLEM

Shape: A Bottom Left Corner Coordinate: (2,0)
Shape: B Bottom Left Corner Coordinate: (5,0)
Shape: C Bottom Left Corner Coordinate: (0,0)
Shape: E Bottom Left Corner Coordinate: (2,2)
node count: 4
revision count: 24

Additionally, the driver will print out an ascii representation of the solution to the circuit board problem. It will not show all occupied spaces for each rectangle, but rather just where the lower left corner for each rectangle is located. The rest of the occupied squares can then be deduced with knowledge about each rectangle's dimensions. The dimensions are the same as those described in the example given in the instructions.

9 Profiling

I tested my solution with inferencing and without, and the number of assignments explored was greater when run without inferencing. With inferencing, thel map problem tried 7 assignments, indiciated by "node count", and the circuit board problem tried 4 assignments. Without inferencing, these numbers increased to 12 and 23 respectively, so inferencing was definitely a big help. Additionally, MRV reduced the number of revisions that inferencing would call, so that also seemed to improve efficiency.