Constraint Satisfaction Problem Solution

Nicholas Golini

November 8, 2016

1 Introduction

This solution goes through the solutions to two Constraint Satisfaction Problems, a map coloring problem, and a circuit-board layout problem. The solution will also walk through a generic constrain satisfaction solver, and will show how the map coloring problem and the circuit-board problem are solved using the generic solver. A constraint satisfaction problem is a problem that defines a set of variables, a domain of different values that the variables can be assigned as, and a set of constraints which map a pair of possible solution assignments to a given pair of variables. For each problem, the solution will go through how the problem is set up as a CSP, and how it converts the CSP into a generic form to be solved by the CSP generic solver.

2 Map Coloring Problem

The Map Coloring problem is such that given a map with different territories (or states), we want to color in each territory with a color so that no two bordering territories share the same color. This solution will create a map to represent the territories of Australia, and will attempt to color in the territories with no more than three colors.

2.1 Variables

The variables of the Map Coloring Problem is a list of all of the territories. In my code, the territory names are hard-coded in as "WA" for Western Australia, "NT" for Northern Territory, "Q" for Queensland, "NSW" for New South Wales, "V" for Victoria, "SA" for Southern Australia, and "T" for Tasmania. Since the program is not reading in a picture of the map, it does not know which territories are neighbors and which are not, so this is hard-coded into the problem, and will be discussed in the constraints section. Since the variables are saved within a list, variables$\{WA, NT, Q, NSW, V, SA, T\}$ can be simply mapped to $\{0, 1, 2, 3, 4, 5, 6\}$ for the generic solver, where variable(0) = "WA", variable(1) = "NT", and so on. The generic CSP is going to come up with a solution in the form of an int array, where the index corresponds to its variable in the string array, and the value of an index corresponds to a value form the variable's domain.

2.2 Domain

In the Australia Map Coloring problem, we are trying to color in each territory with no more than three colors: red, green, and blue. Since each territory can be assigned to either red, green and blue, each variable's domain is $\{red, green, blue\}$. In order to make this more generic, domain values can be mapped to a unique int, where the int represents the color it is mapped to. For example, red = 0, green = 1, and blue = 2. So in the generic problem solver, each variable's domain is $\{0, 1, 2\}$.

2.3 Constraints

The constraints in the map solving problem are quite simple. If territories A and B are not neighbors, then A and B can either be the same color or different colors. If territories A and B are neighbors, then A and B must be colored differently. The constraint data structure used in this problem will be a HashMap with keys being lists of Strings, representing the two territories that the constraints are for, and corresponding values being a list of pairs of possible values from the domain for the two variables. For example, since Western Australia and Southern Australia are neighbors, the list (”WA”, ”SA”) will be mapped to all pairs of values within the variables' domains such that the values are not the same. Thus (”WA”, ”SA”) will be mapped to the domain $\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$. If two territories are not neighbors, then they the above domain $\cup\{(red, red), (green, green), (blue, blue)\}$. As stated before, the solution does not read in a picture of a map so it does not know which territories are neighbors or not. Due to this, the constraints for the map coloring problem will be hard-coded in.

In order to generalize the constraints map, it shall be converted to using the int representations for the variable and domains. The conversion for the constraint map from String representation to integer representation takes place in the function: constraintsToInt. This function virtually creates an identical map where the string representations of the variables and domains will be converted to their integer representation. For the above example between ”WA” and ”SA”, the integer conversions are as follows: ”WA” = 0 and ”SA” = 5, given from the variable mapping, and ”red” = 0, ”green” = 1, and ”blue” = 2. Thus the entry for (”WA”, ”SA”) would be: $(0, 5) \to \{(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)\}$. This constraint HashMap will be then wrapped inside a class called Constraint, which holds the generic constraints HashMap and has a function isSatisfied(), that takes in a current assignment and checks if each assignment of variables complies with the constraints. If it does not, the function returns 0. If it does comply with the constraints, but is not a complete assignment, the function returns 1. If the assignment complies with the constraints and is a complete assignment, then a solution to the problem has been found and the function returns 2. Once the variables, domain, and constraints are generalized, they can be passed into the generic problem solver to be solved.

3 Generic Constraint Satisfaction Solver

The generic constraint satisfaction problem solver requires a generalized list of variables, domain, and constraints, each where variable and variable value is an integer. In order to access the variables, domains, and constraints from each problem, a node will be created for each problem that holds the generalized variables, domain, and constraints of the problem. For example, in the map constraint problem, there is a MapNode variable that holds the map's variables, domains, and constraints. This MapNode, which will be called startNode, is initialized after the problem has converted all of its variables, domains, and constraints to their generic integer representation. This information is piped into variables within the generic problem solver at the beginning of the initial call to the Backtrack algorithm helper function, called BackTrackingSearch. Once the variables, domains, and constraints are initialized within the generic solver, we can start the backtrack search for a solution to the problem. One important note to make is how the assignment is created. The assignment variable is an integer array, where the index of a spot in the assignment represents the corresponding variable, and the value of the assignment array at an index is the value assigned to that variable from its domain.An unassigned variable is represented by a -1.

3.1 Backtracking Search

Backtracking without the use of heuristics and Inferences is simply a brute force algorithm that will test every possible assignment of values to variables. Backtracking is a recursive depth first algorithm that chooses a value for one variable at a time, and backtracks on that variable when a later variable has no legal value to be assigned. For example, if 0 is chosen for the assignment of variable X, which results in a failure to assign a legal value to variable Y, then the algorithm will backtrack on all of its assignments made since assigning variable X, and will try again using the next value for X, which would be 1. This method of backtracking will eventually try every possible variable assignment until a complete solution is found. If the algorithm finds a valid solution to the problem, the solution is returned, and if no solution is found, null is returned. The generic backtracking solution is shown below:

```java
public synchronized int[] BackTrack(int[] assign, CSP csp){
    int var;
    System.out.println(this.constraints);
    System.out.println(this.domain);
    if(constraints.isSatisfied(assign, true, domain) == 2){
      return assign;
    var = ChooseUnassignedVar(assign);
    List<Integer> colors = domain.get(var);
    for(int color : colors){
      if(constraints.isSatisfied(assign, true, this.domain) == 2){
        return assign;
      }
      assign[var] = color;
      if(constraints.isSatisfied(assign, true, this.domain) > 0) {
        BackTrack(assign, csp);
      }
// restore constraints
      if(constraints.isSatisfied(assign, true, this.domain) != 2) {
        assign[var] = -1;
      }
    }
    return null;
}
```

3.2 Heuristics

Backtracking can be improved using different techniques of choosing the next variable to find an assignment for. Currently, the backtracking algorithm simply finds an arbitrary unassigned variable from the assignment to assign next. One technique that can be used is the Minimum-Remaining values (MRV) heuristic. The MRV heuristic finds the variable with the minimum remaining values that it can be assigned to for the next assignment. In other words, it finds the variable with the fewest legal values that it can be assigned to. By doing this, if a variable assignment is going to fail, it will now fail sooner (do fewer depth first recursions). This is because if a variable cannot be assigned anything, then that variable will be chosen as the MRV, and failure for the assignment will be found immediately, resulting in fewer searches of the other variables with a bad assignment. The code to find the variable with the MRV is below:

```java
public int MRVChooser(int[] assign, CSP csp){
  int var = -1;
  int fewest = 100;
  for(int k : this.domain.keySet()){
    if(this.domain.get(k).size() < fewest && assign[k] == -1)     {
      var = k;
      fewest = this.domain.get(k).size();
    }
  }
  return var;
}
```

Another useful heuristic is the degree heuristic. The degree heuristic finds the variable that is involved in the largest number of constraints on other unassigned variables. By choosing the variable with the largest number of constraints on other unassigned variables, the heuristic is virtually minimizing the branching factor of future choices, thus reducing the number of depth first recursion calls when assigning the rest of the variables. To implement this, if DH is being used in backtracking, I first create a HashMap called varDegMap, mapping variables to the number of constraints it is involved in for unassigned variables. Once this map is created, I just have to recurse through the list of variables and choose the one with the largest degree.

3.3 Inferences

When we assign a value to a variable, whether it is part of a complete solution or not, we can make inferences about the rest of the variables' domains that can speed up the Backtracking algorithm. To do this, we want to implement an Arc Consistency algorithm, MAC3. First, I will describe what arc consistency means. An arc simply represents a directed correlation from one variable to another, A to B. In order for A to be arc consistent with respect to B, for every possible value in the domain of A, there must be at least one value in B's domain that fits the constraint between A and B. If A is arc consistent with respect to B, then we know that when choosing a value for variable A, any value of B can be chosen and the constraints will be met. If A and B are each arc consistent with respect to the other, then any value in A's domain can be chosen with any value in B's domain. This is helpful in our Backtracking algorithm, because when we assign a variable, X, to a value at each iteration of the for loop in Backtrack, we can make every other variable arc consistent with respect to X. In other words, we can get rid of all of the values in the domain of each unassigned variable, Y, that make Y arc inconsistent with respect to X. By doing so, we know that each value in the unassigned variables' domains do not interfere with the current assignment of variables, resulting in the Backtrack function looking only at variables that fit with the constraints of the current assignment.

This is implemented by calling the algorithm MAC3 inside the Backtrack function like so (replacing lines 14-16 in the backtracking method above):

```
if(constraints.isSatisfied(assign, false, this.domain) > 0) {
  if(MAC3(csp, var, assign)) { // if inferences do not lead to failure
    System.out.println("HERE");
    deleteFromDomain();
  }
  BackTrack(assign, csp);
```

When MAC3 is called, it is given a variable who's value was just updated and assignment as arguments. MAC3 creates a queue that will hold arcs to be checked for arc consistency, and is initially filled up with arcs going from each unassigned variable, xj, to the variable passed into MAC3, xi. Once the queue is filled with the initial arcs, it goes into a while loop that runs until every arc in the queue is arc consistent with respect to xi, or until the domain of a variable has been completely wiped out. In the case where MAC3 deletes every value from a variable's, xj, domain, we know that given the current assignment of xi, there is no value in xj in order to make xj arc consistent with xi, thus we know that the assignment for xi is incorrect. This is where MAC3 really speeds up the Backtrack function. If a current assignment to xi is going to make it so that xj has no value in its domain to fit the constraints between xi and xj, MAC3 will return false and the next possible assignment for xi will be attempted. Without the use of MAC3, Backtrack will keep making its depth first calls until it tries all of the assignments for xj in its domain in which it then tries the next assignment for xi, which may take quite a long time.

MAC3 uses the helper function, REVISE, to actually do the deletion of values in variable domains that are arc inconsistent. After each new arc is popped off of the queue in MAC3, revise is called with the arguments of the two ends of the arc, xi and xj. Revise looks at each value, x, in the domain of Xi, and sees if there is a value, y, in the domain of xj where (x, y) satisfy constraints between xi and xj. If there does exist a y value that satisfies the constraint, then xj is arc consistent with respect to xi and x can be kept in the domain of xi. If there does not exist a y, however, then the arc is inconsistent, and x must be deleted from xi's domain to make the arc consistent. If any x value is taken out of the domain of xi, revise returns true, indicating the MAC3 needs to two more things: 1) MAC3 needs to check if revise deleted the entire domain of a variable, and 2) it needs to create arcs between the variables that would affected by the changing domain and add them to the queue. Like state before, if an entire domain is deleted as a result of the revise function, then the current assignment being tested is guaranteed to be an incorrect assignment, so MAC3 will return false. If revise does not delete the entire domain of a variable, then MAC3 makes an arc from xi's neighbors, xk, to xi. This is necessary because if xi's domain changes, the arc consistencies between xi and its neighbors xk will also be changed and thus must be checked.

If MAC3 returns false, then we do not want to update the domain of any variables, because a false MAC3 indicates that the assignment chosen by backtrack is incorrect, thus any inference made with respect to the assigned variable must be forgotten. To only delete values from domains when MAC3 returns true, I keep track of all of the values to be deleted from variables' domains in a HashMap during revise. In this case, revise does not do any deleting from the domain itself, it just compiles a HashMap of all the values that need to be deleted from the domain IF MAC3 is true. If MAC3 returns true in the Backtrack function, then a new function, deleteFromDomain() deletes the values found from revise from the domain, and backtrack keeps recursing with the found inferences. If MAC3 returns false, no inferences are added to the domain, and the backtrack keeps working to find a solution. The MAC3 and Revise functions are below:

```java
public synchronized boolean MAC3(CSP csp, int var, int[] assign){
  System.out.println("IN AC3");
  System.out.println("Domain before AC3" + domain);
  removedFromDomain = new HashMap<>();
  Stack<List<Integer>> q = new Stack<>();
  for(int i = 0; i < assign.length; i++){
    if(assign[i] == -1){
      List<Integer> arc = new ArrayList<>();
      arc.add(i); arc.add(var);
      q.add(arc);
      System.out.println(arc.toString());
    }
  }
  int xi, xj;
  while(!q.isEmpty()){
    removedFromDomain = new HashMap<>();
    List<Integer> X = q.pop();
    xi = X.get(0);
    xj = X.get(1);
    boolean rev = Revise(csp, xi, xj);
    if(rev){
      int xx = removedFromDomain.get(xi).size();
      int yy = domain.get(xi).size();
      if(xx == yy){
        return false;
      }
      List<Integer> neighbors = varMap.get(xi);
      if(neighbors != null) {
        for (int xk : neighbors) {
          if(xk != xj) {
          List<Integer> newarc = new ArrayList<>();
          newarc.add(xk);
          newarc.add(xi);
          q.add(newarc);
        }
      }
    }
   }
  }
  return true;
 }
```

```
public synchronized boolean Revise(CSP csp, int xi, int xj){
  boolean revised = false;
  List<Integer> vars = new ArrayList<>();
  vars.add(xj); vars.add(xi);
  int x;
  for(Iterator<Integer> itr = this.domain.get(xi).iterator(); itr.hasNext();){
    revised = false;
    x = itr.next();
    boolean inc = false;
    List<Integer> vals = new ArrayList<>();
    List<Integer> ys = domain.get((xj));
    for(int y : ys){
      vals.add(0, x); vals.add(1, y);
      if(constraints.conMap.get(vars).contains(vals)){
        inc = true;
      }
    }
    if(!inc){
      if(removedFromDomain.containsKey(xi)) {
        List<Integer> newVal = removedFromDomain.get(xi);
          if(!newVal.contains(x))
            newVal.add(x);
            removedFromDomain.put(xi, newVal);
          } else{
            List<Integer> newVal = new ArrayList<>();
            newVal.add(x);
            removedFromDomain.put(xi, newVal);
          }
        revised = true;
      }
    }
    return revised;
  }
```

⋆ See Appendix to see effectiveness of heuristics and inference findings

4 Circuit-Board Problem

In the circuit-board problem, we are given a circuit board of height m and width n, and a list of variables with different dimensions. The solution for the circuit-board problem is a set of locations for each piece to be located such that each piece is completely on the board, and no two pieces are overlapping.
One more note about the Circuit-Board problem is that board indices are not that of a double array with an x and y index. They are all assigned to a single int index, starting from the bottom left corner being 0, incrementing from left to right, and when a line is finished, the next index is at the next higher up row to the far left. An example is below:

$$08, 09, 10, 11$$

$$04, 05, 06, 07$$

$$00, 01, 02, 03$$

4.1 Variables

The variables in the Circuit-Board problem are defined as a list of (w, h) dimensions for the different pieces. The variable list maps each variable (w, h) dimension to an integer corresponding to an index in the assignment array.

4.2 Domain

The domain and constraints of the pieces for the circuit board problem are a little more involved than that of the map coloring problem. Given an m x n board with pieces of dimension (x_i, y_i), we first need to find the domain of the pieces. This domain will be all of the indices that the piece can go on. For reference, the index assigned to a piece is where the bottom left corner of the piece goes. For example if a 2 by 2 piece was at index 0 on a 3 by 10 board, the piece would be at 0, and take up the spots 0, 1, 10, and 11. The DomainCB data structure from my code maps a pair of dimensions, i.e. (2, 3) for the 2 by 3 block, to a list of every spot it can fit in. To find these spots, I loop through all of the indices on the board, and pass the dimensions of the block and possible index into a function called doesPieceFit, which calculates the number of horizontal and vertical spots from that index left on the board, and if the piece's dimensions fit within those available spots, then the piece can fit on that index while staying completely on the board. One example domain entry for a 2 by 5 piece on a 3 by 10 board is:

$$[2, 5] = [0, 1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15]$$

In order for the domain to be used in the generic problem solver, each key to the domain is converted to its index in the variable list. The values can all stay the same.

4.3 Constraints

The constraints of the problem map two pieces to all of the possible pairs of locations where the two pieces fit on the board, and do not overlap each other. The constraint map is created in the function createConstraints(), which loops through the variable list twice in a double for loop in order to get a mapping from each piece to ever other piece. When two pieces, A and B, are in consideration for creating constraints between them, another double for loop is entered in which each pair of locations for the two variables are considered. When it has two potential spots for the two variables, a function, doPiecesFit is called, passing in the two variables and the two potential spots for the variables. doPiecesFit creates two lists, one for each of the variables, and contains all of the spots that the two variables would take up at their respective possible spots. The two lists are then run through a disjoint java function, which returns true if the sets are disjoint, and false if there is overlap. If this call is true, then the pieces do fit together, and are added to the constraint for the two variables. One example constraint mapping between two variables is below:

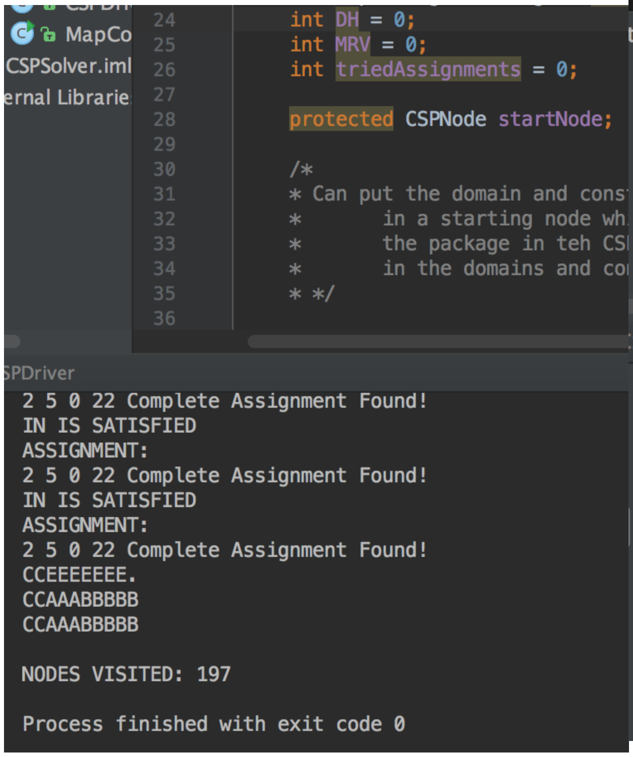
$$[[3,2],[1,7]] = [[0,2],[0,3],[0,12],[0,13],[0,22],[0,23],[1,3],[1,13],[1,23],$$
$$[7,0],[7,10],[7,20],[8,0],[8,1],[8,10],[8,11],[8,20],[8,21]]$$

This mapping is giving a list of pairs where the 3 by 2 and 1 by 7 blocks can go on the circuit board without overlapping each other. In order for this constraint structure to be used in the generic constraint solver, each set of dimensions in the key set of the constraint hashmap are mapped to the integer variable representation of the block, so that the keys for the generic constraints is a pair of two variable indices.

5 Appendix

5.1 Effects of Heuristics and Inference

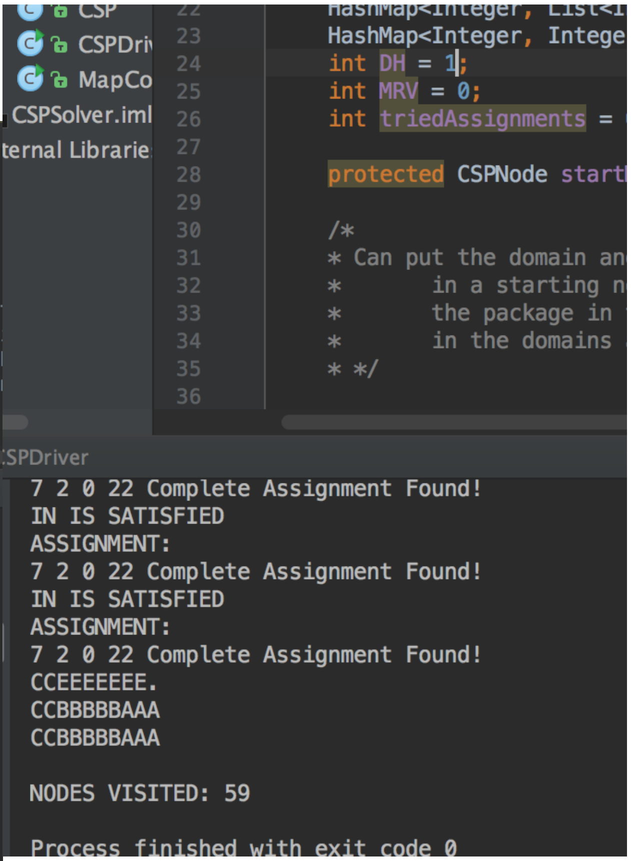
Variables Attempted No Heuristic and with Degree Heuristic



```
int DH = 0;
int MRV = 0;
int triedAssignments = 0;

protected CSPNode startNode;

/*
* Can put the domain and cons
*       in a starting node wh
*       the package in teh CS
*       in the domains and co
* */
```

```
2 5 0 22 Complete Assignment Found!
IN IS SATISFIED
ASSIGNMENT:
2 5 0 22 Complete Assignment Found!
IN IS SATISFIED
ASSIGNMENT:
2 5 0 22 Complete Assignment Found!
CCEEEEEEE.
CCAAABBBBB
CCAAABBBBB

NODES VISITED: 197

Process finished with exit code 0
```

Figure 1: This screenshot shows that without any heuristics, the circuit board CSP attempted 197 variable assignments.

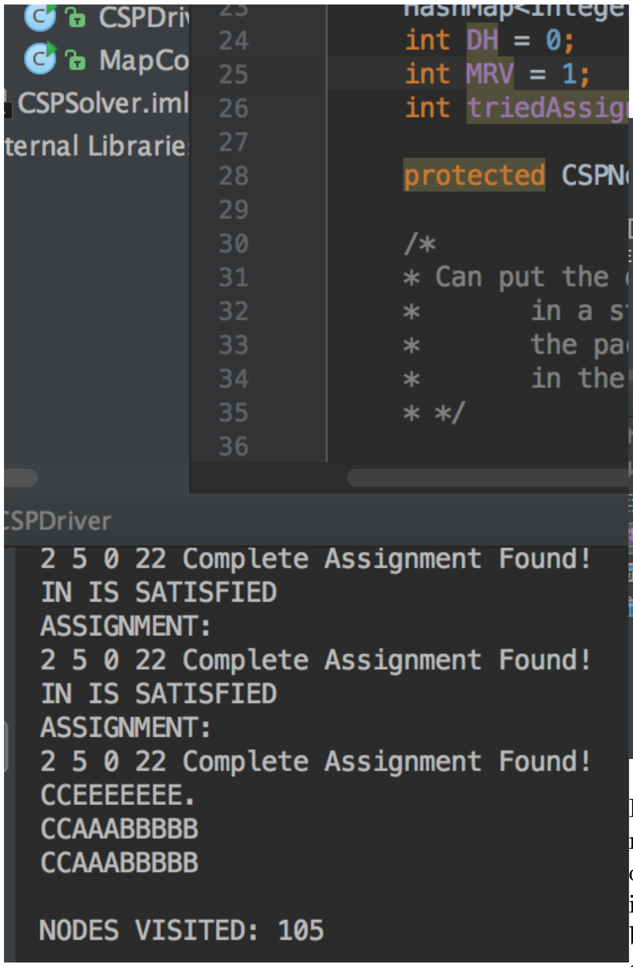


```
HashMap<Integer, List<I
HashMap<Integer, Intege
int DH = 1;
int MRV = 0;
int triedAssignments =

protected CSPNode start

/*
* Can put the domain an
*       in a starting n
*       the package in
*       in the domains
* */
```

```
7 2 0 22 Complete Assignment Found!
IN IS SATISFIED
ASSIGNMENT:
7 2 0 22 Complete Assignment Found!
IN IS SATISFIED
ASSIGNMENT:
7 2 0 22 Complete Assignment Found!
CCEEEEEEE.
CCBBBBBAAA
CCBBBBBAAA

NODES VISITED: 59

Process finished with exit code 0
```

Figure 2: With the degree heuristic implemented, the circuit-board CSP attempted only 59 variable assignments. This heuristic has such a large effect because the branching factor of this problem is very high at the beginning, but gets smaller as the backtracking keeps going, thus it takes out the high branching factor at the beginning.

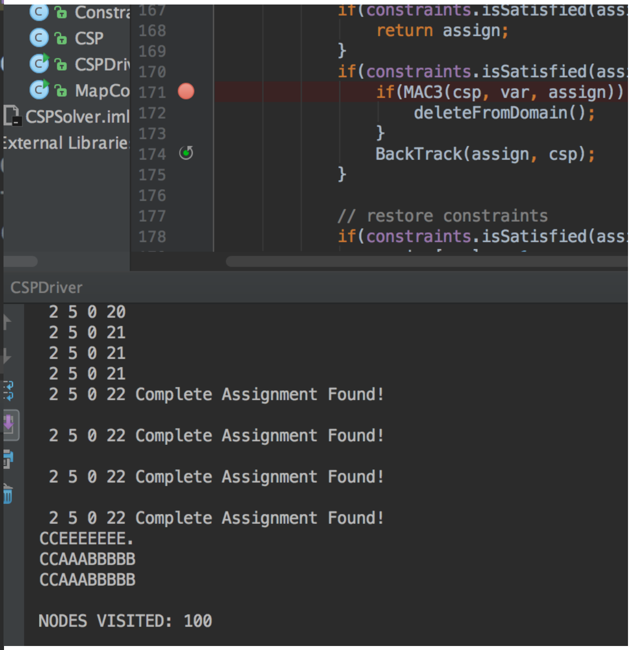
Variables Attempted with MRV Heuristic and with MAC3 Inference (without other heuristics)



```
    HashMap<Intege
    int DH = 0;
    int MRV = 1;
    int triedAssig

    protected CSPN

    /*
    * Can put the
    *       in a s
    *       the pa
    *       in the
    * */
```

```
2 5 0 22 Complete Assignment Found!
IN IS SATISFIED
ASSIGNMENT:
2 5 0 22 Complete Assignment Found!
IN IS SATISFIED
ASSIGNMENT:
2 5 0 22 Complete Assignment Found!
CCEEEEEEE.
CCAAABBBBB
CCAAABBBBB

NODES VISITED: 105
```

Figure 3: With the MRV heuristic, 105 variable assignments were attempted. This is definitely not as good as the degree heuristic, but it still cuts the number of attempted variables almost in half.



```
        if(constraints.isSatisfied(ass
            return assign;
        }
        if(constraints.isSatisfied(ass
            if(MAC3(csp, var, assign))
                deleteFromDomain();
            }
            BackTrack(assign, csp);
        }

        // restore constraints
        if(constraints.isSatisfied(ass
```

```
 2 5 0 20
 2 5 0 21
 2 5 0 21
 2 5 0 21
 2 5 0 22 Complete Assignment Found!

 2 5 0 22 Complete Assignment Found!

 2 5 0 22 Complete Assignment Found!

 2 5 0 22 Complete Assignment Found!
CCEEEEEEE.
CCAAABBBBB
CCAAABBBBB

NODES VISITED: 100
```

Figure 4: The MAC3 inference also diminishes the number of attempted variables by nearly one half. It decreases the number of variables attempted because it takes away values from the domains that cannot be possible given a certain variable assignment, thus decreasing the number of attempted variable assignments.