CHARLOTTE

Solving Constraint satisfaction problems (CSP) - Map Coloring

PROJECT DOCUMENTATION REPORT

Project By:

1.) **Devashri Gadgil - 801243925**

2.) **Varad Deshpande - 801243927**

3.) **Akshay Bheda - 801196169**

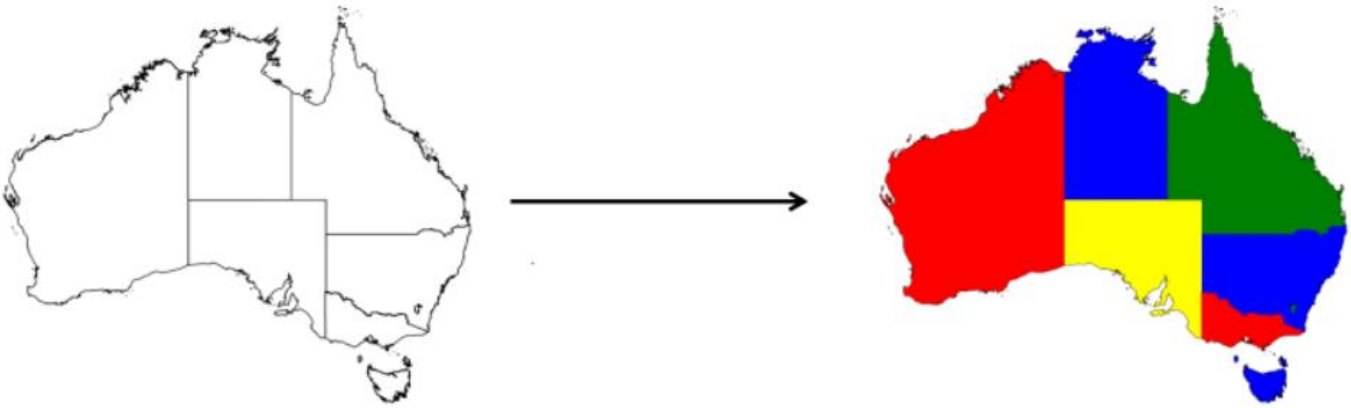
Table of Contents

Intelligent Systems Project 3

Solving Constraint satisfaction problems (CSP) - Map Coloring

Problem Formulation

The map coloring problem deals with the problem of coloring an n-state map with the given k colors such that no two neighboring states should be assigned the same color. Here, k is the minimum number of colors that can be used to solve the map coloring problem. k is called the chromatic number of the map.



The initial state of the map is the map with no color assigned to any state.

The goal state if the map with every state assigned a color such that no two states that share the same boundary, are neighbors in other words, have the same color.

The most important factor here is to check the consistency of any color m to be assigned to any state n when any of the neighbor of n has been assigned any color. We maintain a consistency graph to store the neighboring states and check for this consistency.

We can solve the map coloring problem using the following methods:

1. Backtracking: We go on adding a color from its domain to a state in a randomly ordered list of states. We backtrack as we find inconsistency in any assignment and change any of the previous assignment.
2. Forward Checking: We follow the same process as backtracking only that we remove any assigned color from the domain of its neighboring states. We backtrack in case of inconsistency or if any state has a null domain.
3. Forward Checking with singleton domain propagation: In this case, we find the next vertex with single domain value and assign that to the vertex and then backtrack to other unassigned vertices. Backtracking happens in case of inconsistency.

The heuristics for the map coloring problem are as follow:
1. Minimum Remaining Values: Choose the state with the lowest number of colors available in its domain.
2. Degree Heuristic: Choose the state with the most constraints.
3. Least Constraining Values: Choose the state with least constraints.

Overall Program Structure

We have implemented the n queens' algorithm to solve this problem using **python**

Global Variables

Variable Name	**Description**
states_australia	This contains the list of states of Australia
neighbours_australia	This contains the list of all Neighboring states mapped to each state of Australia
states_usa	This contains the list of states of USA
neighbours_usa	This contains the list of all Neighboring states mapped to each state of USA
noOfBacktracks	This is global variable to keep track of number of backtracks

Functions and Procedures

Function Name	**Description**
initColorDict	This is to initialize color dictionary
initDomain	This is to initialize domain
Backtrack	This function is for simple backtracking
Forwardcheck	This function is for simple backtracking with forward check
ForwardcheckWithSingletonPropogation	This function is for simple backtracking with f/w check and singleton propagation
reduceDomain	this function reduces the domain for heuristic function
check	This function checks if we can reduce the domain for forward checking
checkSingletonPropogation	This function checks singleton propagation
reduceSingletonDomain	This function reduces singleton domain
minRemainingValueHeuristic	This function calculates the min remaining value
leastConstrainingValueHeuristic	This function calculates the least constraining value
BacktrackWithHeuristics	This function is for backtracking with heuristics
ForwardcheckWithHeuristics	This function is for backtracking with heuristics with f/w check
ForwardcheckWithSingletonPropogationAndHeuristics	This function is for backtracking with heuristics and with f/w check and singleton propagation
getChromaticNumber	This function returns the chromatic number
checkConstraint	The function checks the constraint
showTime	This function will calculate timing and print the elapsed time

Source Code for Map Coloring

```
import random
import copy
import pandas as pd
import matplotlib.pyplot as plt
import geopandas
from datetime import datetime

states_australia = ['Western Australia', 'Northern Territory', 'South Australia',
'Queensland', 'New South Wales', 'Victoria', 'Tasmania']

neighbours_australia = {
    'Western Australia' : ['Northern Territory', 'South Australia'],
    'Northern Territory' : ['Western Australia', 'South Australia', 'Queensland'],
    'South Australia' : ['Western Australia', 'Northern Territory', 'Queensland', 'New South
Wales', 'Victoria'],
    'Queensland' : ['Northern Territory', 'South Australia', 'New South Wales'],
    'New South Wales' : ['Queensland', 'Victoria', 'South Australia'],
    'Victoria' : ['New South Wales', 'South Australia'],
    'Tasmania' : []
}

states_usa = [
    'Alabama','Alaska','Arkansas','Arizona',
    'California','Colorado','Connecticut',
    'Delaware','Florida',
    'Georgia',
    'Hawaii',
    'Iowa','Idaho','Illinois','Indiana',
    'Kansas','Kentucky',
    'Louisiana',
    'Massachusetts','Maryland','Maine','Michigan','Minnesota','Missouri','Mississippi','Monta
na',
    'North Carolina','North Dakota','Nebraska','New Hampshire','New Jersey','New
Mexico','Nevada','New York',
    'Ohio','Oklahoma','Oregon',
    'Pennsylvania',
    'Rhode Island',
    'South Carolina','South Dakota',
    'Tennessee','Texas',
    'Utah',
    'Virginia','Vermont',
    'Washington','Wisconsin','West Virginia','Wyoming'
    ]

neighbours_usa = {
```

```
    'Alabama' : ['Florida' , 'Georgia' , 'Mississippi' , 'Tennessee'],
    'Alaska' : [],
    'Arkansas' : ['Louisiana' , 'Missouri' , 'Mississippi' , 'Oklahoma' , 'Tennessee' ,
'Texas'],
    'Arizona' : ['California' , 'Colorado' , 'New Mexico' , 'Nevada' , 'Utah'],
    'California' : ['Arizona' , 'Nevada' , 'Oregon'],
    'Colorado' : ['Arizona' , 'Kansas' , 'Nebraska' , 'New Mexico' , 'Oklahoma' , 'Utah' ,
'Wyoming'],
    'Connecticut' : ['Massachusetts' , 'New York' , 'Rhode Island'],
    'Delaware' : ['Maryland' , 'New Jersey' , 'Pennsylvania' ],
    'Florida' : ['Alabama' , 'Georgia' ],
    'Georgia' : ['Alabama' , 'Florida' , 'North Carolina' , 'South Carolina' , 'Tennessee' ],
    'Hawaii' : [ ],
    'Idaho' : ['Montana' , 'Nevada' , 'Oregon' , 'Utah' , 'Washington' , 'Wyoming' ],
    'Illinois' : ['Iowa' , 'Indiana' , 'Michigan', 'Kentucky' , 'Missouri' , 'Wisconsin' ],
    'Indiana' : ['Illinois' , 'Kentucky' , 'Missouri' , 'Ohio' ],
    'Iowa' : ['Illinois' , 'Minnesota' , 'Missouri' , 'Nebraska' , 'South Dakota' ,
'Wisconsin' ],
    'Kansas' : ['Colorado' , 'Missouri' , 'Nebraska' , 'Oklahoma' ],
    'Kentucky' : ['Illinois' , 'Indiana' , 'Missouri' , 'Ohio' , 'Tennessee' , 'Virginia' ,
'West Virginia' ],
    'Louisiana' : ['Arkansas' , 'Mississippi' , 'Texas' ],
    'Massachusetts' : ['Connecticut' , 'New Hampshire' , 'New York' , 'Rhode Island' ,
'Vermont' ],
    'Maryland' : [ 'Delaware' , 'Pennsylvania' , 'Virginia' , 'West Virginia' ],
    'Maine' : ['New Hampshire' ],
    'Michigan' : ['Indiana' , 'Ohio' , 'Wisconsin','Minnesota','Illinois' ],
    'Minnesota' : ['Iowa' , 'North Dakota' ,'Michigan', 'South Dakota' , 'Wisconsin' ],
    'Missouri' : ['Arkansas' , 'Iowa' , 'Illinois' , 'Kansas' , 'Kentucky' , 'Nebraska' ,
'Oklahoma' , 'Tennessee' ],
    'Mississippi' : ['Alabama' , 'Arkansas' , 'Louisiana' , 'Tennessee' ],
    'Montana' : ['Idaho' , 'North Dakota' , 'South Dakota' , 'Wyoming' ],
    'North Carolina' : ['Georgia' , 'South Carolina' , 'Tennessee' , 'Virginia' ],
    'North Dakota' : ['Minnesota' , 'Montana' , 'South Dakota' ],
    'Nebraska' : ['Colorado' , 'Iowa' , 'Kansas' , 'Missouri' , 'South Dakota' , 'Wyoming' ],
    'New Hampshire' : ['Massachusetts' , 'Maine' , 'Vermont' ],
    'New Jersey' : ['Delaware' , 'New York' , 'Pennsylvania' ],
    'New Mexico' : ['Arizona' , 'Colorado' , 'Oklahoma' , 'Texas' , 'Utah' ],
    'Nevada' : ['Arizona' , 'California' , 'Idaho' , 'Oregon' , 'Utah' ],
    'New York' : ['Connecticut' , 'Massachusetts' , 'New Jersey' , 'Pennsylvania' ,
'Vermont', 'Rhode Island' ],
    'Ohio' : ['Indiana' , 'Kentucky' , 'Michigan' , 'Pennsylvania' , 'West Virginia' ],
    'Oklahoma' : ['Arkansas' , 'Colorado' , 'Kansas' , 'Missouri' , 'New Mexico' , 'Texas' ],
    'Oregon' : ['California' , 'Idaho' , 'Nevada' , 'Washington' ],
    'Pennsylvania' : ['Delaware' , 'Maryland' , 'New Jersey' , 'New York' , 'Ohio' , 'West
Virginia' ],
    'Rhode Island' : ['Connecticut' , 'Massachusetts', 'New York' ],
    'South Carolina' : ['Georgia' , 'North Carolina' ],
```

```
    'South Dakota' : ['Iowa' , 'Minnesota' , 'Montana' , 'North Dakota' , 'Nebraska' ,
'Wyoming' ],
    'Tennessee' : ['Alabama' , 'Arkansas' , 'Georgia' , 'Kentucky' , 'Missouri' ,
'Mississippi' , 'North Carolina' , 'Virginia' ],
    'Texas' : ['Arkansas' , 'Louisiana' , 'New Mexico' , 'Oklahoma' ],
    'Utah' : ['Arizona' , 'Colorado' , 'Idaho' , 'New Mexico' , 'Nevada' , 'Wyoming' ],
    'Virginia' : [ 'Kentucky' , 'Maryland' , 'North Carolina' , 'Tennessee' , 'West Virginia'
],
    'Vermont' : ['Massachusetts' , 'New Hampshire' , 'New York' ],
    'Washington' : [ 'Idaho' , 'Oregon' ],
    'Wisconsin' : ['Iowa' , 'Illinois' , 'Michigan' , 'Minnesota' ],
    'West Virginia' : ['Kentucky' , 'Maryland' , 'Ohio' , 'Pennsylvania' , 'Virginia' ],
    'Wyoming' : ['Colorado' , 'Idaho' , 'Montana' , 'Nebraska' , 'South Dakota' , 'Utah' ]
}

noOfBacktracks = 0

#Function to inilialize assigned colors
def initColorDict(states):
    assignedColor = {}
    for state in states:
        assignedColor[state] = 'Nil'
    return assignedColor

#Function to initialize domain
def initDomain(states, noOfColors):
    domain = {}
    if noOfColors == 1:
        color = ['R']
    elif noOfColors == 2:
        color = ['R', 'G']
    elif noOfColors == 3:
        color = ['R', 'G', 'B']
    elif noOfColors == 4:
        color = ['R', 'G', 'B', 'Y']
    else:
        color = ['R', 'G', 'B', 'Y', 'P']
    for state in states:
        domain[state] = copy.deepcopy(color)
    return domain

def Backtrack(states, neighbours, colors, domain):
    global noOfBacktracks
    # Check if Successful
    if all(value != 'Nil' for value in colors.values()):
        return "Success"
    # Pick a state to color
    currentState = states[0]
```

```
    currentNeighbors = neighbours[currentState]
    occupiedColors = list( map(colors.get, currentNeighbors))
    for color in domain[currentState]:
        if color not in occupiedColors:
            # assign consistent color
            colors[currentState] = color
            # Temporarily remove currentState
            states.remove(currentState)
            #Recusrively call the function
            output = Backtrack(states, neighbours, colors, domain)
            if output != "Failure":
                return "Success"
            colors[currentState] = 'Nil'
            # add currentState back since assignment failed
            states.append(currentState)
    if colors[currentState] == 'Nil':
        noOfBacktracks = noOfBacktracks + 1
        return "Failure"


def Forwardcheck(states, neighbours, colors, domain):
    global noOfBacktracks
    if all(value != 'Nil' for value in colors.values()):
        return "Success"
    currentState = states[0]
    #states.remove(currentState)
    currentNeighbors = neighbours[currentState]
    output = 'Success'
    occupiedColors = list( map(colors.get, currentNeighbors))
    if 'Nil' in occupiedColors:
        occupiedColors.remove('Nil')
    for color in domain[currentState]:
        if color not in occupiedColors:
            # assign consistent color
            colors[currentState] = color
            # Temporarily remove currentState
            states.remove(currentState)
            # check if domain can be reduced
            result = check(color,currentNeighbors, colors)
            if not result:
                prevDomain = copy.deepcopy(domain)
                # Reduce domain
                reduceDomain(color, currentNeighbors, colors)
                output = Forwardcheck(states, neighbours, colors, domain)
                if output != "Failure":
                    return "Success"
                # Restore the domain
                domain = prevDomain
```

```
            colors[currentState] = 'Nil'
            # Add the state back since it was a failed assignment
            states.append(currentState)
    if colors[currentState] == 'Nil':
        noOfBacktracks = noOfBacktracks + 1
        return "Failure"

def ForwardcheckWithSingletonPropogation(states, neighbours, colors, domain):
    global noOfBacktracks
    if all(value != 'Nil' for value in colors.values()):
        return "Success"
    currentState = states[0]
    #states.remove(currentState)
    currentNeighbors = neighbours[currentState]
    output = 'Success'
    occupiedColors = list( map(colors.get, currentNeighbors))
    if 'Nil' in occupiedColors:
        occupiedColors.remove('Nil')
    for color in domain[currentState]:
        if color not in occupiedColors:
            # assign consistent color
            colors[currentState] = color
            # Temporarily remove currentState
            states.remove(currentState)
            # check if domain reduction can be applied
            result = check(color,currentNeighbors, colors)
            if not result:
                prevDomain = copy.deepcopy(domain)
                # Reduce the domain
                reduceDomain(color, currentNeighbors, colors)
                # Check if singleton propogation can happen
                singleton = reduceSingletonDomain(currentNeighbors, neighbours, colors)
                if singleton:
                    output = ForwardcheckWithSingletonPropogation(states, neighbours, colors,
domain)
                    if output != "Failure":
                        return "Success"
                # Restore the domain
                domain = prevDomain
            colors[currentState] = 'Nil'
            # Add the state back since it was a failed assignment
            states.append(currentState)
    if colors[currentState] == 'Nil':
        noOfBacktracks = noOfBacktracks + 1
        return "Failure"

def reduceDomain(color, currentNeighbors, colors):
    for neighbor in currentNeighbors:
```

```
        if colors[neighbor] == 'Nil' and color in domain[neighbor]:
            domain[neighbor].remove(color)


def check(color, currentNeighbors, colors):
    for neighbor in currentNeighbors:
        if colors[neighbor] == 'Nil' and color in domain[neighbor]:
            if len(domain[neighbor]) == 1:
                return True
    return False

def reduceSingletonDomain(currentNeighbors, neighbors, colors):
    reduceStates = []
    for neighbor in currentNeighbors:
        if len(domain[neighbor]) == 1 and colors[neighbor] == 'Nil':
            reduceStates.append(neighbor)

    while reduceStates:
        state = reduceStates.pop(0)
        for neighbor in neighbors[state]:
            if colors[neighbor] == 'Nil' and domain[state][0] in domain[neighbor]:
                domain[neighbor].remove(domain[state][0])
                if len(domain[neighbor]) == 0:
                    return False
                if len(domain[neighbor]) == 1:
                    reduceStates.append(neighbor)
    return True

def minRemainingValueHeuristic(states, domain, neighbours):
    states.sort(key=lambda x: (len(domain[x]),-len(neighbours[x])))
    currentSelection = states[0]
    return currentSelection

def leastConstrainingValueHeuristic(currentState, domain, neighbors):
    currentDomain = domain[currentState]
    currentNeighbors = neighbors[currentState]
    orderedDomain ={}
    for color in currentDomain:
        count = 0
        for neighbor in currentNeighbors:
            if color in domain[neighbor]:
                count = count + 1
        orderedDomain[color] = count

    # Sort
    orderedDomain = dict(sorted(orderedDomain.items(), key=lambda item: item[1]))
    return list(orderedDomain.keys())
```

```
def BacktrackWithHeuristics(states, neighbours, colors, domain):
    global noOfBacktracks
    if all(value != 'Nil' for value in colors.values()):
        return "Success"
    # Use minimum heuristics( and degree heuristics ) to select next unassigned variable
    currentState = minRemainingValueHeuristic(states, domain, neighbours)
    currentNeighbors = neighbours[currentState]
    occupiedColors = list( map(colors.get, currentNeighbors))
    # Use LCV heuristic to get the color
    orderedDomain = leastConstrainingValueHeuristic(currentState, domain, neighbours)
    for color in orderedDomain:
        if color not in occupiedColors:
            # assign consistent color
            colors[currentState] = color
            # Temporarily remove currentState
            states.remove(currentState)
            output = BacktrackWithHeuristics(states, neighbours, colors, domain)
            if output != "Failure":
                return "Success"
            colors[currentState] = 'Nil'
            # add currentState back since assignment failed
            states.append(currentState)
    if colors[currentState] == 'Nil':
        noOfBacktracks = noOfBacktracks + 1
        return "Failure"

def ForwardcheckWithHeuristics(states, neighbours, colors, domain):
    global noOfBacktracks
    if all(value != 'Nil' for value in colors.values()):
        return "Success"
    # Use minimum heuristics( and degree heuristics ) to select next unassigned variable
    currentState = minRemainingValueHeuristic(states, domain, neighbours)
    currentNeighbors = neighbours[currentState]
    output = 'Success'
    occupiedColors = list( map(colors.get, currentNeighbors))
    if 'Nil' in occupiedColors:
        occupiedColors.remove('Nil')
    # Use LCV heuristic to get the color
    orderedDomain = leastConstrainingValueHeuristic(currentState, domain, neighbours)
    for color in orderedDomain:
        if color not in occupiedColors:
            # assign consistent color
            colors[currentState] = color
            # Temporarily remove currentState
            states.remove(currentState)
            # check if any domain can be reduced
            result = check(color,currentNeighbors, colors)
            if not result:
```

```
                prevDomain = copy.deepcopy(domain)
                # Reduce domain
                reduceDomain(color, currentNeighbors, colors)
                output = ForwardcheckWithHeuristics(states, neighbours, colors, domain)
                if output != "Failure":
                    return "Success"
                # Restore domain
                domain = prevDomain
            colors[currentState] = 'Nil'
            # add currentState back since assignment failed
            states.append(currentState)
    if colors[currentState] == 'Nil':
        noOfBacktracks = noOfBacktracks + 1
        return "Failure"


def ForwardcheckWithSingletonPropogationAndHeuristics(states, neighbours, colors, domain):
    global noOfBacktracks
    if all(value != 'Nil' for value in colors.values()):
        return "Success"
    # Use minimum heuristics( and degree heuristics ) to select next unassigned variable
    currentState = minRemainingValueHeuristic(states, domain, neighbours)
    currentNeighbors = neighbours[currentState]
    output = 'Success'
    occupiedColors = list( map(colors.get, currentNeighbors))
    if 'Nil' in occupiedColors:
        occupiedColors.remove('Nil')
    # Use LCV heuristic to get the color
    orderedDomain = leastConstrainingValueHeuristic(currentState, domain, neighbours)
    for color in orderedDomain:
        if color not in occupiedColors:
            # assign consistent color
            colors[currentState] = color
            # Temporarily remove currentState
            states.remove(currentState)
            # check if any domain can be reduced
            result = check(color,currentNeighbors, colors)
            if not result:
                prevDomain = copy.deepcopy(domain)
                # Reduce domain
                reduceDomain(color, currentNeighbors, colors)
                # Aplly singleton propogation
                singleton = reduceSingletonDomain(currentNeighbors, neighbours, colors)
                if singleton:
                    output = ForwardcheckWithSingletonPropogationAndHeuristics(states,
neighbours, colors, domain)
                    if output != "Failure":
                        return "Success"
```

```
                # Rsstore domain if filure occurs
                domain = prevDomain
            colors[currentState] = 'Nil'
            # add currentState back since assignment failed
            states.append(currentState)
    if colors[currentState] == 'Nil':
        noOfBacktracks = noOfBacktracks + 1
        return "Failure"

#Function to get the minimum chromatic number of the map
def getChromaticNumber(states, neighbors):
    copyStates = copy.deepcopy(states)
    count = 0
    while 1:
        count = count + 1
        copyStates = copy.deepcopy(states)
        colors = initColorDict(states)
        domain = initDomain(states, count)
        result = Backtrack(copyStates, neighbors, colors, domain)
        if result == 'Success':
            break
    return count

def checkConstraint(state, neighbors, color, colors):
        for neighbor in neighbors[state]:
            if colors[neighbor] == color:
                return False
        return True

# Function to get elapsed time for a particular run
def showTime():
    endTime = datetime.now()
    elapsedTime = (endTime - startTime).microseconds / 1000
    print("Elapsed Time: %sms" % (str(elapsedTime)))

#Function to color the given map
def plotMap(country,colors):
    # url of our shape file - USA
    print("Ready to plot map for USA")
    if country == 'USA':
        path="C:/code/states_21basic/"
        # # load the shape file using geopandas
        states = geopandas.read_file(path+'states.shp')
        states = states.to_crs("EPSG:3395")
        ax2 = states.boundary.plot(figsize=(12,12), edgecolor=u'gray')
        if colors is not None:
            for k,v in colors.items():
                if v == 'R':
```

```
                    states[states.STATE_NAME == k].plot(edgecolor=u'gray',
color='red',ax=ax2)
                elif v == 'B':
                    states[states.STATE_NAME == k].plot(edgecolor=u'gray',
color='blue',ax=ax2)
                elif v == 'G':
                    states[states.STATE_NAME == k].plot(edgecolor=u'gray',
color='green',ax=ax2)
                else :
                    states[states.STATE_NAME == k].plot(edgecolor=u'gray',
color='yellow',ax=ax2)

            plt.show()
    else :
        print("Ready to plot map for AUS")
        path="C:/code/aus_basic/"
        # # load the shape file using geopandas
        states = geopandas.read_file(path+'STE_2016_AUST.shp')
        states = states.to_crs("EPSG:3395")
        ax2 = states.boundary.plot(figsize=(12,12), edgecolor=u'gray')
        if colors is not None:
            for k,v in colors.items():
                if v == 'R':
                    states[states.STE_NAME16 == k].plot(edgecolor=u'gray',
color='red',ax=ax2)
                elif v == 'B':
                    states[states.STE_NAME16 == k].plot(edgecolor=u'gray',
color='blue',ax=ax2)
                elif v == 'G':
                    states[states.STE_NAME16 == k].plot(edgecolor=u'gray',
color='green',ax=ax2)
                else :
                    states[states.STE_NAME16 == k].plot(edgecolor=u'gray',
color='yellow',ax=ax2)

            plt.show()


print("Choose map: \n1. Australia \n2. USA")
mapChoice = int(input())
print("Select if you want to run with Heuristics / without heursitics: \n1 for without
heuristic \n2 for with heuristic")
runWithHeuristic = int(input())
print("Select \n1 for Depth first search only  \n2 for Depth first search + forward checking
\n3 for Depth first search + forward checking + propagation through singleton domains")
algorithm = int(input())
startTime = datetime.now()
if(mapChoice == 1):
```

```
    states = copy.deepcopy(states_australia)
    neighbours = neighbours_australia
    min_number = getChromaticNumber(states, neighbours)
    print("Minimum no of colors required for Australia map: ", min_number)
    if(runWithHeuristic==1):
        if(algorithm == 1):
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            noOfBacktracks = 0
            result = Backtrack(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('AUS',colors)
            else:
                print("Failure")
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
        elif(algorithm == 2):
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = Forwardcheck(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('AUS',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
        else:
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = ForwardcheckWithSingletonPropogation(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('AUS',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
```

```
                showTime()
    else:
        if(algorithm == 1):
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = BacktrackWithHeuristics(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('AUS',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
        elif(algorithm == 2):
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = ForwardcheckWithHeuristics(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('AUS',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
        else:
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = ForwardcheckWithSingletonPropogationAndHeuristics(states, neighbours,
colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('AUS',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
```

```
elif(mapChoice == 2):
    states = copy.deepcopy(states_usa)
    neighbours = neighbours_usa
    min_number = getChromaticNumber(states, neighbours)
    print("Minimum no of colors required for USA map: ", min_number)
    if(runWithHeuristic==1):
        if(algorithm == 1):
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            noOfBacktracks = 0
            result = Backtrack(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('USA',colors)
            else:
                print("Failure")
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()

        elif(algorithm == 2):
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = Forwardcheck(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('USA',colors)
            else:
                print("Failure")
                print(colors)
                showTime()
                print("No. of Backtracks: ", noOfBacktracks)
        else:
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = ForwardcheckWithSingletonPropogation(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('USA',colors)
            else:
                print("Failure")
```

```
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
    else:
        if(algorithm == 1):
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = BacktrackWithHeuristics(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('USA',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
        elif(algorithm == 2):
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = ForwardcheckWithHeuristics(states, neighbours, colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('USA',colors)
            else:
                print("Failure")
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
        else:
            noOfBacktracks = 0
            colors = initColorDict(states)
            domain = initDomain(states, min_number)
            result = ForwardcheckWithSingletonPropogationAndHeuristics(states, neighbours,
colors, domain)
            if result == 'Success':
                print(colors)
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
                plotMap('USA',colors)
            else:
                print("Failure")
                print(colors)
```

```
                print("No. of Backtracks: ", noOfBacktracks)
                showTime()
else:
    print("Choose correct map.")
    exit()
```

Output: Australia Map

Without Heuristic

Algorithm	Time Required (in milliseconds)	Number of Backtracks
Depth first search only	3.99	2
Depth first search with forward checking	2	0
Depth first search with forward checking and propagation through singleton domains	5	0

With Heuristic

Algorithm	Time Required (in milliseconds)	Number of Backtracks
Depth first search only	2.5	0
Depth first search with forward checking	3.8	0
Depth first search with forward checking and propagation through singleton domains	3.977	0

Sample Input/Output

Choose map:

1. Australia

2. USA

1

Select if you want to run with Heuristics / without heuristic:

1 for without heuristic

2 for with heuristic

2

Select

1 for Depth first search only

2 for Depth first search + forward checking

3 for Depth first search + forward checking + propagation through singleton domains

3

Minimum no of colors required for Australia map: 3

{'South Australia': 'R', 'Queensland': 'G', 'Victoria': 'G', 'New South Wales': 'B', 'Western Australia': 'G', 'Tasmania': 'R', 'Northern Territory': 'B'}

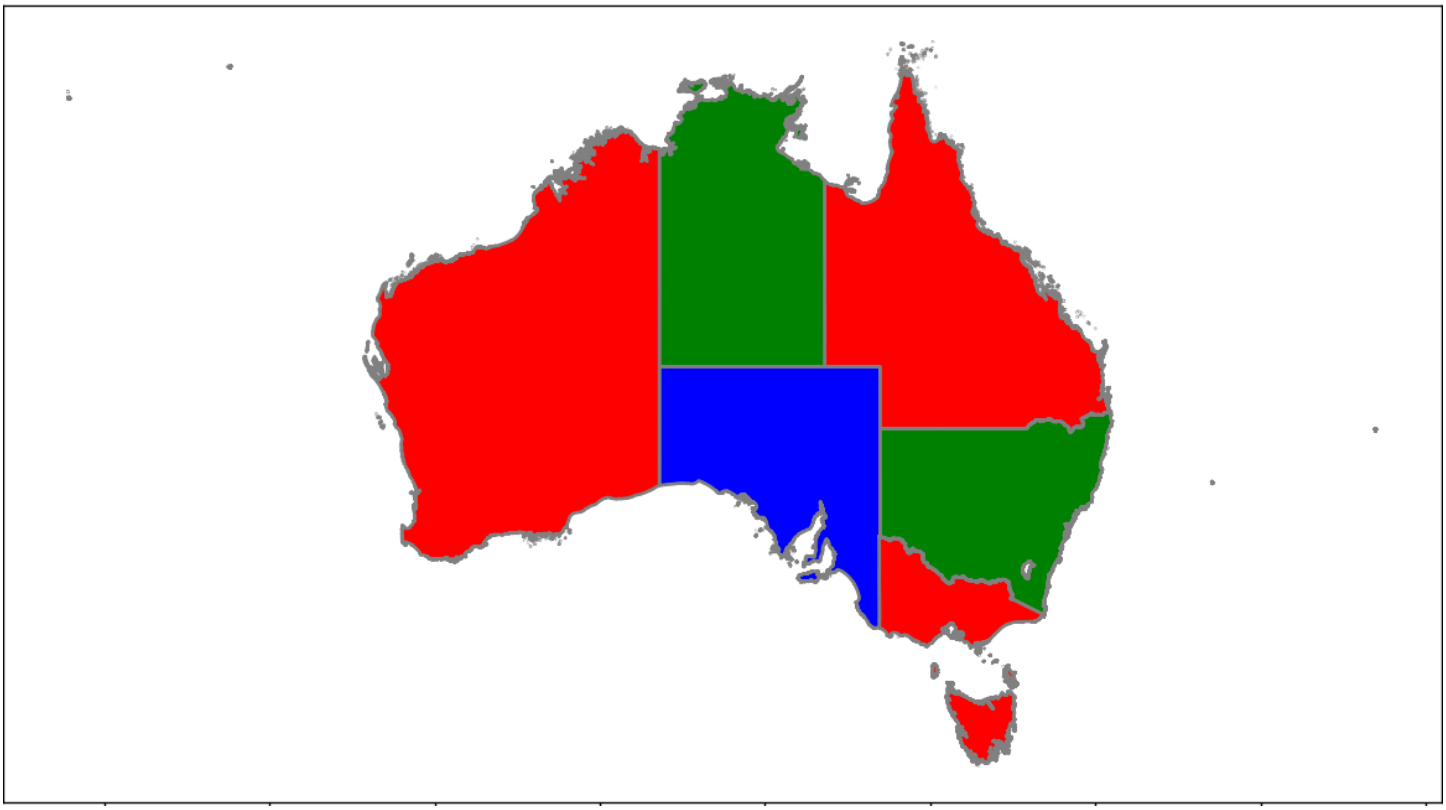
No. of Backtracks: 0

Elapsed Time: 3.977ms

Ready to plot map for USA

Ready to plot map for AUS

Graph UI Output:



Output: USA Map

Without Heuristic

Algorithm	Time Required (in milliseconds)	Number of Backtracks
Depth first search only	260	17054
Depth first search with forward checking	680.16	2124
Depth first search with forward checking and propagation through singleton domains	712.445	896

With Heuristic

Algorithm	Time Required (in milliseconds)	Number of Backtracks
Depth first search only	308.08	20
Depth first search with forward checking	427.45	0
Depth first search with forward checking and propagation through singleton domains	727.48	0

Sample Input/Output

Choose map:

1. Australia

2. USA

2

Select if you want to run with Heuristics / without heursitics:

1 for without heuristic

2 for with heuristic

2

Select

1 for Depth first search only

2 for Depth first search + forward checking

3 for Depth first search + forward checking + propagation through singleton domains

3

Minimum no of colors required for USA map: 4

{'Alabama': 'B', 'Alaska': 'R', 'Arkansas': 'B', 'Arizona': 'Y', 'California': 'R', 'Colorado': 'R', 'Connecticut': 'B', 'Delaware': 'R', 'Florida': 'G', 'Georgia': 'R', 'Hawaii': 'R', 'Iowa': 'B', 'Idaho': 'R', 'Illinois': 'G', 'Indiana': 'Y', 'Kansas': 'B', 'Kentucky': 'B', 'Louisiana': 'G', 'Massachusetts': 'G', 'Maryland': 'Y', 'Maine': 'G', 'Michigan': 'B', 'Minnesota': 'G', 'Missouri': 'R', 'Mississippi': 'R', 'Montana': 'G', 'North Carolina': 'B', 'North Dakota': 'B', 'Nebraska': 'G', 'New Hampshire': 'R', 'New Jersey': 'G', 'New Mexico': 'B', 'Nevada': 'B', 'New York': 'R', 'Ohio': 'R', 'Oklahoma': 'G', 'Oregon': 'G', 'Pennsylvania': 'B',

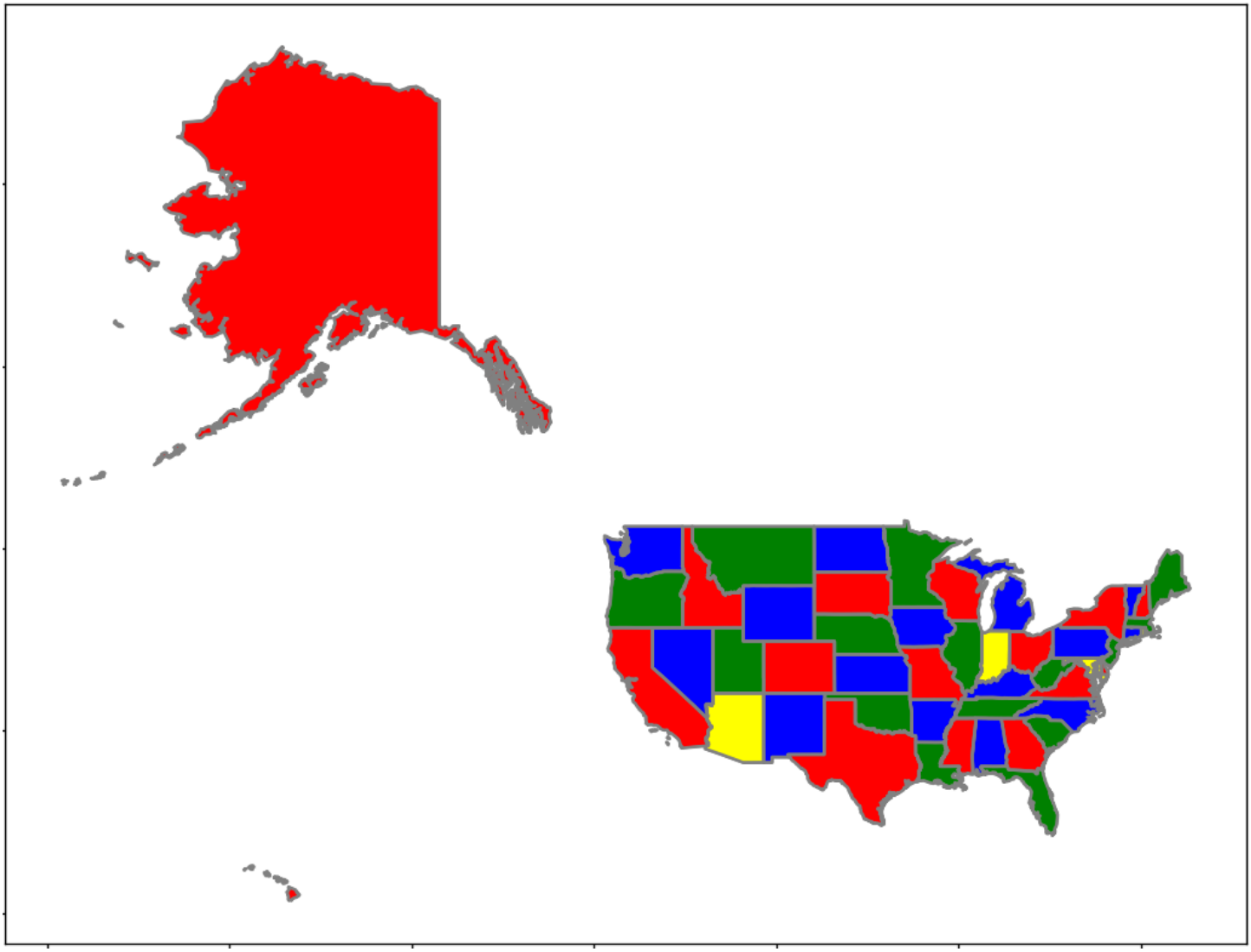
'Rhode Island': 'Y', 'South Carolina': 'G', 'South Dakota': 'R', 'Tennessee': 'G', 'Texas': 'R', 'Utah': 'G', 'Virginia': 'R', 'Vermont': 'B', 'Washington': 'B', 'Wisconsin': 'R', 'West Virginia': 'G', 'Wyoming': 'B'}

No. of Backtracks: 0

Elapsed Time: 348.093ms

Ready to plot map for USA

Graph UI Output



Conclusion

Thus, we can conclude that using heuristics improves the performance but increases the runtime as we need to keep track of other variables.

The chromatic number gives the minimum number of colors required to solve a particular map. In our case,

Australia needs 3 colors hence the chromatic number is 3 and the USA map needs 4 colors, so the chromatic number is 4.

Steps to run the program

Prerequisite:

Download the code folder from canvas

Unzip the code folder and store it on C Drive folder ONLY (otherwise update the path in code on line 407 and 426. Search for path variable.

You can use either python 3.9 or 3.10.

If you are using 3.9, use the files named as cp39 when installing the dependencies

If you are using 3.10, use the files named as cp310 when installing the dependencies

Steps to run on a windows machine

1. Install Python version 3+

2. Install latest version of PIP package installer

3. Install latest version of numpy using PIP (pip install numpy)

4. Install latest version of pandas using PIP (pip install pandas)

5. Install latest version of matplotlib using PIP(pip install matplotlib)

6. Install geopandas and its dependencies as follows

a. Unzip dependencies folder from main code folder

b. Go to the folder where the binaries are extracted.

c. Open cmd as administrator from the extracted folder

d. The following order of installation using pip install is necessary. Be careful with the filename. It should work if the filename is correct: (Tip: Type “pip install” followed by a space and type the first two letters of the binary and press Tab. (e.g. pip install gd(press Tab))

pip install GDAL-3.3.3-cp39-cp39-win_amd64.whl

pip install pyproj-3.3.0-cp39-cp39-win_amd64.whl

pip install Fiona-1.8.20-cp39-cp39-win_amd64.whl

pip install Shapely-1.8.0-cp39-cp39-win_amd64.whl

pip install geopandas-0.10.2-py2.py3-none-any

7. Open the command prompt where the python code file is stored

8. Run py mapcoloring.py

Note: If you get any error, please feel free to reach any of the group member.