University of North Carolina Charlotte

Department of Computer Science

College of Computing and Informatics

Project Title: Map coloring

Subject: Intelligent Systems

Term: Fall 2017

By Dhwani Contractor

Guided by Dr. Jing Xiao

Index

Problem Defination

Note that you are to apply both forward checking with MRV and Min-Conflicts local search algorithms to solve the n-region, at most 4 color, map coloring problem.
You need to construct the performance table comparing the two algorithms for the largest n's you can manage. Discuss the results.

Constraint Satisfaction Problem:

A constraint satisfaction problem consists of three components, X,D, and C:
X is a set of variables, {X1, . . . ,Xn}.
D is a set of domains, {D1, . . . ,Dn}, one for each variable.
C is a set of constraints that specify allowable combinations of values.
Each domain Di consists of a set of allowable values, {v1, . . . , vk} for variable Xi. Each constraint Ci consists of a pair <scope, rel >, where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

Map coloring:

We are given the task of colouring each region red, green, or blue in such a way that no neighbouring regions have the same colour. To formulate this as a CSP, we define the variables to be the regions X = {WA, NT, Q, NSW, V, SA, T}.

The domain of each variable is the set Di = {red , green, blue}. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:
C = {SA ≠ WA, SA ≠ NT, SA ≠ Q,SA ≠ NSW, SA ≠ V, WA ≠ NT,NT ≠ Q,Q ≠ NSW, NSW ≠ V } .

Here we are using abbreviations; SA ≠ WA is a shortcut for <(SA,WA),SA _= W>, where SA ≠ WA can be fully enumerated in turn as {(red , green), (red , blue), (green, red), (green, blue), (blue, red), (blue, green)} .

Model

To color any map with n number of region, the map should be practically feasible. In other words, it is not possible that all regions of the map are connected to all regions or most of the regions. For example, in the map of USA, California cannot be connected to New York. California is somehow connected to Oregon, Nevada and Arizona. These 3 states will also be connected to the nearer states of the west side.

In this project, instead of naming each region of the map, all the regions have been numbered from 1, 2, 3… n and number of neighbours have been restricted to 8 means a region can have maximum 8 neighbours. Map is created in such a way that i^{th} number of region can have neighbours from which has region-number from i-4 to i+4.

In this project, an n x n matrix is created which indicates the map. For each row i, if j^{th} element is 1 it means j^{th} element is neighbour of i^{th} element.

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

This matrix indicates:
Neighbours of 1→3, 4
Neighbours of 2→5, 6
Neighbours of 3→1, 4, 6, 7
Neighbours of 4→1, 3, 5, 7
Neighbours of 5→2, 4, 7, 8
Neighbours of 6→2, 3, 7, 9, 10
Neighbours of 7→3, 4, 5, 6, 8, 9
Neighbours of 8→7, 9, 10
Neighbours of 9→5, 6, 7, 8, 10
Neighbours of 10→6, 8, 9

Approach & Method

Forward checking

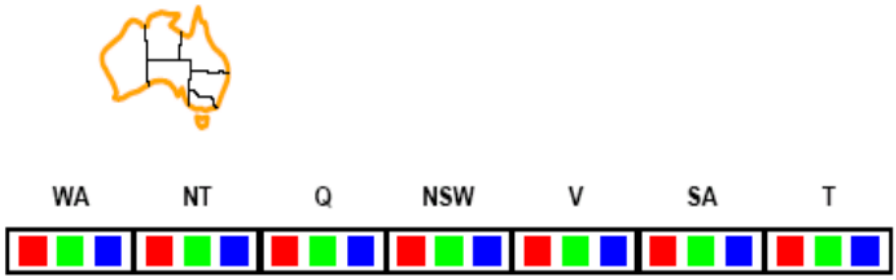
Forward checking algorithm keeps track of remaining legal values for unassigned variables.
Terminate search direction when a variable has no legal values.

Minimum Remaining Value:

Minimum Remaining Value Heuristic function chooses the variable with the fewest legal values.
In other words, it chooses the most constraint region to be color first.

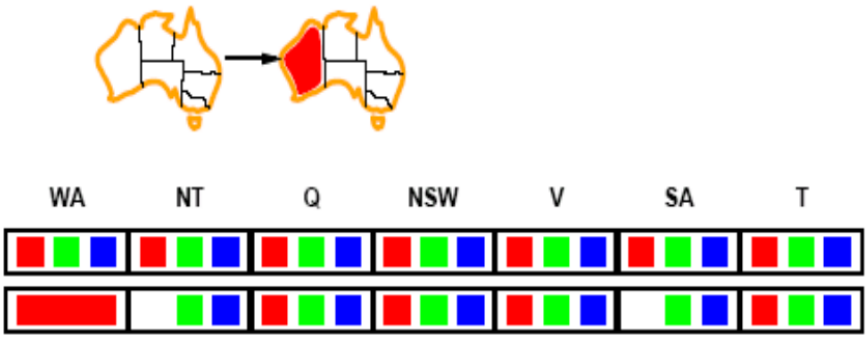
Step: 1

- All the colors are legal for each region in the Map of Australia.



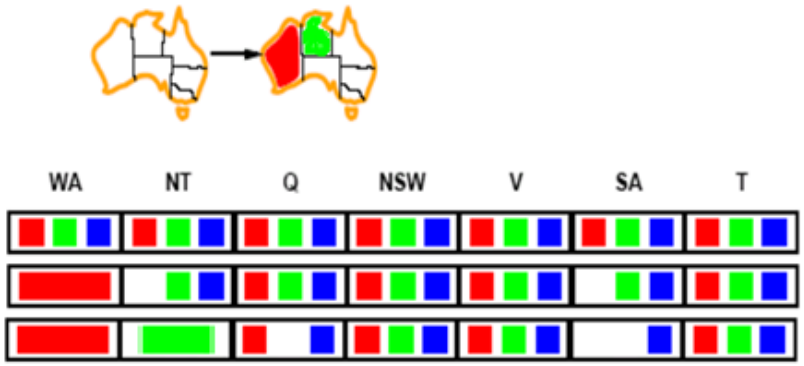
Step: 2

- Assign WA = Red then NT & SA can no longer be Red as they are neighbours of WA.



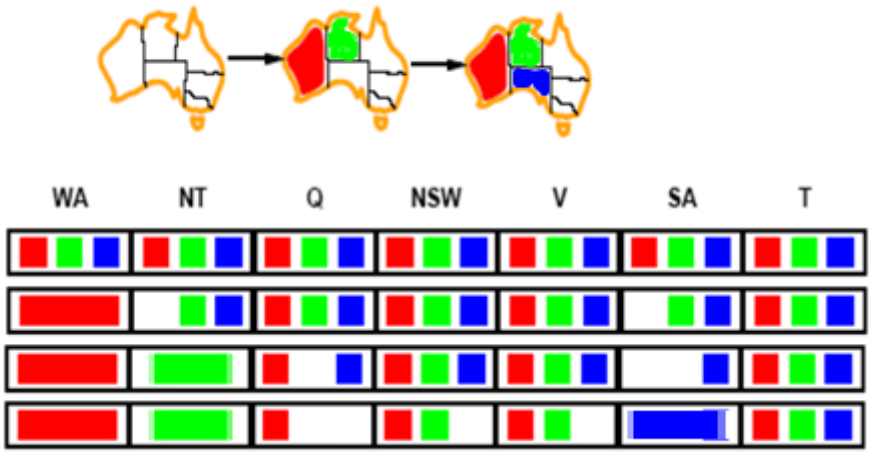
Step: 3

- Assign NT= Green as it has **Minimum Remaining Value** (MRV=2) from the colors.
- Now, SA & Q will no longer have Green color.



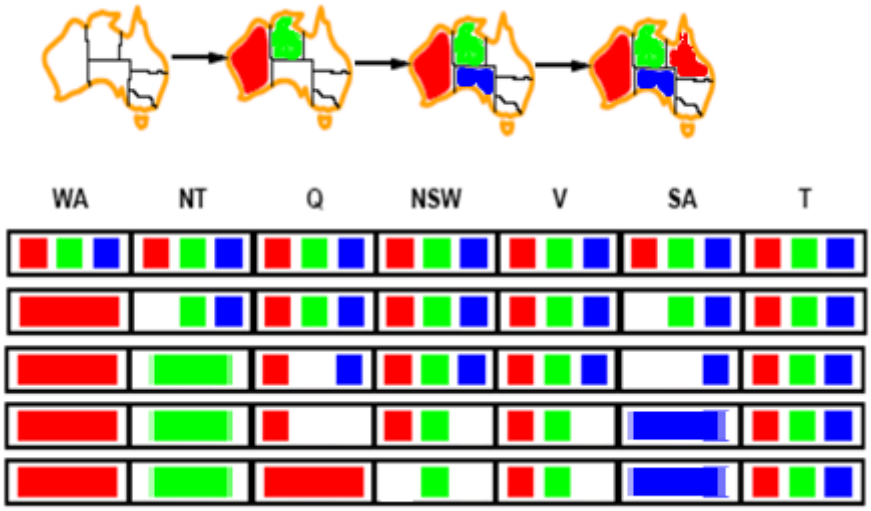
Step: 4

- Assign SA= Blue as it has **Minimum Remaining Value** (MRV=1) from the colors.
- Now, NSW, V & Q will no longer have Blue color.



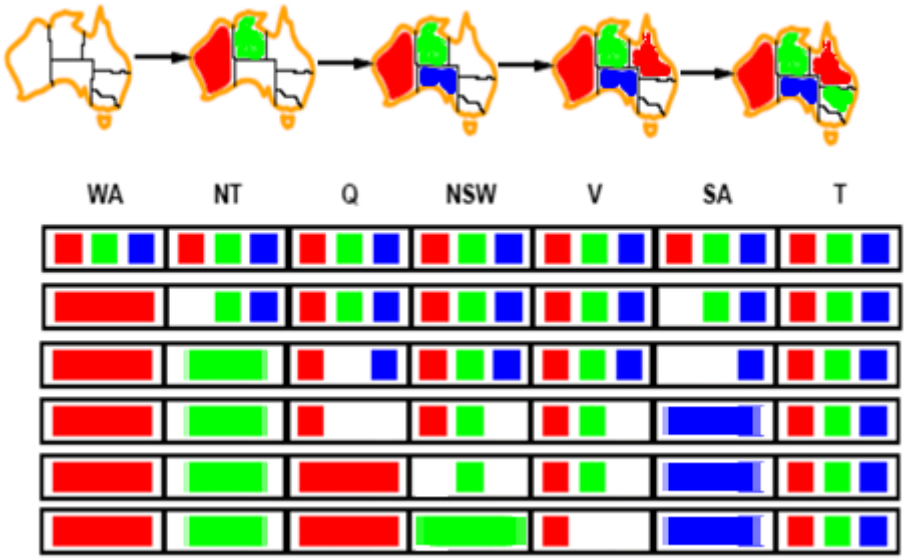
Step: 5

- Assign Q= Red as it has **Minimum Remaining Value** (MRV=1) from the colors.
- Now, NSW will no longer have Red color.



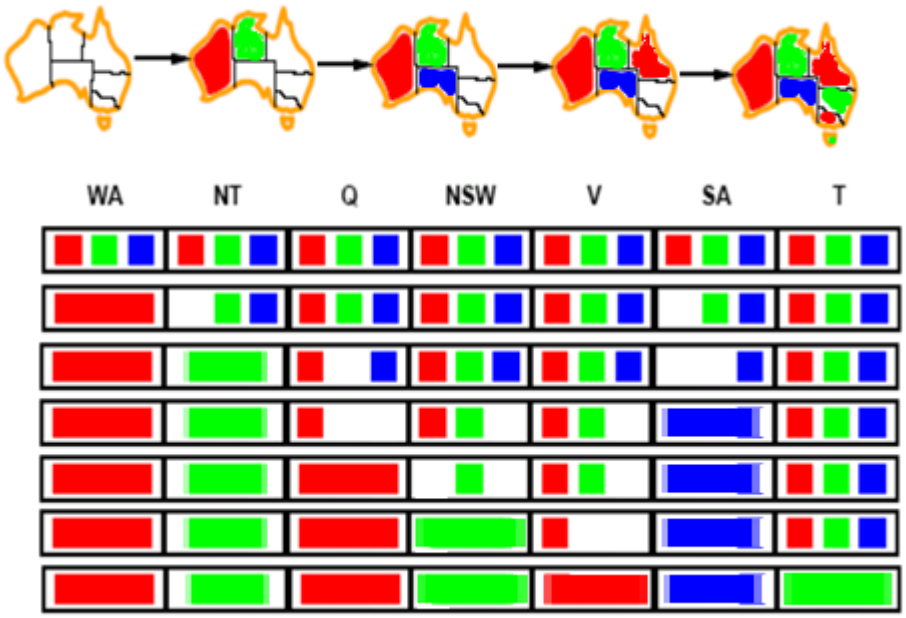
Step: 6

- Assign NSW= Green as it has **Minimum Remaining Value** (MRV=1) from the colors.
- Now, V will no longer have Green color.



Step: 7

- Assign V= Red as it has **Minimum Remaining Value** (MRV=1) from the colors.
- Assign any of the color to T as it has no constraint means all values of colors are possible for T.



Min-Conflict Algorithm

The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for $i = 1$ to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set $var = value$ in $current$
  return *failure*

Hill climbing is used as Local search in this project.

Random restart occurs when the algorithm reaches its Local max.

Performance Measurement

For n=10

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.007	0.02
0.013	0.064
0.009	0.025

For n=20

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.039	0.049
0.073	0.252
0.048	0.069

For n=30

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.072	0.378
0.147	0.389
0.065	0.156

For n=40

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.139	0.212
0.109	0.302
0.113	0.286
0.102	0.517

For n=50

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.119	0.885
0.148	0.392
0.136	0.435
0.16	4.153

For n=60

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.136	0.731
0.157	1.866
0.152	3.231
0.182	5.624

For n=70

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.134	2.336
0.166	Reached Max Steps for random restart
0.152	0.829
0.161	9.154
0.16	3.147

For n=80

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.192	16.864
0.227	10.084
0.176	13.854
0.198	1.406

For n=90

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.217	12.888
0.212	2.361
0.215	10.972
0.234	10.545
0.246	39.587

For n=100

Time taken by Forward Checking with MRV (second)	Time taken by Hill-Climbing with Minimum Conflict (second)
0.278	15.312
0.241	Reached Max Steps for random restart
0.234	40.015
0.243	22.383
0.247	5.321

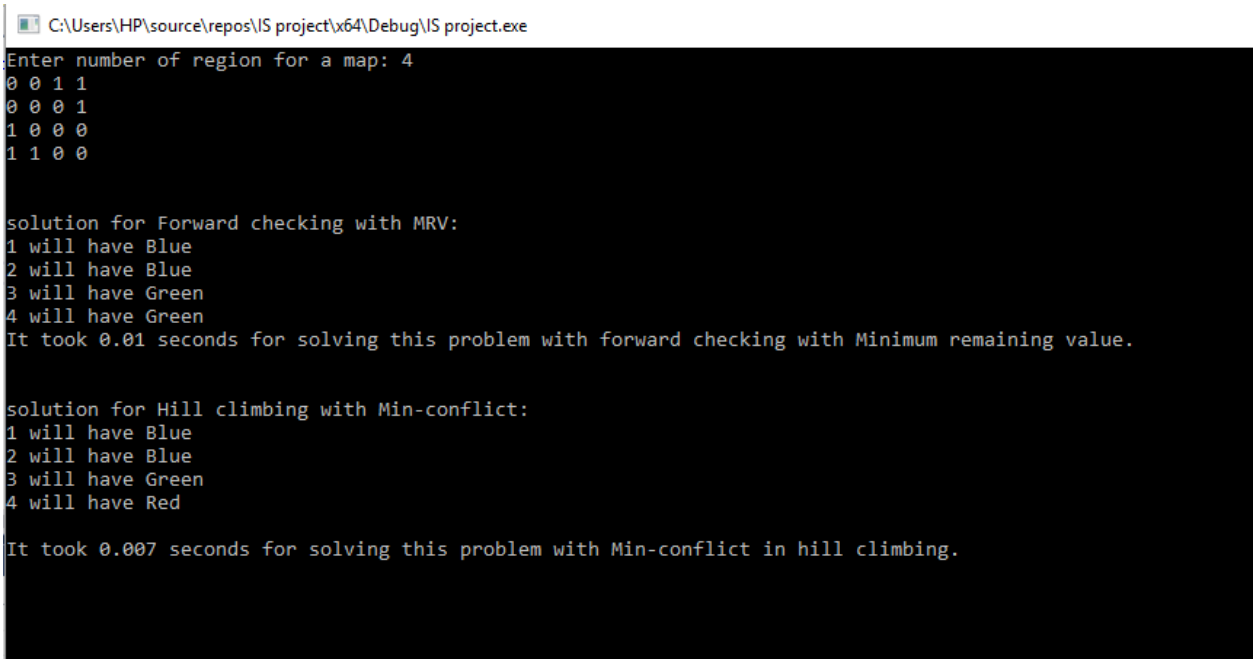
No of Region	Average Time taken by Forward Checking with MRV (second)	Average Time taken by Hill-Climbing with Minimum Conflict (second)
10	0.00967	0.0363
20	0.0533	0.1233
30	0.09467	0.307
40	0.11575	0.32925
50	0.14075	1.173
60	0.156	2.863
70	0.15175	3.8665
80	0.19825	10.5525
90	0.2248	15.2746
100	0.2486	20.75775

Note: Maximum random restarts allowed for these readings are 20.

Examples for Testing

The code randomly generates the feasible map. Following are the examples and screenshots of the results.

For n=4



For n=15

Enter number of region for a map: 15

0 0 1 1 0 0 0 0 0 0 0 0 0 0 0

0 0 1 0 1 1 0 0 0 0 0 0 0 0 0

1 1 0 1 0 0 1 0 0 0 0 0 0 0 0

1 0 1 0 0 0 1 1 0 0 0 0 0 0 0

0 1 0 0 0 1 1 1 0 0 0 0 0 0 0

0 1 0 0 1 0 1 0 0 0 0 0 0 0 0

0 0 1 1 1 1 0 1 1 0 1 0 0 0 0

0 0 0 1 1 0 1 0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 1 0 1 1 1 0 0 0

0 0 0 0 0 0 0 0 1 0 0 1 0 1 0

0 0 0 0 0 0 1 0 1 0 0 0 1 0 0

0 0 0 0 0 0 0 0 1 1 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 1 0 0 1 1

0 0 0 0 0 0 0 0 0 1 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 0 1 1 0 0

solution for Forward checking with MRV:

1 will have Blue

2 will have Blue

3 will have Green

4 will have Red

5 will have Red

6 will have Green

7 will have Blue

8 will have Green

9 will have Red

10 will have Blue

11 will have Green

12 will have Green

13 will have Blue

14 will have Green

15 will have Red

It took 0.035 seconds for solving this problem with forward checking with Minimum remaining value.

solution for Hill climbing with Min-conflict:

1 will have Blue

2 will have Blue

3 will have Yellow

4 will have Red

5 will have Red

6 will have Yellow

7 will have Green

8 will have Blue

9 will have Red

10 will have Blue

11 will have Yellow

12 will have Yellow

13 will have Blue

14 will have Red

15 will have Green

It took 0.076 seconds for solving this problem with Min-conflict in hill climbing.

A map is solvable for any number of n.
The program doesn't give the solution for the following two situations

1. If the map is practically not possible.
2. If the map is feasible but the hill-climbing algorithm's random restarts have exceeded the set value.

Outline of the work performed

From the Performance Measurement, forward checking algorithm with Minimum Remaining Value (MRV) is much better than Hill climbing algorithm with min-conflict because of following two reasons:

1. For any n, time for taken by forward checking by each time is almost equal to the average of its all readings. While Min-conflict algorithm is unpredictable because of the random restarts. If the randomly generated initial state is closer to the solution then it will give output in lesser time but if it is not close than it may end up having many random restarts.
2. As the n increases, the time taken by Forward checking does not increases drastically as Min-conflict.

Code

```
#include<iostream>
#include<vector>
#include<time.h>
#include<map>
#include<string>
#define WITHOUT_COLOR "NoColor"


using namespace std;

int c = 0;
int maxstep;

clock_t t_fc, t_hill;

//Generates random number within 0 and given limit
int getRandom(int limit) {
	return rand() % limit;
}
class Region {
public:
	int number;
	string color;
	vector<int> neighbourRegion;
	map<string, bool> possibleColors;
	//Initialization of Forward Checking
	void initFC(int i) {
		number = i + 1;
		color = WITHOUT_COLOR; // The color assigned to the region, initially colorless
                                                        //Initially all colors are available for
this region
		possibleColors["Blue"] = true;
		possibleColors["Yellow"] = true;
		possibleColors["Green"] = true;
		possibleColors["Red"] = true;

	}
	//Initialization of Min-Conflict Algo
	void initMinConflict(int i) {
		number = i + 1;
		gencolor(getRandom(4));
	}
	//Assigns color to the region
	void gencolor(int cnum) {
		if (cnum == 0)
			color = "Blue";
		else if (cnum == 1)
			color = "Yellow";
		else if (cnum == 2)
			color = "Green";
		else if (cnum == 3)
			color = "Red";
	}
	//Assigns neighbours from the generated map.
	void genNeighbour(int i, int size, int** Map) {

		for (int j = 0; j < size; j++) {
			if (Map[i][j] == 1) {
				neighbourRegion.push_back(j + 1);
			}
		}
	}
	//Returns the Region-number of neighbours
	vector<int> getNeighbors() {
```

```
			return neighbourRegion;
		}
		//Checks if the region has color or not
		bool hasColor() {
			if (color == WITHOUT_COLOR)
				return false;
			return true;
		}
		//Updates colors of neighbour region in MRV
		void updatePossibleColors(bool flag, string colorName) {
			possibleColors[colorName] = flag;
		}

		//Returns to quantity of available cores for this region
		int getPossibleColors() {
			int cont = 0;
			for (map<string, bool>::iterator it = possibleColors.begin(); it !=
possibleColors.end(); ++it) {
				if (it->second)
					cont += 1;
			}
			return cont;
		}
		//Returns to name of available colors for this region
		vector<string> getPossibleColorsname() {
			vector<string> col;
			for (map<string, bool>::iterator it = possibleColors.begin(); it !=
possibleColors.end(); ++it) {
				if (it->second)
					col.push_back(it->first);
			}
			return col;
		}

};

class Map {
public:
		vector<Region> regionList;
		vector<string> colors;
		//Initialization of colors
		void MapcolorInt() {
			colors.push_back("Blue");
			colors.push_back("Yellow");
			colors.push_back("Green");
			colors.push_back("Red");
		}
		//Counts the number of conflicts in the given map
		int getHeuristic(int** M) {
			int heuristic = 0;
			for (int i = 0; i < regionList.size(); i++) {
				for (int j = 0; j <= i; j++) {
					if (M[i][j] == 1 && regionList.at(i).color ==
regionList.at(j).color) {
						heuristic++;
						//cout << i + 1 << " & " << j +1<< " have same color";
					}

				}
				//cout << "\n";
			}
			return heuristic;
		}
		//Prints the solution for the given map
		void printSolution() {
			for (int i = 0; i < regionList.size(); i++) {
```

```
                    cout << regionList.at(i).number << " will have " <<
regionList.at(i).color << "\n";
                }
        }
        //Chooses the region with the fewest possible colors
        int selectVariableMVR() {
                int mrv = colors.size() + 1;
                int variable = 0;
                for (int i = 0; i < regionList.size(); i++) {
                        if (!regionList[i].hasColor()) {
                                if (regionList[i].getPossibleColors() < mrv) {
                                        variable = i;
                                        mrv = regionList[i].getPossibleColors();
                                }
                        }
                }
                return variable;
        }
        void hillclimbiing(int** M) {
                //Get the conflicts for the given map with the colors assigned to it
                        int H = getHeuristic(M);
                        //If there is no conflicts then print the solution
                        if (H == 0) {
                                cout << "\n\nsolution for Hill climbing with Min-conflict: \n";
                                printSolution();
                                t_hill = clock() - t_hill;
                                cout << "\nIt took " << (float)t_hill / CLOCKS_PER_SEC << "
seconds for solving this problem with Min-conflict in hill climbing.\n";
                                exit(0);
                        }
                        else {
                                int nextH = H;
                                for (int i = 0; i < regionList.size(); i++) {
                                        //save the original color to assign it back if the new
Heuristic is bigger than older one.
                                        string originalColor = regionList.at(i).color;
                                        //Check heuristic for all the colors and assign the color
with the least heuristic.
                                        for (int j = 0; j < colors.size(); j++) {
                                                if (colors.at(j) != originalColor) {
                                                        regionList.at(i).color = colors.at(j);
                                                        int newH = getHeuristic(M);
                                                        if (newH < nextH) {
                                                                nextH = newH;
                                                                hillclimbiing(M);
                                                        }
                                                }
                                        }
                                        regionList.at(i).color = originalColor;
                                }
                                H = getHeuristic(M);
                                //If no such combination found for the values assigned to the map
then local max has been reached.
                                if (H != 0) {
                                        //Check if number of random restarts has not exceeded the
maxstep.
                                        //This loop make sure that the algorithm doesn't run for
longer period of time.
                                        while (c < maxstep) {
                                        c++;
                                        cout << "\nLocal Max Reached " <<c << " times with
Heuristic " << H;
                                        //
                                        for (int i = 0; i < regionList.size(); i++) {
                                                regionList[i].initMinConflict(i);
```

```
                                        regionList[i].genNeighbour(i, regionList.size(),
M);
                                        }
                                hillclimbiing(M);
                                }
                                //If still the solution is not found than maximum number
of restarts than map can be unsolvable.
                                cout << "\nTry another map because this map is not
solvable";
                                exit(0);
                        }

                }

}

        void ForwardChecking() {
                for (int j = 0; j < regionList.size(); j++) {
                        //Find the region which has MINIMUM REMAINING VALUES
                        int mvr = selectVariableMVR();

                        //Get the possible color names for that region
                        vector<string> possColorsList = regionList[mvr].getPossibleColorsname();

                        //assign one of the possible colors to the region
                        regionList.at(mvr).color = possColorsList[0];

                        //Find the neighbours of the region to whom the color is just assigned.
                        vector<int> neighbor = regionList.at(mvr).getNeighbors();

                        //Remove the assigned color from the possible colors of its neighbours
                        for (int i = 0; i < neighbor.size(); i++) {
                                if (!regionList[neighbor[i] - 1].hasColor()) {
                                        regionList[neighbor[i] - 1].updatePossibleColors(false,
regionList.at(mvr).color);
                                }
                        }
                }
                //Once all the regions are colored then the solution will be printed
                cout << "\n\nsolution for Forward checking with MRV: \n";
                printSolution();
        }
};

//To print the map
void printMap(int** M, int n) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j< n; j++) {
                        cout << M[i][j] << " ";
                }
                cout << "\n";
        }
}


//Create map function generates a practically feasible map.
int** createMap(int n) {
        int** newMap = new int*[n];
        for (int i = 0; i < n; ++i)
                newMap[i] = new int[n];

        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        if (i == j)newMap[i][j] = 0;
                        else
                        {
```

```
                        newMap[i][j] = -1;
                    }
                }
        }
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (newMap[i][j] == -1) {
                        if (j == i - 4 || j == i - 3 || j == i - 2 || j == i - 1 || j ==
i + 1 || j == i + 2 || j == i + 3 || j == i + 4) {
                            newMap[i][j] = getRandom(2);
                            newMap[j][i] = newMap[i][j];
                        }
                        else
                        {
                            newMap[i][j] = 0;
                        }
                    }

                }
        }

        printMap(newMap, n);
        return newMap;
}


int main()
{
        int n;

        srand(time(NULL));
        cout << "Enter number of region for a map: ";
        cin >> n;
        int** M = new int*[n];
        M = createMap(n);

        //Making 2 different regions because FC and Min-conflict algorithm starts with
different initiallizations.
        vector<Region> Region_fc(n), Region_hill(n);

        //initialization for the regions in FC includes giving numbers to the regions, giving
"NoColor" to the present color of each region
        //making possibility of all color=1 and put the values of neighbors from the map.
        //initialization for the regions Hill climbing with min-conflict includes giving
numbers to the regions, giving random color to each region
        //and put the values of neighbors from the map.

        for (int i = 0; i < n; i++) {
                Region_fc[i].initFC(i);
                Region_fc[i].genNeighbour(i, n, M);
                Region_hill[i].initMinConflict(i);
                Region_hill[i].genNeighbour(i, n, M);
        }

        Map map_fc,map_hill;
        map_fc.regionList = Region_fc;
        //Initialize colors with red, blue, green, yellow
        map_fc.MapcolorInt();
        t_fc = clock();
        //Forward checking function of Map class actually assigns the values to all the
regions.
        map_fc.ForwardChecking();
        t_fc = clock() - t_fc;
        cout<<"It took "<< (float)t_fc / CLOCKS_PER_SEC <<" seconds for solving this problem
with forward checking with Minimum remaining value.\n";
```

```
        map_hill.regionList = Region_hill;
        //Initialize colors with red, blue, green, yellow
        map_hill.MapcolorInt();
        //for maximum 20 random restarts.
        maxstep = 20;
        t_hill = clock();
        //Hill climbing algorithm was used as local search algorithm
        map_hill.hillclimbiing(M);

        return 0;
}
```

External Documentation

Classes:

1. **Region**

→Basically, this class saves the neighbours of the region, color assigned to it and the given number of the region instead on naming them.

→For forward checking it also saves the possible colors.

2. **Map**

→It saves list of all Regions in the map and the possible colors.

Functions:

For Forward checking:

1. getpossiblecolors
   →It will return a vector of the region-numbers of the region.
2. selectVariableMVR
   →This function will find that which region has the Minimum remaining possible colors and will return the region number.
   →Note that this function will not consider the region whose color has already been assigned to it.

3. ForwardChecking
   →This is the main function for Forward checking and it utilizes all the other function to find the solution.
   →Process:
   - Find the region which has MINIMUM REMAINING VALUES.
   - Get the possible color names for that region.
   - Assign one of the possible colors to the region.
   - Find the neighbours of the region to whom the color is just assigned.
   - Remove the assigned color from the possible colors of its neighbours.
   - Once all the regions are colored then the solution will be printed.

For Min-conflict:

1. getHeuristic

→It counts number of conflicts in the map. Here, the number of conflicts means how many pairs of the region is having the same color.

2. HillClimbing

→This is the main function for hillclimbing with minimum conflicts and it utilizes all the other function to find the solution.

→Process:

- Get the conflicts for the given map with the colors assigned to it.
- If there is no conflicts then print the solution.
- Else..
- Save the original color to assign it back if the new Heuristic is bigger than older one.
- Check heuristic for all the colors and assign the color with the least heuristic.
- If no such combination found for the values assigned to the map then local max has been reached.
- Check if number of random restarts has not exceeded the maxstep to make sure that the algorithm doesn't run for longer period of time.
- If random restart occurs maximum times and still the solution is not found than maximum number of restarts than map can be unsolvable.

Future Scope

Somehow the condition for generating the map restricts many possible maps. Hence the intelligence for generating the feasible map can be improved.

References

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: a Modern Approach*. 3rd ed., Prentice-Hall, 2010.