Constraint Satisfaction Solution

Boying Shi

February 26, 2014

1 Introduction

In this homework, we are asked to deal with constraint satisfaction problem. Constraint satisfaction problems (CSPs) are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods[1]. I finished both building design structure for solving general CSP and applying two certain problems on it and solve those by backtracking search algorithm as well as improved backtracking search algorithm. The two certain problems are, one is the traditional map coloring problem and the other one is circuit-board layout problem. My program and solve these two problems and show different running time of them for different search algorithm.

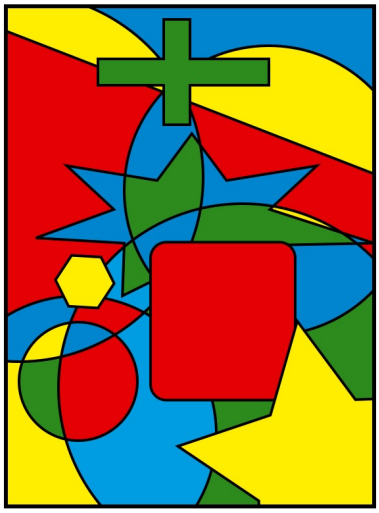

Figure 1: Problem that can be solved from CSP[2]

2 Project Design

This assignment is the most special one among all assignments given till now because it is the first time that I need to design and build the project all by myself without provided code given.

To build the problem model, I need to come up with the object oriented class relationship first. Since CSP is a standard search problem, an `ConstraintSearchProblem` class is needed to represent the whole general problem. And a standard CSP search problem is composed with three tuples: `variables`, `domain` and `constraints`. So easily, I need to build `variables` class, 'domain' class and `constraints` class just as Figure 2 shows.

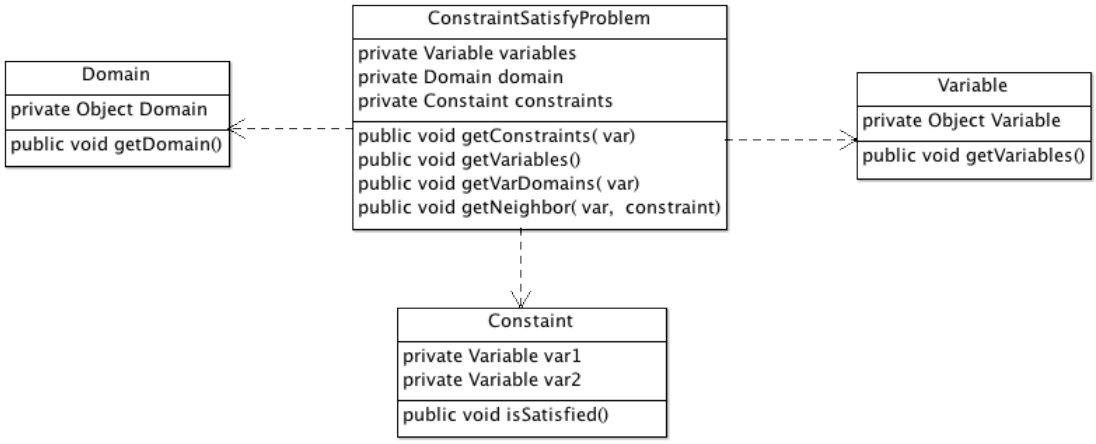

Figure 2: Class Diagram

However, as mentioned in homework design note, all variables in real world can be represented as integers and all domains can also be represented as integers. Therefore, in my project, I do not create a `variables` class and `constraints` class, instead, I only used integer to represent each. For constraints, here I use binary constraint to represent each pair of variables and in `constraints` class, I also had a `isSatisfied()` function, to check if the current two variables has the proper assignment. Also in my `ConstraintSatisfyProblem` class, I need to represent the relationships between the three tuples. Here I have, each variable has its domain list and each variable has its constraint list. Therefore, the my class relationship graph is like Figure 2 shows: `variables` class, 'domain' class and `constraints` class just as Figure 3 shows.

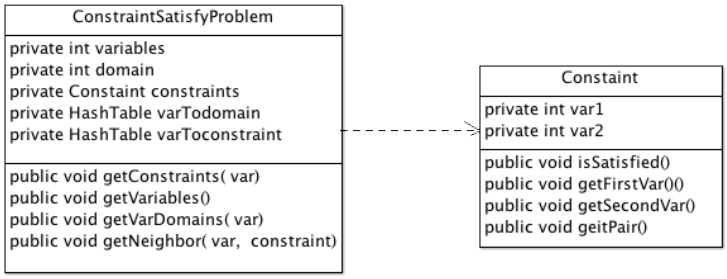

Figure 3: Class Diagram

But, even with such design, I can only have one particular constraint for a particular problem. To get a general problem solver, I will have to deal with different constraint and especially different `isSatisfied()` function. Therefore, only single `Constraint` class is not enough, I should add a interface of Constraint and different constraint for different real problem will implement this interface and then my model becomes: `variables` class, 'domain' class and `constraints` class just as Figure 4 shows.

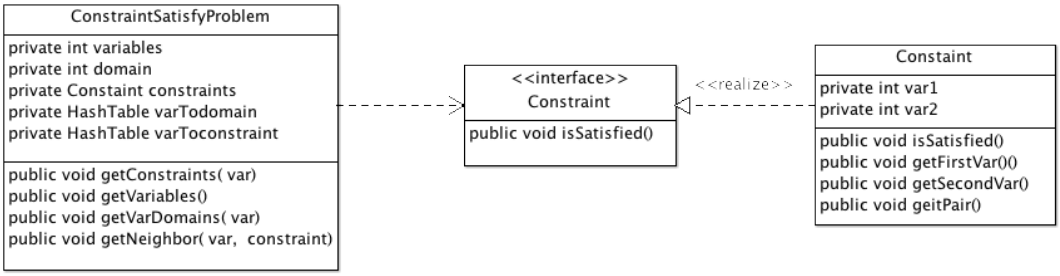

Figure 4: Class Diagram

Then, to finish my work of solving the CSP problem, I also need a class to deal with the assignment work for each pair of variable and value, a BackTracking search class deal with the simply search problem, a MRVBackTracking search class, a DegreeBackTracking Search and a MAC3BackTracking Search. Then, my class relationship final look is as followed:

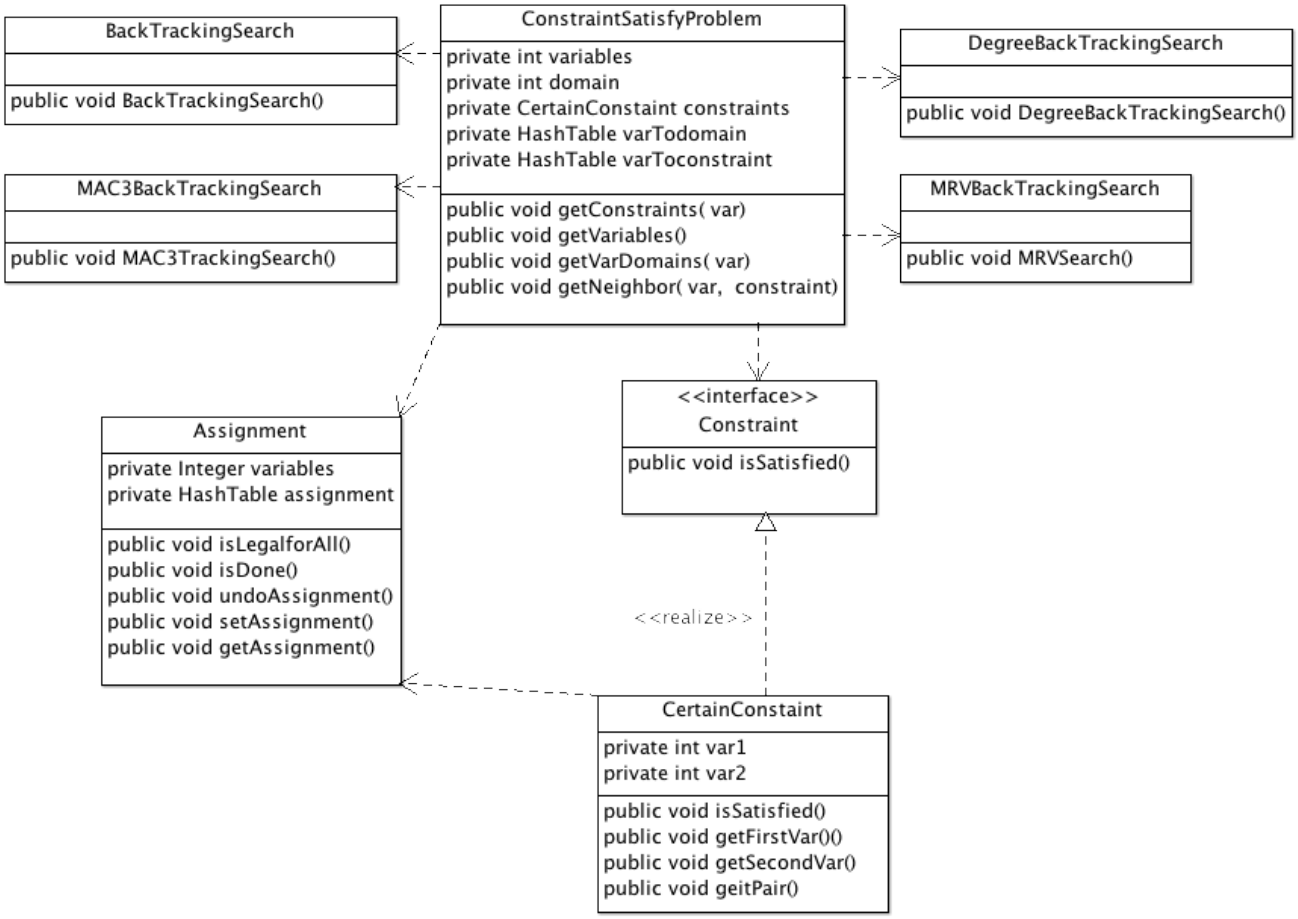

Figure 5: Class Diagram

3 Implemenation

In this section, I will show how I implement those classes one by one and then give test result for map-coloring problem and circuit-board layout problem.

ConstraintSatisfyProblem.java

The ConstraintSatisfyProblem.java is the general solver to get the initial values for solving certain problem. The code of this part is shown below:

Listing 1: ConstraintSatisfactionProblem

```
1  public class ConstraintSatisfactionProblem {
2  //variable list
3  private ArrayList<Integer> var;
4  //bianary constraint
5  private ArrayList<Constraint> constraints;
6  //each variable has a list of constraints
7  private Hashtable<Integer, ArrayList<Constraint>> relation;
8  //each variable has a list of domains
9  private Hashtable<Integer, ArrayList<Integer>> vartodom;
10
11 public ConstraintSatisfactionProblem(ArrayList<Integer> variableList,
       Hashtable<Integer, ArrayList<Integer>> domainMap,ArrayList<Constraint> c){
12   this.var = variableList;
13   this.constraints = c;
14   this.relation = new Hashtable<Integer, ArrayList<Constraint>>();
15   this.vartodom = new Hashtable<Integer,ArrayList<Integer>>();
16
17   //give initial empty value for each key
18   for (int eachVariable : variableList) {
19     this.relation.put(eachVariable, new ArrayList<Constraint>());
20     this.vartodom.put(eachVariable, new ArrayList<Integer>());
21     }
22
23   //create constraint list for each variable
24   for(Constraint cu: c){
25     for (int v: cu.getPair()){
26       relation.get(v).add(cu);
27         }
28       }
29   this.vartodom = domainMap;
30   }
31
32 //get current variable constraint list
33 public ArrayList<Constraint> getConstraints(int var) {
34   return this.relation.get(var);
35   }
36
37 //get variable list
38 public ArrayList<Integer> getVariables() {
39   return this.var;
40 }
```

```
41
42 public void setVarDomain(int v,  ArrayList<Integer> d){
43   this.vartodom.put(v, d);
44 }
45
46 public ArrayList<Integer> getVarDomain(int v){
47   return this.vartodom.get(v);
48 }
49
50 public int getNeighbor(int var, Constraint constraint) {
51   int[] pair = constraint.getPair();
52     if (var == pair[0])
53       return pair[1];
54     else if (var == pair[1])
55       return pair[0];
56     return -100;
57    }
58 }
```

Then, for each time, when the client pass variables, domains of each variable as well as each constraint to it, it can using assigned backtracking search to deal with the problem and then give a solution. In this class, I use two Hashtable to help me store two relationships, one is the relationship between variable to its domain list and the other one is the relationship between variable to its constraint list. At first, I just pass a Array list of domain into the constructor, but it can only deal with the problem will the same and unchanged domain, after testing for backtracking search with heuristic, I realized that I cannot only pass the unchanged domain list, I need to record every variable's domain list since they will change during searching. Then, I use the Hashtable to store it.

Constraint.java

Here is my interface of constraint, also, at first I did not realized how important an interface is, until I find I need to solve a new problem rather than a map-coloring problem, so I modified my code, and add an interface to deal with the problem, the code is as below:

Listing 2: interface Constraint

```
1 public interface Constraint {
2   int getFirstVar();
3   int getSecondVar();
4   int getPair();
5   boolean isSatisfiedWith(Assignment assignment);
6 }
```

For the map coloring problem and circuit-board layout problem, they will implement different constraint class, I will show they implementation later in each section.

Assignment.java

In this class, it deals with the assignment problem about variable to value and the code is as shown below:

Listing 3: Assignment.java

```
1  public class Assignment {
2  Hashtable<Integer, Integer> assignment;
3
4  //initial assignment with each variable of value -15
5  public Assignment(ArrayList<Integer> variableList) {
6    this.assignment =  new Hashtable<Integer, Integer>();
7
8    int index = 0;
9    for (int eachVariable : variableList) {
10     this.variables[index] = eachVariable;
11     this.assignment.put(eachVariable, -15);
12     index++;
13     }
14   }
15
16 //set
17 public void setAssignment(int var, int dom){
18   this.assignment.put(var, dom);
19 }
20
21 //get
22 public int getAssignment(int var) {
23   return this.assignment.get(var);
24 }
25
26 //check if all constraints are satisfied
27 public boolean isLegalforAll(ArrayList<Constraint> constraints) {
28   for (Constraint c : constraints){
29     if (!c.isSatisfied(this)){
30       return false;
31         }
32       }
33     return true;
34   }
35
36 //check if all variable has assigned to a value
37 public boolean isDone(ArrayList<Integer> varlist) {
38   for (int eachVariable : varlist) {
39     if (this.assignment.get(eachVariable) == -15)
40       return false;
41       }
42     return true;
43   }
44   }
```

BackTracking.java

For the beginning part, I need to implement the original back tracking search algorithm. Back tracking search is very alike to recursive DFS, it search a legal path a the beginning to the depth.Backtracking is a recursive algorithm. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. Several variants of backtracking exists[3].

Here, I implemented the original back tracking search without heuristic:

Listing 4: back tracking search

```
1   public Assignment BackTrackingSearch(ConstraintSatisfactionProblem
        csp,Assignment assignment) {
2     Assignment result = null;
3     //check if all the assignments have done, and return value
4     if (assignment.isDone(csp.getVariables())) {
5       result = assignment;
6       return result;
7       } else {
8         int var = selecVar(assignment, csp);
9
10        for (int value:csp.getVarDomain(var)) {
11          assignment.setAssignment(var, value);
12          if (assignment.isLegal(csp.getConstraints(var))) {
13            result = BackTrackingSearch(csp, assignment);
14            if (result != null)
15              return result;
16              }
17              assignment.undo(var);
18            }
19          }
20          return null;
21        }
22
23  //select a unassigned variable
24   private int selecVar(Assignment assignment,
25     ConstraintSatisfactionProblem csp) {
26     for (int var : csp.getVariables()) {
27       if (assignment.getAssignment(var)==-15)
28         return var;
29           }
30           return -100;
31       }
```

4 Map-coloring Problem

This map-coloring problem is a traditional problem that belongs to the famous four color theorem. In mathematics, the four color theorem, or the four color map theorem, states that, given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color[4].

4.1 Implementation

To deal it with my CSP, here I need to abstract it into the three tuples problem, that is to say, I need to get what are the variables and domains and constraints in map-coloring for Australia. Since my CSP only deals with variables that are integer, then I need to make all 7 parts of Australia into integers, therefore, I assigned them as below(as well as what I did with domain):

Listing 5: variable list and domain list

```
1  public static int WA = 0;
2  public static int SA = 1;
3  public static int NT = 2;
4  public static int Q = 3;
5  public static int NSW = 4;
6  public static int V = 5;
7  public static int T = 6;
8
9  public static int RED = 0;
10 public static int GREEN = 1;
11 public static int BLUE = 2;
```

After this preparation, I write a `MapColoringClient.java` to initial those values and form a mapProblem that is a ConstraintSatisfyProblem. After applying BackTracking Search, I can finally get a result. Here is my MapColoringClient.java code:

Listing 6: MapColoringClient.java

```
1  //form variable list
2  ArrayList<Integer> variables = new ArrayList<Integer>();
3  variables.add(NSW);
4  variables.add(NT);
5  variables.add(Q);
6  variables.add(SA);
7  variables.add(T);
8  variables.add(V);
9  variables.add(WA);
10
11 //form domain list
12 ArrayList<Integer> domainlist = new ArrayList<Integer>();
13 domainlist.add(RED);
14 domainlist.add(GREEN);
15 domainlist.add(BLUE);
16
17 //form relationship between domain list and variable
18 Hashtable<Integer,ArrayList<Integer>> domains = new
       Hashtable<Integer,ArrayList<Integer>>();
19 for(int eachvariable: variables)
```

```
20    domains.put(eachvariable, domainlist);
21
22  //form constraint
23  ArrayList<mapConstraint> cons = new ArrayList<mapConstraint>();
24  cons.add(new mapConstraint(WA,NT));
25  cons.add(new mapConstraint(WA,SA));
26  cons.add(new mapConstraint(NT,SA));
27  cons.add(new mapConstraint(NT,Q));
28  cons.add(new mapConstraint(SA,Q));
29  cons.add(new mapConstraint(SA,NSW));
30  cons.add(new mapConstraint(SA,V));
31  cons.add(new mapConstraint(Q,NSW));
32  cons.add(new mapConstraint(NSW,V));
33
34  //initial problem
35  ConstraintSatisfactionProblem mapProblem = new
        ConstraintSatisfactionProblem(variables,domains,cons);
36
37  //Solve problem with original BTS
38  Assignment assignment = new Assignment(variables);
39  BackTracking search = new BackTracking();
40  Assignment result = search.recursiveBackTrackingSearch(mapProblem,
        assignment);
41  double endTime = System.nanoTime();
42  double time = (endTime - startTime)/1000000;
43
44  //Print out result
45  System.out.println("Original BTS: "+result);
46  System.out.println("Time only run search: "+time);
```

Also, I need to show the constraint for map coloring. And especially the isSatified() function. For this problem, if two assignment of variables have different color, then it can return true else return false. The code is shown here:

Listing 7: mapConstraint.java

```
1   //initial binary constraint
2   public Constraint(int var1, int var2) {
3     this.var1 = var1;
4     this.var2 = var2;
5     list.add(var1);
6     list.add(var2);
7     }
8
9   //if two variable assigned to different value then return true
10  public boolean isSatisfied(Assignment assignment) {
11    int v1 = assignment.getAssignment(var1);
12    int v2 = assignment.getAssignment(var2);
13
14    return v1==-15||(v1!=v2);
15          }
```

4.2 Test result

The test result for only a signal original Back Tracking Search is below:

Listing 8: result of original backtracking search

```
Original BTS: {WA=GREEN, SA=BLUE, NT=RED, Q=GREEN, NSW=RED, V=GREEN, T=RED}
Time only run search: 1.581824
```

From this test result, it can concludes that my previous code works well and can finally find a right assignment to the map. For deeper discussion, I then need to add heuristic to check if the search algorithm can be improved or not.

5 Adding Heuristic[5]

In my previous search algorithm, every search I will pick up an unassigned variable and to go on my searching process, also I will use the defaulted order to get value and I did not try to think about reduce the possible search space in the future . Then, there comes 3 kinds of heuristic methods.

How to pick up an unassigned variable

Here it comes to the first kind of heuristic, I need to pick up an unassigned variable that is better than others. The representative of this method is minimum remaining values(MRV) heuristic. The key idea of this heuristic is to choose the variable with the fewest legal values. And here is my change in my BackTracking search code:

Listing 9: MRV

```
1  private ArrayList<Integer> MRVHeuristic(ConstraintSatisfactionProblem csp,
       Assignment assignment) {
2    ArrayList<Integer> result = new ArrayList<Integer>();
3    int mrv = Integer.MAX_VALUE;
4    for (int var : csp.getVariables()) {
5      if (assignment.getAssignment(var)==-15) {
6        int num = csp.getVarDomain(var).size();
7          if (num <= mrv) {
8            if (num < mrv) {
9              result.clear();
10             mrv = num;
11             }
12             result.add(var);
13             }
14           }
15         }
16         return result;
17         }
```

Also, to improve a little bit of MRV, I also need to choose a better beginning variable to assign value. To deal with this, I implement Degree heuristic. And the main idea is to select variable that is involved in the largest number of constraints on other unassigned variables. And my code is below:

Listing 10: Degree heuristic

```
1  private ArrayList<Integer> DegreeHeuristic(ArrayList<Integer> vars,
2    Assignment assignment, ConstraintSatisfactionProblem csp) {
```

```
3   ArrayList<Integer> result = new ArrayList<Integer>();
4   int maxDegree = Integer.MIN_VALUE;
5     for (int var : vars) {
6       int degree = 0;
7       for (Constraint constraint : csp.getConstraints(var)) {
8         int neighbor = csp.getNeighbor(var, constraint);
9         //calculate branch numbers
10        if (assignment.getAssignment(var)==-15&&
              csp.getVarDomain(neighbor).size() > 1)
11                                      degree++;
12                        }
13          if (degree > maxDegree) {
14            result.clear();
15            maxDegree = degree;
16            }
17            result.add(var);
18          }
19        }
20      return result;
21    }
```

These are the two representatives for help better choose variables to be assigned.

Order of trying value

Not only the order to choose variables that matters the running time of the search, the order of trying different values to assigned the variables also matters. The main idea of it is as below: Given a variable, choose the least constraining value which means the one that rules out the fewest values in the remaining variables. Then it will leave more flexibility to the remaining assignment works. Since this heuristic is more useful for a lot of value to assigned to the variable, and for map coloring problem, I only have 3 values, so I did not implement here.

How to reduce the possible search space

In previous heuristics, it only cares about the current situation and do not think about the future possible situation. In other word, if there is an assignment will finally lead to a failure search, then this assignment should not be taken into consideration. One of solving this problem is forward checking. The main idea of it is to keep track of remaining legal values for unassigned variables and stop searching when any variable has no legal values. And the other one that I solved here is MAC-3 heuristic. It helps to check the neighbor's neighbor domains with the current ones and to get if they are satisfied with each other. If not, delete that possible value in the domain. And here is my code of it adding to my original back track search:

Listing 11: MAC3

```
1  public void updateDomain(ConstraintSatisfactionProblem csp, int
       var,Assignment a){
2  //queue to store neighbor and neighbor's neighbor
3  Queue<Constraint> queue = new LinkedList<Constraint>();
4
5  //get current neighbor of this variable
6  ArrayList<Constraint> currentc = csp.getConstraints(start);
7  ArrayList<Integer> neiblist = new ArrayList<Integer>();
8  for(Constraint eachc:currentc){
```

```
9     int neighbor = csp.getNeighbor(var, eachc);
10    neiblist.add(neighbor);
11  }
12
13  //get this neighbor's neighbor
14  if (neiblist!=null){
15    for(int i = 0; i < neiblist.size();i++){
16      if (a.getAssignment(i)==-15){
17        ArrayList<Constraint> Nextc = csp.getConstraints(neiblist.get(i));
18        for(Constraint eachc:Nextc){
19          int nextneighbor = csp.getNeighbor(neiblist.get(i), eachc);
20          Constraint node = new Constraint(neiblist.get(i),nextneighbor);
21          queue.add(node);
22          }
23        }
24      }
25    }
26
27  while (!queue.isEmpty()){
28    Constraint currentNode = queue.poll();
29    if(removed(currentNode,csp)){
30      ArrayList<Constraint> newconstraints =
            csp.getConstraints(currentNode.getSecondVar());
31        for(int i =0;i<newconstraints.size();i++){
32          if (newconstraints!=null){
33            ArrayList<Integer> NextNeibolist = new ArrayList<Integer>();
34
35            for(Constraint eachc:newconstraints){
36              int nextneighbor = csp.getNeighbor(currentNode.getSecondVar(),
                    eachc);
37              NextNeibolist.add(nextneighbor);
38              Constraint node = new
                    Constraint(currentNode.getSecondVar(),nextneighbor);
39              queue.add(node);
40              }
41            }
42          }
43        }
44      }
45  }
46
47
48  //removed that pair leads to failure
49  public boolean removed(Constraint currentNode, ConstraintSatisfactionProblem
        csp){
50    int v1= currentNode.getFirstVar();
51    int v2 = currentNode.getSecondVar();
52    int result = 0;
53    ArrayList<Integer> d1 = csp.getVarDomain(v0);
54    ArrayList<Integer> d2 = csp.getVarDomain(vR);
55
```

```
56    for (int i = 0; i < d2.size(); i ++){
57      int flag = 0;
58      for (int j = 0; j < d1.size(); j ++){
59        if(d2.get(i) != d1.get(j)){
60          flag = 1;
61            }
62          }
63        //do remove
64        if(flag == 0){
65          result = 1;
66          d2.remove(i);
67          csp.setVarDomain(v2, d2);
68          }
69        }
70
71    //return if removed or not
72    if (result == 1){
73      return true;
74      }else{
75      return false;
76      }
77  }
```

Test result of adding heuristics

Here is my result of running the 4 back tracking search:

Listing 12: Test Result

```
Original BTS: {WA=GREEN, SA=BLUE, NT=RED, Q=GREEN, NSW=RED, V=GREEN, T=RED}
Time only run search: 2.057984

MRV BTS: {WA=GREEN, SA=BLUE, NT=RED, Q=GREEN, NSW=RED, V=GREEN, T=RED}
Time only run search: 1.056256

Degree BTS: {WA=GREEN, SA=BLUE, NT=RED, Q=GREEN, NSW=RED, V=GREEN, T=RED}
Time only run search: 0.897024

MAC3 BTS: {WA=GREEN, SA=BLUE, NT=RED, Q=GREEN, NSW=RED, V=GREEN, T=RED}
Time only run search: 1.14688
```

From the result list above, it clearly prove what I analyzed previously. Since the original BTS will just get a defaulted order to pick up the variable to be assigned as well as has no modified order to try value let along to think about future possibilities. Therefore, it runs slowest comparing with other three heuristics. And MRV change the order to pick up the unassigned variable and help to reduce time of searching. On the base of MRV, I added Degree heuristic, and the result shows that it runs a little bit faster than MRV. Finally MAC3 runs quicker than original BTS but slower than MRV and Degree. I think the result is that it really did a "thinking" to find the solution path, so it runs faster than the original one without "thinking". While MAC3 think too much than MRV and Degree heuristic, so for this simply 3 color assigning problem, it performs a little slow. But, I think if there are more variables and more colors, then MAC3 will perform better.

Test result of Europe

From Piazza, a kind classmate post a test case of coloring map of Europe. Here is my result of running the 4 back tracking search:

Listing 13: Test Result

```
Original BTS: {Portugal=RED, Spain=GREEN, Belgium=RED, France=GREEN,
    Monaco=GREEN, Germany=BLUE, Andora=RED, Switzerland=RED, Italy=BLUE,
    Netherlands=GREEN, United Kingdom=RED, Ireland=GREEN, Denmark=RED,
    Norway=GREEN, Sweeden=RED}
Time only run search: 1.554944

MRV BTS: {Portugal=RED, Spain=GREEN, Belgium=RED, France=GREEN, Monaco=GREEN,
    Germany=BLUE, Andora=RED, Switzerland=RED, Italy=BLUE, Netherlands=GREEN,
    United Kingdom=RED, Ireland=GREEN, Denmark=RED, Norway=GREEN, Sweeden=RED}
Time only run search: 1.28896

Dgree BTS: {Portugal=RED, Spain=GREEN, Belgium=RED, France=GREEN,
    Monaco=GREEN, Germany=BLUE, Andora=RED, Switzerland=RED, Italy=BLUE,
    Netherlands=GREEN, United Kingdom=RED, Ireland=GREEN, Denmark=RED,
    Norway=GREEN, Sweeden=RED}
Time only run search: 1.03808

MAC3 BTS: {Portugal=RED, Spain=GREEN, Belgium=RED, France=GREEN,
    Monaco=GREEN, Germany=BLUE, Andora=RED, Switzerland=RED, Italy=BLUE,
    Netherlands=GREEN, United Kingdom=RED, Ireland=GREEN, Denmark=RED,
    Norway=GREEN, Sweeden=RED}
Time only run search: 1.172992
```

From this test result, the ones with heuristics function all get a quick running time and especially MAC3 heuristic has a quicker running time than MRV since there are more variables in this problem and MAC3 can show its efficiency.

6 Circuit board layout problem

This is a interesting problem that I need to using CSP to solve circuit board assignment problem. When I first saw this problem, I just realized I need to general variables and constraints and domains at first.

6.1 Implementation

Since I store variables and domains as integer type, I need to transfer this board information into just integer. So, each board has different height and width, this will represents its variable. But I need a signal integer to represent it and I put it like, if a board is 2*3, then I will store it as a variable to be 23. And then, the decade will be height and the unite will be width.

Then, I come to think about the domain representation to Integer. I choose to store all possible positions (x,b) of the board's top left corner on the big board.(The position means, the first row with index 0 and the first column with index 0, then (0,0) means the first row first column) Then I will return my solution by return each board top left corner, I can figure out how to assign all the board on the big board. My code of applying it to CSP is as below:

Listing 14: Circuit client

```
1  //assign each board information to a signal integer. 2*3=23 2*5=25 3*2=32
       1*7=17
2  public static int A = 23;
3  public static int B = 25;
4  public static int C = 32;
5  public static int E = 17;
6
7 //form variable list
8  ArrayList<Integer> variables = new ArrayList<Integer>();
9  variables.add(A);
10 variables.add(B);
11 variables.add(C);
12 variables.add(E);
13
14 //get each board all possible position of top left corner
15 ArrayList<Integer> domainlistA = new ArrayList<Integer>();
16 for(int j=0;j<20;j=j+10){
17   for(int i=j;i<8+j;i++){
18     domainlistA.add(i);
19     }
20   }
21
22 ArrayList<Integer> domainlistB = new ArrayList<Integer>();
23 for(int j=0;j<20;j=j+10){
24   for(int i=j;i<6+j;i++){
25     domainlistB.add(i);
26     }
27   }
28
29 ArrayList<Integer> domainlistC = new ArrayList<Integer>();
30 for(int i=0;i<9;i++){
31   domainlistC.add(i);
32   }
33
```

```
34
35 ArrayList<Integer> domainlistE = new ArrayList<Integer>();
36 for(int j=0;j<30;j=j+10){
37   for(int i=j;i<4+j;i++){
38     domainlistE.add(i);
39       }
40   }
41
42 //form each variable with its domain
43 Hashtable<Integer,ArrayList<Integer>> domains = new
       Hashtable<Integer,ArrayList<Integer>>();
44 domains.put(A, domainlistA);
45 domains.put(B, domainlistB);
46 domains.put(C, domainlistC);
47 domains.put(E, domainlistE);
48
49 //form constraint, every board with each other is a constraint
50 ArrayList<boardConstraint> cons = new ArrayList<boardConstraint>();
51 cons.add(new boardConstraint(A,B));
52 cons.add(new boardConstraint(A,C));
53 cons.add(new boardConstraint(A,E));
54 cons.add(new boardConstraint(B,C));
55 cons.add(new boardConstraint(B,E));
56 cons.add(new boardConstraint(C,E));
57
58 //form problem
59 ConstraintSatisfactionProblem mapProblem = new
       ConstraintSatisfactionProblem(variables,domains,cons);
60
61 //run problem
62 double startTime0 = System.nanoTime();
63 Assignment assignment = new Assignment(variables);
64 BackTracking search = new BackTracking();
65 Assignment result = search.recursiveBackTrackingSearch(mapProblem,
       assignment);
66 double endTime0 = System.nanoTime();
67 double time0 = (endTime0 - startTime0)/1000000;
68
69 //print result
70 System.out.println("Circuit Problem: "+result);
71 System.out.println("Time only run search: "+time0);
```

Also, I need to modify a boardConstraint to implement Constraint interface, the one I need to check is the isSatifed() function, and here it means that if there is a board assigned on the big board, then if there comes another board, then it cannot overlap with the former assigned board and my code is here:

Listing 15: isStatisfied()

```
1   public boolean isSatisfied(Assignment assignment) {
2     //get assignment value for var1 and var2
3     int v1 = assignment.getAssignment(var1);
4     int v2 = assignment.getAssignment(var2);
5
```

```
6       //get height and width of var1
7       int height1  = this.var1/10;
8       int weight1 = this.var1\%10;
9
10      //get height and width of var2
11      int height2  = this.var2/10;
12      int weight2 = this.var2\%10;
13
14      //initial assigned position of var1 and var2
15      ArrayList<Integer> v1place = new ArrayList<Integer>();
16      ArrayList<Integer> v2place = new ArrayList<Integer>();
17
18      //check when they are assigend
19      if(v1!=-15&&v2!=-15){
20        for(int i = 0; i < height1;i++){
21          for(int j = 0; j < weight1; j++){
22            v1place.add(v1 + i*10 + j);
23            }
24          }
25
26       for(int i = 0; i <  height2;i++){
27         for(int j = 0; j < weight2; j++){
28           v2place.add(v2 + i*10 + j);
29             }
30           }
31
32       for (int i = 0;i < v1place.size();i++){
33         if (v2place.contains(v1place.get(i))){
34           return false;
35             }
36           }
37         }
38         return true;
39
40    }
```

6.2 Test Result

For the example test case I can figure out a solution. And to remind me of the it, I put the test case here:

Listing 16: Test Case

```
A = 2X3
B = 2X5
C = 3X2
E = 1X7
```

The test result of assigning circuit board with back tracking search is shown below:

Listing 17: Test Result

```
Circuit Problem: {A=0, B=3, C=8, E=20}
```

```
Time only run search: 2.51008
```

In my test result, A will assign to 0 means its top left corner point will assign to position(0,0). And the top left corner of B will assign to (0,3). The top left corner of C will assigned to (0,8). And the top left corner of E will be (2,0). Then the graph is like:

Listing 18: Test Result

```
**********                    AAABBBBBCC
**********       ===>         AAABBBBBCC
**********                    EEEEEEE*CC
```

After applying it with heuristics functions, the running result is as below:

Listing 19: Test Result

```
Circuit Problem: {A=0, B=3, C=8, E=20}
MRV Time only run search: 2.37021

Circuit Problem: {A=0, B=3, C=8, E=20}
Degree Time only run search: 2.13105

Circuit Problem: {A=0, B=3, C=8, E=20}
MAC3 Time only run search: 2.42018
```

And the test result of running time has the same trending with I run map coloring problem, the original search will run the slowest and the Degree heuristic will quicker than MRV heuristic. Due to too few number of domain list, the running time of MAC3 runs quicker than original search problem but runs slower than other two heuristic functions.

Moreover, I also have a test case to lay out circuit board, the test case is:

Listing 20: Test Case

```
A = 3X2
B = 2X2
C = 1X3
E = 2X3
```

The test result of it is shown below:

Listing 21: Test Result

```
Circuit Problem: {A=0, B=2, C=4, E=14}
Original BTS Time only run search: 2.31296

Circuit Problem: {A=0, B=2 C=4, E=14}
MRV Time only run search: 1.74319

Circuit Problem: {A=0, B=2 C=4, E=14}
Degree Time only run search: 1.52152

Circuit Problem: {A=0, B=2 C=4, E=14}
MAC3 Time only run search: 1.68412
```

And the graph is like:

Listing 22: Test Result

```
**********                    AABBCCC***
**********        ===>        AABBEEE***
**********                    AA**EEE***
```

7 Citation

[1].http://en.wikipedia.org/wiki/Constraint_satisfaction_problem
[2].http://upload.wikimedia.org/wikipedia/commons/8/8a/Four_Colour_Map_Example.svg
[3].http://en.wikipedia.org/wiki/Constraint_satisfaction_problem
[4].http://en.wikipedia.org/wiki/Four_color_theorem
[5].http://www.ics.uci.edu/~smyth/courses/cs271/topic5_constraint_satisfaction.ppt.

Code Work

Got idea from discussion on Piazza.
Discussed with TA Yinan Zhang.