Intelligent Systems Project-3 Report

CSP Map coloring problem

Team Members:

Shreya Vinodh	**801149383**
Utkarsha Gurjar	**801149356**
Afreen	**801136632**

Map coloring problem:

Map coloring problem is the constraint satisfaction problem. Constraint satisfaction problem is the process of getting a solution to a set of constraints that impose conditions that the variables must satisfy.

The map coloring problem is that by using fewer colors, color the given map such that no two states sharing the same border should have the same color. In this project, we are considering 2 maps, USA and Australia.

Here, we make use of chromatic number. It means the least number of colors needed to color the vertices of the graph, such that vertices sharing the same border should not have the same color. In case of both the USA and Australia maps, 3 chromatic colors are used.

It can be solved by various methods. Here, we have shown it using depth first search, depth first search using forward checking and depth first search using singleton.

Depth-first search always expands the deepest node in the current frontier of the search tree. Depth first search using forward checking checks consistency between assigned and non-assigned states. Depth first search using singleton checks if any state is left with just one color then it is assigned first.

Map coloring problem formulation:

Initial state:

A map containing states.

Heuristic:

The heuristic cost h, is the number of adjacent states have same colors.

Goal state:

A map in which the states sharing the same border have different colors and the least no. of colors are used

Goal Test :

When the heuristic h of a state is zero, that state is considered to be the goal.

Program structure

Create an object of class Map start
Class Map initializes the total no. of states, state neighbours, assigns colors to each state

Start calls main function defined in class Map main()
main() asks the user about the choice of map and then displays the colors assigned to each state of the chosen map, number of backtracks and the time taken for executing each method.

After getting the inputs, implement DFS backtrack algorithm DFS_backtrack()
Then implement DFS with forward check DFS_forwardCheck()
Then implement DFs forward check with singleton DFS_forwardSingleton()

Global variables

Aus_state_mapping: List of the states of Australia in key-value pair
Aus_edges: The neighbouring states of each state of Australia map is mentioned
Us_state_mapping: Contains the list of the states of USA in key-value pair
Us_edges: The neighbouring states of each state of USA map is mentioned
Color_list: Contains the list of all colors in key-value pair
Status: contains status of each state
Parent: to keep a track of the parent

Function/procedure to compute the heuristic function

1. The function "degreeConst()" computes the degree constraint heuristic function.
2. The function "leastConst()" computes the least constraining value heuristic function.
3. The function "MRV()" computes the minimum remaining value heuristic function.

Other functions/procedures

Apart from the above mentioned functions, there are few more functions used:

1. The constructor of "Map" class initializes the total no of states, state neighbours, assigns colors to each state.
2. The function "assignDomain()" assigns the domains to each state.
3. The function "assignTrack()" keep the track of the parent from which the children's domain have been cut.
4. The function "assignColor()" checks with the neighbouring states and then assign different color than the neighbouring state.
5. The function "checkConstraint()" checks the color of the neighbours, if different then return false
6. The function "colorMap()" checks the domain and assigns a color to each state
7. The functions "DFS_backtrack()" and "DFSvisit_backtrack" implements DFS backtracking algorithm.

8. The function “DFS_forwardCheck” and “DFSvisit_forward” implements DFS with forward checking.
9. The functions “DFS_forwardSingleton” and “DFSvisit_Singleton” implements DFS with forward checking with singleton.

Program source codes

WITHOUT HEURISTIC:

```
import time
import random

# Australia Map
aus_state_mapping = {0: 'Western Australia', 1: 'Northern Territory', 2: 'South Australia', 3:
'Queensland',
                     4: 'New South Wales',
                     5: 'Victoria', 6: 'Tasmania'}

# neighbouring edges
aus_edges = {
    0: [1, 2],
    1: [0, 2, 3],
    2: [0, 1, 3, 4, 5],
    3: [1, 2, 4],
    4: [2, 3, 5],
    5: [2, 4],
    6: []
}
# USA Map
us_state_mapping = {0: 'Washington', 1: 'Oregon', 2: 'California', 3: 'Idaho', 4: 'Nevada', 5:
'Arizona', 6: 'Utah',
                    7: 'Montana',
                    8: 'Wyoming', 9: 'Colorado', 10: 'New Mexico', 11: 'North Dakota', 12: 'South
Dakota',
                    13: 'Nebraska', 14: 'Kansas',
                    15: 'Oklahoma', 16: 'Texas', 17: 'Minessota', 18: 'Iowa', 19: 'Missouri', 20:
'Arkansas',
                    21: 'Lousiana', 22: 'Wisconsin',
                    23: 'Illinois', 24: 'Mississippi', 25: 'Michigan', 26: 'Indiana', 27: 'Kentucky', 28:
'Tennessee',
                    29: 'Alabama', 30: 'Ohio',
                    31: 'West Virginia', 32: 'Virgnia', 33: 'North Carolina', 34: 'South Carolina', 35:
'Georgia',
                    36: 'Florida',
                    37: 'Pennsylvania', 38: 'Maryland', 39: 'Delaware', 40: 'New Jersey', 41: 'New York',
                    42: 'Connecticut', 43: 'Hawaii', 44: 'Massachusetts',
                    45: 'Rhode Island', 46: 'Vermont', 47: 'New Hamsphire', 48: 'Maine', 49: 'Alaska'}

# neighbouring edges of USA
us_edges = {
```

```
    0: [3, 1],
    1: [0, 3, 4, 2],
    2: [1, 4, 5],
    3: [0, 1, 4, 6, 8, 7],
    4: [1, 2, 5, 6, 3],
    5: [2, 4, 6, 9, 10],
    6: [3, 4, 5, 10, 9, 8],
    7: [3, 8, 12, 11],
    8: [7, 3, 6, 9, 13, 12],
    9: [8, 6, 5, 10, 15, 14, 13],
    10: [5, 6, 9, 15, 16],
    11: [7, 12, 17],
    12: [11, 7, 8, 13, 18, 17],
    13: [18, 12, 8, 9, 14, 19],
    14: [13, 9, 15, 19],
    15: [16, 20, 19, 14, 9, 10],
    16: [10, 15, 20, 21],
    17: [11, 12, 18, 22],
    18: [17, 12, 13, 19, 23, 22],
    19: [18, 13, 14, 15, 20, 27, 28, 23],
    20: [19, 15, 16, 21, 24, 28],
    21: [16, 20, 24],
    22: [17, 18, 23, 25],
    23: [22, 18, 19, 27, 26],
    24: [29, 21, 28, 20],
    25: [22, 26, 30],
    26: [25, 23, 27, 30],
    27: [26, 23, 19, 28, 32, 31, 30],
    28: [27, 19, 20, 24, 29, 35, 33, 32],
    29: [28, 24, 36, 35],
    30: [25, 26, 27, 31, 37],
    31: [30, 27, 32, 37, 38],
    32: [38, 31, 27, 28, 33],
    33: [32, 28, 35, 34],
    34: [33, 35],
    35: [29, 28, 33, 34, 36],
    36: [29, 35],
    37: [30, 31, 41, 40, 38, 39],
    38: [31, 32, 39, 37],
    39: [40, 38, 37],
    40: [39, 37, 41],
    41: [37, 40, 42, 44, 45, 46],
    42: [40, 41, 44, 45],
    43: [],
    44: [42, 47, 41, 45, 46],
    45: [44, 42],
    46: [41, 44, 47],
    47: [46, 44, 48],
    48: [47],
    49: []
}
```

```
color_list = {1: 'Red', 2: 'Blue', 3: 'Green', 4: 'Yellow', 5: 'Cyan', 6: 'Magenta', 7: 'Black', 8: 'White'}


class Map:
    # initialize total no of states, state neighbours, assigns colors to each state
    def __init__(self, tot_states, state_neighbors):
        self.tot_states = tot_states
        self.state_neighbors = state_neighbors
        self.chromatic_num = 0
        self.color = [0] * self.tot_states  # 1 color for each state
        self.backtrack_count = 0

    # assigns the domains
    def assignDomain(self, num):
        domain_dict = {}
        for i in range(self.tot_states):
            domain_dict[i] = {}
            for n in range(1, num + 1):
                domain_dict[i][n] = 2
        return domain_dict

     #to keep a track from which state the current state has cut out
    def assignTrack(self, num):
        track = {}
        for i in range(self.tot_states):
            track[i] = {}
            for n in range(1, num + 1):
                track[i][n] = -1
        return track

    # check with the neighbouring states and then assign different color than the neighbouring
state
    def assignColor(self, num, domain_dict):
        visited = []
        backtracks = 0
        for state in self.state_neighbors:
            if state not in visited:
                for color in domain_dict[state]:
                    if self.checkConstraint(state, color) == False:
                        self.color[state] = color
                        visited.append(state)
                        break
                for neighbor in self.state_neighbors[state]:
                    if neighbor not in visited:
                        for color in domain_dict[neighbor]:
                            if self.checkConstraint(neighbor, color) == False:
                                self.color[neighbor] = color
                                visited.append(neighbor)
                                break
                            else:
```

```
                    backtracks += 1
        return max(self.color)

    # check the color of the neighbours, if different then return false
    def checkConstraint(self, state, color):
        for j in self.state_neighbors[state]:
            if self.color[j] == color:
                return True
        return False

    # checks the domain and assigns the color to each state
    def colorMap(self, max_num):
        domain_dict = self.assignDomain(max_num)
        min_chrom = self.assignColor(max_num, domain_dict)
        return min_chrom

    # algorithm for implementing DFS backtracking
    def DFS_backtrack(self, chrom_num, random_list):
        domain_dict = self.assignDomain(chrom_num)
        global status
        global parent
        status = {}
        parent = {}
        self.backtrack_count = 0
        for state in self.state_neighbors:
            status[state] = 10
            parent[state] = -1

        for rnd in random_list:
            state = rnd
            if status[state] == 10:
                bTrack = self.DFSvisit_backtrack(state, domain_dict)
                if bTrack == -1:
                    break
        return bTrack


    def DFSvisit_backtrack(self, state, domain_dict):
        global status
        global parent
        assigned = 0
        status[state] = 20
        for c in domain_dict[state]:
            if self.checkConstraint(state, c) == False:
                if assigned == 0:
                    self.color[state] = c
                    domain_dict[state][c] = 1
                    assigned = 1
            else:
                if assigned == 0:
                    self.backtrack_count += 1
```

```
            domain_dict[state][c] = 0
        if assigned == 0:
            return -1

        for neighbor in self.state_neighbors[state]:
            if status[neighbor] == 10:
                parent[neighbor] = state
                bTrack = self.DFSvisit_backtrack(neighbor, domain_dict)
                if bTrack == -1:
                    return bTrack
        status[state] = 30
        return self.backtrack_count

    # algorithm for implementing DFS forward checking
    def DFS_forwardCheck(self, chrom_num, random_list):
        global status
        global parent
        domain_dict = self.assignDomain(chrom_num)
        track = self.assignTrack(chrom_num)
        status = {}
        parent = {}
        self.backtrack_count = 0
        for state in self.state_neighbors:
            status[state] = 10
            parent[state] = -1

        for rnd in random_list:
            state = rnd
            if status[state] == 10:
                bTrack = self.DFSvisit_forward(state, domain_dict, track)
        return bTrack

    def DFSvisit_forward(self, state, domain_dict, track):
        global status
        global parent
        assigned = 0
        prev_c = -1
        color = {}
        status[state] = 20
        for c in domain_dict[state]:
            if domain_dict[state][c] == 2:
                if assigned == 0:
                    self.color[state] = c
                    domain_dict[state][c] = 1
                    assigned = 1
                    self.reduceDomain(state, c, domain_dict, track)

            elif domain_dict[state][c] == 1:
                prev_c = c

        if assigned == 1 and prev_c != -1:
```

```
        self.undo(state, prev_c, domain_dict, track)
        color[prev_c] = 2

    if assigned == 0:
        self.backtrack_count += 1
        if parent[state] != -1:
            self.DFSvisit_forward(parent[state], domain_dict, track)

    for neighbor in self.state_neighbors[state]:
        if status[neighbor] == 10:
            parent[neighbor] = state
            self.DFSvisit_forward(neighbor, domain_dict, track)
    status[state] = 30
    return self.backtrack_count

  # for reducing the domains
def reduceDomain(self, state, c, domain_dict, track):
    for neighbor in self.state_neighbors[state]:
        domain_dict[neighbor][c] = 0
        if track[neighbor][c] == -1:
            track[neighbor][c] = state

  # for undoing the states
def undo(self, state, c, domain_dict, track):
    for neighbor in self.state_neighbors[state]:
        if track[neighbor][c] == state and domain_dict[neighbor][c] == 0:
            domain_dict[neighbor][c] = 2

# algorithm for implementing Singleton
def DFS_forwardSingleton(self, chrom_num, random_list):
    global status
    global parent
    domain_dict = self.assignDomain(chrom_num)
    track = self.assignTrack(chrom_num)
    status = {}
    parent = {}
    self.backtrack_count = 0
    for state in self.state_neighbors:
        status[state] = 10
        parent[state] = -1

    for rnd in random_list:
        state = rnd
        if status[state] == 10:
            bTrack = self.DFSvisit_Singleton(state, domain_dict, track)
    return bTrack

def DFSvisit_Singleton(self, state, domain_dict, track):
    global status
    global parent
    assigned = 0
```

```
    prev_c = -1
    color = {}
    status[state] = 20
    for c in domain_dict[state]:
        if domain_dict[state][c] == 2:
            if assigned == 0:
                self.color[state] = c
                domain_dict[state][c] = 1
                assigned = 1
                self.reduceDomainSingleton(state, c, domain_dict, track)
        elif domain_dict[state][c] == 1:
            prev_c = c

    if assigned == 1 and prev_c != -1:
        self.undoSingleton(state, prev_c, domain_dict, track)
        color[prev_c] = 2

    if assigned == 0:
        self.backtrack_count += 1
        if parent[state] != -1:
            self.DFSvisit_Singleton(parent[state], domain_dict, track)

    for neighbor in self.state_neighbors[state]:
        if status[neighbor] == 10:
            parent[neighbor] = state
            self.DFSvisit_Singleton(neighbor, domain_dict, track)
    status[state] = 30
    return self.backtrack_count

def reduceDomainSingleton(self, state, c, domain_dict, track):
    for neighbor in self.state_neighbors[state]:
        check = domain_dict[neighbor][c]
        domain_dict[neighbor][c] = 0
        if check == 2:
            colorS = self.checkSingleton(neighbor, domain_dict)
            if colorS > 0:
                self.reduceDomainSingleton(neighbor, colorS, domain_dict, track)
        if track[neighbor][c] == -1:
            track[neighbor][c] = state

def checkSingleton(self, neighbor, domain_dict):
    color_dict = domain_dict[neighbor]
    count = 0
    temp_c = 0
    for key in color_dict:
        if color_dict[key] == 2:
            count += 1
            temp_c = key
    if count != 1:
        temp_c = 0
```

```
        return temp_c

    def undoSingleton(self, state, c, domain_dict, track):
        for neighbor in self.state_neighbors[state]:
            if track[neighbor][c] == state and domain_dict[neighbor][c] == 0:
                domain_dict[neighbor][c] = 2

#Main method which takes the input from the user
def main():
    map_choice = input('Select map to be colored : \n1.Australia\n2.US\n')
    print()
    if map_choice == '1':
        state_mapping = aus_state_mapping
        edges = aus_edges
        m = Map(len(edges), edges)
        min_possible = m.colorMap(5)
        m.chromatic_num = min_possible
        print("Minimum chromatic number possible for Australia map = ", m.chromatic_num)
        print()

    elif map_choice == '2':
        state_mapping = us_state_mapping
        edges = us_edges
        m = Map(len(edges), edges)
        min_possible = m.colorMap(5)
        m.chromatic_num = min_possible
        print("Minimum chromatic number possible for US map = ", m.chromatic_num)
        print()

    # Generate a random list of states
    random_list = []
    states = []
    states_colored = []
    for i in range(m.tot_states):
        random_list.append(i)
    random.shuffle(random_list)
    for s in random_list:
        states.append(state_mapping[s])
    print("Search starts from state: ", states[0])
    print()

    # Print the Results
    print("1.DFS Only:")
    print()
    start = time.process_time()
    m.backtracks = 0
    m.backtracks = m.DFS_backtrack(m.chromatic_num, random_list)
    end = time.time_ns()
    assigned = []
    print("Number of backtracks = ", m.backtracks)
    for colors in m.color:
```

```
            assigned.append(color_list[colors])
        print("Colors assigned for states in original order = ", assigned)
        print("Time taken", (end - start), 'ns')
        print()

        print("2.DFS with Forward Check:")
        print()
        start = time.process_time()
        m.backtracks = 0
        m.backtracks = m.DFS_forwardCheck(m.chromatic_num, random_list)
        end = time.time_ns()
        assigned = []
        print("Number of backtracks = ", m.backtracks)
        for colors in m.color:
            assigned.append(color_list[colors])
        print("Colors assigned for states in original order = ", assigned)
        print("Time taken", (end - start), 'ns')
        print()

        print("3.DFS with Forward Check and propagation through Singleton domains:")
        print()
        start = time.process_time()
        m.backtracks = 0
        m.backtracks = m.DFS_forwardSingleton(m.chromatic_num, random_list)
        end = time.time_ns()
        assigned = []
        print("Number of backtracks = ", m.backtracks)
        for colors in m.color:
            assigned.append(color_list[colors])
        print("Colors assigned for states in original order = ", assigned)
        print("Time taken", (end - start), 'ns')
        print()

    main()
```

WITH HEURISTIC:

```
 import time
import random

# Australia Map
aus_state_mapping = {0: 'Western Australia', 1: 'Northern Territory', 2: 'South Australia', 3:
'Queensland',
                     4: 'New South Wales',
                     5: 'Victoria', 6: 'Tasmania'}


aus_edges = {
    0: [1, 2],
    1: [0, 2, 3],
```

```
    2: [0, 1, 3, 4, 5],
    3: [1, 2, 4],
    4: [2, 3, 5],
    5: [2, 4],
    6: []
}

# USA Map
us_state_mapping = {0: 'Washington', 1: 'Oregon', 2: 'California', 3: 'Idaho', 4: 'Nevada', 5:
'Arizona', 6: 'Utah',
                    7: 'Montana',
                    8: 'Wyoming', 9: 'Colorado', 10: 'New Mexico', 11: 'North Dakota', 12: 'South Dakota',
                    13: 'Nebraska', 14: 'Kansas',
                    15: 'Oklahoma', 16: 'Texas', 17: 'Minessota', 18: 'Iowa', 19: 'Missouri', 20: 'Arkansas',
                    21: 'Lousiana', 22: 'Wisconsin',
                    23: 'Illinois', 24: 'Mississippi', 25: 'Michigan', 26: 'Indiana', 27: 'Kentucky', 28:
'Tennessee',
                    29: 'Alabama', 30: 'Ohio',
                    31: 'West Virginia', 32: 'Virgnia', 33: 'North Carolina', 34: 'South Carolina', 35:
'Georgia',
                    36: 'Florida',
                    37: 'Pennsylvania', 38: 'Maryland', 39: 'Delaware', 40: 'New Jersey', 41: 'New York',
                    42: 'Connecticut', 43: 'Hawaii', 44: 'Massachusetts',
                    45: 'Rhode Island', 46: 'Vermont', 47: 'New Hamsphire', 48: 'Maine', 49: 'Alaska'}

us_edges = {
    0: [3, 1],
    1: [0, 3, 4, 2],
    2: [1, 4, 5],
    3: [0, 1, 4, 6, 8, 7],
    4: [1, 2, 5, 6, 3],
    5: [2, 4, 6, 9, 10],
    6: [3, 4, 5, 10, 9, 8],
    7: [3, 8, 12, 11],
    8: [7, 3, 6, 9, 13, 12],
    9: [8, 6, 5, 10, 15, 14, 13],
    10: [5, 6, 9, 15, 16],
    11: [7, 12, 17],
    12: [11, 7, 8, 13, 18, 17],
    13: [18, 12, 8, 9, 14, 19],
    14: [13, 9, 15, 19],
    15: [16, 20, 19, 14, 9, 10],
    16: [10, 15, 20, 21],
    17: [11, 12, 18, 22],
    18: [17, 12, 13, 19, 23, 22],
    19: [18, 13, 14, 15, 20, 27, 28, 23],
    20: [19, 15, 16, 21, 24, 28],
    21: [16, 20, 24],
    22: [17, 18, 23, 25],
    23: [22, 18, 19, 27, 26],
    24: [29, 21, 28, 20],
```

```
    25: [22, 26, 30],
    26: [25, 23, 27, 30],
    27: [26, 23, 19, 28, 32, 31, 30],
    28: [27, 19, 20, 24, 29, 35, 33, 32],
    29: [28, 24, 36, 35],
    30: [25, 26, 27, 31, 37],
    31: [30, 27, 32, 37, 38],
    32: [38, 31, 27, 28, 33],
    33: [32, 28, 35, 34],
    34: [33, 35],
    35: [29, 28, 33, 34, 36],
    36: [29, 35],
    37: [30, 31, 41, 40, 38, 39],
    38: [31, 32, 39, 37],
    39: [40, 38, 37],
    40: [39, 37, 41],
    41: [37, 40, 42, 44, 45, 46],
    42: [40, 41, 44, 45],
    43: [],
    44: [42, 47, 41, 45, 46],
    45: [44, 42],
    46: [41, 44, 47],
    47: [46, 44, 48],
    48: [47],
    49: []
}

color_list = {0: 'CHECK!!!', 1: 'Red', 2: 'Blue', 3: 'Green', 4: 'Yellow', 5: 'Cyan', 6: 'Magenta', 7:
'Black', 8: 'White'}

class Map:
    def __init__(self, tot_states, state_neighbors):
        self.tot_states = tot_states
        self.state_neighbors = state_neighbors
        self.chromatic_num = 0
        self.color = [0] * self.tot_states  # 1 color for each state
        self.backtrack_count = 0

    def assignDomain(self, num):
        domain_dict = {}
        for i in range(self.tot_states):
            domain_dict[i] = {}
            for n in range(1, num + 1):
                domain_dict[i][n] = 2
        return domain_dict

    def assignTrack(self, num):
        track = {}
        for i in range(self.tot_states):
            track[i] = {}
            for n in range(1, num + 1):
```

```
            track[i][n] = -1
    return track

def assignColor(self, num, domain_dict):
    visited = []
    backtracks = 0
    for state in self.state_neighbors:
        if state not in visited:
            for color in domain_dict[state]:
                if self.checkConstraint(state, color) == False:
                    self.color[state] = color
                    visited.append(state)
                    break
            for neighbor in self.state_neighbors[state]:
                if neighbor not in visited:
                    for color in domain_dict[neighbor]:
                        if self.checkConstraint(neighbor, color) == False:
                            self.color[neighbor] = color
                            visited.append(neighbor)
                            break
                        else:
                            backtracks += 1
    return max(self.color)

def checkConstraint(self, state, color):
    for j in self.state_neighbors[state]:
        if self.color[j] == color:
            return True
    return False

def colorMap(self, max_num):
    domain_dict = self.assignDomain(max_num)
    min_chrom = self.assignColor(max_num, domain_dict)
    return min_chrom

def DFS_backtrack(self, chrom_num, heuristic):
    domain_dict = self.assignDomain(chrom_num)
    global status
    global parent
    status = {}
    parent = {}
    self.backtrack_count = 0
    for state in self.state_neighbors:
        status[state] = 10
        parent[state] = -1

    for state in self.state_neighbors:
        if status[state] == 10:
            bTrack = self.DFSvisit_backtrack(state, domain_dict, heuristic)
            if bTrack == -1:
                break
```

```
    return bTrack

def DFSvisit_backtrack(self, state, domain_dict, heuristic):
    global status
    global parent
    assigned = 0
    status[state] = 20

    if heuristic == '3':
        c = self.leastConst(state, domain_dict)
        if c != -1:
            self.color[state] = c
            domain_dict[state][c] = 1
            assigned = 1
        else:
            self.backtrack_count += 1
            return self.backtrack_count
    else:
        for c in domain_dict[state]:
            if self.checkConstraint(state, c) == False:
                if assigned == 0:
                    self.color[state] = c
                    domain_dict[state][c] = 1
                    assigned = 1
            else:
                if assigned == 0:
                    self.backtrack_count += 1
                domain_dict[state][c] = 0
        if assigned == 0:
            return -1

    if heuristic == '1':
        neighbor = self.MRV(state, domain_dict)
        if neighbor == -1:
            return self.backtrack_count
        else:
            parent[neighbor] = state
            bTrack = self.DFSvisit_backtrack(neighbor, domain_dict, heuristic)
            if bTrack == -1:
                return bTrack
    elif heuristic == '2':
        neighbor = self.degreeConst(state, domain_dict)
        if neighbor == -1:
            return self.backtrack_count
        else:
            parent[neighbor] = state
            bTrack = self.DFSvisit_backtrack(neighbor, domain_dict, heuristic)
            if bTrack == -1:
                return bTrack
    else:
        for neighbor in self.state_neighbors[state]:
```

```
            if status[neighbor] == 10:
                parent[neighbor] = state
                bTrack = self.DFSvisit_backtrack(neighbor, domain_dict, heuristic)
                if bTrack == -1:
                    return bTrack
        status[state] = 30
        return self.backtrack_count

    def DFS_forwardCheck(self, chrom_num, heuristic):
        global status
        global parent
        domain_dict = self.assignDomain(chrom_num)
        track = self.assignTrack(chrom_num)
        status = {}
        parent = {}
        self.backtrack_count = 0
        for state in self.state_neighbors:
            status[state] = 10
            parent[state] = -1

        for state in self.state_neighbors:
            if status[state] == 10:
                bTrack = self.DFSvisit_forward(state, domain_dict, track, heuristic)
        return bTrack

    def DFSvisit_forward(self, state, domain_dict, track, heuristic):
        global status
        global parent
        assigned = 0
        prev_c = -1
        color = {}
        status[state] = 20

        if heuristic == '3':
            for c1 in domain_dict[state]:
                if domain_dict[state][c1] == 1:
                    prev_c = c1

            c = self.leastConst(state, domain_dict)
            if c != -1:
                self.color[state] = c
                domain_dict[state][c] = 1
                assigned = 1
                self.reduceDomain(state, c, domain_dict, track)
            else:
                assigned = 0
        else:
            for c in domain_dict[state]:
                if domain_dict[state][c] == 2:
                    if assigned == 0:
                        self.color[state] = c
```

```
                    domain_dict[state][c] = 1
                    assigned = 1
                    self.reduceDomain(state, c, domain_dict, track)
                elif domain_dict[state][c] == 1:
                    prev_c = c

        if assigned == 1 and prev_c != -1:
            self.undo(state, prev_c, domain_dict, track)
            color[prev_c] = 2

        if assigned == 0:
            self.backtrack_count += 1
            if parent[state] != -1:
                self.DFSvisit_forward(parent[state], domain_dict, track, heuristic)

        if heuristic == '1':
            neighbor = self.MRV(state, domain_dict)
            if neighbor == -1:
                return self.backtrack_count
            else:
                parent[neighbor] = state
                self.DFSvisit_forward(neighbor, domain_dict, track, heuristic)
        elif heuristic == '2':
            neighbor = self.degreeConst(state, domain_dict)
            if neighbor == -1:
                return self.backtrack_count
            else:
                parent[neighbor] = state
                self.DFSvisit_forward(neighbor, domain_dict, track, heuristic)
        else:
            for neighbor in self.state_neighbors[state]:
                if status[neighbor] == 10:
                    parent[neighbor] = state
                    self.DFSvisit_forward(neighbor, domain_dict, track, heuristic)
        status[state] = 30
        return self.backtrack_count

    def reduceDomain(self, state, c, domain_dict, track):
        for neighbor in self.state_neighbors[state]:
            domain_dict[neighbor][c] = 0
            if track[neighbor][c] == -1:
                track[neighbor][c] = state

    def undo(self, state, c, domain_dict, track):
        for neighbor in self.state_neighbors[state]:
            if track[neighbor][c] == state and domain_dict[neighbor][c] == 0:
                domain_dict[neighbor][c] = 2

    def DFS_forwardSingleton(self, chrom_num, heuristic):
        global status
        global parent
```

```
    domain_dict = self.assignDomain(chrom_num)
    track = self.assignTrack(chrom_num)
    status = {}
    parent = {}
    self.backtrack_count = 0
    for state in self.state_neighbors:
        status[state] = 10
        parent[state] = -1

    for state in self.state_neighbors:
        if status[state] == 10:
            bTrack = self.DFSvisit_Singleton(state, domain_dict, track, heuristic)
    return bTrack

def DFSvisit_Singleton(self, state, domain_dict, track, heuristic):
    global status
    global parent
    assigned = 0
    prev_c = -1
    color = {}
    status[state] = 20

    if heuristic == '3':
        for c1 in domain_dict[state]:
            if domain_dict[state][c1] == 1:
                prev_c = c1

        c = self.leastConst(state, domain_dict)
        if c != -1:
            self.color[state] = c
            domain_dict[state][c] = 1
            assigned = 1
            self.reduceDomainSingleton(state, c, domain_dict, track)
        else:
            assigned = 0
    else:
        for c in domain_dict[state]:
            if domain_dict[state][c] == 2:
                if assigned == 0:
                    self.color[state] = c
                    domain_dict[state][c] = 1
                    assigned = 1
                    self.reduceDomainSingleton(state, c, domain_dict, track)
            elif domain_dict[state][c] == 1:
                prev_c = c

    if assigned == 1 and prev_c != -1:
        self.undoSingleton(state, prev_c, domain_dict, track)
        color[prev_c] = 2

    if assigned == 0:
```

```
            self.backtrack_count += 1
            if parent[state] != -1:
                self.DFSvisit_Singleton(parent[state], domain_dict, track, heuristic)

        if heuristic == '1':
            neighbor = self.MRV(state, domain_dict)
            if neighbor == -1:
                return self.backtrack_count
            else:
                parent[neighbor] = state
                self.DFSvisit_Singleton(neighbor, domain_dict, track, heuristic)
        elif heuristic == '2':
            neighbor = self.degreeConst(state, domain_dict)
            if neighbor == -1:
                return self.backtrack_count
            else:
                parent[neighbor] = state
                self.DFSvisit_Singleton(neighbor, domain_dict, track, heuristic)
        else:
            for neighbor in self.state_neighbors[state]:
                if status[neighbor] == 10:
                    parent[neighbor] = state
                    self.DFSvisit_Singleton(neighbor, domain_dict, track, heuristic)
        status[state] = 30
        return self.backtrack_count

    def reduceDomainSingleton(self, state, c, domain_dict, track):
        for neighbor in self.state_neighbors[state]:
            check = domain_dict[neighbor][c]
            domain_dict[neighbor][c] = 0
            if check == 2:
                colorS = self.checkSingleton(neighbor, domain_dict)
                if colorS > 0:
                    self.reduceDomainSingleton(neighbor, colorS, domain_dict, track)
            if track[neighbor][c] == -1:
                track[neighbor][c] = state

    def checkSingleton(self, neighbor, domain_dict):
        color_dict = domain_dict[neighbor]
        count = 0
        temp_c = 0
        for key in color_dict:
            if color_dict[key] == 2:
                count += 1
                temp_c = key
        if count != 1:
            temp_c = 0

        return temp_c

    def undoSingleton(self, state, c, domain_dict, track):
```

```
        for neighbor in self.state_neighbors[state]:
            if track[neighbor][c] == state and domain_dict[neighbor][c] == 0:
                domain_dict[neighbor][c] = 2

    # to select a neighbour that has the largest number of constraints on their unassigned
neighbours
    def degreeConst(self, state, domain_dict):
        global status
        adj = {}
        max = 0
        selected = -1
        for neighbor in self.state_neighbors[state]:
            if status[neighbor] == 10:
                count = 0
                for n in self.state_neighbors[neighbor]:
                    if status[n] == 10:
                        count += 1
                if count > max:
                    max = count
                    selected = neighbor
        return selected

    # to find the color that rules out fewest choices for the neighbors
    def leastConst(self, state, domain_dict):
        global status
        min = 99
        selected = -1
        for color in domain_dict[state]:
            if self.checkConstraint(state, color) == True:
                continue
            count = 0
            if domain_dict[state][color] == 2:
                for neighbor in self.state_neighbors[state]:
                    if status[neighbor] == 10:
                        if domain_dict[neighbor][color] == 2:
                            count += 1
                if min > count and count > 0:
                    min = count
                    selected = color
        return selected

    def MRV(self, state, domain_dict):
        global status
        min = 99
        selected = -1
        for neighbor in self.state_neighbors[state]:
            if status[neighbor] == 10:
                count = 0
                for color in domain_dict[neighbor]:
                    if domain_dict[neighbor][color] == 2:
                        count += 1
```

```
            if min > count and count > 0:
                min = count
                selected = neighbor
        return selected


# MAIN
def main():
    map_choice = input('Select map to be colored : \n1.Australia\n2.US\n')
    print()
    if map_choice == '1':
        state_mapping = aus_state_mapping
        edges = aus_edges
        m = Map(len(edges), edges)
        min_possible = m.colorMap(5)
        m.chromatic_num = min_possible
        print("Minimum chromatic number possible for Australia map = ", m.chromatic_num)
        print()

    elif map_choice == '2':
        state_mapping = us_state_mapping
        edges = us_edges
        m = Map(len(edges), edges)
        min_possible = m.colorMap(5)
        m.chromatic_num = min_possible
        print("Minimum chromatic number possible for US map = ", m.chromatic_num)
        print()

    heuristic = input(
        'Select a heuristic to be used:\n1.Minimum Remaining Values (MRV)\n2.Degree
Constraint\n3.Least constraint Value\n')
    print()

    # Print the Results
    print("1.DFS Only:")
    print()
    start = time.time_ns()
    m.backtracks = 0
    m.backtracks = m.DFS_backtrack(m.chromatic_num, heuristic)
    end = time.time_ns()
    assigned = []
    print("Number of backtracks = ", m.backtracks)
    for colors in m.color:
        assigned.append(color_list[colors])
    print("Colors assigned for states in original order = ", assigned)
    print("Time taken", (end - start), 'ns')
    print()

    print("2.DFS with Forward Check:")
    print()
    start = time.time_ns()
```

```
    m.backtracks = 0
    m.backtracks = m.DFS_forwardCheck(m.chromatic_num, heuristic)
    end = time.time_ns()
    assigned = []
    print("Number of backtracks = ", m.backtracks)
    for colors in m.color:
        assigned.append(color_list[colors])
    print("Colors assigned for states in original order = ", assigned)
    print("Time taken", (end - start), 'ns')
    print()

    print("3.DFS with Forward Check and propagation through Singleton domains:")
    print()
    start = time.time_ns()
    m.backtracks = 0
    m.backtracks = m.DFS_forwardSingleton(m.chromatic_num, heuristic)
    end = time.time_ns()
    assigned = []
    print("Number of backtracks = ", m.backtracks)
    for colors in m.color:
        assigned.append(color_list[colors])
    print("Colors assigned for states in original order = ", assigned)
    print("Time taken", (end - start), 'ns')
    print()


main()
```

Execution results
WITHOUT HEURISTICS:

FOR AUSTRALIA:

1.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Search starts from state:  Northern Territory

1.DFS Only:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  0
Colors assigned for states in original order =  ['Blue', 'Red', 'Green', 'Blue', 'Red', 'Blue', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  5
Colors assigned for states in original order =  ['Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Red']
Time taken 994000 ns
```

2.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Search starts from state:  New South Wales

1.DFS Only:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 995400 ns

2.DFS with Forward Check:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  0
Colors assigned for states in original order =  ['Green', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Red']
Time taken 0 ns
```

3.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Search starts from state:  Victoria

1.DFS Only:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  0
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  0
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red']
Time taken 0 ns
```

4.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Search starts from state:  Northern Territory

1.DFS Only:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  0
Colors assigned for states in original order =  ['Blue', 'Red', 'Green', 'Blue', 'Red', 'Blue', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  5
Colors assigned for states in original order =  ['Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Red']
Time taken 998500 ns
```

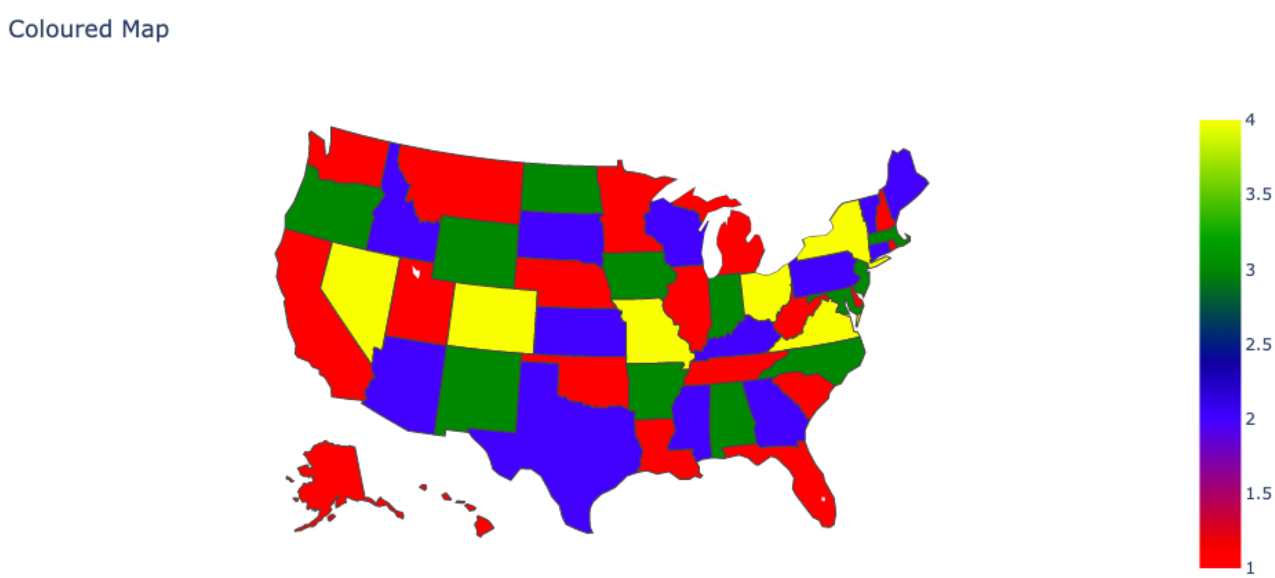
Below table shows the number of backtracks, time taken in nanoseconds for DFS, DFS with forward checking and DFS with singleton:

Start state	DFS	DFS+FC	DFS + S	
Northern Territory	4	0	5	**No.of backtracks**
New South Wales	4	4	0	
Victoria	4	0	0	
Northern Territory	4	0	5	
Northern Territory	0	0	994000	

New South Wales	995400	0	0	**Time taken (nanoseconds)**
Victoria	0	0	0	
Northern Territory	0	0	998500	

USA:

With visualisation:



<u>Without visualisation</u>

1.

```
Select map to be colored :
1.Australia
2.US
2

Minimum chromatic number possible for US map =  4

Search starts from state:  Mississippi

1.DFS Only:

Number of backtracks =  55
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  1
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Blue', 'Red', 'Green', 'Yellow', 'Red', 'Green', 'Blue', 'Red', 'Blue', 'Yellow',
 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Green', 'Blue', 'Green', 'Blue', 'Blue', 'Yellow', 'Red', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Yellow', 'Red', 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Yellow', 'Red', 'Blue', 'Red']
Time taken 996900 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  3
Colors assigned for states in original order =  ['Blue', 'Green', 'Red', 'Red', 'Blue', 'Yellow', 'Green', 'Green', 'Blue', 'Red', 'Blue', 'Red', 'Yellow',
 'Green', 'Blue', 'Green', 'Red', 'Green', 'Blue', 'Yellow', 'Blue', 'Green', 'Red', 'Green', 'Red', 'Green', 'Yellow', 'Red', 'Green', 'Blue', 'Green',
 'Green', 'Yellow', 'Red', 'Blue', 'Yellow', 'Red', 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Yellow', 'Red', 'Green', 'Yellow', 'Yellow', 'Red', 'Green', 'Red']
Time taken 0 ns
```

2.

```
Select map to be colored :
1.Australia
2.US
2

Minimum chromatic number possible for US map =  4

Search starts from state:  Tennessee

1.DFS Only:

Number of backtracks =  55
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 996200 ns

2.DFS with Forward Check:

Number of backtracks =  17
Colors assigned for states in original order =  ['Blue', 'Green', 'Yellow', 'Red', 'Blue', 'Red', 'Yellow', 'Blue', 'Green', 'Blue', 'Red', 'Red', 'Yellow',
 'Red', 'Green', 'Red', 'Blue', 'Blue', 'Green', 'Yellow', 'Green', 'Red', 'Red', 'Red', 'Blue', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Blue', 'Red']
Time taken 996700 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  48
Colors assigned for states in original order =  ['Yellow', 'Blue', 'Blue', 'Red', 'Yellow', 'Green', 'Blue', 'Blue', 'Green', 'Blue', 'Yellow', 'Red',
 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Blue', 'Green', 'Yellow', 'Blue', 'Yellow', 'Red', 'Red', 'Green', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Yellow',
 'Red', 'Yellow', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Blue', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Blue', 'Red']
Time taken 1996100 ns
```

3.

```
Select map to be colored :
1.Australia
2.US
2

Minimum chromatic number possible for US map =  4

Search starts from state:  Rhode Island

1.DFS Only:

Number of backtracks =  55
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 996900 ns

2.DFS with Forward Check:

Number of backtracks =  5
Colors assigned for states in original order =  ['Blue', 'Green', 'Red', 'Red', 'Blue', 'Green', 'Yellow', 'Blue', 'Green', 'Blue', 'Red', 'Red', 'Yellow',
 'Red', 'Green', 'Yellow', 'Blue', 'Blue', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red', 'Blue', 'Blue', 'Green', 'Yellow', 'Green', 'Yellow', 'Red', 'Blue',
 'Yellow', 'Yellow', 'Red', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Red', 'Green', 'Red', 'Red']
Time taken 997300 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  23
Colors assigned for states in original order =  ['Yellow', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Green', 'Yellow', 'Blue',
 'Yellow', 'Red', 'Yellow', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Yellow', 'Yellow', 'Red', 'Yellow', 'Blue',
 'Red', 'Yellow', 'Green', 'Red', 'Blue', 'Green', 'Green', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 998000 ns
```

4.

```
Select map to be colored :
1.Australia
2.US
2

Minimum chromatic number possible for US map =  4

Search starts from state:  South Carolina

1.DFS Only:

Number of backtracks =  55
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 997500 ns

2.DFS with Forward Check:

Number of backtracks =  5
Colors assigned for states in original order =  ['Blue', 'Green', 'Red', 'Red', 'Blue', 'Green', 'Yellow', 'Blue', 'Green', 'Blue', 'Red', 'Red', 'Yellow',
 'Red', 'Green', 'Yellow', 'Blue', 'Blue', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red', 'Blue', 'Blue', 'Green', 'Yellow', 'Green', 'Yellow', 'Red', 'Green',
 'Red', 'Blue', 'Red', 'Blue', 'Blue', 'Yellow', 'Blue', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Blue', 'Red']
Time taken 997800 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  19
Colors assigned for states in original order =  ['Blue', 'Green', 'Blue', 'Red', 'Yellow', 'Green', 'Blue', 'Blue', 'Green', 'Red', 'Yellow', 'Red', 'Yellow',
 'Blue', 'Green', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Yellow', 'Blue', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Blue', 'Green', 'Blue', 'Red', 'Green',
 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Yellow', 'Blue', 'Red', 'Blue', 'Green', 'Blue', 'Red', 'Yellow', 'Yellow', 'Blue', 'Red', 'Blue', 'Red']
Time taken 999500 ns
```

Below table shows the number of backtracks, time taken in nanoseconds for DFS, DFS with forward checking and DFS with singleton:

Start state	DFS	DFS+FC	DFS + S	
Mississippi	55	1	3	**No.of backtracks**
Tennessee	55	17	48	
Rhode Island	55	5	23	
South Carolina	55	5	19	
Mississippi	0	996900	0	

Tennessee	996200	996700	1996100	**Time taken (nanoseconds)**
Rhode Island	996900	997300	998000	
South Carolina	997500	997800	999500	

WITH HEURISTIC:
For Australia,

1.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
1

1.DFS Only:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  0
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  1
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Green', 'Red', 'Red']
Time taken 0 ns
```

2.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
2

1.DFS Only:

Number of backtracks =  4
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 996600 ns

2.DFS with Forward Check:

Number of backtracks =  0
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  1
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Red', 'Green', 'Red', 'Red']
Time taken 0 ns
```

3.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
3

1.DFS Only:

Number of backtracks =  2
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  7
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  13
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns
```

4.

```
Select map to be colored :
1.Australia
2.US
1

Minimum chromatic number possible for Australia map =  3

Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
3

1.DFS Only:

Number of backtracks =  2
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  7
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  13
Colors assigned for states in original order =  ['Red', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Red']
Time taken 997000 ns
```

Below table shows the number of backtracks, time taken in nanoseconds for DFS, DFS with forward checking and DFS with singleton:

Method used	**DFS**	**DFS+FC**	**DFS + S**	
MRV	4	0	1	
DC	4	0	1	**No.of backtracks**
LCV	2	7	13	

LCV	2	7	13	
MRV	0	0	0	**Time taken (nanoseconds)**
DC	996600	0	0	
LCV	0	0	0	
LCV	0	0	997700	

FOR USA:

1.

```
Minimum chromatic number possible for US map =  4

Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
1

1.DFS Only:

Number of backtracks =  55
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue', 'Red', 'Blue',
 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red', 'Yellow', 'Green', 'Red', 'Blue',
 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  0
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Red', 'Blue', 'Green', 'Red',
 'Green', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Green', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Red', 'Green', 'Blue', 'Yellow', 'Red', 'Green', 'Blue',
 'Green', 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Red', 'Blue', 'Red', 'Red']
Time taken 994500 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  1
Colors assigned for states in original order =  ['Red', 'Green', 'Blue', 'Blue', 'Red', 'Green', 'Yellow', 'Yellow', 'Red', 'Blue', 'Red', 'Red', 'Blue', 'Green', 'Red',
 'Green', 'Blue', 'Green', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Green', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Red', 'Green', 'Blue', 'Yellow', 'Red', 'Green', 'Blue',
 'Green', 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Red', 'Blue', 'Red', 'Red']
Time taken 0 ns
```

2.

```
Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
3

1.DFS Only:

Number of backtracks =  8
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
  'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  176
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Red', 'Yellow', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green',
 'Yellow', 'Red', 'Yellow', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red',
 'Blue', 'Red']
Time taken 1990200 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  232
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow',
  'Red', 'Yellow', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue',
 'Red']
Time taken 1993000 ns
```

3.

```
Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
3

1.DFS Only:

Number of backtracks =  8
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
  'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 999900 ns

2.DFS with Forward Check:

Number of backtracks =  176
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Red', 'Yellow', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green',
 'Yellow', 'Red', 'Yellow', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red',
 'Blue', 'Red']
Time taken 3950600 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  232
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow',
  'Red', 'Yellow', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue',
 'Red']
Time taken 3989200 ns
```

4.

```
Select a heuristic to be used:
1.Minimum Remaining Values (MRV)
2.Degree Constraint
3.Least constraint Value
3

1.DFS Only:

Number of backtracks =  8
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Green', 'Blue',
  'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow', 'Red',
 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red']
Time taken 0 ns

2.DFS with Forward Check:

Number of backtracks =  176
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Red', 'Yellow', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green',
 'Yellow', 'Red', 'Yellow', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Red',
 'Blue', 'Red']
Time taken 1990200 ns

3.DFS with Forward Check and propagation through Singleton domains:

Number of backtracks =  232
Colors assigned for states in original order =  ['Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue', 'Red', 'Red', 'Green', 'Yellow', 'Green', 'Blue',
 'Yellow', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Yellow', 'Green', 'Red', 'Blue', 'Red', 'Blue', 'Red', 'Green', 'Blue', 'Red', 'Green', 'Yellow',
  'Red', 'Yellow', 'Green', 'Red', 'Yellow', 'Red', 'Blue', 'Green', 'Red', 'Green', 'Yellow', 'Blue', 'Red', 'Green', 'Red', 'Blue', 'Yellow', 'Blue',
 'Red']
Time taken 1993000 ns
```

Below table shows the number of backtracks, time taken in nanoseconds for DFS, DFS with forward checking and DFS with singleton:

Method used	DFS	DFS+FC	DFS + S	
MRV	55	0	1	**No.of backtracks**
DC	8	176	232	
LCV	8	176	232	
LCV	8	176	232	

MRV	0	994500	0	**Time taken (nanoseconds)**
DC	0	1990200	1993000	
LCV	999900	3950800	3989200	
LCV	0	1990200	1993000	