A PROJECT REPORT ON

“Map Coloring CSP using Minimum Remaining Value and Min-conflict heuristics and relative performance evaluation”

BY
Abhishek Kabra 800888484

UNDER THE GUIDANCE OF
Dr. Jing Xiao

`

DEPARTMENT OF COMPUTER SCIENCE

COLLEGE OF COMPUTING AND INFORMATICS

CHARLOTTE, NORTH CAROLINA - 28262

Fall 2015

1. Objective:

According to the four color theorem, given any separation of plane into contiguous region, no more than four colors are required to color the regions in the map such that no two adjacent regions have the same color. Objective of this project is to solve map coloring problem using Forward Checking with Minimum Remaining Value (MRV) heuristic and by local search using Min-Conflict heuristics. Map coloring problem can be modeled as a Constraint Satisfaction Problem (CSP).

2. Constraint Satisfaction Problem (CSP):

CSP's are special type of mathematical problems where a state of the problem must satisfy constraints defined in the problem. In CSP, final solution of the problem is important than the optimal path to the solution. Therefore for CSP, search can be made easier and efficient by eliminating large portion of the search space by identifying variable-value pairs which violates the constraints.

Definition of CSP: A constraint satisfaction problem is consist of 3 elements.

$X = \{X_1, \ldots, X_n\}$ is a set of variables,
$D = \{D_1, \ldots, D_n\}$ is a set of the respective domains of values, and
$C = \{C_1, \ldots, C_m\}$ is a set of constraints.

- Domain D_i for a variable is defines the set of all possible values a variable X_i can take.
  Types of Domains:
  i. Discrete
  ii. Continuous
  iii. Finite
  iv. Infinite

- Constraint C_i is a pair of <scope, relation>.
  - Scope is a tuple of variables participating in constraint.
  - Relation is a list of all combinations of variable values. Relation can be explicitly specified by a list of variable values or as a relation between values such as "not equal to".

  Example: Let X_1, X_2 be the variables and let domain be {A, B}, then constraint can be specified in following ways:

    - Using explicit list of variable values: ⟨(X1,X2), [(A,B), (B,A)]⟩
    - Using relation between variable values: ⟨(X1,X2),X1≠ X2⟩
  - Types of Constraints:
    - Unary: These type of constraints involve 1 variable
    - Binary: These type of constraints involves two variable.

Solution: Notion of solution to CSP can be defined by assignment of values to the all the variables. Assignments can be following types:

- Partial Assignment: Only some variables of the problem are assigned values.
- Consistent Assignment: An assignment that does not violate any constraints.
- Complete Assignment: All the variables are assigned value.
- Solution: Complete and consistent is a solution to CSP.

Some CSP have objective function to maximize.

Solving CSP: Various search techniques can be used to solve CSP. Most widely used techniques are

1. **Backtracking Search:** Backtracking search is nothing but a Depth First Search in context of CSP. In backtracking search for CSP, an unassigned variable is selected at a time and is assigned a value. Algorithm backtracks when there are no legal value is left to assign to variable. Since it is DFS, algorithm tries to assign new value for variable from previous call.

2. **Constraint Propagation:** Basic idea behind constraint propagation is to maintain local consistency. It helps to convert problem in simpler problem. Constraint propagation can used as preprocessing step in searching algorithm.

3. **Local Search:** Unlike backtracking search, local search method starts with complete random assignment of variables and iteratively improve the assignments by trying to satisfy constraints.

3. Formulation of map coloring problem as CSP:

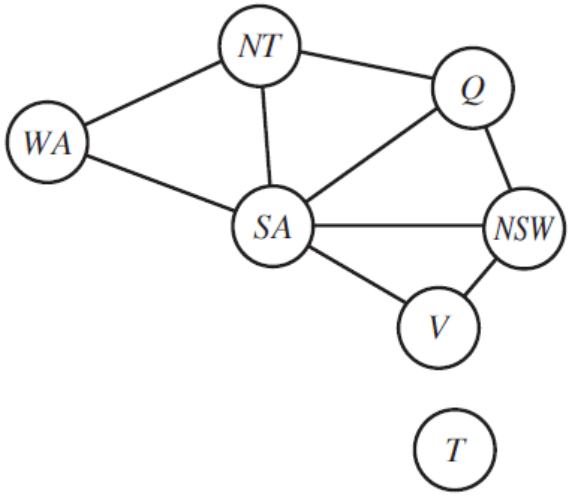
- **Variables:**
  Regions in the map to be colored defines the set of variables of the problem.
  Example: X = {List of countries in the world} or {List of states in a country}

- **Domain:**
  Domain for map coloring problem is a set of legal color values regions can take.
  Example of domain: D = {Red, Green, Blue, Yellow}

  Map coloring problem has discrete and finite domain which makes it easier to solve.

- **Constraints:**
  Constraint for map coloring problem is that no two adjacent regions can have same color values.
  Example: Lets R1 and R2 be two adjacent regions in a map then constraint can be written as $\langle (R1, R2), R1 \neq R2 \rangle$

Constraint graph can be used to visualize constraints. Nodes in the graph represent region and edge represent constraints.
Example:



- **Solution:** Solution to the map coloring problem is assignment of colors to the all the regions in the map such that no two adjacent regions have same color.

4. Map coloring using Forward Checking with MRV heuristic:

This is essentially a backtracking search algorithm. However, it can be observed that simple uninformed backtracking search is inefficient. Time and space requirements make simple backtracking search infeasible for solving CSP. Let's consider we have n variable and d number of domain values. At top level of the search tree we have n*d possible assignments. At next level we will have (n-1)*d assignments. Hence for n variables we end up creating a tree with $n! * d^n$ leaves.

We know that in CSP, only the last solution is important and not the order of application. Therefore, we only need to consider one variable at each level. Using this knowledge, we will have tree with d^n leaves. Even though we have reduced size of the tree, d^n is a big number. For example, for a map with 5 regions and we will need have tree with 4^5 leaf nodes i.e. 1024.

Efficiency of uninformed backtracking search can be improved using heuristic. Beauty of CSP is that we can improve efficiency without even domain specific knowledge.

- **Minimum Remaining Value:** MRV heuristic helps to select an unassigned variable. Basic idea for MRV heuristic is to select a variable that has least legal remaining domain values. It is also called as Most Constrained Variable or fail-first heuristic.
  For map coloring problem, MRV heuristic selects a region with fewest color options available. Whenever in the algorithm a region has find a region with no legal color values left, it backtracks.

- **Degree Heuristic**: It can observed that MRV heuristic cannot help us to select first region from the map for coloring since at the start of the algorithm every region has 4 legal color values left. In such cases, a region with maximum neighbors is selected as this region in involved in maximum number of constraints.

  MRV heuristic has precedence over degree heuristic, however degree heuristic is used a tie-breaker when 2 regions has same number of legal color values left.

- **Forward Checking**: In Map Coloring CSP constraint is that no 2 adjacent regions can have same color. This knowledge about CSP can be used to improve efficiency of search to a great extent. In forward checking, as soon as the color is assigned to a region, currently assigned color is removed from the domain values of neighboring regions. This helps in maintaining arc consistency in the constraint graph.

Program structure:

1. **Technologies used:** This project is a web based application. Technologies used are mainly HTML5, JavaScript, CSS. It also uses D3 and topojson JavaScript libraries.

   - **D3.js:** D3.js is a JavaScript library for manipulating documents based on data. D3 helps to create visualization based on data using HTML, SVG, and CSS.
   - **topojson:** It is an extension of GeoJson, an open standard to create GIS files.
   - Algorithms are writeen in JavaScripts.

2. **Global variables:**
   - **countries:** This array is used to store list of countries from input json file of world map.
   - **neighbours:** 2D array to store neighbors of each country in the world map. This also read from input world map json file.
   - **listOfRemNode:** This array stores list of remaining unassigned regions at any point of time in program.
   - **mapNodes:** Array to store objects of class MapNode.
   - **numOfDeadEnds:** Integer to measure dead ends encountered during program. This essentially gives us number of times the algorithm backtracks.
   - **numOfRecursiveCalls:** Integer to count the number of recursive calls made during the program.
     Last two variable are used to measure the performance of algorithm.

3. **Data structure:**
   - **class MapNode:** This is a class for a node in a constraint graph. An instance of MapNode class represents a region in map.

     Below are the data members of the class:
     - **mapArrayIndex:** Stores the index of this region in input array of countries from json file
     - **color:** String to store the current color assignment of region
     - **legalColors:** Array to hold list of allowed colors for the region
     - **isColorSet:** Boolean to distinguish if region is an assigned or unassigned variable
     - **neighbours:** Array to hold list of neighboring regions. This array serves the purpose of adjacency list.

     Below are the methods for MapNode class:
     - **forwardChecking():** This method maintains arc consistency with neighboring regions by deleting recently used color value from legalColors array of neighboring regions.

     - **setMapNodeColor():** Method sets the color to the region and sets boolean variable is isColorSet to true. A call to method forwardChecking() is given at the end of this method.

     - **unsetMapNodeColor():** Method to unset the color to a region if algorithm hits dead end and backtracks. Also sets boolean isColorSet to false. Method restoreNeighbours() is called at the end of this method.

     - **restoreNeighbours():** Adds recently unset color value of the region to the domain of neighboring regions.

     - **deleteValueFromDomain():** Helper method that deletes input color from the array legalColors and is called in forwardChecking method.

     - **satisfyConstraint():** This method checks if selected color value for the region satisfies all the constraints. This method return true if selected color value for the region is not the color neighboring regions otherwise returns false.

4. **Global Functions:**
   - **fcMRV():** Recursive function implementing backtracking search with forward checking and minimum remaining value heuristic.

- **pickUnassignedVariable():** Function picks unassigned variable from listOfRemNode based on MRV and Degree heuristic.

- **colorMapWithFCMRV():** This is main method in JavaScript called from HTML. This method returns list of map regions with corresponding color assigned.

5. Map coloring using local search with Min-conflict heuristic:

Map coloring using local search start with complete assignment of colors to regions in the map. During initial assignment of color for each region is a color is selected randomly from domain. Local search algorithm iteratively improves color assignments by reducing total number of conflicts. Total conflicts measures the number of adjacent regions having same color. In each iteration, a conflicted region is selected randomly. For randomly selected region, all color values in domain are tried and color that minimizes conflicts with neighboring region is selected and set to region. This is called **min-conflict heuristic** as it minimizes total number of conflicts. Same process is repeated until total conflicts becomes 0 or maximum threshold number of iterations are reached.

Local minima: Search may get stuck in local minima meaning that number of conflict does not improve over iterations. In such cases random restarts is used to restart algorithm from initial assignment.

Connected Component: By Identifying connected components of constraint graph, problem is divided into independent sub problems. Dividing problem into sub problems reduces the overall time complexity of the program. Total work without decomposition is $O(d^n)$ whereas with independent connected component total work is reduced to $O(d^c n/c)$. In case world map, connected components can be continents or islands. Topological sorting is done to get independent component of the problem.

Program structure:

1. **Global Variables:**
   - **countries:** This array is used to store list of countries from input json file of world map.
   - **neighbours:** 2D array to store neighbors of each country in the world map. This also read from input world map json file.
   - **listOfRemNode:** This array stores list of remaining unassigned regions at any point of time in program.
   - **connectedComponents:** Array to store connected components of map.
   - **mapNodesMC:** Array to store objects of class McMapNode.
   - **numOfRestarts:** Integer to measure number of random restarts required for solving problem.

- **numOfTotalSteps:** Integer to count the number of iteration during program for all components.

Last two variables are used to measure the performance of algorithm.

2. **Data structure:**
   - **class McMapNode:** This is a class for a node in a constraint graph. An instance of McMapNode class represents a region in map.

     Below are the data members of the class:
     - **mapArrayIndex:** Stores the index of this region in input array of countries from json file
     - **color:** String to store the current color assignment of region
     - **legalColors:** Array to hold list of allowed colors for the region
     - **isColorSet:** Boolean to distinguish if region is an assigned or unassigned variable
     - **isVisited:** Boolean used in topological sorting to get connected components.
     - **neighbours:** Array to hold list of neighboring regions. This array serves the purpose of adjacency list.

   Below are the methods for McMapNode class:
     - **assignRandomColor():** Assigns random color to the region from list of allowed colors.
     - **setColor(color):** Sets input color to the region.

3. **Functions:**
   - **randomRestart():** Sets random colors from domain to all the regions in the map.
   - **getConnectedComponents():** Returns the list of connected components of map.
   - **traverseDepth():** Recursive function to do topological sorting.
   - **solveWithMinConflict():** Solves CSP for each component using Min-Conflict heuristic.
   - **getNumOfConflictsOfComponents():** Returns the total number of conflicts for each component.
   - **getConflictOfNode():** Returns number of conflicts of each region.
   - **pickConflictedVariable():** randomly picks a conflicted node

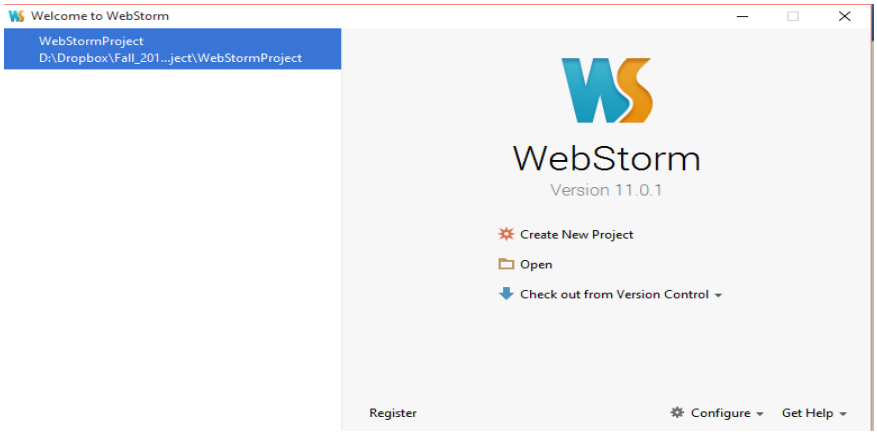
- **selectValWithMinConflict():** From the domain of allowed colors selects a value for region which minimizes conflicts.

- **colorMapWithMinConflict():** This is a main function called from HTML.

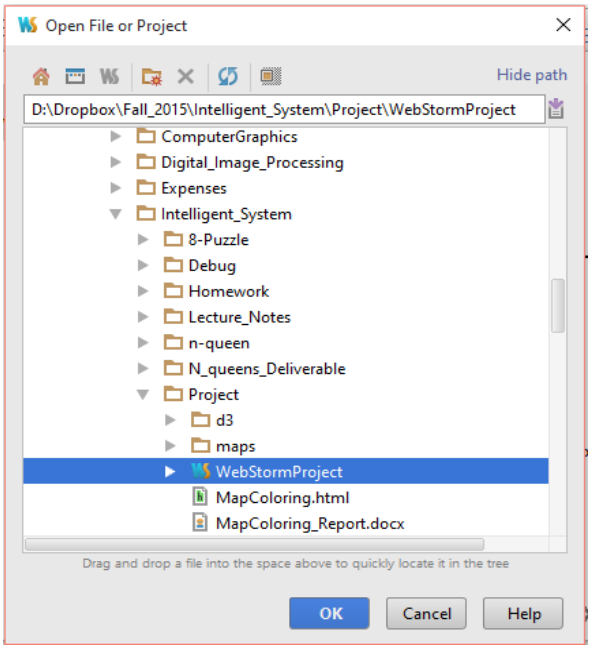
6. Application

- **How to setup & How to run:**

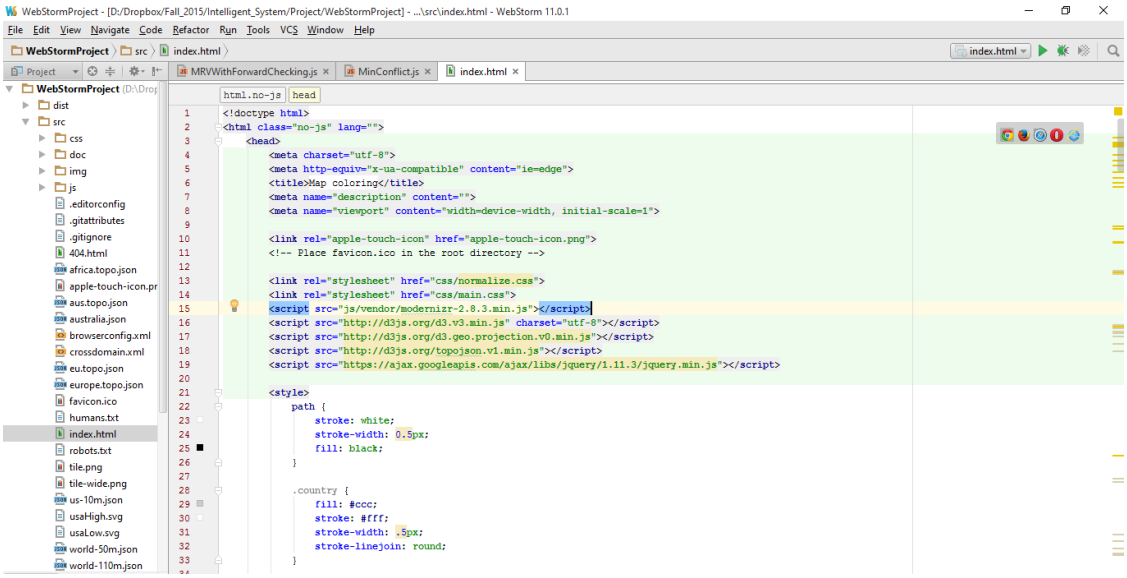
1. Since this is a web-based application, we need to setup local web server.
2. Download web storm JavaScript IDE from below link.
   https://www.jetbrains.com/webstorm/download/
3. Start webstorm IDE:



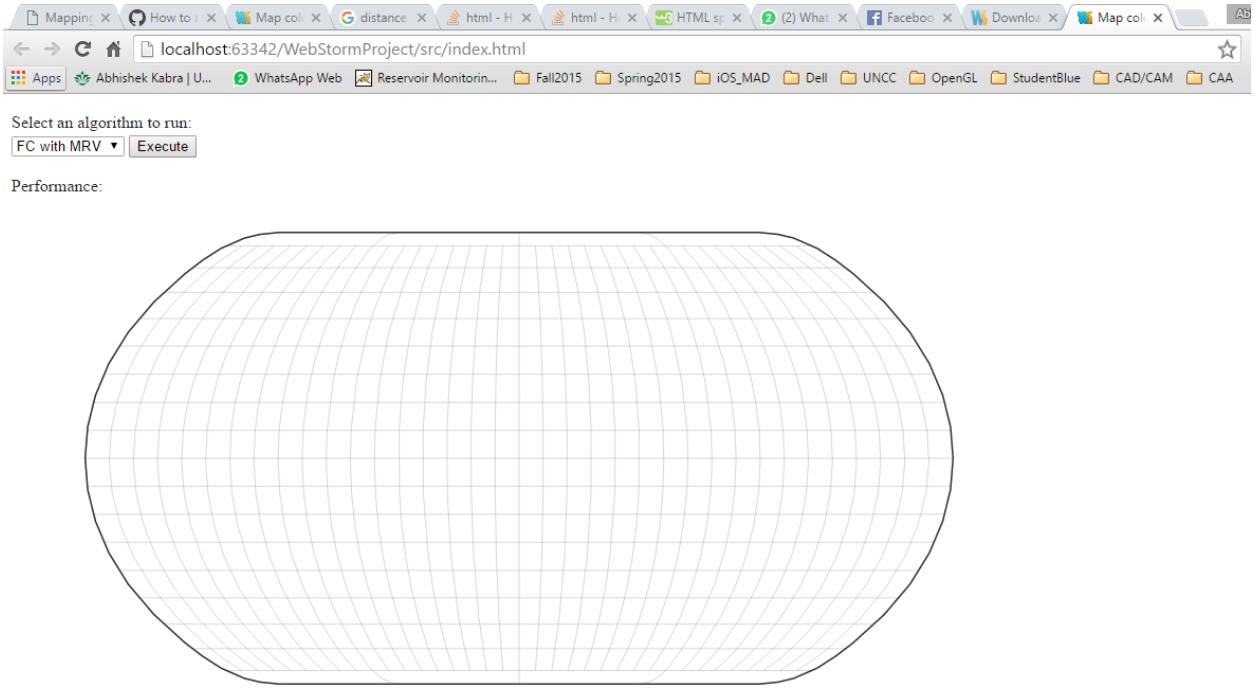
4. Click on open. Browse to project folder submitted on moodle. Click on OK.



5. Go to index.html and click on Chrome icon.



6. Application will open in browser



7. Select the algorithm from dropdown and click on execute.
8. **Refresh page**, select other algorithm to run again and click on execute.

- **How it works:**
    1. After clicking on execute button, algorithm reads **world-50m.json** file from project director and parse it using D3.
    2. D3 parses json file and returns list of countries and corresponding neighbors of each country.
    3. Depending on algorithm selected in the dropdown algorithm determines color for each country and returns an array of countries with corresponding color. This array is iterated to color each region on map.

7. Results: In order to analyze and compare performance of two algorithms, following key performance indicators are selected:

- **Forward checking with MRV:**
    1. Total number of times algorithm backtracks (domain wipeout)
    2. Total number of recursive calls made during execution
- **Local search using min-conflict heuristic:**
    1. Total number of random restarts
    2. Total number of steps during execution

Algorithm was tested on various maps having number of region in the range of 11 to 241. Below is sample relative performance table for both algorithms.

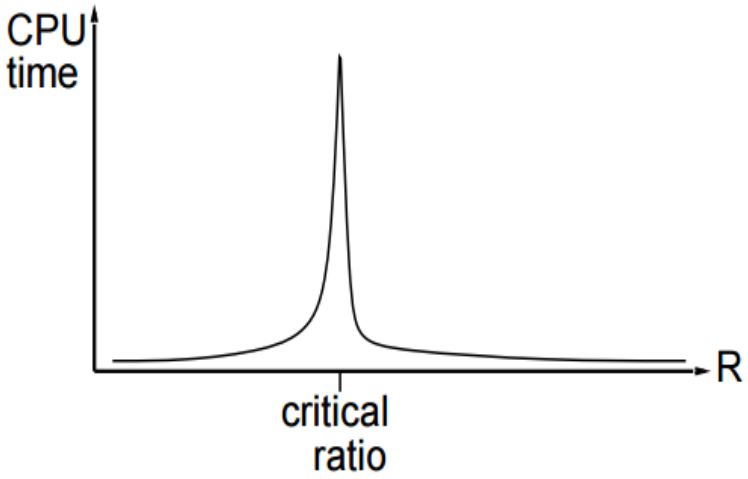
Number of regions	Forward checking with MRV		Local search with Min-conflict	
	Number of recursive calls	Number of backtracks	Number of random restarts	Number of total steps
11	12	0	0	8
53	54	0	0	87
58	59	0	2	4700
241	242	0	4	32051

As it evident from the result that forward checking with MRV performs better compared to local search using min-conflict heuristic.

Forward checking with MRV, colors all the regions (tested up to 241) without even having to backtrack due to domain wipeout.

On the other hand, local search using Min-conflict is inefficient for map coloring CSP. Performance of local search is critically dependent on initial random start. Hence performance can be vastly different during each time the algorithm is executed. Local search method also suffers from local minima. Critical ratio of number of constraints to number variables also plays an important role in performance of local search methods.

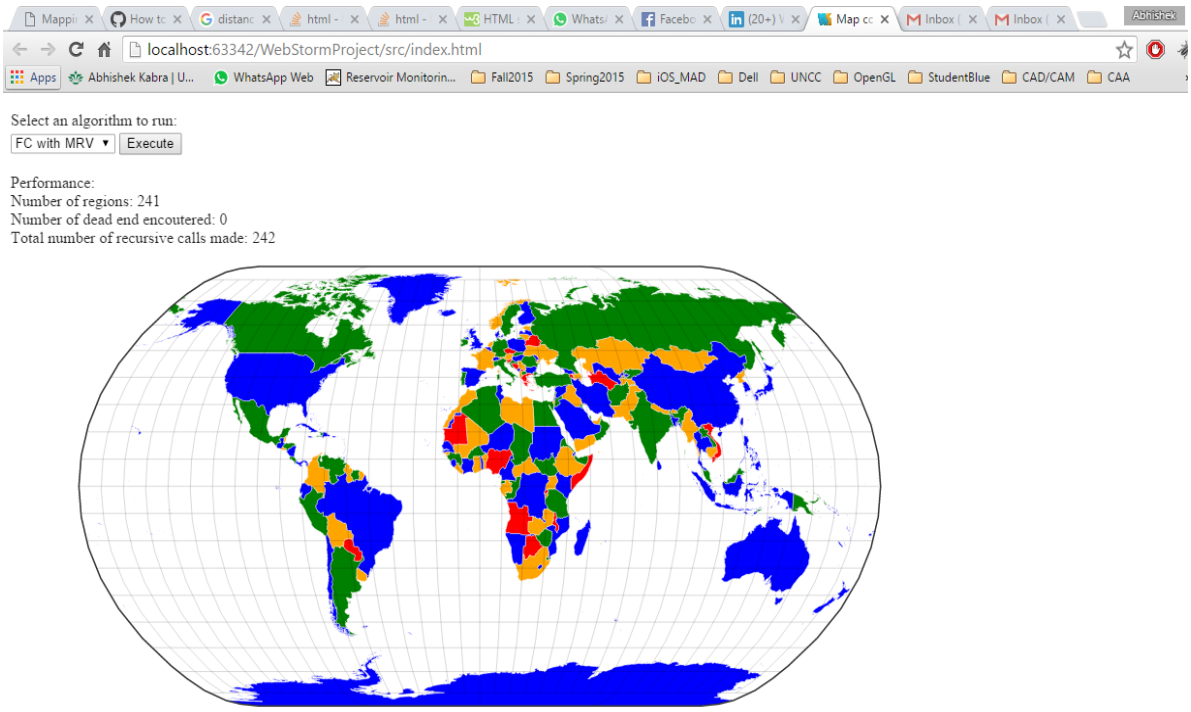
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



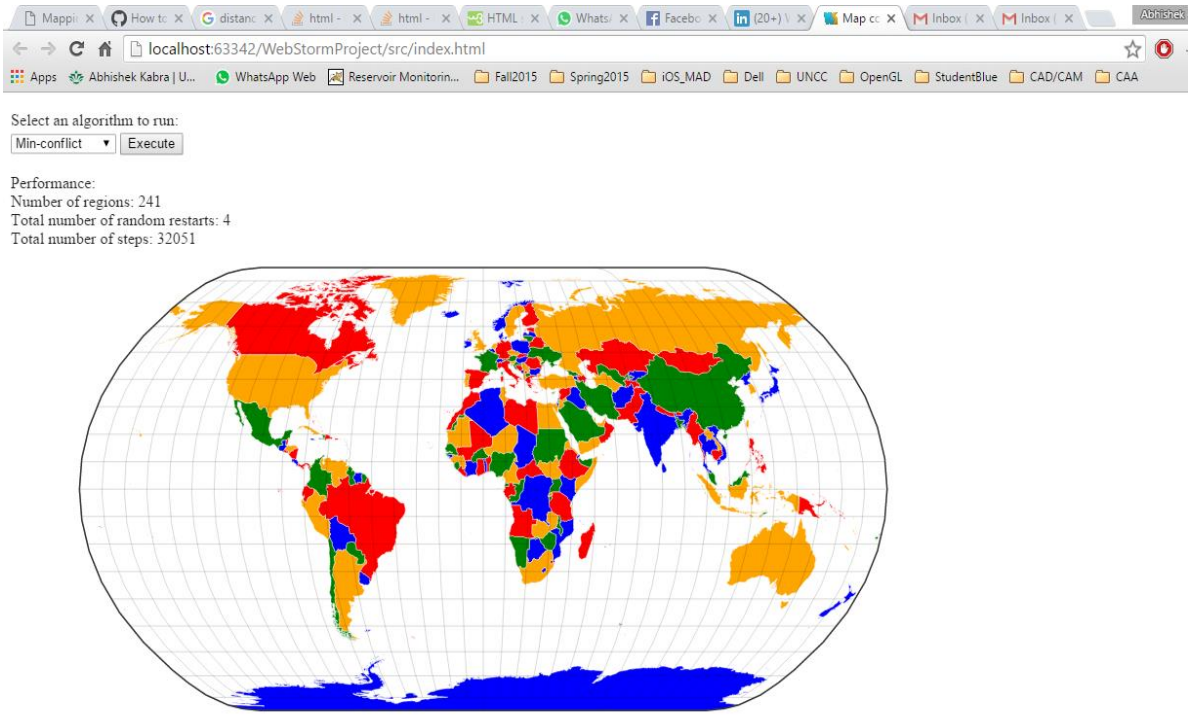
Conclusion: It is advisable to use forward checking with minimum value remaining heuristic for map coloring problem. Forward checking with minimum value remaining heuristic has predictable performance standards.

Below are images of few sample runs:

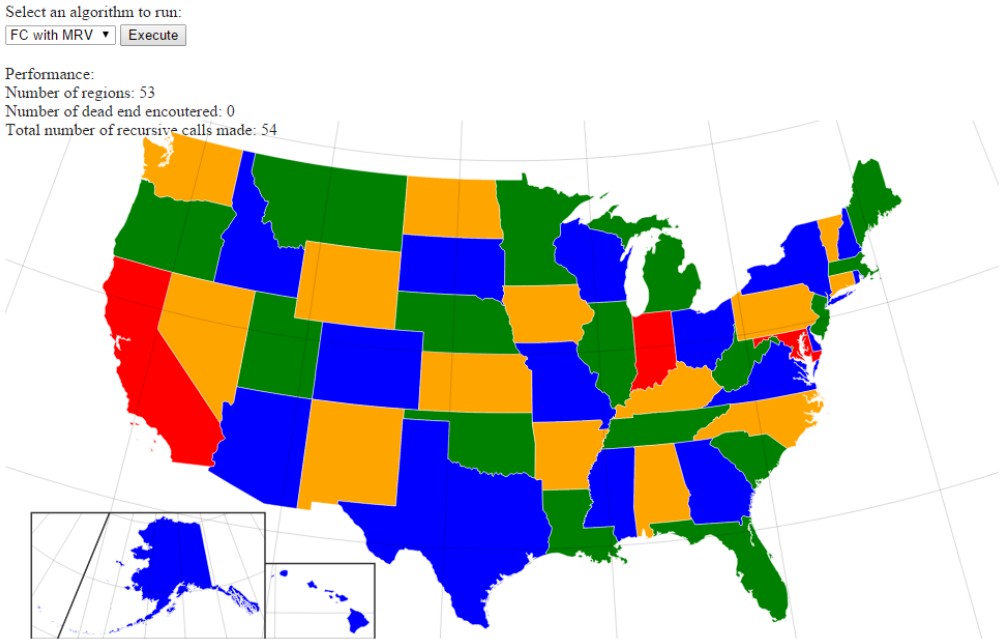
1. World map using Forward checking with MRV heuristic:



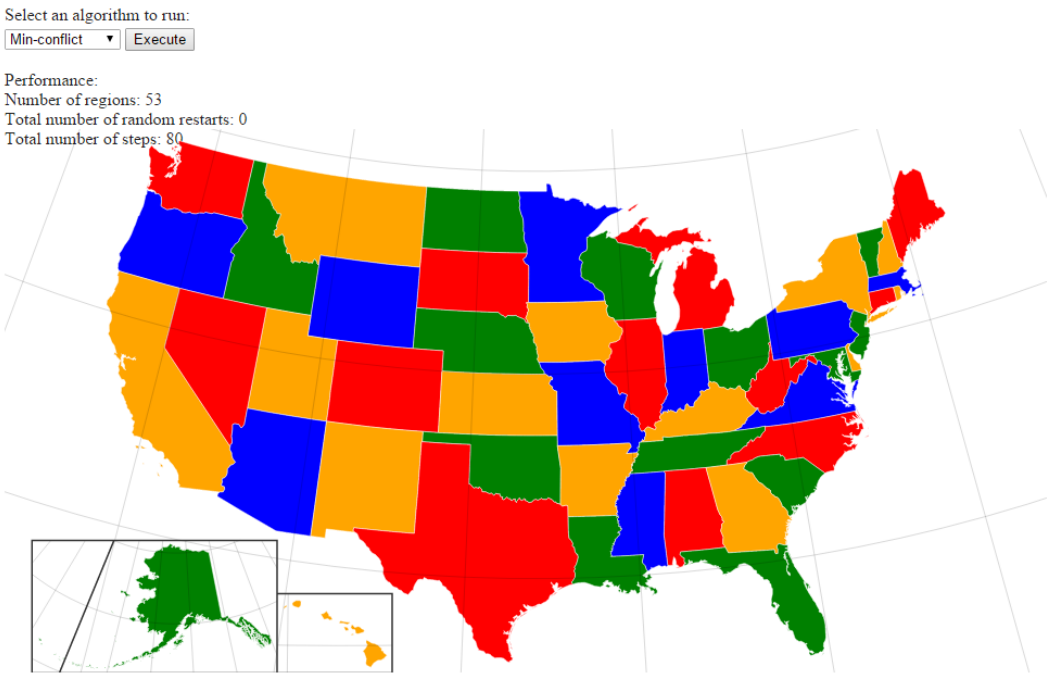
2. World map using local search with Min-Conflict heuristic:



3. USA states map using Forward checking with MRV heuristic:



4. USA states map using local search with Min-Conflict heuristic:



Future scope:

1. Animating how algorithm makes color assignments during runtime.

References:

1. Text book: Artificial Intelligence A Modern Approach, *Stuart Russell and Peter Norvig*
2. https://en.wikipedia.org/wiki/Constraint_satisfaction_problem
3. http://d3js.org/
4. https://en.wikipedia.org/wiki/GeoJSON#TopoJSON

Source code:

1. **Index.html:**

```
<!doctype html>
<html class="no-js" lang="">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <title>Map coloring</title>
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="apple-touch-icon" href="apple-touch-icon.png">
        <!-- Place favicon.ico in the root directory -->
        <link rel="stylesheet" href="css/normalize.css">
        <link rel="stylesheet" href="css/main.css">
        <script src="js/vendor/modernizr-2.8.3.min.js"></script>
        <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
        <script src="http://d3js.org/d3.geo.projection.v0.min.js"></script>
        <script src="http://d3js.org/topojson.v1.min.js"></script>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
        <style>
            path {
                stroke: white;
                stroke-width: 0.5px;
                fill: black;
            }

            .country {
                fill: #ccc;
                stroke: #fff;
                stroke-width: .5px;
                stroke-linejoin: round;
            }

            .graticule {
                fill: none;
                stroke: #000;
                stroke-opacity: .3;
                stroke-width: .5px;
            }

            .graticule.outline {
                stroke: #333;
                stroke-opacity: 1;
                stroke-width: 1.5px;
            }

        </style>
        <script>
            $(document).ready(function(){
                displayMap();

                $("#executeBtn").click(function(){
                    loadData();

                });
            });
        </script>
    </head>
    <body>
```

```
        <!-- Add your site or application content here -->

        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/{{JQUERY_VERSION}}/jquery.min.js"></script>
        <script>window.jQuery || document.write('<script src="js/vendor/jquery-
{{JQUERY_VERSION}}.min.js"><\/script>')</script>
        <script src="js/plugins.js"></script>
        <script src="js/main.js"></script>
        <script src="js/MRVWithForwardChecking.js"></script>
        <script src="js/MinConflict.js"></script>

        <div>
            <p style=margin-bottom:0px>Select an algorithm to run: </p>
            <select id = "algorithm">
                <option value = "1">FC with MRV</option>
                <option value = "2">Min-conflict</option>
            </select>
            <button id = "executeBtn">Execute</button>
        </div>
        <div>
            <p style=margin-bottom:-20px;>Performance: </p>
            <p id="numRegions" style=margin-bottom:-20px;></p>
            <p id="kpi1" style=margin-bottom:-20px></p>
            <p id="kpi2" style=margin-bottom:-20px></p>
        </div>

        <script>
            var svg, path,projection;
            var countries, neighbours;
            var coloredMap = [];

            function loadData(){
                d3.json("world-50m.json", function(error, world) {
                    countries = topojson.feature(world,   world.objects.countries).features,
                        neighbours = topojson.neighbors(world.objects.countries.geometries);

                    var option = $("#algorithm").val();
                    runAlgorithm(option);

                });
//                d3.json("us-10m.json", function(error, usa) {
//                    countries = topojson.feature(usa, usa.objects.states).features,
//                        neighbours = topojson.neighbors(usa.objects.states.geometries);
//                    var option = $("#algorithm").val();
//                    runAlgorithm(option);
//                });

//                d3.json("australia.json", function(error, aus) {
//                     countries = topojson.feature(aus, aus.objects.states).features,
//                        neighbours = topojson.neighbors(aus.objects.states.geometries);
//                    var option = $("#algorithm").val();
//                    runAlgorithm(option);
//                });
//
//                d3.json("eu.topo.json", function(error, europe) {
//                    countries = topojson.feature(europe, europe.objects.europe).features,
//                        neighbours = topojson.neighbors(europe.objects.europe.geometries);
//                    var option = $("#algorithm").val();
//                    runAlgorithm(option);
//                });
//
//                d3.json("africa.topo.json", function(error, africa) {
//                    countries = topojson.feature(africa, africa.objects.collection).features,
//                        neighbours = topojson.neighbors(africa.objects.collection.geometries);
//                    var option = $("#algorithm").val();
//                    runAlgorithm(option);
//                });

            }
```

```
function displayMap() {
    var width = 960,
        height = 500;

    projection = d3.geo.albersUsa(),
    projection = d3.geo.naturalEarth(),
        color = d3.scale.category20(),
        graticule = d3.geo.graticule();

    path = d3.geo.path()
        .projection(projection);

    svg = d3.select("body").append("svg")
        .attr("width", width)
        .attr("height", height);

    svg.append("path")
        .datum(graticule)
        .attr("class", "graticule")
        .attr("d", path);

    svg.append("path")
        .datum(graticule.outline)
        .attr("class", "graticule outline")
        .attr("d", path);
}


function tempColor(d, i) {
    return coloredMap[i].color;
}

function runAlgorithm(selAlgo){
        coloredMap = [];
        if(selAlgo == 1){
            coloredMap = colorMapWithFCMRV(this.countries, this.neighbours);
        }
        else{
            coloredMap = colorMapWithMinConflicts(this.countries, this.neighbours);
        }

        var countries = svg.selectAll(".country")
            .data(this.countries)
            .enter().insert("path", ".graticule")
            .attr("class", "country")
            .attr("d", path);

            countries.style("fill", function (d, i) {
                return tempColor(d, i);
            });
            updatePerformanceParameters(selAlgo);
}
```

```
            function updatePerformanceParameters(selAlgo){
                document.getElementById("numRegions").innerHTML ="Number of regions: " +
countries.length;
                if(selAlgo == 1) {
                    document.getElementById("kpi1").innerHTML = "Number of dead end encoutered: "
+ numOfDeadEnds;
                    document.getElementById("kpi2").innerHTML = "Total number of recursive calls
made: " + numOfRecursiveCalls;
                    numOfDeadEnds = 0; numOfRecursiveCalls = 0;
                }
                else{
                    document.getElementById("kpi1").innerHTML = "Total number of random restarts:
" + numOfRestarts;
                    document.getElementById("kpi2").innerHTML = "Total number of steps: " +
numOfTotalSteps;
                    numOfRestarts = 0; numOfTotalSteps = 0;
                }

            }

        </script>

    </body>
</html>
```

2. MRVWithForwardChecking.js:

```
/**
 * Created by Abhishek on 12/1/2015.
 */
//Stores list of contraint graph nodes (in this case node of state/country)
var mapNodes = [];

//Stores list of remaining unassiagned variable
var listOfRemNode = [];

//Array that stores neighbours for each graph node (adjacency list)
var neighbours = [];

//Stores list of countries / states from json file
var countries = [];

var numOfDeadEnds = 0;
var numOfRecursiveCalls = 0;

//Class of constraint graph node
function MapNode (mapArrayIndex) {
    this.mapArrayIndex = mapArrayIndex;
    this.color = "Black"; // Default color
    this.legalColors = ["Red", "Green", "Blue", "Orange"]; //Allowed colors
    this.legalColors.sort();
    this.isColorSet =false;
    this.neighbours = neighbours[mapArrayIndex]; //neighbour of this node
}

//Returns color of the node
MapNode.prototype.getColor = function(){
    return this.color;
}

//Sets color to the graph(map) node
MapNode.prototype.setMapNodeColor = function(i){
    this.isColorSet = true;
    this.color = this.legalColors[i];
    /*As soon as color is assinged to a node do forward checking
    i.e. remove currently used color from domain of neighobouring
    variables*/
    this.forwardChecking();
}

/* Unset node color if dead end is encountered in
in search due to domain wipeout*/
MapNode.prototype.unSetMapNodeColor = function(){
    this.isColorSet = false;
    /*Add currently unset domain value for neighboring nodes as
    neighbouring nodes can have recently unset value*/
    this.restoreNeighbours();
    this.color = "Black";
}

/*Adds currently unset domain value for neighboring nodes as
 neighbouring nodes can have recently unset value*/
MapNode.prototype.restoreNeighbours = function() {
    for (var i = 0; i < this.neighbours.length; i++) {
        var neighbour = mapNodes[this.neighbours[i]];
        if (neighbour.isColorSet == false) {
            var index =neighbour.legalColors.indexOf(this.color)
            if(index == -1)
            {
                neighbour.legalColors.push(this.color);
                neighbour.legalColors.sort();
            }
        }
    }
}
```

```
/*Updates the domain value for neighbouring nodes*/
MapNode.prototype.forwardChecking = function(){
    for(var i = 0; i<this.neighbours.length; i++){
        var neighbour = mapNodes[this.neighbours[i]];
        if(neighbour.isColorSet == false)
        {
            neighbour.deleteValueFromDomain(this.color);
        }
    }
}

/*Deletes input domain value from domain*/
MapNode.prototype.deleteValueFromDomain = function(color){
    var index = this.legalColors.indexOf(color);
    if(index >= 0)
    {
        this.legalColors.splice(index, 1);
    }

}

/*Checks if assignments satisfies constraint that no two
neighbouring nodes can have same color,
returns true if constraints are satisfied*/
MapNode.prototype.satisfyConstraint = function(color){
    for(var i = 0; i<this.neighbours.length; i++){
        var neighbour = mapNodes[this.neighbours[i]];
        if(neighbour.isColorSet == true)
        {
            if(neighbour.color == color)
            {
                return false;
            }
        }
    }
    return true;
}

/*Iterates through list of countries/States and creates an
array of objects of custom class MapNode*/
function  createCountryNode(){
    for(var i = 0; i < countries.length; i++){
        mapNodes[i]  = new MapNode(i);
        listOfRemNode[i] = mapNodes[i];
    }
}

/*Main function called from HTML file*/
function colorMapWithFCMRV(iCountries, iNeighbours){
    countries = iCountries;
    neighbours = iNeighbours;

    createCountryNode();

    //Does Minimum remaining value using forward checking
    fcMRV();
    console.log("Number of dead-ends encountered: " + numOfDeadEnds);
    console.log("Number of recursrive calls: " + numOfRecursiveCalls);
    return mapNodes;

}
```

```
/*Recursive function that solves CSP using minimum remaining value
and forward checking*/
function fcMRV() {

    numOfRecursiveCalls++;
    if(listOfRemNode.length <=0){
        return mapNodes;
    }

    //nextNode is null if all variables are assigned
    var nextNode = pickUnassignedVariable();
    if(nextNode == null){
        return null;
    }

    //Domain wipeout -- Algorithm backtracks
    if(nextNode.legalColors.length <=0){
        numOfDeadEnds++;
        return null;
    }

    //Loop to try all domain values
    for(var i = 0; i<nextNode.legalColors.length; i++){
        var consSatisfied =nextNode.satisfyConstraint(nextNode.legalColors[i]);
        if(consSatisfied == true) {
            nextNode.setMapNodeColor(i);
            removeColoredNodeFromList(nextNode);
            var result = fcMRV();
            if (result == null) {
                /*Since that result is null, means current assignments creates inconsistent
                assignments. Unset current assignment. Try next next domain value*/
                nextNode.unSetMapNodeColor();
                listOfRemNode.push(nextNode);
            }
            else{
                return mapNodes; /*If result is not null meaning that we have found a solution*/
            }
        }
    }
    return null;
}

/*Selects the unassigned variables with minimum remaining value.
Uses degree heuristic as a tie breaker.*/
function pickUnassignedVariable(){
    var mrvNodes = [];
    var currentMRV = 4;
    for(i = 0; i < listOfRemNode.length; i++) {
        var currentMapMode = listOfRemNode[i];
        if(currentMapMode.legalColors.length < currentMRV) {
            if(currentMapMode.isColorSet == false) {
                currentMRV = currentMapMode.legalColors.length;
                while (mrvNodes.length) {
                    mrvNodes.pop();
                }
                mrvNodes.push(currentMapMode);
            }
        }
        else if(currentMapMode.legalColors.length == currentMRV){
            if(currentMapMode.isColorSet == false){
                mrvNodes.push(currentMapMode);
            }
        }
    }

    return getNodeWithHighestDegree(mrvNodes);
}
```

```
/*Returns the node with highest degree i.e.
a node with highest number of neighbours*/
function getNodeWithHighestDegree(listOfNodes){

    if(listOfNodes.length <= 0)
        return null;

    var indCountriesWithMaxNeighbour = 0;
    for(i = 0; i < listOfNodes.length; i++){
        if(getNumOfUnAssignedNeighbours(listOfNodes[indCountriesWithMaxNeighbour])<
getNumOfUnAssignedNeighbours(listOfNodes[i]))
        {
            indCountriesWithMaxNeighbour = i;
        }
    }
    return listOfNodes[indCountriesWithMaxNeighbour];
}

/*Deletes a node from list of remaining nodes to be colored*/
function removeColoredNodeFromList(node){

    var index = listOfRemNode.indexOf(node);
    if(index >= 0){
        listOfRemNode.splice(index, 1);
    }

}

/*Returns number of unassigned neighboring nodes
of input node*/
function getNumOfUnAssignedNeighbours(node){
    var numNeighbours = 0;
    for(var i = 0; i < node.neighbours.length; i++)
    {
        var neighbourNode = mapNodes[node.neighbours[i]];
        //if(neighbourNode.isColorSet == false){
            numNeighbours++;
        //}
    }
    return numNeighbours;
}
```

3. MinConflict.js:

```
/**
 * Created by Abhishek on 12/4/2015.
 */
//Stores list of constraint graph nodes (in this case node of state/country)
var mapNodesMC = [];

//Stores list of remaining unassigned variable
var listOfRemNode = [];

//Array that stores neighbours for each graph node (adjacency list)
var neighbours = [];

//Stores list of countries / states from json file
var countries = [];

var connectedComponents = [];

var numOfRestarts = 0;
var numOfTotalSteps = 0;

//Class of constraint graph node
function McMapNode(mapArrayIndex) {
    this.isVisited = false;
    this.mapArrayIndex = mapArrayIndex;
    this.color = "Black"; // Default color
    this.legalColors = ["Red", "Green", "Blue", "Orange"]; //Allowed colors
    this.legalColors.sort();
    this.isColorSet =false;
    this.neighbours = neighbours[mapArrayIndex]; //neighbour of this node
}

/*Assigns color to node randomly from domain*/
McMapNode.prototype.assignRandomColor = function(colors){
    var i = Math.floor((Math.random() * (colors.length)) + 0);
    this.color = colors[i];
}

/*Helper method to set color*/
McMapNode.prototype.setColor = function(color){
    this.color = color;
}

/*Assigns random colors to all the nodes in the input list
Used to do random restart if search gets stuck in local minima*/
function randomStart(listOfNodes){
    var node;
    for (var i= 0; i < listOfNodes.length; i++){
        var node = listOfNodes[i];
        node.assignRandomColor(node.legalColors);
    }
}


/*Iterates through list of countries/States and creates an
 array of objects of custom class McMapNode*/
function  createCountryNodeMC(){
    for(var i = 0; i < countries.length; i++){
        mapNodesMC[i]  = new McMapNode(i);
    }
}
```

```
/*Main function called from HTML*/
function colorMapWithMinConflicts(iCountries, iNeighbours){
    countries = iCountries;
    neighbours = iNeighbours;
    createCountryNodeMC();
    randomStart(mapNodesMC);
    getConnectedComponents();

    for (var i = 0; i < connectedComponents.length; i++){
        var component = connectedComponents[i];
        var result = solveCSPWithMinConflict(component);
        if(result == false) {
            break;
        }
    }
    console.log("Number of total steps of by algorithm: " + numOfTotalSteps);
    console.log("Number of random restarts required by algorithm: " + numOfRestarts);

    return mapNodesMC;
}

/*Gets connected component of constraint graph*/
function getConnectedComponents(){
    for(var i = 0; i< mapNodesMC.length; i++){
        var node = mapNodesMC[i];
        if(node.isVisited == false){
            var listOfNodes = [];
            traverseDepth(node, listOfNodes);
            connectedComponents.push(listOfNodes);
        }
    }
}

/*recursive function to do topological sorting*/
function traverseDepth(node,listOfNodes){
    node.isVisited = true;
    listOfNodes.push(node);
    for(var i = 0; i < node.neighbours.length; i++){
        var neighbour = mapNodesMC[node.neighbours[i]];
        if(neighbour.isVisited == false){
            //neighbour.isVisited = true;
            traverseDepth(neighbour,listOfNodes);
        }
    }
}

/*solves CSP for each component*/
function solveCSPWithMinConflict(component){
    var totalConflicts = 0;
    var restartAfter = component.length * 50;
    for(var i = 1; i <= 1000000; i++){
        numOfTotalSteps++;
        if(i % restartAfter == 0){
            numOfRestarts++;
            randomStart(component);
        }
        totalConflicts = getNumOfConflictsOfComponent(component);
        /*If total conflicts are 0, meaning solution is found*/
        if(totalConflicts == 0){
            console.log("Steps required for component of size :" + component.length + " is " +
i);
            return true;
        }
        var node = pickConflictedVariable(component);
        var color = selectValWithMinConflicts(node);
        node.setColor(color);
    }

    return false;
}
```

```
/*Returns total number conflicts for input component*/
function getNumOfConflictsOfComponent(component){
    var totalConflicts = 0;
    for(var i = 0; i < component.length; i++){
        totalConflicts += getConflictOfNode(component[i]);
    }
    return totalConflicts;
}


function getConflictOfNode(node){
    var numOfConflicts = 0;
    for(var i = 0; i<node.neighbours.length; i++){
        var neighbour = mapNodesMC[node.neighbours[i]];
        if(neighbour.color == node.color)
        {
            numOfConflicts++;
        }
    }
    return numOfConflicts;
}

/*Randomly picks conflicted*/
function pickConflictedVariable(component){

    var maxConflict = 0;
    var maxConflictVar = null;

    while (maxConflictVar == null){
        var i = Math.floor((Math.random() * (component.length)) + 0);
        var totalConflicts = getConflictOfNode(component[i]);
        if(totalConflicts > 0){
                maxConflictVar = component[i];
        }
    }
    return maxConflictVar;
}

/*Function selects value from the domain which
minimizes total number of conflict with neighvoring nodes*/
function selectValWithMinConflicts(node){
    var currentColor = node.color;
    var minConflicts = node.neighbours.length;
    for(var i = 0; i < node.legalColors.length; i++){
        if(node.legalColors[i] == node.color){
            continue;
        }
        var numOfConflicts = 0;
        for(var j = 0; j<node.neighbours.length; j++){
            var neighbour = mapNodesMC[node.neighbours[j]];
            if(neighbour.color == node.legalColors[i])
            {
                numOfConflicts++;
            }
        }
        if(numOfConflicts <= minConflicts){
            minConflicts = numOfConflicts;
            currentColor = node.legalColors[i];
        }
    }
    return currentColor;
}
```