Constraint Satisfaction Problems

Kevin Farmer

November 5, 2016

1 Introduction

For this assignment, we look at the class of problems called constraint satisfaction problems. The problems consist of some form of constraints that limit possible valid solutions, and the goal is to find some valid solution. Often there are many possible solutions, and it is only necessary to find one of them. For this assignment, there are two primary problems. The first is to color a map with a limited number of colors, such that no two neighboring areas (states, countries, etc) are the same color. The second is to be given a list of circuits of differing dimensions, and a circuitboard. We must then find some way to layout the circuits such that all of the circuits fit on the board, and no circuits are overlapping. In addition to this, I completed an extension to solve a Sudoku puzzle when given a starting board.

In order to do this, I implemented a framework in `ConstraintSatisfactionProblem.java` which holds the code for the searches, and I then extend this framework for each of the given problems.

2 Constraints

In order to represent the constraints for the problem, I created the class `Constraint.java`. This class maps a pair of variables to the set of valid pairs of values for those two variables. It has the method `isSatisfied(int[] assignment, int var)` , which is given the current assignment and the variable that was just updated. It then checks if there are any conflicts between that variable and it's neighboring variables. There is also a method `getConstraintList` which returns the set of valid values for a given pair of variables.

3 Simple Solver

When the driver program calls `solve` on a CSP, some initial setup is done and then `recursiveBacktrackSolver` is called to do the actual search. When none of the heuristics are enabled, this function uses `getNextVar` to get the next unassigned var in the list, iterates through the domain of this variable and assigns the next value. It then checks if the constraints are satisfied, and if so then it recurses and the next variable will have a value assigned to it. It will continue to do this until finding some valid complete solution, or it will assigned every possible combination of values and then return failure.

```
1  //A recursive DFS through the search tree
2  private int[] recursiveBacktrackSolver(int[] assignment, int totalAssigned, HashMap<
     Integer, List<Integer>> domains) {
3
4    if (totalAssigned == numVar) {
5      System.out.println("Found solution");
6      return assignment;
7    }
8
9    int varToAssign;
10   if (mrv)
11     varToAssign = getMrvVar(assignment, domains);
```

```
12      else
13        varToAssign = getNextVar(assignment);
14
15      List<Integer> domainBeforeAssign = domains.get(varToAssign);
16      List<Integer> domain;
17
18      if (lcv) {
19        sortByLCV(domainBeforeAssign, varToAssign, domains);
20      }
21
22      //Assign value to variable
23      Iterator<Integer> iter = domainBeforeAssign.iterator();
24      while(iter.hasNext()) {
25        int val = (int) iter.next();
26
27        assignment[varToAssign] = val;
28        totalAssignments++;
29
30        //After making an assignment, set the domain to be that value
31        domain = new ArrayList<Integer>();
32        domain.add(val);
33        domains.put(varToAssign, domain);
34
35
36        //Check for conflict
37        if ( !allConstraints.isSatisfied(assignment, varToAssign) ) {
38          assignment[varToAssign] = UNASSIGNED;
39          domain = domainBeforeAssign; //Reverse changes to domain
40          continue;
41        }
42
43        if (mac3) {
44          if (!runMAC3(assignment, varToAssign, domains)) {
45            //If MAC-3 fails
46            assignment[varToAssign] = UNASSIGNED;
47            domain = domainBeforeAssign; //Reverse changes to domain
48            continue;
49          }
50        }
51
52
53        int[] sol = recursiveBacktrackSolver(assignment, totalAssigned+1, copyDomains(domains)
      );
54        if (sol != null) {
55          return sol;
56        }
57
58      }
59
60      assignment[varToAssign] = UNASSIGNED;
61      domain = domainBeforeAssign; //Reverse changes to domain
62
63      return null;
64    }
```

As one can see, there is also the option to enable heuristics as well which can hasten the search.

4 Minimum Remaining Values

The first heuristic is Minimum Remaining Values, which is used to determine which variable we should attempt to assign values to next. It does this by comparing the size of the domains for each variable, and

returning the variable with the smallest domain. This speeds the search by causing it to "fail-fast" if some assignment for that variable is invalid. If some variable has a very limited domain, and as such is likely to have conflicts with neighboring variables, then if we wait to assign a value to that variable, then if that assignment is invalid then we may spend a significant amount of time in a branch of the search tree that has no valid solution. By causing variables to "fail-fast" the search will more quickly determine if a branch has no solution and will waste less time in that branch.

```
1  //Returns the unassigned variable with the minimum remaining values
2  private int getMrvVar(int[] assignment, HashMap<Integer, List<Integer>> domains) {
3
4    int var = 0;
5    int num;
6    int min = Integer.MAX_VALUE;
7
8    for (int i = 0; i < numVar; i++) {
9      if (assignment[i] != UNASSIGNED)
10       continue;
11
12     List<Integer> vals = domains.get(i);
13     num = vals.size();
14     if (num < min) {
15       min = num;
16       var = i;
17     }
18   }
19
20   return var;
21 }
```

5 Least-Constraining Value

We can also use the least-constraining value method to determine which values we should assign to a given variable first. As opposed to MRV, LCV uses the "fail-slow" approach to deciding a value. In order to do this, it examines the domains of each of the neighboring variables and counts the number of times that each value appears. I then use this to sort the domain, that way the LCV method only needs to run once each time we recurse and are assigning to a new variable. By using the "fail-slow" approach, at each step of the process we are limiting ourselves the least for choices further down in that subtree. This means that the subtree is actually larger, and is more likely to contain a valid solution. However, if we have some partial assignment that is not part of a valid solution, then we will still explore the entire subtree below this partial assignment. In this case, using LCV will actually be slower in this subtree because we are now performing a sort at each level in addition to the search.

```
1  //Sorts the domain so that the value that appears least in neighboring domains is first
2  private void sortByLCV(List<Integer> domain, int varToAssign, HashMap<Integer, List<
     Integer>> domains) {
3    HashSet<Pair> constraintList;
4
5    int[] numAdj = new int[numVal+1]; //+1 b/c Sudoku values are offset by 1
6    for (int i = 0; i < numAdj.length; i++)
7      numAdj[i] = 0;
8
9    for (int i = 0; i < numVar; i++) {
10     constraintList = allConstraints.getConstraintList(new Pair(varToAssign, i));
11     if (constraintList == null) {
12       //then not adjacent
13       continue;
14     } else {
```

```
        List<Integer> adjDomain = domains.get(i);

        //Increment for each val in adjacent domain
        for (int adjVal : adjDomain) {
          numAdj[adjVal]++;
        }
      }
    }

    //Sort domain based on values found
    Collections.sort(domain, new Comparator<Integer>() {
        public int compare(Integer o1, Integer o2) {
            return (Integer.valueOf(numAdj[o1])).compareTo(numAdj[o2]);
        }
    });
  }
```

6 MAC-3

The MAC-3 algorithm is a method of constraint propogation that can be interleaved with the search in order to reduce the remaining domains after an assignment has been made. The basic idea of it is that there is a pair of arcs between pairs of variables. For an arc from var1 to var2 to be consistent, then for every value in the domain of var1, there must be some valid value in the domain of var2 that exists in the constraint table. If there is not for some value, then we delete this value from the domain of var1.

For the MAC-3 algorithm, we add all arcs which point to the variable that we just assigned a value to. We then pop the first element off the list, and enfore consistency. If we need to make a deletion to do so, then we add all arcs pointing to this variable to the queue as well. We continue this process until the queue is empty, and at that point all arcs should be consistent.

```
  //MAC-3 algorithm for constraint propagation
  private boolean runMAC3(int[] assignment, int var, HashMap<Integer, List<Integer>> domains
    ) {

    LinkedList<Pair> queue = new LinkedList<Pair>();

    //Add arcs to queue that point to var
    for (int i = 0; i < numVar; i++) {
      if (assignment[i] != UNASSIGNED)
        continue;
      Pair arc = new Pair(i, var);
      if (allConstraints.getConstraintList(arc) != null) {
        queue.add(arc);
        //System.out.println(arc);
      }
    }

    //Pop all arcs, enforce consistency
    while (!queue.isEmpty()) {
      Pair arc = queue.poll();
      int var1 = arc.getV1();
      int var2 = arc.getV2();

      List<Integer> domain1 = domains.get(var1);
      List<Integer> domain2 = domains.get(var2);

      HashSet<Pair> constrains = allConstraints.getConstraintList(arc);

      boolean madeDeletion = false;

```

```
30        //For each value in the domain, check consistency
31        Iterator<Integer> iter = domain1.iterator();
32        while (iter.hasNext()) {
33          int val1 = iter.next();
34          boolean hasLegalPairing = false;
35
36          //check if some valid pairing in domain2
37          for (int val2 : domain2) {
38            Pair vals = new Pair(val1, val2);
39            if (constrains.contains(vals))
40              hasLegalPairing = true;
41          }
42
43          //If not consistent for this val1 in domain1, then remove val1
44          if (!hasLegalPairing) {
45            iter.remove();
46            madeDeletion = true;
47            valsDeleted++;
48          }
49
50        }
51
52        if (domain1.size() == 0)
53          return false;
54
55        //If made a deletion, add all arcs pointing to var1 in arc
56        if (madeDeletion) {
57          for (int i = 0; i < numVar; i++) {
58            Pair newArc = new Pair(i, var1);
59            if (allConstraints.getConstraintList(newArc) != null) {
60              queue.add(newArc);
61            }
62          }
63        }
64      }
65
66      return true;
67    }
```

7 Map Coloring

With the framework in place to solve these constraint satisfaction problems, I now actually implement the first of them in `MapColoringCSP.java`. To do this, I take in a List of country names, a list of colors, and a Hashtable of borders between countries. I then build the initial domains, which in this case is just the entire list of possible colors for each country. In my `buildConstraints` method, I just create a list of valid pairs of colors for each pair of vertices, and in this case any pair of colors which are not the same is valid. Because of this, I actually could have been more memory efficient by having all pairs of neighboring countries point to the same list of valid pairs of colors, however this problem could also be modified so that certain countries have a more limited choice of colors and in that case this would no longer be a valid option.

Once we have the number of variable and values, the initial domains, and the constraints, the search can then be run on this specific problem. In order to print an assignment in a readable form, the integer of a variable can be used to index into the list of countries, and that of a color can be done for the color list as well. That way `assignment[0] = 0` can be used to represent: Western Australia: Red".

8 Circuit Board

In `CircuitBoardCSP.java`, I take in the dimensions of the circuitboard, the number of circuits, and the dimensions of each circuit as arguments. As part of this problem, I need a way to convert a two dimensional coordinate system into a single integer value. The method I decided to use was to assign each coordinate in the grid a unique value starting at 0, and going up to (area of board - 1). Take the following board for example:

2	8	9	10	11
1	4	5	6	7
0	0	1	2	3
	0	1	2	3

Each cell is assigned the corresponding value. I created the following two methods to convert from value to coordinates and back.

```
1  private int coordToVal(int x, int y) {
2    return x + (width * y);
3  }
4
5  private int[] valToCoord(int val) {
6    int x = val % width;
7    int y = val / width;
8    int coord[] = {x, y};
9    return coord;
10 }
```

Once I had this in place, just needed to find all locations such that if the bottom left corner of a circuit was a (x,y) then the entire circuit is within the board. All such values make up the domain for each circuit. In order to build the constraints, for each pair of variables I looped through their domains and checked if each pair of values was valid using the method `isValidPair`. This method determines the smallest and greatest x and y values for each circuit, and then checks if the x-coordinates overlap and the y-coordinates overlap. If they both do, then this is not a valid pair.

```
1  //Returns false if the two circuits overlap
2  private boolean isValidPair(Pair pair, Pair vals) {
3    Pair circ1 = circuits[pair.v1];
4    Pair circ2 = circuits[pair.v2];
5    int[] coord1 = valToCoord(vals.v1);
6    int[] coord2 = valToCoord(vals.v2);
7
8    int x1start = coord1[0];
9    int y1start = coord1[1];
10   int x1end = x1start+circ1.v1-1;
11   int y1end = y1start+circ1.v2-1;
12
13   int x2start = coord2[0];
14   int y2start = coord2[1];
15   int x2end = x2start+circ2.v1-1;
16   int y2end = y2start+circ2.v2-1;
17
18   boolean xOverlap = false, yOverlap = false;
19
20   if (x1start <= x2end && x1end >= x2start)
21     xOverlap = true;
22   if (y1start <= y2end && y1end >= y2start)
23     yOverlap = true;
24
25
26   if (xOverlap && yOverlap)
27     return false;
```

```
28      else
29        return true;
30    }
```

9 Comparing Results

Now that those two problems have been implemented, we can compare the difference in searches that utilize the different heuristics. For this I use the CircuitBoardCSP, because the search is more complex which makes it easier to see differences in the statistics. Something to note is that for the MapColoringCSP though, that without some form of inference that MRV and LCV cannot speed up the search because the domains for each variable are the same. Something else to keep in mind is that when running the same search multiple times, the actual execution time will differ somewhat. The following table lists execution time, number of partial assignments visited, and number of values deleted from domains.

MRV	LCV	MAC-3	Time(ms)	Visited	Deleted
F	F	F	1401	11236	0
T	F	F	1269	552	0
F	T	F	1253	329	0
F	F	T	1413	26	3139
T	T	F	1233	250	0
T	F	T	1404	10	1437
F	T	T	1406	10	934
T	T	T	1429	10	1460

Here is some example output from each combination of methods. As you can see, the execution time is around the same range, but is the quickest when using only MRV, LCV, or both. This is because using MAC-3 for constraint propagation greatly reduces the number of visted states, but it is a fairly slow algorithm and so most of this benefit is offset in terms of runtime. When using MAC-3 in conjunction to either MRV or LCV or both, only 10 partial assignments are visited, which is the minimum as this is being run using 10 circuits.

If there is no possible solution, then using MRV or MAC-3 will determine this much quicker than the basic solver, however LCV which actually be slower in most cases.

10 Extenstion: Sudoku

For an extension to this assignment, I also implemented a `SudokuCSP` which can be used to solve a given Sudoku puzzle. If you do not know, Sudoku is a 9x9 puzzle that starts with some initial cells in the grid filled in. You must then fill in the rest of the cells in the grid. In addition to this, the grid is broken up into 9 areas of 3x3 cells. Each cell must be filled with a number 1 through 9, but two cells in the same section, row or column cannot share the same number.

In order to implement this, I take in an initial 9x9 array of ints representing the board, some of which may be filled with a starting value. I then use each cell in the grid as a variable. If the cell has an initial value, then the domain of that variable contains one that one possible value. Otherwise, the inital domain for each variable is the integers 1 through 9. When building the constraints, I can check if two variables with coordinates (x1, y1) and (x2, y2) must be constrained with the following code:

```
1     if (x1 == x2 || y1 == y2) { //In same row or same column
2       canConflict = true;
3     } else if (x1/3 == x2/3 && y1/3 == y2/3) { //In same square of board
4       canConflict = true;
5     }
```

If so, then I build the constraints such that the two variables cannot hold the same value. Sudoku poses an interesting problem because unlike the other two, several of the variables are limited to a single value. This means that the basic solver is incredibly ineffective without the heuristics, because it is quite possible for it attempt an assignment of all the variables, and if the last one has a set value that causes a conflict then this can cause the search to take a very long time. I found two starting boards on the website www.websudoku.com that I used for testing this problem. On the easy difficulty, the basic solver took 26,274 milliseconds. For the "evil" difficulty, the solver took 1,407,907 milliseconds, which is over 23 minutes. However, the use of heuristics, especially MRV, makes an enormous difference in performance. Some more data from the "evil" difficulty follows:

MRV	LCV	MAC-3	Time(ms)	Visited	Deleted
F	F	F	1,407,907	11707559739236	0
T	F	F	387	217,294	0
F	T	F	1,258,711	702530501	0
F	F	T	1533	44547	259326
T	F	T	58	113	949
T	T	T	60	114	1011

The LCV made a small difference in runtime, but enabling either MRV or MAC-3 made the program run orders of magnitude quicker, with MRV making the largest difference. This makes sense, because with MRV enabled the search will go through the variables with preassigned values first, which means there is no longer an issue with trying to assign to one of those late in the search and having that cause conflicts. Using MAC-3 for constraint propogation also helps by removing values from domains that can cause potential conflicts.

I think one of the reasons for these results is that the way I have the problem set up, there is not initial arc consistency. Some potential ways to speed the basic search would be that when I am creating the initial domains, I check for pre-assigned values in conflicting variables and do not add a value to a variables domain if it is certain to always cause conflicts with one of the pre-assigned cells. Another way could be to the the AC-3 algorithm on the initial domains before beginning the recursive search. This would do what I just described, removing values from domains that are guaranteed to cause conflicts.