# REPORT

## Project 3: Constraint satisfaction problems (CSP)

## (Map Coloring)

## Submitted By

Jawad Chowdhury, (ID# 801135477)

# Map Coloring Problem Formulation

Here, in this project, we have worked on the implementation of a Map Coloring Problem. To solve this problem, we have used Constraint Satisfaction approach with different strategy and heuristics.

In a general sense, a constraint satisfaction problem is a scenario where we have a set of variables (nodes) and each of them has a certain domain (possible values to assign) and a set of constraints. Our goal is to find out an allowable combination (satisfying the constraint) of variables and with the values.

**Map Coloring Problem**: The map coloring is a problem where there are different areas/regions in a map and taking a bunch of colors we need to color each of these regions in such a way so that no two neighboring regions has the same color.

**Constraint Graph**: we can formulate the map coloring problem in a constraint graph, and as each of the constraint relates only two of the regions (variables), so we have a bunch of binary constraints (edges) to satisfy.

In this project we have used different strategies to solve the problem, such as-

**DFS** – This is the basic strategy, it does not propagate the process for any further pruning. The main check for this strategy is to check whether the current assigned value to the current variable does not violate any past assignments.

**DFS + Forward Check** – This is a strategy which is based on the DFS but here there are additional checks. After assigning the value, according to this strategy, we eliminate any domain value to the unassigned variables that is not consistent with the current assignment.

**DFS + Forward Check + Singleton** – In case of DFS + Forward Check, if any neighbor has a domain with only one value, then that neighbor needs to be propagated further. And in case of the propagation, if we locate the similar case again, the singleton (domain with single value) propagation continues further.

# Program Structure (Variables, Functions-Procedures & Heuristics)

In my implementation, I have used 6 separate scripts to isolate the implementations.
**1.csp_dfs.py** : for basic DFS strategy

**2.csp_dfs_fs.py** : for DFS + Forward Search (Forward Check)

**3.csp_dfs_singleton.py** : for DFS + Forward Search (Forward Check) + propagation through singleton domain.

**4.csp_heuristic_dfs.py** : for basic DFS strategy with heuristics

**5.csp_heuristic_dfs_fs.py** : for DFS + Forward Search (Forward Check) with heuristics

**6.csp_heuristic_dfs_singleton.py** : for DFS + Forward Search (Forward Check) + propagation through singleton domain with heuristics.

I have used mainly 2 global variables, one for the list of vertices and the other one is for the list of constraints. Also, the graph object G is declared as global to have an easier access throughout the implementation.

**Functions-Procedures**: The mentionable functions-procedures that I have used in this implementation are as follows-

**1.check_constraint(n,c)** - to check whether the assignment of color 'c' on node 'n' violates any assignment or not.

**2.get_next_with_heuristic() –** this is an utility function to find out the best next possible variable to use according to the heuristics of MRV and Degree heuristic.

**3.get_color_with_heuristic(n) –** this function returns the best possible color for the node 'n' according to the heuristic of Least Constraining Value.

**4.get_removed_domain(n,c) -** for the assignment of color 'c' to the node 'n', this function eliminates the inconsistent values from the neighboring variables.

**5.color_map(i,n)**- this is actually kind of the run function which executes recursively until it reaches to a base case.

**6.check_map_color()-** this function tries to satisfy the constraint problem using different number of colors starting from 1 to certain max limit. Whenever this function gets a success, we get the chromatic number $\chi(G)$, which is the smallest number of colors needed to color G.

**Heuristics:** Here I have used 3 different heuristic to formulate the preferred order of values and variables.

**MRV & Degree Heuristics:** These 2 heuristics are being used in the function **get_next_with_heuristic().** So among the unassigned variables we order them first according to Minimum Remaining Value heuristic. Then among those ordered variables we use the Degree Heuristic as the second priority. MRV gives us the variables with Minimum possible domain value and Degree Heuristic gives us Variables which are mostly constrained.

**LCV (Least Constraining Value):** I have used this heuristic in the function **get_color_with_heuristic(n).** So the strategy is to pick up the value from the domain which rules out the fewest possible value from the neighboring variables.

# Execution Results (According to the Specification)

The following Table is for **WITHOUT** heuristics –
The cell values are given as (**x(G), #of backtrack, time-duration(seconds)**)

| Map | Strategy(with Heu) | Trial 1 | Trial 2 | Trial 3 | Trial 4 |
|---|---|---|---|---|---|
| Australia | DFS | **(3, 8, 1.33)** | **(3, 5, 1.12)** | **(3, 8, 1.23)** | **(3, 4, 1.13)** |
| | DFS + FC | **(3, 2, 1.11)** | **(3, 2, 1.21)** | **(3, 0, 1.17)** | **(3, 2, 1.20)** |
| | DFS + FC + Singleton | **(3, 0, 1.13)** | **(3, 0, 1.07)** | **(3, 0, 1.20)** | **(3, 0, 1.13)** |
| USA | DFS | **(4, 98, 1.27)** | **(4, 55, 1.11)** | **(4, 61, 1.21)** | **(4, 59, 1.10)** |
| | DFS + FC | **(4, 11, 1.17)** | **(4, 2, 1.10)** | **(4, 8, 1.13)** | **(4, 6, 1.12)** |
| | DFS + FC + Singleton | **(4, 0, 1.46)** | **(4, 0, 1.15)** | **(4, 0, 1.22)** | **(4, 0, 1.11)** |

The following Table is for **WITH** heuristics –
The cell values are given as (**x(G), #of backtrack, time-duration(seconds)**)

| Map | Strategy(without Heu) | Trial 1 | Trial 2 | Trial 3 | Trial 4 |
|---|---|---|---|---|---|
| Australia | DFS | **(3, 7, 1.33)** | **(3, 8, 1.17)** | **(3, 13, 1.18)** | **(3, 8, 1.15)** |
| | DFS + FC | **(3, 0, 1.29)** | **(3, 0, 1.13)** | **(3, 0, 1.18)** | **(3, 0, 1.27)** |
| | DFS + FC + Singleton | **(3, 0, 1.24)** | **(3, 0, 1.15)** | **(3, 0, 1.10)** | **(3, 0, 1.11)** |
| USA | DFS | **(4, 58, 1.79)** | **(4, 61, 2.40)** | **(4, 58, 1.96)** | **(4, 57, 2.15)** |
| | DFS + FC | **(4, 0, 1.24)** | **(4, 0, 1.26)** | **(4, 0, 1.13)** | **(4, 0, 1.24)** |
| | DFS + FC + Singleton | **(4, 0, 1.27)** | **(4, 0, 1.14)** | **(4, 0, 1.25)** | **(4, 0, 1.18)** |

## Source Code

I have 6 separated scripts for different strategies-

Without heuristic Basic DFS,
Without heuristic DFS + Forward Check,
Without heuristic DFS + Forward Check + Singleton,
With heuristic Basic DFS,
With heuristic DFS + Forward Check,
With heuristic DFS + Forward Check + Singleton.

The core part of each of these implementations are same. They only vary on the strategy according to which the algorithm runs. Considering the space, I have put only one script in the report for example.

The implementation of **With heuristic DFS + Forward Check + Singleton** is given below which reflects the source file of "**csp_heuristic_dfs_fs_singleton.py**"

==========

**csp_heuristic_dfs_fs_singleton.py**

```
list_node = []
list_constraint = []
f = open("map_aus.txt", "r") # need to use "map_aus.txt" for australia or
"map_usa.txt" for usa color mapping
lines = f.readlines()
import random
random.shuffle(lines)
lines = [line.replace('\n', '').replace(' ','').replace('[','').replace(']','') for
line in lines]
for line in lines:
    list_elems = line.split(':')
    s = list_elems[0]
    list_d = list_elems[1].split(',')
    if s not in list_node and s != '':
        list_node.append(s)
    for d in list_d:
        if d not in list_node and d != '':
            list_node.append(d)
    for d in list_d:
        if d != s and (s, d) not in list_constraint and (d, s) not in list_constraint
and s!='' and d!='':
```

```
            list_constraint.append((s,d))
print(list_node, list_constraint)

def check_constraint(n, c):
    """
    This function is being used to check the binary constraint satisfaction for the
assignment of n=c
    """
    ok = True
    for cons in list_constraint:
        if cons[0] == n:
            other_node = cons[1]
            if G.nodes[other_node]['color'] == c:
                ok = False
                return ok
        elif cons[1] == n:
            other_node = cons[0]
            if G.nodes[other_node]['color'] == c:
                ok = False
                return ok
    return ok

def get_next_with_heuristic():
    """
    This function is being used to get the best neighbor according to MRV and Degree
heuristics
    """
    node_not_visited = []
    for n in G.nodes():
        if G.nodes[n]['color'] == '':
            remaining_value = len(G.nodes[n]['domain']) # For MRV
            degree_heuristic = G.degree(n) # For Degree heuristic
            e = (n, remaining_value, degree_heuristic)
            node_not_visited.append(e)
    node_not_visited.sort(key=lambda x: (x[1],-x[2])) # sorting based on priority of
MRV > Degree heuristic
    return node_not_visited[0][0] if len(node_not_visited) > 0 else None

def get_color_with_heuristic(n):
    """
    This function is being used to get best possible color value according to Least
Constraining Value heuristic
    """
    lease_constraining_value = []
    neighbors = list(G.neighbors(n))
    for c in G.nodes[n]['domain']:
        rules_out = 0
        for node in list(G.neighbors(n)):
            if G.nodes[node]['color'] == '' and c in G.nodes[node]['domain']:
                rules_out+=1
        e = (c, rules_out)
        lease_constraining_value.append(e)
    lease_constraining_value.sort(key=lambda x: x[1])
    return [tup[0] for tup in lease_constraining_value]
```

```
def get_removed_domain(n,c):
    """
    This function is being used for elimination of domain values which are
inconsistent to the assignment of n=c
    """
    result = {}
    for cons in list_constraint:
        if cons[0] == n:
            other_node = cons[1]
            if c in G.nodes[other_node]['domain']:
                G.nodes[other_node]['domain'].remove(c)
                if other_node not in result.keys():
                    result[other_node] = [c]
                else:
                    result[other_node].append(c)
                if len(G.nodes[other_node]['domain']) == 1:
                    temp_c = G.nodes[other_node]['domain'][0]
                    temp_i = list_node.index(other_node)
                    temp_result = get_removed_domain(temp_i, temp_c)
                    for k,v in temp_result.items():
                        if k not in result.keys():
                            result[k] = v
                        else:
                            result[k] = list(set(result[k]+v))
        elif cons[1] == n:
            other_node = cons[0]
            if c in G.nodes[other_node]['domain']:
                G.nodes[other_node]['domain'].remove(c)
                if other_node not in result.keys():
                    result[other_node] = [c]
                else:
                    result[other_node].append(c)
                if len(G.nodes[other_node]['domain']) == 1:
                    temp_c = G.nodes[other_node]['domain'][0]
                    temp_i = list_node.index(other_node)
                    temp_result = get_removed_domain(temp_i, temp_c)
                    for k,v in temp_result.items():
                        if k not in result.keys():
                            result[k] = v
                        else:
                            result[k] = list(set(result[k]+v))
    return result

def color_map(i,n):
    """
    This function recursively runs the algorithm until it reaches to a base case
    """
    global no_bt
    if i == len(G.nodes()):
        return True

    # use heuristic LCV
    sorted_color_domain = get_color_with_heuristic(n)
    for c in sorted_color_domain:
        G.nodes[n]['color'] = c
```

```
        ok = check_constraint(n, c)
        if ok:
            domain_remove = get_removed_domain(n,c)

            # use heuristic MRV, Degree
            next_n = get_next_with_heuristic()
            success = color_map(i+1, next_n)
            if success:
                return success
            for k,l in domain_remove.items():
                for v in l:
                    if v not in G.nodes[k]['domain']:
                        G.nodes[k]['domain'].append(v)
            no_bt +=1
        G.nodes[n]['color'] = ''
        no_bt +=1
    return False

def check_map_color(G=None, list_color=[]):
    """
    This is an outer function to find out whether mapcoloring for G with list_color
is possible or not.
    """
    # list node, # list_constraint, # list_color
    import random
    success = color_map(0, list_node[random.randint(0, len(list_node)-1)])
    return success

if __name__=='__main__':
    import time, copy
    start_time = time.time()
    max_no_color = 10
    success_with_max_no_color = False
    for no_color in range(1, max_no_color + 1):
        list_color = [c for c in range(no_color)]
        import networkx as nx
        import matplotlib.pyplot as plt

        G = nx.Graph()
        G.add_nodes_from(list_node, color='')
        G.add_edges_from(list_constraint)
        for n in G.nodes:
            G.nodes[n]['domain'] = copy.deepcopy(list_color)

        # for n in G.nodes():
        #     print(G.nodes[n]['domain'], G.nodes[n]['color'])
        no_bt = 0
        success = check_map_color(G, list_color)
        if success:
            print("Success with X(G) = ", no_color, " and No of Backtrack = ", no_bt)
            # for n in G.nodes():
            #     print(n, G.nodes[n]['color'])
            success_with_max_no_color = True
            break
        print()
```

```
    print()
if not success_with_max_no_color:
    print('Failed, need more than %s colors !!!' % (max_no_color))
print("--- %s seconds ---" % (time.time() - start_time))
```