Constraint Satisfaction Problem

Daniel Whitcomb

March 3, 2014

1 Introduction

Constraint satisfaction problems consider variables that can take values from a given domain within a set of constraints. These types of problems often are useful in graph-coloring(map coloring), space optimization(circuit board placement), and time optimization(class schedules) situations. In this report I will display a framework that can be used to setup and solve basic constraint satisifaction problems. I will also discuss possible optimization methods of constraint satisfaction searches.

2 Constraint Satisfaction Framework

The CSP framework is made of 5 parts: `ConstraintSatisfactionProblem.java`, `Variable.java`, `Constraint.java`, `Value.java`, and `Backtracker.java`. These items mirror the terms used in standard constraint satisfaction problems. The person writing their own problem needs to either implement or extend all of these classes except for `Backtracker.java`.

2.1 ConstraintSatisfactionProblem

`ConstraintSatisfactionProblem.java` is the superclass that a developer should extend when writing a simulation. Here is its implementation:

```java

import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.Random;
import java.util.Set;

public class ConstraintSatisfactionProblem {

	//These all will need objects eventually
	//I wanted to make it more general than
	//an array of ints

	//keys are variables
	//Value is HashMap with two values:
```

```java
        //map.get("constraints") gets constraint set
        //map.get("domain") gets domain set
        public int nodes;
        HashSet<Variable> returnedVars;


        public ConstraintSatisfactionProblem(){}

        public HashMap<String, List<Object>> putVariable(Variable variable, List<Object> constraints, List<Object> domain, List<Object> neighbors, HashMap<Variable, HashMap<String, List<Object>>> world){

        //puts the variable into the map
            HashMap<String, List<Object>> newMap = new HashMap<String, List<Object>>();
            newMap.put("constraints", constraints);
            newMap.put("domain", domain);

            if(neighbors != null){
                newMap.put("neighbors", neighbors);
            }

            world.put(variable, newMap);

            return newMap;
        }

        public boolean constraintsFulfilled(Variable variable, HashMap<Variable, HashMap<String, List<Object>>> world){
            //In case there are multiple constraints, the entire list is iterated over
            List<Object> constraints = world.get(variable).get("constraints");

            for(Object c : constraints){
                if(!((Constraint)c).isSatisfied(variable)){
                    return false;
                }
            }

            return true;

        }

    //Checks the world to make sure the constraints for all variables are satisfied
        public boolean constraintsGood(LinkedList<Variable> list, HashMap<Variable, HashMap<String, List<Object>>> world){

            if(list.size() == 0){
                return true;
            }

```

```java
                for(Variable var : list){
                        if(!constraintsFulfilled(var, world))
                                return false;
                }
                return true;
        }

        public void printSolvedProblem(LinkedList<Variable> variables){
                System.out.println("====================");
                for(Variable var : variables){
                        System.out.println(var.toString());
                }
        }

    //Implemented here because this method should be overriden in any implementation
        public Variable getNext(Variable currentVar){
                return null;
        }

    //A useful helper method if the user doesn't care what is
    //Gotten next
        public Variable randomVariable(Set<Variable> list){
                int size = list.size();
                int rand = new Random().nextInt(size);
                int i = 0;

                for(Variable var : list){
                        if(i == rand){
                                return var;
                        }
                        i++;
                }

                return null; //It will never reach this point.
        }
}
```

The main world of the method is of the type *HashMap< Variable, HashMap<String, List<Object>>>*. The outer hashmap holds the variable set as the keys, and the values are a HashMap of Lists. There are two required lists: *constraints* (a list of *Constraint* objects and *domain* (a list of *Value* objects). There is also an optional list, textitneighbors, in case the problem is working with a graph or not. In these problems, I feel the graph setup is common enough where it is worth addinging it as an option. The method `putVariables()`, takes a variable, puts the lists containing the constraints, the domain, and the neighbors (if provided) into a HashMap, and puts them into the World HashMap.

The class also handles high level constraint checking, which all gets called in `Backtracker.java`. The method `constraintsGood()` calls `constraintsFulfilled()` for each variable in the world. `constraintsFulfilled()` calls the `isSatisfied()` method that is required by the *Constraint* interface in `Constraint.java`.

The instance variable *returnedVars* keeps track of what variables have been returned to the backtracker, so the same variable doesn't get returned twice accidentally. `printSolvedProblem` does exactly what you think. Each time a solution is found, it prints the values of all the variables. If the user needs a better print method, they can overwrite it themselves.

2.2 Object Frameworks

The three object frameworks include: `Variable.java`, `Value.java`, and `Constraint.java`. These files handle the structure of what needs to be implemented by the

The following is the content of the above classes:

```java
public class Variable {

	public String designation;
	public Value val = null;

	//This should always be overriden!!!
	public Variable copy(){
		return null;
	}

	public String toString(){
		String str = designation + ": " + val;
		return str;
	}
}

public interface Value {

	public Value copy();
	public Object getValue();
	public String toString();
	public boolean equals(Object o);
}

public interface Constraint{

	public boolean isSatisfied(Variable variable);
	public Constraint copy();

}
```

I would have liked to make *Variable* and interface, but I wanted to require the user to use some instance variables that are used during backtracking, but with an interface, you can only define static and final variables. so I decided to use a class and have the user extend it instead of implement. Otherwise, for *Value* and *Constraint*, they could be interfaces. These definitions become important when we look at the backtracker because they are used to make generalizations about what types are being used in the search.

3 Backtracking

Backtracking involves setting the values of a variable, moving to the next some and setting a random value, and checking if the combination works. If it does, then the pattern continues until the values for the variables no longer fulfill the constraints, or a solution to the problem has been found.

Backtracking is most often implemented recursively because it uses a depth-first tree search. Here is my implementation of the backtracker:

```
public class Backtracker{

        public ConstraintSatisfactionProblem problem;

        public Backtracker(ConstraintSatisfactionProblem n_problem){
                problem = n_problem;
        }

        public void runBacktrack(HashMap<Variable, HashMap<String,
            List<Object>>> newWorld){

                LinkedList<Variable> unfilled = new
                    LinkedList<Variable>();
                LinkedList<Variable> filled = new LinkedList<Variable>();
                unfilled.addAll(newWorld.keySet());

                backtracking(newWorld, filled, unfilled, null);

                System.out.println("Solutions found: " + problem.nodes);

        }

        public void backtracking(HashMap<Variable, HashMap<String,
            List<Object>>> world,
                        LinkedList<Variable> filled, LinkedList<Variable>
                            unfilled, Variable currentVar){

        //Base case!
                if(problem.constraintsGood(filled, world)){
                        if(unfilled.size() == 0){
                                problem.printSolvedProblem(filled);
                                problem.nodes++;
                                return;
                        }
                } else {
                        return;
                }

        //Recursive case!
                Variable var = problem.getNext(currentVar);

                if(var == null){
                        var = unfilled.pop();
                } else {
                        unfilled.remove(var);
```

```
42 			}
43 
44 			filled.push(var);
45 
46 			for(Object val : world.get(var).get("domain")){
47 				var.val = ((Value)val);
48 
49 				backtracking(world, filled, unfilled, var);
50 
51 			}
52 
53 			var.val = null;
54 			filled.remove(var);
55 			problem.returnedVars.remove(var);
56 			unfilled.add(var);
57 		}
```

The method gets started with `runBacktrack()` at line 9. The system keeps two LinkedLists, *filled* and *unfilled*. *Filled* holds the *Variable* objects that have a value assigned to them, *unfilled* holds the objects whose values are null. The lists represent disjoint subsets of the keyset from *world*, the wrapper objet discussed in section 2. These lists get passed between different levels in the tree, being updated when values are set or unset. Initially, all the variables are put into *unfilled*.

Once inside the recursion, the base case starts at line 24. It uses the `constraintsGood()` method that was defined in `ConstraintSatisfactionProblem.java`. It takes the variables in the *filled* list, and checks all of their constraints to make sure nothing is wrong. If there is a problem, the tree is pruned at that point, and the method returns. If there isn't a problem, and all the variables have a value (indicated by the size of *unfilled* being 0) then a solution has been reached, at which point the solution is printed and the method returns.

If a solution for the problem has not been reached, then the method moves into the recursive case. Here is the chance for the simulation writer to build their own way of getting the next variable to be searched. The `getNext()` method is a highly suggested method for the user to implement. The given one in `ConstraintSatisfactionProblem.java` always returns null, in which case the head of the *unfilled* list is popped to get the next variable. If the user wants a say in how the variables should be explored, `getNext()` needs to be implemented.

Once a variable is chosen, it is removed from *unfilled*, and added to *filled*. The recursion happens in a loop in which the values in the domain of the variable are cycled through. Each time the variable takes on the value, and `backtraking()` is called with the next value in place.

4 Map-Coloring Problem

4.1 Implementation

Graph coloring is a way of giving values to vertexs based on certain constraints. A real life example of this is found in cartography. Zones on maps that are bordering one another are often made different colors so it is easier to distiguish the border between them. This type of problem fits our constraint satisfaction framework.

To solve it, I have written MapColoringProblem.java, which extends ConstraintSatisfactionProblem. Here is the implementation:

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.HashSet;
import java.util.Set;
import java.util.ArrayList;


public class MapColoringProblem extends ConstraintSatisfactionProblem{

        HashMap<Variable, HashMap<String, List<Object>>> world;

        public MapColoringProblem(HashMap<Variable, HashMap<String,
             List<Object>>> n_world){
                world = n_world;
                returnedVars = new HashSet<Variable>();
        }

        @Override
        public Variable getNext(Variable currentVar){

                if(currentVar == null){
                        Variable var = randomVariable(world.keySet());
                        while(returnedVars.contains(var)){
                                var = randomVariable(world.keySet());
                        }
                        returnedVars.add(var);
                        return var;
                }

                for(Object var : world.get(currentVar).get("neighbors")){
                        if(!returnedVars.contains(var)){
                                returnedVars.add(((Variable)var));
                                return (Variable)var;
                        }
                }

                return null;
        }


        private class MapVariable extends Variable{

                public MapVariable(String n_designation, Value n_val){
                        designation = n_designation;
                        val = n_val;
                }

                @Override
```

```
		public Variable copy(){
			MapVariable newVar = new MapVariable(designation,
			    val);
			return newVar;
		}
	}

	private class MapConstraint implements Constraint{

		public boolean isSatisfied(Variable variable) {

			List<Object> adjList =
			    world.get(variable).get("neighbors");
			if(adjList == null){
				return true;
			}

			for(Object neighbor : adjList){

				Value val = ((Variable)neighbor).val;
				if(val == null){
					continue;
				}
				if(variable.val.equals(val)){
					return false;
				}
			}

			return true;
		}

		public Constraint copy() {
			Constraint con = new MapConstraint();
			return con;
		}

	}

	private class MapValue implements Value{

		public String val;

		public MapValue(String n_val){
			val = n_val;
		}

		public Value copy() {
			return new MapValue(val);
		}

		public Object getValue() {
			return val;
		}

```

```
101             public String toString(){
102                     return val;
103             }
104
105             public boolean equals(Object o){
106                     return val.equals(((Value)o).getValue());
107             }
108
109     }
```

MapColoringProblem follows the rules I set in section 2 for the correct use of the framework. Starting off, I have overridden the getNext() method at line 19. The new version selects a neighbor of the current variable to return. It also makes sure the variable has not been returned before. If the current variable is null, or there are no neighbors, null is returned.

The most important parts of the problem stem from the three objects held within the world. MapVariable is the implementation of the Value interface we saw earlier; there is nothing interesting here besides a constructor and a copy method. These variables aren't too complicated. It's a similar story with MapValue. It mainly consists of the constructor and some helper methods(equals, toString, etc). In this case, the most important part of the world is the implementation of Constraint. MapConstraint handles the single constraint of the problem; neighboring variables cannot have the same color. The Constraint interface requires a isSatisfied() method. In MapConstraint, the isSatisfied() method gets the neighbors of the given variable and compares the values of each, making sure they are not the same.

4.2 Testing

There has been some discussion about whether to return all the solutions or just one. The previous time I implemented a backtracking algorithm (N-Queens in CS10) we returned all the solutions found by the algorithm, so I decided to do the same.

Here is the setup code for my MapColoringProblem test:

```
public static void main(String[] args){

        //World
        HashMap<Variable, HashMap<String, List<Object>>> world =
                        new HashMap<Variable, HashMap<String,
                            List<Object>>>();

        //Setup of variables
        MapColoringProblem mpProblem = new MapColoringProblem(world);
        MapVariable wa = mpProblem.new MapVariable("Western Australia",
            null);
        MapVariable nt = mpProblem.new MapVariable("Northern Territory",
            null);
        MapVariable q = mpProblem.new MapVariable("Queensland", null);
        MapVariable sa = mpProblem.new MapVariable("South Australia",
            null);
```

```java
        MapVariable nsw = mpProblem.new MapVariable("New South Wales", null);
        MapVariable v = mpProblem.new MapVariable("Victoria", null);
        MapVariable t = mpProblem.new MapVariable("Tazmania", null);

        //Setup Neighbor lists
        List<Object> wa_neighbor = new ArrayList<Object>(Arrays.asList(nt, sa));
        List<Object> nt_neighbor = new ArrayList<Object>(Arrays.asList(wa, sa, q));
        List<Object> q_neighbor = new ArrayList<Object>(Arrays.asList(nt, sa, nsw));
        List<Object> sa_neighbor = new ArrayList<Object>(Arrays.asList(wa, nt, q, nsw, v));
        List<Object> nsw_neighbor = new ArrayList<Object>(Arrays.asList(q, sa, v));
        List<Object> v_neighbor = new ArrayList<Object>(Arrays.asList(sa, nsw));
        List<Object> t_neighbor = new ArrayList<Object>();

        //Setup constraint and domain
        //They all have the same constraint and domain so there only needs
        //to be one object for the whole group

        //Constraint checks
        MapConstraint mConstraint = mpProblem.new MapConstraint();
        List<Object> constraintList = new ArrayList<Object>(Arrays.asList(mConstraint));
        //Three color problem domain values
        //colors are r, g, b
        Value rVal = mpProblem.new MapValue("R");
        Value gVal = mpProblem.new MapValue("G");
        Value bVal = mpProblem.new MapValue("B");

        List<Object> mapDomain = new ArrayList<Object>(Arrays.asList(rVal, gVal, bVal));

        //Put variables in the world
        mpProblem.putVariable(wa, constraintList, mapDomain, wa_neighbor, world);
        mpProblem.putVariable(nt, constraintList, mapDomain, nt_neighbor, world);
        mpProblem.putVariable(q, constraintList, mapDomain, q_neighbor, world);
        mpProblem.putVariable(sa, constraintList, mapDomain, sa_neighbor, world);
        mpProblem.putVariable(nsw, constraintList, mapDomain, nsw_neighbor, world);
        mpProblem.putVariable(v, constraintList, mapDomain, v_neighbor, world);
        mpProblem.putVariable(t, constraintList, mapDomain, t_neighbor, world);

```

```java
        new Backtracker(mpProblem).runBacktrack(world);
}
```

These code blocks setup all the necessary variables for a map coloring problem for the Australian territories. The problem is 3-colorable meaning that the least amount of colors useable where bordering territories will not have the same color is 3. The first block of code sets up the individual variables, which act as the nodes of the graph. This problem requires that edges between nodes exist because the constraint needs them. The edges have been defined in the next block where ArrayLists of adjacent nodes are setup. Since the constraints for the variables are the same, only one constraint object is needed, but is put into a list because there may be multiple constaint objects being passed into the world. The initial domain for all the variables is the same too, so only one domain is created and passed into the world with multiple references. Finally, `putVariable()` puts all 7 variables into the world, and backtrack is called.

Here is the output for the setup. The format is in "TERRITORY: COLOR", the possible colors are R, G, and B (red, green, and blue).

```
====================
Tazmania: R
Victoria: G
New South Wales: R
Queensland: G
South Australia: B
Western Australia: G
Northern Territory: R
====================
Tazmania: G
Victoria: G
New South Wales: R
Queensland: G
South Australia: B
Western Australia: G
Northern Territory: R
====================
Tazmania: B
Victoria: G
New South Wales: R
Queensland: G
South Australia: B
Western Australia: G
Northern Territory: R
====================
Tazmania: R
Victoria: B
New South Wales: R
Queensland: B
South Australia: G
Western Australia: B
Northern Territory: R
====================
Tazmania: G
Victoria: B
```

```
New South Wales: R
Queensland: B
South Australia: G
Western Australia: B
Northern Territory: R
====================
Tazmania: B
Victoria: B
New South Wales: R
Queensland: B
South Australia: G
Western Australia: B
Northern Territory: R
====================
Tazmania: R
Victoria: R
New South Wales: G
Queensland: R
South Australia: B
Western Australia: R
Northern Territory: G
====================
Tazmania: G
Victoria: R
New South Wales: G
Queensland: R
South Australia: B
Western Australia: R
Northern Territory: G
====================
Tazmania: B
Victoria: R
New South Wales: G
Queensland: R
South Australia: B
Western Australia: R
Northern Territory: G
====================
Tazmania: R
Victoria: B
New South Wales: G
Queensland: B
South Australia: R
Western Australia: B
Northern Territory: G
====================
Tazmania: G
Victoria: B
New South Wales: G
Queensland: B
South Australia: R
Western Australia: B
Northern Territory: G
====================
```

```
Tazmania: B
Victoria: B
New South Wales: G
Queensland: B
South Australia: R
Western Australia: B
Northern Territory: G
====================
Tazmania: R
Victoria: R
New South Wales: B
Queensland: R
South Australia: G
Western Australia: R
Northern Territory: B
====================
Tazmania: G
Victoria: R
New South Wales: B
Queensland: R
South Australia: G
Western Australia: R
Northern Territory: B
====================
Tazmania: B
Victoria: R
New South Wales: B
Queensland: R
South Australia: G
Western Australia: R
Northern Territory: B
====================
Tazmania: R
Victoria: G
New South Wales: B
Queensland: G
South Australia: R
Western Australia: G
Northern Territory: B
====================
Tazmania: G
Victoria: G
New South Wales: B
Queensland: G
South Australia: R
Western Australia: G
Northern Territory: B
====================
Tazmania: B
Victoria: G
New South Wales: B
Queensland: G
South Australia: R
Western Australia: G
```

```
Northern Territory: B
Solutions found: 18
Nodes Visited: 157
```

5 Circuit Board Positioning

Positions chips on a circuit board brings up a similar problem to map coloring. The chips have multiple possible locations, variable size, and they all need to fit on in a 2D space without any overlap.

5.1 Implementation

Once again we can use the CSP framework to solve this kind of problem. My implementation in `CircuitBoardProblem.java` handles the problem. In this implementation, the locations of each chip are represented by 2D coordinates that correspond to locations on the board. The locations represent the bottom left corner of each chip.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;

public class CircuitBoardProblem extends ConstraintSatisfactionProblem{


	HashMap<Variable, HashMap<String, List<Object>>> world;
	public int BOARD_HEIGHT = 3;
	public int BOARD_WIDTH = 10;

	public CircuitBoardProblem(HashMap<Variable, HashMap<String,
	    List<Object>>> n_world){
		world = n_world;
		returnedVars = new HashSet<Variable>();
	}

	@Override
	public Variable getNext(Variable currentVar){
		return null;
	}

	private class CircuitBoardVariable extends Variable{

		public int width;
		public int height;

		public CircuitBoardVariable(String n_designation, Value
		    n_val, int n_width, int n_height){
			designation = n_designation;
```

```
                        val = n_val;

                        width = n_width;
                        height = n_height;
                }

                public Variable copy(){
                        return new CircuitBoardVariable(designation, new
                            CircuitBoardValue(((CircuitBoardValue)val).x_val,
                            ((CircuitBoardValue)val).y_val), width,
                            height);
                }
        }

        private class CircuitBoardValue implements Value{

                //Value represents the x and y coordinates
                //of the bottom left
                public int x_val;
                public int y_val;

                public CircuitBoardValue(int n_x_val, int n_y_val){
                        x_val = n_x_val;
                        y_val = n_y_val;
                }

                public Value copy() {
                        return new CircuitBoardValue(x_val, y_val);
                }

                public String toString(){
                        return "(" + x_val + ", " + y_val + ")";
                }

        }

        private class CircuitBoardConstraint implements Constraint{

                public boolean isSatisfied(Variable variable) {
                        CircuitBoardVariable var =
                            (CircuitBoardVariable)variable;
                        CircuitBoardValue val = (CircuitBoardValue)var.val;

                        //Check if base point is off board
                        if(val.x_val < 0 || val.y_val < 0 || val.x_val >=
                            BOARD_WIDTH || val.y_val >= BOARD_WIDTH) {
                                return false;
                        }

                        //Check if the rest of the chip is off board
                        if(val.x_val + var.width > BOARD_WIDTH ||
                            val.y_val + var.height > BOARD_HEIGHT){
                                return false;
                        }
```

```java

                //Check if there is overlap with any other chip
                for(Variable otherVar : world.keySet()){
                    if(isOverlap(var, ((CircuitBoardVariable)otherVar))){
                        return false;
                    }
                }

                return true;
            }

            private boolean isOverlap(CircuitBoardVariable var1, CircuitBoardVariable var2){
                if(var1 == var2){
                    return false;
                }

                CircuitBoardValue val1 = (CircuitBoardValue)var1.val;
                CircuitBoardValue val2 = (CircuitBoardValue)var2.val;

                if(val1 == null | val2 == null){
                    return false;
                }

                if(val1.x_val == val2.x_val && val1.y_val == val2.y_val){ return true; }

                for(int i=val1.x_val; i < var1.width + val1.x_val; i++){
                    for(int v=val1.y_val; v < var1.height + val1.y_val; v++){
                        if(i >= val2.x_val && i < val2.x_val + var2.width && v >= val2.y_val && v < val2.y_val + var2.height){
                            return true;
                        }
                    }
                }

                return false;
            }

            public Constraint copy() {
                return new CircuitBoardConstraint();
            }
        }

        public void printSolvedProblem(LinkedList<Variable> variables){
            System.out.println("============");
            String[][] array = new String[BOARD_WIDTH][BOARD_HEIGHT];
```

```
		for(int i = 0; i < BOARD_WIDTH; i++){
			for(int v=0; v < BOARD_HEIGHT; v++){
				for(Variable var1 : world.keySet()){
					CircuitBoardVariable var =
					    (CircuitBoardVariable)var1;
					CircuitBoardValue val =
					    (CircuitBoardValue)var.val;

					if(val.x_val == i && val.y_val == v){
						for(int j=0;
						    j<var.width;j++){
							for(int b=0;
							    b<var.height;b++){
								array[i+j][v+b]
								    =
								    var.designation;
							}
						}
					}
				}
			}
		}

		for(int i=0; i < BOARD_HEIGHT; i++){
			for(int v=0; v < BOARD_WIDTH; v++){
				if(array[v][i] == null){
					System.out.format(".");
				} else {
					System.out.format(array[v][i]);
				}
			}
			System.out.format("\n");
		}

		System.out.println("=============");
		System.out.println("");
	}



	public static void main(String[] args){

		//World
		HashMap<Variable, HashMap<String, List<Object>>> world =
				new HashMap<Variable, HashMap<String,
				    List<Object>>>();

		//Setup of variables
		CircuitBoardProblem cbProblem = new
		    CircuitBoardProblem(world);
		CircuitBoardVariable a = cbProblem.new
		    CircuitBoardVariable("a", null, 3, 2);
		CircuitBoardVariable b = cbProblem.new
```

```
                CircuitBoardVariable("b", null, 5, 2);
169         CircuitBoardVariable c = cbProblem.new
                CircuitBoardVariable("c", null, 2, 3);
170         CircuitBoardVariable e = cbProblem.new
                CircuitBoardVariable("e", null, 7, 1);
171
172
173         //Setup constraint and domain
174         //They all have the same constraint and domain so there
                only needs
175         //to be one object for the whole group
176
177         //Constraint checks
178         CircuitBoardConstraint cbConstraint = cbProblem.new
                CircuitBoardConstraint();
179         List<Object> constraintList = new
                ArrayList<Object>(Arrays.asList(cbConstraint));
180
181         //Three color problem domain values
182         //colors are r, g, b
183
184         List<Object> domain = new ArrayList<Object>();
185
186         for(int i=0; i < cbProblem.BOARD_WIDTH; i++){
187                 for(int v=0; v < cbProblem.BOARD_HEIGHT; v++){
188                         domain.add(cbProblem.new
                                CircuitBoardValue(i, v));
189                 }
190         }
191
192         //Put variables in the world
193         cbProblem.putVariable(a, constraintList, domain, null,
                world);
194         cbProblem.putVariable(b, constraintList, domain, null,
                world);
195         cbProblem.putVariable(c, constraintList, domain, null,
                world);
196         cbProblem.putVariable(e, constraintList, domain, null,
                world);
197
198
199         new Backtracker(cbProblem).runBacktrack(world);
200     }
201 }
```

This code is formatted similiarly to the map-coloring problem. The main differences in the implementation stems from the nature of the problem. This problem doesn't require a graph structure, so in the setup code (see line 159), it doesn't use the optional "neighbor" list that the map-coloring problem utilizes. Due to this, the `getNext()` method does not do anything in particular. It returns null because the next node to check is not important and doesn't help the search.

The setup for the variables is a bit more complex than the map-coloring problem. Each variable (see line 25) has *width* and *height* variables, because each chip will have different sizes, but the sizes are held in the *Variable* extension because they are unchanging. Similar to the map-coloring variable, the circuit board variable holds a designation - once again used for identification purposes - and a value *val* - used to hold a instance of *CircuitBoardValue*.

CircuitBoardValue - as previously mentioned - holds its value in two variables *x_val* and *y_val*, which represent the location of the bottom left point of the chip.

The domain of each variable (the chips) is every possible location on the board. If there is a 10x3 board, then the domain would be all values (X, textitY where $0 >= X > 10$ and where $0 >= Y > 3$.

The constraint for the problem is described in the code at line 91. This is were the most important part ofthe implementation comes in; no two chips can overlap, so by cross-checking the points one chip covers, with the boundaries of other variables, we can determine if a location for a chip is legal. The method `isOverlap()` handles checking if two variables are overlapping.

5.2 Testing

The code in the `main()` method is the setup code for the problem. To test the problem, I setup the example board outlined in the assignment - a 10x3 board with four chips of varying sizes. The individual chips are represented as follows

```
aaa
aaa

bbbbb
bbbbb

cc
cc
cc

eeeeeee
```

The backtracking solver found 16 possible solutions for the problem, many of which are just slight variations of one another. The chips are represented by the ASCII art above. Positions that are not covered are represented by ”.”s.

```
============
bbbbbaaacc
bbbbbaaacc
eeeeeee.cc
============

============
bbbbbaaacc
bbbbbaaacc
.eeeeeeecc
============

============
```

```
eeeeeee.cc
bbbbbaaacc
bbbbbaaacc
=============

=============
.eeeeeeecc
bbbbbaaacc
bbbbbaaacc
=============

=============
ccbbbbbaaa
ccbbbbbaaa
cceeeeeee.
=============

=============
ccbbbbbaaa
ccbbbbbaaa
cc.eeeeeee
=============

=============
cceeeeeee.
ccbbbbbaaa
ccbbbbbaaa
=============

=============
cc.eeeeeee
ccbbbbbaaa
ccbbbbbaaa
=============

=============
aaabbbbbcc
aaabbbbbcc
eeeeeee.cc
=============

=============
aaabbbbbcc
aaabbbbbcc
.eeeeeeecc
=============

=============
eeeeeee.cc
aaabbbbbcc
aaabbbbbcc
=============

=============
```

```
.eeeeeeecc
aaabbbbbcc
aaabbbbbcc
=============

=============
ccaaabbbbb
ccaaabbbbb
cceeeeeee.
=============

=============
ccaaabbbbb
ccaaabbbbb
cc.eeeeeee
=============

=============
cceeeeeee.
ccaaabbbbb
ccaaabbbbb
=============

=============
cc.eeeeeee
ccaaabbbbb
ccaaabbbbb
=============

Solutions found: 16
```

6 Maintaining Arch Consistency

In order to speed up the the backtracking search, after each value is set, the domains of every further variable can be updated based on the possible values of the new situation. This allows the search to not even go down a branch of the tree that it knows it will not initially be able to search. This decreases search time drastically.

I implemented a version of the MAC-3 algorithm with the map-coloring problem that I discussed earlier. At the start of each recursive method call of `backtracking()`, the user of the framework is allowed to override a method called `updateDomains()`, because MAC-3 works differently for every problem, I required the user to make the decision about how to implement the consistency.

The *Value* interface has been changed to a class, because I need to require a boolean variable *toBeConsidered*. This variable tells the backtracker whether or not to look at this value as a possibility. It allows me to "remove" values from the domain without actually removing them. I

The following is the implementation of the `updateDomains()` method for the map-coloring problem:

```
public void updateDomains(LinkedList<Variable> filled,
    LinkedList<Variable> unfilled){
		if(filled.size() == 0 || unfilled.size() == 0){
			return;
		}

		for(Variable var : filled){
			List<Object> neighbors =
			    world.get(var).get("neighbors");
			if(neighbors == null){
				continue;
			}

			for(Object n : neighbors){
				MapVariable neighbor = (MapVariable)n;

				for(Object v :
				    world.get(neighbor).get("domain")){
				    MapValue val = (MapValue)v;
				    if(val.equals(var.val)){
					val.toBeConsidered = false;
				    }
				}
			}
		}

		for(Object var : unfilled.toArray()){
			int i = 0;
			MapValue chosenVal = null;
			Variable variable = (Variable)var;
			for(Object v : world.get(variable).get("domain")){
				MapValue val = (MapValue)v;

				if(val.toBeConsidered){
					i++;
					chosenVal = val;
				}
			}

			if(i == 1 && chosenVal != null){
				variable.val = chosenVal;
				unfilled.remove(variable);
				filled.add(variable);
				returnedVars.add(variable);
			}
		}

		for(Variable var : filled){
			for(Object v : world.get(var).get("domain")){
				MapValue val = (MapValue)v;

				val.toBeConsidered = true;
			}
		}
```

```
}
```

The method does three things. First, it takes the variables that already have set values, and updates the domains of their neighbors to do the brunt of the MAC-3 algorithm. Next, it checks if any of the variables without values set, only have a domain of one possible value. If there is only one possible value, then the value of the variable gets set. Finally, all the values in the domains of the filled values get set to true because once the recursion doubles back on itself, the arch-consistency needs to be maintained.

6.1 Testing

Unfortunately, the code is not behaving exactly as it should. In testing for the MAC-3, the same number of solutions for the Australia problem were not being returned.

Here is the output from the test:

```
====================
Tazmania: R
Victoria: R
Western Australia: R
Northern Territory: G
Queensland: R
South Australia: B
New South Wales: G
====================
Tazmania: G
Victoria: R
Western Australia: R
Northern Territory: G
Queensland: R
South Australia: B
New South Wales: G
====================
Tazmania: B
Victoria: R
Western Australia: R
Northern Territory: G
Queensland: R
South Australia: B
New South Wales: G
Solutions found: 3
Nodes Visited: 15
```