

# PROJECT DOCUMENTATION REPORT

**ON**

**SOLVING Map Coloring Problem using CSP**

**in the context of USA and Australia maps.**

**PROGRAMMING PROJECT 3**

**ITCS 6150 - Intelligent Systems**

Project Guidance By

**Dewan T. Ahmed, Ph.D.**

**SUBMITTED BY:**

|  |  |  |
| --- | --- | --- |
| **Priyanka Pandit Jadhav (801167642)** | **Maheswara Rao Lekkala (801167702)** | **Bharadwaj Tirunagaru (801167910)** |
| [**pjadhav7@uncc.edu**](mailto:pjadhav7@uncc.edu) | [**mlekkala@uncc.edu**](mailto:mlekkala@uncc.edu) | [**btirunag@uncc.edu**](mailto:btirunag@uncc.edu) |
| **704-905-1666** | **980-833-5010** | **980-729-4465** |

**Date: 04/23/2020**

**MAP COLORING PROBLEM STATEMENT:**

**INTRODUCTION TO CSP:**

A CSP is a problem composed of a finite set of variables each of which has a finite domain of values and a set of constraints. Each constraint is defined over some subset of the original set of variables and restricts the values these variables can simultaneously take. The task is to find an assignment of a value for each variable such that the assignments satisfy all the constraints in some problems the goal is to find all such assignments. Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search.

**GRAPH COLORING:**

Given a graph G = (V, E) and an integer k, a k-coloring of G is a one-one mapping of vertices to colors, such that adjacent vertices are assigned to different colors. The Minimum Graph Coloring Problem (Min–GCP) consists in finding the minimum k such that a k-coloring exists. Such minimum k is known as the chromatic number of G and is denoted by χ(G), or simply by χ. Min–GCP is NP-hard. The chromatic number is bounded from below by the size of the maximum clique of G, known as the clique number ω(G) which is equal to χ(G) when G is a perfect graph. Chromatic number is the minimum number of colors required to satisfy the constraint that adjacent vertices do not have same color.

**MAP COLORING:**

The Map coloring problem is like the graph colouring problem. In map coloring the constraint is that states which are adjacent to each other i.e. share a border should not have the same color. In the case for map colouring of Australia the chromatic number is 3 meaning if number of colors used is less than 3 then we would have some states which cannot be assigned colors as their domain size would reduce to 0. In the case of USA map colouring chromatic number is 4.

**MAP COLORING SOLUTION APPROACHES:**

Thereare 3 approaches we have used in this project DFS, DFS with forward checking, DFS with forward checking and propagation through singleton domain.

**Depth first search (BACKTRACKING):**

**T**he concept in backtracking applied to map coloring is that once any variables domain reaches 0 then the algorithm goes back to previous states and see if using other options in the domain would yield a color for the one with the empty set in its domain. Every time the algorithm checks the next state only after reaching there, there are no prechecks done.

**BACKTRACKING WITH FORWARD CHECKING:**

The concept here is exactly same as backtracking only that next check is pre-checked making the algorithm smarter than Just BACKTRACKING. Number of backtracks are significantly reduced.

**BACKTRACKING WITH FORWARD CHECKING THROUGH SINGLETON DOMAINS:**

Here the algorithmchecks among all possibilities of next states and choses the one with domain value equal to 1 and propagates to the next unassigned variables from the one with domain =1. Number of backtracks are further reduced and the algorithm is relatively faster.

**HEURISTICS USED:**

There may be several options or nodes to choose from the next states. Any one of the states may be chosen at random and we could progress the map colouring algorithm. With using heuristics, we get an order of choosing the variables depending on some factors. Some of the most used heuristics are as follows below.

1. **MINIMUM REMAINING VALUES:**

In this heuristic propagation follows in the order of those nodes with least number of values in its domain. With respect to map coloring problem If one state has 2 permissible values in its domain and another state has 3 then the state with 2 values would be chosen first , here permissible refers to reducing domain size because of constraints imposed that adjacent states cannot have same color.

2. **DEGREE HEURISTIC:**

The idea here is assign a value to the variable that is involved in the largest number of constraints on other unassigned variables. It is often used as a means to reduce the number of same next possibilities ie as a tie breaker with Minimum remaining values heuristic to choose the best next when all next nodes have the same number of domain values after a variable assignment is done using MRV.

3. **LEAST CONSTRAINING VALUE:**

Here the chosen heuristic rules out the fewest values in the remaining variables.

Variables used:

counterForBacktrack: to store the total number of backtracks

staring\_time and ending\_time: to calculate total run duration

color\_dictionary and m: chromatic number and respective colors

State\_dictionary: list of states in usa or aus

result\_dictionary: color assigned as program procedes

**Code:**

WithoutHeuristics

# Python program for solution of M Coloring

# problem using dfs\_backtracking

from datetime import datetime

print(*"Please Enter the Country\nAmerica / Australia"*)

starting\_time = datetime.now()

country = input()

counterForBacktrack = 0

class **dfs\_backtracking**():

def **\_\_init\_\_**(*self*, vertices):

*self*.V = vertices

*self*.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def **check\_is\_safe**(*self*, v, colour, c):

for i in range(*self*.V):

if *self*.graph[v][i] == 1 and colour[i] == c:

return False

return True

def **graph\_colour**(*self*, m, colour, v):

global counterForBacktrack

if v == *self*.V:

return True

for c in range(1, m+1):

if *self*.check\_is\_safe(v, colour, c) == True:

colour[v] = c

if *self*.graph\_colour(m, colour, v+1) == True:

return True

colour[v] = 0

else:

counterForBacktrack+=1

def **graph\_colouring**(*self*, m):

colour = [0] \* *self*.V

if *self*.graph\_colour(m, colour, 0) == False:

return False

# Print the solution

print(*"Solution exist and Following are the assigned colours:"*)

for idx, val in enumerate(colour):

result\_dictionary[state\_dictionary[str(idx+1)]] = color\_dictionary[str(val)]

return True

if country == *'America'*:

change\_position=[[18,24,25,36,42,43],[9,10,24,42],[],[5,6,28,31,44],[21,32,39],[3,28,37],[3,16,27,31,34,36,50],[20,30,38],[1,10],[1,9,33,40,42],[],[26,28,37,44,47,50],[14,15,17,22,25,49],[13,17,22,35],[13,23,25,27,41,49],[6,25,27,36],[13,14,25,35,42,46,48],[4,24,43],[29],[8,38,46,48],[7,29,32,39,45],[13,14,23,35,49],[15,22,34,41,49],[1,18,42,43],[4,13,15,16,17,27,36,42],[12,34,41,50],[6,15,16,25,41,50],[3,5,12,37,44],[19,21,45],[8,32,38],[3,6,36,43,44],[7,21,30,38,39,45],[10,23,26,40,41,42,46],[23,26,41],[14,17,22,38,48],[4,6,16,25,31,43],[5,12,28,47],[8,20,30,32,35,48],[7,21,32],[10,33],[15,23,26,27,34,50],[1,4,10,17,24,25,33],[4,18,31,36],[3,6,12,28,31,50],[21,29,32],[17,20,33,42,48],[12,37],[17,20,35,38,46],[13,15,20,22,23],[6,12,26,27,41,44]]

positions = []

for i in range(0,50) :

individual=[]

for j in range(0,50) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary={}

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

state\_dictionary[*"1"*] = *"AL"*

state\_dictionary[*"2"*] = *"AK"*

state\_dictionary[*"3"*] = *"AZ"*

state\_dictionary[*"4"*] = *"AR"*

state\_dictionary[*"6"*] = *"CO"*

state\_dictionary[*"5"*] = *"CA"*

state\_dictionary[*"7"*] = *"CT"*

state\_dictionary[*"8"*] = *"DE"*

state\_dictionary[*"9"*] = *"FL"*

state\_dictionary[*"10"*] = *"GA"*

state\_dictionary[*"11"*] = *"HI"*

state\_dictionary[*"12"*] = *"ID"*

state\_dictionary[*"13"*] = *"IL"*

state\_dictionary[*"14"*] = *"IN"*

state\_dictionary[*"15"*] = *"IA"*

state\_dictionary[*"16"*] = *"KS"*

state\_dictionary[*"17"*] = *"KY"*

state\_dictionary[*"18"*] = *"LA"*

state\_dictionary[*"19"*] = *"ME"*

state\_dictionary[*"20"*] = *"MD"*

state\_dictionary[*"21"*] = *"MA"*

state\_dictionary[*"22"*] = *"MI"*

state\_dictionary[*"23"*] = *"MN"*

state\_dictionary[*"24"*] = *"MS"*

state\_dictionary[*"25"*] = *"MO"*

state\_dictionary[*"26"*] = *"MT"*

state\_dictionary[*"27"*] = *"NE"*

state\_dictionary[*"28"*] = *"NV"*

state\_dictionary[*"29"*] = *"NH"*

state\_dictionary[*"30"*] = *"NJ"*

state\_dictionary[*"31"*] = *"NM"*

state\_dictionary[*"32"*] = *"NY"*

state\_dictionary[*"33"*] = *"NC"*

state\_dictionary[*"34"*] = *"ND"*

state\_dictionary[*"35"*] = *"OH"*

state\_dictionary[*"36"*] = *"OK"*

state\_dictionary[*"37"*] = *"OR"*

state\_dictionary[*"38"*] = *"PA"*

state\_dictionary[*"39"*] = *"RI"*

state\_dictionary[*"40"*] = *"SC"*

state\_dictionary[*"41"*] = *"SD"*

state\_dictionary[*"42"*] = *"TN"*

state\_dictionary[*"43"*] = *"TX"*

state\_dictionary[*"44"*] = *"UT"*

state\_dictionary[*"45"*] = *"VT"*

state\_dictionary[*"46"*] = *"VA"*

state\_dictionary[*"47"*] = *"WA"*

state\_dictionary[*"48"*] = *"WV"*

state\_dictionary[*"49"*] = *"WI"*

state\_dictionary[*"50"*] = *"WY"*

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

# Driver Code

g = dfs\_backtracking(50)

g.graph = positions

m=4 ## chromataic number

g.graph\_colouring(m)

if country == *'Australia'*:

change\_position = [[3,4,6],[3,4,7],[1,2,4],[1,2,3,4,6,7],[],[1,4],[2,4]]

positions = []

for i in range(0,7) :

individual=[]

for j in range(0,7) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary = {}

state\_dictionary[*"1"*] = *"New South Wales"*

state\_dictionary[*"2"*] = *"Northern Territory"*

state\_dictionary[*"3"*] = *"Queensland"*

state\_dictionary[*"4"*] = *"South Australia"*

state\_dictionary[*"5"*] = *"Tasmania"*

state\_dictionary[*"6"*] = *"Victoria"*

state\_dictionary[*"7"*] = *"Western Australia"*

color\_dictionary = {*"1"*: *"red"*, *"2"*: *"green"*, *"3"*: *"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

# Driver Code

g = dfs\_backtracking(7)

g.graph = positions

m=3 ## chromataic number

g.graph\_colouring(m)

ending\_time = datetime.now()

difference = ending\_time - starting\_time

print(*"THE TOTAL TIME TOOK FOR EXECUTION ----------------"*,str(difference.total\_seconds()))

print(*"The number of dfs\_backtracking steps happened:"*+str(counterForBacktrack))

for key, value in result\_dictionary.items():

print(*"{} ==> {}"*.format(key,value))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# problem using dfs backtracking and forward Checking

from datetime import datetime

print(*"Please Enter the Country\nAmerica / Australia"*)

starting\_time = datetime.now()

country = input()

counterForBacktrack = 0

class **dfs\_forwardchecking**():

def **\_\_init\_\_**(*self*, vertices):

*self*.V = vertices

*self*.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def **check\_is\_safe**(*self*, v, colour, c):

for i in range(*self*.V):

if *self*.graph[v][i] == 1 and colour[i] == c:

return False

return True

def **get\_the\_neighbors**(*self*, state):

neighbors = []

for i in range(*self*.V):

if *self*.graph[state][i] == 1:

neighbors.append(i)

return neighbors

# A recursive utility function to solve m

def **garph\_colour**(*self*, m, colour, v):

global counterForBacktrack

try:

if v == *self*.V:

return True

if not domain\_dictionary[v+1]:

return False

for c in domain\_dictionary[v+1]:

if *self*.check\_is\_safe(v, colour, c) == True:

colour[v] = c

neighbors = *self*.get\_the\_neighbors(v)

for neighbor in neighbors:

if c in domain\_dictionary[neighbor+1]:

domain\_dictionary[neighbor+1].remove(c)

if *self*.garph\_colour(m, colour, v+1) == True:

return True

for neighbor in neighbors:

a = neighbor+1

if c not in domain\_dictionary[a]:

domain\_dictionary[a].append(c)

domain\_dictionary[a].sort()

colour[v] = 0

else:

counterForBacktrack+=1

except Exception as e:

print(*"something wrong"*, e)

def **graph\_colouring**(*self*, m):

colour = [0] \* *self*.V

if *self*.garph\_colour(m, colour, 0) == False:

return False

# Print the solution

print(*"Solution exist and Following are the colours that we have assinged:"*)

for idx, val in enumerate(colour):

result\_dictionary[state\_dictionary[str(idx+1)]] = color\_dictionary[str(val)]

return True

def **dictionary\_of\_domain**(n):

for key, value in enumerate(state\_dictionary):

integers = list(range(1,n))

domain\_dictionary[key+1] = integers

if country == *'America'*:

change\_position=[[9,10,24,42],[],[5,6,28,31,44],[18,24,25,36,42,43],[3,28,37],[3,16,27,31,34,36,50],[21,32,39],[20,30,38],[1,10],[1,9,33,40,42],[],[26,28,37,44,47,50],[14,15,17,22,25,49],[13,17,22,35],[13,23,25,27,41,49],[6,25,27,36],[13,14,25,35,42,46,48],[4,24,43],[29],[8,38,46,48],[7,29,32,39,45],[13,14,23,35,49],[15,22,34,41,49],[1,18,42,43],[4,13,15,16,17,27,36,42],[12,34,41,50],[6,15,16,25,41,50],[3,5,12,37,44],[19,21,45],[8,32,38],[3,6,36,43,44],[7,21,30,38,39,45],[10,23,26,40,41,42,46],[23,26,41],[14,17,22,38,48],[4,6,16,25,31,43],[5,12,28,47],[8,20,30,32,35,48],[7,21,32],[10,33],[15,23,26,27,34,50],[1,4,10,17,24,25,33],[4,18,31,36],[3,6,12,28,31,50],[21,29,32],[17,20,33,42,48],[12,37],[17,20,35,38,46],[13,15,20,22,23],[6,12,26,27,41,44]]

positions = []

for i in range(0,50) :

individual=[]

for j in range(0,50) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary={}

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

state\_dictionary[*"1"*] = *"AL"*

state\_dictionary[*"2"*] = *"AK"*

state\_dictionary[*"3"*] = *"AZ"*

state\_dictionary[*"4"*] = *"AR"*

state\_dictionary[*"6"*] = *"CO"*

state\_dictionary[*"5"*] = *"CA"*

state\_dictionary[*"7"*] = *"CT"*

state\_dictionary[*"8"*] = *"DE"*

state\_dictionary[*"9"*] = *"FL"*

state\_dictionary[*"10"*] = *"GA"*

state\_dictionary[*"11"*] = *"HI"*

state\_dictionary[*"12"*] = *"ID"*

state\_dictionary[*"13"*] = *"IL"*

state\_dictionary[*"14"*] = *"IN"*

state\_dictionary[*"15"*] = *"IA"*

state\_dictionary[*"16"*] = *"KS"*

state\_dictionary[*"17"*] = *"KY"*

state\_dictionary[*"18"*] = *"LA"*

state\_dictionary[*"19"*] = *"ME"*

state\_dictionary[*"20"*] = *"MD"*

state\_dictionary[*"21"*] = *"MA"*

state\_dictionary[*"22"*] = *"MI"*

state\_dictionary[*"23"*] = *"MN"*

state\_dictionary[*"24"*] = *"MS"*

state\_dictionary[*"25"*] = *"MO"*

state\_dictionary[*"26"*] = *"MT"*

state\_dictionary[*"27"*] = *"NE"*

state\_dictionary[*"28"*] = *"NV"*

state\_dictionary[*"29"*] = *"NH"*

state\_dictionary[*"30"*] = *"NJ"*

state\_dictionary[*"31"*] = *"NM"*

state\_dictionary[*"32"*] = *"NY"*

state\_dictionary[*"33"*] = *"NC"*

state\_dictionary[*"34"*] = *"ND"*

state\_dictionary[*"35"*] = *"OH"*

state\_dictionary[*"36"*] = *"OK"*

state\_dictionary[*"37"*] = *"OR"*

state\_dictionary[*"38"*] = *"PA"*

state\_dictionary[*"39"*] = *"RI"*

state\_dictionary[*"40"*] = *"SC"*

state\_dictionary[*"41"*] = *"SD"*

state\_dictionary[*"42"*] = *"TN"*

state\_dictionary[*"43"*] = *"TX"*

state\_dictionary[*"44"*] = *"UT"*

state\_dictionary[*"45"*] = *"VT"*

state\_dictionary[*"46"*] = *"VA"*

state\_dictionary[*"47"*] = *"WA"*

state\_dictionary[*"48"*] = *"WV"*

state\_dictionary[*"49"*] = *"WI"*

state\_dictionary[*"50"*] = *"WY"*

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(5)

# Driver Code

g = dfs\_forwardchecking(50)

g.graph = positions

m=4 ## chromataic number

if country == *'Australia'*:

change\_position = [[3,4,6],[3,4,7],[1,2,4],[1,2,3,4,6,7],[],[1,4],[2,4]]

positions = []

for i in range(0,7) :

individual=[]

for j in range(0,7) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary = {}

state\_dictionary[*"1"*] = *"New South Wales"*

state\_dictionary[*"2"*] = *"Northern Territory"*

state\_dictionary[*"3"*] = *"Queensland"*

state\_dictionary[*"4"*] = *"South Australia"*

state\_dictionary[*"5"*] = *"Tasmania"*

state\_dictionary[*"6"*] = *"Victoria"*

state\_dictionary[*"7"*] = *"Western Australia"*

color\_dictionary = {*"1"*: *"red"*, *"2"*: *"green"*, *"3"*: *"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(4)

# Driver Code

g = dfs\_forwardchecking(7)

g.graph = positions

m=3 ## chromataic number

g.graph\_colouring(m)

ending\_time = datetime.now()

difference = ending\_time - starting\_time

print(*"THE TOTAL TIME TOOK FOR EXEC ----------------"*,str(difference.total\_seconds()))

print(*"The number of backtracking steps happened:"*+str(counterForBacktrack))

for key, value in result\_dictionary.items():

print(*"{} => {}"*.format(key,value))

# problem using dfs backtracking forward checking and singleton

import random

from datetime import datetime

print(*"Please Enter the Country\nAmerica / Australia"*)

starting\_time = datetime.now()

country = input()

counterForBacktrack = 0

class **dfs\_singleton**():

def **\_\_init\_\_**(*self*, vertex):

*self*.V = vertex

*self*.graph = [[0 for column in range(vertex)]

for row in range(vertex)]

def **check\_is\_safe**(*self*, v, colour, c):

for i in range(*self*.V):

if *self*.graph[v][i] == 1 and colour[i] == c:

return False

return True

def **retrieve\_neighbors**(*self*, state):

neighbors = []

for i in range(*self*.V):

if *self*.graph[state][i] == 1:

neighbors.append(i)

return neighbors

def **check\_if\_colored**(*self*, colors):

numberofvertex = 0

for color in colors:

if color != 0:

numberofvertex = numberofvertex + 1

if numberofvertex == 7:

return True

elif numberofvertex == 50:

return True

else:

return False

def **check\_ST\_constraints**(*self*, v):

if *self*.check\_if\_colored(colour):

return True

if v == *self*.V:

v = random.choice([a for a in range(len(colour)) if colour[a] == 0])

def **ST\_removedomain\_variables**(*self*, v):

if c in domain\_dictionary[v+1]:

domain\_dictionary[v+1].remove(c)

if c not in domain\_dictionary[v+1]:

domain\_dictionary[v+1].append(c)

domain\_dictionary[v+1].sort()

def **ST\_logic**(*self*, st\_states):

if len(st\_states)==0:

return

state\_tocolor = -1

for key,value in domain\_dictionary.items():

if len(value) == 1 and colour[key-1] == 0:

state\_tocolor = key

break

if state\_tocolor == -1:

current = random.choice([a for a in range(len(colour)) if colour[a] == 0])

if *self*.graph\_colour(m, colour, current) == True:

return True

else:

if *self*.graph\_colour(m, colour, state\_tocolor-1) == True:

return True

def **graph\_colour**(*self*, m, colour, v):

global counterForBacktrack

try:

if v == *self*.V:

return True

if country==*'Australia'*:

list\_st\_states = list(range(*self*.V%7))

if country==*'America'*:

list\_st\_states = list(range(*self*.V%50))

*self*.ST\_logic(list\_st\_states)

if not domain\_dictionary[v+1]:

return False

for c in domain\_dictionary[v+1]:

if *self*.check\_is\_safe(v, colour, c) == True:

colour[v] = c

neighbors = *self*.retrieve\_neighbors(v)

for neighbor in neighbors:

if c in domain\_dictionary[neighbor+1]:

domain\_dictionary[neighbor+1].remove(c)

if *self*.graph\_colour(m, colour, v+1) == True:

return True

for neighbor in neighbors:

a = neighbor+1

if c not in domain\_dictionary[a]:

domain\_dictionary[a].append(c)

domain\_dictionary[a].sort()

colour[v] = 0

else:

counterForBacktrack+=1

except Exception as e:

print(*"something wrong"*, e)

def **garph\_colouring**(*self*, m):

colour = [0] \* *self*.V

if *self*.graph\_colour(m, colour, 0) == False:

return False

# Print the solution

print(*"Solution exist and Following are the assigned colours:"*)

for idx, val in enumerate(colour):

result\_dictionary[state\_dictionary[str(idx+1)]] = color\_dictionary[str(val)]

return True

def **dictionary\_of\_domain**(n):

for key, value in enumerate(state\_dictionary):

integers = list(range(1,n))

domain\_dictionary[key+1] = integers

if country == *'America'*:

change\_position=[[9,10,24,42],[],[5,6,28,31,44],[18,24,25,36,42,43],[3,28,37],[3,16,27,31,34,36,50],[21,32,39],[20,30,38],[1,10],[1,9,33,40,42],[],[26,28,37,44,47,50],[14,15,17,22,25,49],[13,17,22,35],[13,23,25,27,41,49],[6,25,27,36],[13,14,25,35,42,46,48],[4,24,43],[29],[8,38,46,48],[7,29,32,39,45],[13,14,23,35,49],[15,22,34,41,49],[1,18,42,43],[4,13,15,16,17,27,36,42],[12,34,41,50],[6,15,16,25,41,50],[3,5,12,37,44],[19,21,45],[8,32,38],[3,6,36,43,44],[7,21,30,38,39,45],[10,23,26,40,41,42,46],[23,26,41],[14,17,22,38,48],[4,6,16,25,31,43],[5,12,28,47],[8,20,30,32,35,48],[7,21,32],[10,33],[15,23,26,27,34,50],[1,4,10,17,24,25,33],[4,18,31,36],[3,6,12,28,31,50],[21,29,32],[17,20,33,42,48],[12,37],[17,20,35,38,46],[13,15,20,22,23],[6,12,26,27,41,44]]

positions = []

for i in range(0,50) :

individual=[]

for j in range(0,50) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary={}

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

state\_dictionary[*"1"*] = *"AL"*

state\_dictionary[*"2"*] = *"AK"*

state\_dictionary[*"3"*] = *"AZ"*

state\_dictionary[*"4"*] = *"AR"*

state\_dictionary[*"6"*] = *"CO"*

state\_dictionary[*"5"*] = *"CA"*

state\_dictionary[*"7"*] = *"CT"*

state\_dictionary[*"8"*] = *"DE"*

state\_dictionary[*"9"*] = *"FL"*

state\_dictionary[*"10"*] = *"GA"*

state\_dictionary[*"11"*] = *"HI"*

state\_dictionary[*"12"*] = *"ID"*

state\_dictionary[*"13"*] = *"IL"*

state\_dictionary[*"14"*] = *"IN"*

state\_dictionary[*"15"*] = *"IA"*

state\_dictionary[*"16"*] = *"KS"*

state\_dictionary[*"17"*] = *"KY"*

state\_dictionary[*"18"*] = *"LA"*

state\_dictionary[*"19"*] = *"ME"*

state\_dictionary[*"20"*] = *"MD"*

state\_dictionary[*"21"*] = *"MA"*

state\_dictionary[*"22"*] = *"MI"*

state\_dictionary[*"23"*] = *"MN"*

state\_dictionary[*"24"*] = *"MS"*

state\_dictionary[*"25"*] = *"MO"*

state\_dictionary[*"26"*] = *"MT"*

state\_dictionary[*"27"*] = *"NE"*

state\_dictionary[*"28"*] = *"NV"*

state\_dictionary[*"29"*] = *"NH"*

state\_dictionary[*"30"*] = *"NJ"*

state\_dictionary[*"31"*] = *"NM"*

state\_dictionary[*"32"*] = *"NY"*

state\_dictionary[*"33"*] = *"NC"*

state\_dictionary[*"34"*] = *"ND"*

state\_dictionary[*"35"*] = *"OH"*

state\_dictionary[*"36"*] = *"OK"*

state\_dictionary[*"37"*] = *"OR"*

state\_dictionary[*"38"*] = *"PA"*

state\_dictionary[*"39"*] = *"RI"*

state\_dictionary[*"40"*] = *"SC"*

state\_dictionary[*"41"*] = *"SD"*

state\_dictionary[*"42"*] = *"TN"*

state\_dictionary[*"43"*] = *"TX"*

state\_dictionary[*"44"*] = *"UT"*

state\_dictionary[*"45"*] = *"VT"*

state\_dictionary[*"46"*] = *"VA"*

state\_dictionary[*"47"*] = *"WA"*

state\_dictionary[*"48"*] = *"WV"*

state\_dictionary[*"49"*] = *"WI"*

state\_dictionary[*"50"*] = *"WY"*

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(5)

# Driver Code

g = dfs\_singleton(50)

g.graph = positions

m=4

if country == *'Australia'*:

change\_position = [[3,4,6],[3,4,7],[1,2,4],[1,2,3,4,6,7],[],[1,4],[2,4]]

positions = []

for i in range(0,7) :

individual=[]

for j in range(0,7) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary = {}

state\_dictionary[*"1"*] = *"New South Wales"*

state\_dictionary[*"2"*] = *"Northern Territory"*

state\_dictionary[*"3"*] = *"Queensland"*

state\_dictionary[*"4"*] = *"South Australia"*

state\_dictionary[*"5"*] = *"Tasmania"*

state\_dictionary[*"6"*] = *"Victoria"*

state\_dictionary[*"7"*] = *"Western Australia"*

color\_dictionary = {*"1"*: *"red"*, *"2"*: *"green"*, *"3"*: *"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(4)

# Driver Code

g = dfs\_singleton(7)

g.graph = positions

m=3

g.garph\_colouring(m)

ending\_time = datetime.now()

difference = ending\_time - starting\_time

print(*"THE TOTAL TIME TAKEN FOR EXECUTION ----------------"*,str(difference.total\_seconds()))

print(*"The number of backtracking steps happened:"*+str(counterForBacktrack))

for key, value in result\_dictionary.items():

print(*"{} ==> {}"*.format(key,value))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

With Heuristics:

# problem using dfs backtracking and heuristic

from datetime import datetime

print(*"Please Enter the Country\nAmerica / Australia"*)

starting\_time = datetime.now()

country = input()

counterForBacktrack = 0

class **dfs\_backtrack\_heuristic**():

def **\_\_init\_\_**(*self*, vertex):

*self*.V = vertex

*self*.graph = [[0 for column in range(vertex)]

for row in range(vertex)]

## heuristic functions

def **m\_r\_v**(*self*, domain\_dictionary, colours):

least\_val\_states = {0:[],1:[],2:[],3:[],4:[]}

for key, value in domain\_dictionary.items():

if len(value)==0 and colours[key-1] == 0:

least\_val\_states[0].append(key-1)

elif(len(value)==1 and colours[key-1]==0):

least\_val\_states[1].append(key-1)

elif(len(value)==2 and colours[key-1]==0):

least\_val\_states[2].append(key-1)

elif(len(value)==3 and colours[key-1]==0):

least\_val\_states[3].append(key-1)

elif(len(value)==4 and colours[key-1]==0):

least\_val\_states[4].append(key-1)

if len(least\_val\_states[0])>0:

return least\_val\_states[0]

elif len(least\_val\_states[1])>0:

return least\_val\_states[1]

elif len(least\_val\_states[2])>0:

return least\_val\_states[2]

elif len(least\_val\_states[3])>0:

return least\_val\_states[3]

else:

return least\_val\_states[4]

def **DegreeConstraint**(*self*, domain\_dictionary, colours):

Max\_DC = 0 #Degree constraint

Max\_DC\_state = -1

for v in range(*self*.V):

if colours[v]!=0:

continue

counter = 0

for i in range(*self*.V):

if *self*.graph[v][i] == 1:

counter = counter + 1

if counter > Max\_DC:

Max\_DC = counter

Max\_DC\_state = v

return Max\_DC\_state

def **L\_C\_V**(*self*, domain\_dictionary, colours):

Min\_DegreeConstraint = 0

min\_DegreeConstraint\_state = -1

for v in range(*self*.V):

if colours[v]!=0:

continue

counter = 0

for i in range(*self*.V):

if *self*.graph[v][i] == 0:

counter = counter + 1

if counter > Min\_DegreeConstraint:

Min\_DegreeConstraint = counter

min\_DegreeConstraint\_state = v

return min\_DegreeConstraint\_state

def **get\_next\_state**(*self*, domain\_dictionary, colours):

next\_state = 0

next\_MRV\_states = *self*.m\_r\_v(domain\_dictionary, colours)

next\_DC\_states = *self*.DegreeConstraint(domain\_dictionary, colours)

next\_LeastConstraintValue\_states = *self*.L\_C\_V(domain\_dictionary, colours)

if (len(next\_MRV\_states)==1):

next\_state = next\_MRV\_states[0]

elif(next\_DC\_states!=-1):

next\_state = next\_DC\_states

else:

next\_state = next\_LeastConstraintValue\_states

return next\_state

def **check\_is\_safe**(*self*, v, colour, c):

for i in range(*self*.V):

if *self*.graph[v][i] == 1 and colour[i] == c:

return False

return True

def **retrieve\_neighbors**(*self*, state):

neighbours = []

for i in range(*self*.V):

if *self*.graph[state][i] == 1:

neighbours.append(i)

return neighbours

def **if\_coloured**(*self*, colors):

totalvertex = 0

for color in colors:

if color != 0:

totalvertex = totalvertex + 1

if totalvertex == 50:

return True

else:

return False

def **graph\_color**(*self*, m, colour, v):

global counterForBacktrack

try:

if *self*.if\_coloured(colour):

return True

if v == *self*.V:

return True

if not domain\_dictionary[v+1]:

return False

for c in domain\_dictionary[v+1]:

if *self*.check\_is\_safe(v, colour, c) == True:

colour[v] = c

neighbors = *self*.retrieve\_neighbors(v)

next\_state = *self*.get\_next\_state(domain\_dictionary, colour)

if next\_state != -1:

if *self*.graph\_color(m, colour, next\_state) == True:

return True

else:

if *self*.graph\_color(m, colour, v+1) == True:

return True

colour[v] = 0

else:

counterForBacktrack+=1

except Exception as e:

print(*"something wrong"*, e)

def **graph\_colouring**(*self*, m):

colour = [0] \* *self*.V

if *self*.graph\_color(m, colour, 0) == False:

return False

# Print the solution

print(*"Solution exist and Following are the assigned colours:"*)

for idx, val in enumerate(colour):

result\_dictionary[state\_dictionary[str(idx+1)]] = color\_dictionary[str(val)]

return True

def **dictionary\_of\_domain**(n):

for key, value in enumerate(state\_dictionary):

integers = list(range(1,n))

domain\_dictionary[key+1] = integers

if country == *'America'*:

change\_position=[[9,10,24,42],[],[5,6,28,31,44],[18,24,25,36,42,43],[3,28,37],[3,16,27,31,34,36,50],[21,32,39],[20,30,38],[1,10],[1,9,33,40,42],[],[26,28,37,44,47,50],[14,15,17,22,25,49],[13,17,22,35],[13,23,25,27,41,49],[6,25,27,36],[13,14,25,35,42,46,48],[4,24,43],[29],[8,38,46,48],[7,29,32,39,45],[13,14,23,35,49],[15,22,34,41,49],[1,18,42,43],[4,13,15,16,17,27,36,42],[12,34,41,50],[6,15,16,25,41,50],[3,5,12,37,44],[19,21,45],[8,32,38],[3,6,36,43,44],[7,21,30,38,39,45],[10,23,26,40,41,42,46],[23,26,41],[14,17,22,38,48],[4,6,16,25,31,43],[5,12,28,47],[8,20,30,32,35,48],[7,21,32],[10,33],[15,23,26,27,34,50],[1,4,10,17,24,25,33],[4,18,31,36],[3,6,12,28,31,50],[21,29,32],[17,20,33,42,48],[12,37],[17,20,35,38,46],[13,15,20,22,23],[6,12,26,27,41,44]]

positions = []

for i in range(0,50) :

individual=[]

for j in range(0,50) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary={}

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

state\_dictionary[*"1"*] = *"AL"*

state\_dictionary[*"2"*] = *"AK"*

state\_dictionary[*"3"*] = *"AZ"*

state\_dictionary[*"4"*] = *"AR"*

state\_dictionary[*"6"*] = *"CO"*

state\_dictionary[*"5"*] = *"CA"*

state\_dictionary[*"7"*] = *"CT"*

state\_dictionary[*"8"*] = *"DE"*

state\_dictionary[*"9"*] = *"FL"*

state\_dictionary[*"10"*] = *"GA"*

state\_dictionary[*"11"*] = *"HI"*

state\_dictionary[*"12"*] = *"ID"*

state\_dictionary[*"13"*] = *"IL"*

state\_dictionary[*"14"*] = *"IN"*

state\_dictionary[*"15"*] = *"IA"*

state\_dictionary[*"16"*] = *"KS"*

state\_dictionary[*"17"*] = *"KY"*

state\_dictionary[*"18"*] = *"LA"*

state\_dictionary[*"19"*] = *"ME"*

state\_dictionary[*"20"*] = *"MD"*

state\_dictionary[*"21"*] = *"MA"*

state\_dictionary[*"22"*] = *"MI"*

state\_dictionary[*"23"*] = *"MN"*

state\_dictionary[*"24"*] = *"MS"*

state\_dictionary[*"25"*] = *"MO"*

state\_dictionary[*"26"*] = *"MT"*

state\_dictionary[*"27"*] = *"NE"*

state\_dictionary[*"28"*] = *"NV"*

state\_dictionary[*"29"*] = *"NH"*

state\_dictionary[*"30"*] = *"NJ"*

state\_dictionary[*"31"*] = *"NM"*

state\_dictionary[*"32"*] = *"NY"*

state\_dictionary[*"33"*] = *"NC"*

state\_dictionary[*"34"*] = *"ND"*

state\_dictionary[*"35"*] = *"OH"*

state\_dictionary[*"36"*] = *"OK"*

state\_dictionary[*"37"*] = *"OR"*

state\_dictionary[*"38"*] = *"PA"*

state\_dictionary[*"39"*] = *"RI"*

state\_dictionary[*"40"*] = *"SC"*

state\_dictionary[*"41"*] = *"SD"*

state\_dictionary[*"42"*] = *"TN"*

state\_dictionary[*"43"*] = *"TX"*

state\_dictionary[*"44"*] = *"UT"*

state\_dictionary[*"45"*] = *"VT"*

state\_dictionary[*"46"*] = *"VA"*

state\_dictionary[*"47"*] = *"WA"*

state\_dictionary[*"48"*] = *"WV"*

state\_dictionary[*"49"*] = *"WI"*

state\_dictionary[*"50"*] = *"WY"*

dictionary\_of\_domain(5)

# Driver Code

g = dfs\_backtrack\_heuristic(50) #number of states 48

g.graph = positions

m=4 ## chromataic number

if country == *'Australia'*:

change\_position = [[3,4,6],[3,4,7],[1,2,4],[1,2,3,4,6,7],[],[1,4],[2,4]]

positions = []

for i in range(0,7) :

individual=[]

for j in range(0,7) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary={}

color\_dictionary = {*"1"*: *"red"*, *"2"*: *"green"*, *"3"*: *"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

state\_dictionary[*"1"*] = *"New South Wales"*

state\_dictionary[*"2"*] = *"Northern Territory"*

state\_dictionary[*"3"*] = *"Queensland"*

state\_dictionary[*"4"*] = *"South Australia"*

state\_dictionary[*"5"*] = *"Tasmania"*

state\_dictionary[*"6"*] = *"Victoria"*

state\_dictionary[*"7"*] = *"Western Australia"*

dictionary\_of\_domain(4)

# Driver Code

g = dfs\_backtrack\_heuristic(7) #number of states

g.graph = positions

m=3 ## chromataic number

g.graph\_colouring(m)

ending\_time = datetime.now()

difference = ending\_time - starting\_time

print(*"THE TOTAL TIME TOOK FOR EXECUTION ----------------"*,str(difference.total\_seconds()))

print(*"The number of backtracking steps happened:"*+str(counterForBacktrack))

for key, value in result\_dictionary.items():

print(*"{} ==> {}"*.format(key,value))

# problem using dfs backtrack forward checking and singleton heuristic

from datetime import datetime

print(*"Please Enter the Country\nAmerica / Australia"*)

start = datetime.now()

country = input()

counterForBacktrack=0

class **dfs\_forward\_Checking\_Singleton\_heuristic**():

def **\_\_init\_\_**(*self*, vertices):

*self*.V = vertices

*self*.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

## heuristic functions

def **m\_r\_v**(*self*, domain\_dictionary, colours):

least\_val\_states = {0:[],1:[],2:[],3:[],4:[]}

for key, value in domain\_dictionary.items():

if len(value)==0 and colours[key-1] == 0:

least\_val\_states[0].append(key-1)

elif(len(value)==1 and colours[key-1]==0):

least\_val\_states[1].append(key-1)

elif(len(value)==2 and colours[key-1]==0):

least\_val\_states[2].append(key-1)

elif(len(value)==3 and colours[key-1]==0):

least\_val\_states[3].append(key-1)

elif(len(value)==4 and colours[key-1]==0):

least\_val\_states[4].append(key-1)

if len(least\_val\_states[0])>0:

return least\_val\_states[0]

elif len(least\_val\_states[1])>0:

return least\_val\_states[1]

elif len(least\_val\_states[2])>0:

return least\_val\_states[2]

elif len(least\_val\_states[3])>0:

return least\_val\_states[3]

else:

return least\_val\_states[4]

def **DegreeConstraint**(*self*, domain\_dictionary, colours):

Max\_DC = 0 #Degree constraint

Max\_DC\_state = -1

for v in range(*self*.V):

if colours[v]!=0:

continue

counter = 0

for i in range(*self*.V):

if *self*.graph[v][i] == 1:

counter = counter + 1

if counter > Max\_DC:

Max\_DC = counter

Max\_DC\_state = v

return Max\_DC\_state

def **L\_C\_V**(*self*, domain\_dictionary, colours):

Min\_DegreeConstraint = 0

min\_DegreeConstraint\_state = -1

for v in range(*self*.V):

if colours[v]!=0:

continue

counter = 0

for i in range(*self*.V):

if *self*.graph[v][i] == 0:

counter = counter + 1

if counter > Min\_DegreeConstraint:

Min\_DegreeConstraint = counter

min\_DegreeConstraint\_state = v

return min\_DegreeConstraint\_state

def **get\_next\_state**(*self*, domain\_dictionary, colours):

NextState = 0

next\_MRV\_states = *self*.m\_r\_v(domain\_dictionary, colours)

next\_DC\_states = *self*.DegreeConstraint(domain\_dictionary, colours)

next\_LeastConstraintValue\_states= *self*.L\_C\_V(domain\_dictionary, colours)

if (len(next\_MRV\_states)==1):

NextState = next\_MRV\_states[0]

elif(next\_DC\_states!=-1):

NextState = next\_DC\_states

else:

NextState = next\_LeastConstraintValue\_states

return NextState

def **if\_safe**(*self*, v, colour, c):

for i in range(*self*.V):

if *self*.graph[v][i] == 1 and colour[i] == c:

return False

return True

def **retrieve\_neighbors**(*self*, state):

neighbours = []

for i in range(*self*.V):

if *self*.graph[state][i] == 1:

neighbours.append(i)

return neighbours

#singleton methods

def **if\_coloured**(*self*, colors):

totalvertex = 0

for color in colors:

if color != 0:

totalvertex = totalvertex + 1

if totalvertex == 50:

return True

elif totalvertex == 7:

return True

else:

return False

def **check\_singleton\_constraints**(*self*, v):

if *self*.if\_coloured(colour):

return True

if v == *self*.V:

v = random.choice([a for a in range(len(colour)) if colour[a] == 0])

def **singleton\_domain\_variables\_removing**(*self*, v):

if c in domain\_dictionary[v+1]:

domain\_dictionary[v+1].remove(c)

if c not in domain\_dictionary[v+1]:

domain\_dictionary[v+1].append(c)

domain\_dictionary[v+1].sort()

def **singleton\_logic**(*self*, st\_states):

if len(st\_states)==0:

return

state\_tocolor = -1

for key,value in domain\_dictionary.items():

if len(value) == 1 and colour[key-1] == 0:

state\_tocolor = key

break

if state\_tocolor == -1:

current = random.choice([a for a in range(len(colour)) if colour[a] == 0])

if *self*.graph\_colour(m, colour, current) == True:

return True

else:

if *self*.graph\_colour(m, colour, state\_tocolor-1) == True:

return True

def **graph\_colour**(*self*, m, colour, v):

global counterForBacktrack

try:

if v == *self*.V:

return True

if country == *'Australia'*:

list\_st\_states = list(range(*self*.V%7))

if country == *'America'*:

list\_st\_states = list(range(*self*.V%50))

*self*.singleton\_logic(list\_st\_states)

if not domain\_dictionary[v+1]:

return False

for c in domain\_dictionary[v+1]:

if *self*.if\_safe(v, colour, c) == True:

colour[v] = c

neighbors = *self*.retrieve\_neighbors(v)

for neighbor in neighbors:

if c in domain\_dictionary[neighbor+1]:

domain\_dictionary[neighbor+1].remove(c)

NextState = *self*.get\_next\_state(domain\_dictionary, colour)

if NextState != -1:

if *self*.graph\_colour(m, colour, NextState) == True:

return True

else:

if *self*.graph\_colour(m, colour, v+1) == True:

return True

for neighbor in neighbors:

a = neighbor+1

if c not in domain\_dictionary[a]:

domain\_dictionary[a].append(c)

domain\_dictionary[a].sort()

colour[v] = 0

else:

counterForBacktrack +=1

except Exception as e:

print(*"something wrong"*, e)

def **GraphColuringUtil**(*self*, m):

colour = [0] \* *self*.V

if *self*.graph\_colour(m, colour, 0) == False:

return False

# Print the solution

print(*"Solution exist and Following are the colours that have been assingned:"*)

for idx, val in enumerate(colour):

result\_dictionary[state\_dictionary[str(idx+1)]] = color\_dictionary[str(val)]

return True

def **dictionary\_of\_domain**(n):

for key, value in enumerate(state\_dictionary):

integers = list(range(1,n))

domain\_dictionary[key+1] = integers

if country == *'America'*:

change\_position=[[9,10,24,42],[],[5,6,28,31,44],[18,24,25,36,42,43],[3,28,37],[3,16,27,31,34,36,50],[21,32,39],[20,30,38],[1,10],[1,9,33,40,42],[],[26,28,37,44,47,50],[14,15,17,22,25,49],[13,17,22,35],[13,23,25,27,41,49],[6,25,27,36],[13,14,25,35,42,46,48],[4,24,43],[29],[8,38,46,48],[7,29,32,39,45],[13,14,23,35,49],[15,22,34,41,49],[1,18,42,43],[4,13,15,16,17,27,36,42],[12,34,41,50],[6,15,16,25,41,50],[3,5,12,37,44],[19,21,45],[8,32,38],[3,6,36,43,44],[7,21,30,38,39,45],[10,23,26,40,41,42,46],[23,26,41],[14,17,22,38,48],[4,6,16,25,31,43],[5,12,28,47],[8,20,30,32,35,48],[7,21,32],[10,33],[15,23,26,27,34,50],[1,4,10,17,24,25,33],[4,18,31,36],[3,6,12,28,31,50],[21,29,32],[17,20,33,42,48],[12,37],[17,20,35,38,46],[13,15,20,22,23],[6,12,26,27,41,44]]

positions = []

for i in range(0,50) :

individual=[]

for j in range(0,50) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary = {}

state\_dictionary[*"1"*] = *"AL"*

state\_dictionary[*"2"*] = *"AK"*

state\_dictionary[*"3"*] = *"AZ"*

state\_dictionary[*"4"*] = *"AR"*

state\_dictionary[*"6"*] = *"CO"*

state\_dictionary[*"5"*] = *"CA"*

state\_dictionary[*"7"*] = *"CT"*

state\_dictionary[*"8"*] = *"DE"*

state\_dictionary[*"9"*] = *"FL"*

state\_dictionary[*"10"*] = *"GA"*

state\_dictionary[*"11"*] = *"HI"*

state\_dictionary[*"12"*] = *"ID"*

state\_dictionary[*"13"*] = *"IL"*

state\_dictionary[*"14"*] = *"IN"*

state\_dictionary[*"15"*] = *"IA"*

state\_dictionary[*"16"*] = *"KS"*

state\_dictionary[*"17"*] = *"KY"*

state\_dictionary[*"18"*] = *"LA"*

state\_dictionary[*"19"*] = *"ME"*

state\_dictionary[*"20"*] = *"MD"*

state\_dictionary[*"21"*] = *"MA"*

state\_dictionary[*"22"*] = *"MI"*

state\_dictionary[*"23"*] = *"MN"*

state\_dictionary[*"24"*] = *"MS"*

state\_dictionary[*"25"*] = *"MO"*

state\_dictionary[*"26"*] = *"MT"*

state\_dictionary[*"27"*] = *"NE"*

state\_dictionary[*"28"*] = *"NV"*

state\_dictionary[*"29"*] = *"NH"*

state\_dictionary[*"30"*] = *"NJ"*

state\_dictionary[*"31"*] = *"NM"*

state\_dictionary[*"32"*] = *"NY"*

state\_dictionary[*"33"*] = *"NC"*

state\_dictionary[*"34"*] = *"ND"*

state\_dictionary[*"35"*] = *"OH"*

state\_dictionary[*"36"*] = *"OK"*

state\_dictionary[*"37"*] = *"OR"*

state\_dictionary[*"38"*] = *"PA"*

state\_dictionary[*"39"*] = *"RI"*

state\_dictionary[*"40"*] = *"SC"*

state\_dictionary[*"41"*] = *"SD"*

state\_dictionary[*"42"*] = *"TN"*

state\_dictionary[*"43"*] = *"TX"*

state\_dictionary[*"44"*] = *"UT"*

state\_dictionary[*"45"*] = *"VT"*

state\_dictionary[*"46"*] = *"VA"*

state\_dictionary[*"47"*] = *"WA"*

state\_dictionary[*"48"*] = *"WV"*

state\_dictionary[*"49"*] = *"WI"*

state\_dictionary[*"50"*] = *"WY"*

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

dictionary\_of\_domain(5)

# Driver Code

g = dfs\_forward\_Checking\_Singleton\_heuristic(50)

g.graph = positions

m=4 ## chromataic number

if country == *'Australia'*:

change\_position = [[3,4,6],[3,4,7],[1,2,4],[1,2,3,4,6,7],[],[1,4],[2,4]]

positions = []

for i in range(0,7) :

individual=[]

for j in range(0,7) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary = {}

state\_dictionary[*"1"*] = *"New South Wales"*

state\_dictionary[*"2"*] = *"Northern Territory"*

state\_dictionary[*"3"*] = *"Queensland"*

state\_dictionary[*"4"*] = *"South Australia"*

state\_dictionary[*"5"*] = *"Tasmania"*

state\_dictionary[*"6"*] = *"Victoria"*

state\_dictionary[*"7"*] = *"Western Australia"*

color\_dictionary = {*"1"*: *"red"*, *"2"*: *"green"*, *"3"*: *"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(4)

# Driver Code

g = dfs\_forward\_Checking\_Singleton\_heuristic(7) #number of states

g.graph = positions

m=3 ## chromataic number

g.GraphColuringUtil(m)

end = datetime.now()

elapsed = end - start

print(*"THE TOTAL TIME TOOK FOR EXECUTION ----------------"*,str(elapsed.total\_seconds()))

print(*"total number of backtracks taken:"*+str(counterForBacktrack))

for key, value in result\_dictionary.items():

print(*"{} => {}"*.format(key,value))

# problem using dfs backtrack forward checking and heuristic

from datetime import datetime

print(*"Please Enter the Country\nAmerica / Australia"*)

starting\_time = datetime.now()

country = input()

counterForBacktrack = 0

class **dfs\_forwardchecking\_heuristic**():

def **\_\_init\_\_**(*self*, vertexs):

*self*.V = vertexs

*self*.graph = [[0 for column in range(vertexs)]

for row in range(vertexs)]

## heuristic functions

def **m\_r\_v**(*self*, domain\_dictionary, colours):

least\_val\_states = {0:[],1:[],2:[],3:[],4:[]}

for key, value in domain\_dictionary.items():

if len(value)==0 and colours[key-1] == 0:

least\_val\_states [0].append(key-1)

elif(len(value)==1 and colours[key-1]==0):

least\_val\_states [1].append(key-1)

elif(len(value)==2 and colours[key-1]==0):

least\_val\_states [2].append(key-1)

elif(len(value)==3 and colours[key-1]==0):

least\_val\_states [3].append(key-1)

elif(len(value)==4 and colours[key-1]==0):

least\_val\_states [4].append(key-1)

if len(least\_val\_states [0])>0:

return least\_val\_states [0]

elif len(least\_val\_states [1])>0:

return least\_val\_states [1]

elif len(least\_val\_states [2])>0:

return least\_val\_states [2]

elif len(least\_val\_states [3])>0:

return least\_val\_states [3]

else:

return least\_val\_states [4]

def **DegreeConstraint**(*self*, domain\_dictionary, colours):

Max\_DC = 0 #Degree constraint

Max\_DC\_state = -1

for v in range(*self*.V):

if colours[v]!=0:

continue

counter = 0

for i in range(*self*.V):

if *self*.graph[v][i] == 1:

counter = counter + 1

if counter > Max\_DC:

Max\_DC = counter

Max\_DC\_state = v

return Max\_DC\_state

def **L\_C\_V**(*self*, domain\_dictionary, colours):

Min\_DegreeConstraint = 0

min\_degree\_constraint\_state = -1

for v in range(*self*.V):

if colours[v]!=0:

continue

counter = 0

for i in range(*self*.V):

if *self*.graph[v][i] == 0:

counter = counter + 1

if counter > Min\_DegreeConstraint:

Min\_DegreeConstraint = counter

min\_degree\_constraint\_state = v

return min\_degree\_constraint\_state

def **get\_next\_state**(*self*, domain\_dictionary, colours):

next\_state = 0

next\_MRV\_states = *self*.m\_r\_v(domain\_dictionary, colours)

next\_degree\_constraint\_states = *self*.DegreeConstraint(domain\_dictionary, colours)

next\_LeastConstraintValue\_states = *self*.L\_C\_V(domain\_dictionary, colours)

if (len(next\_MRV\_states)==1):

next\_state = next\_MRV\_states[0]

elif(next\_degree\_constraint\_states!=-1):

next\_state = next\_degree\_constraint\_states

else:

next\_state = next\_LeastConstraintValue\_states

return next\_state

def **check\_safe**(*self*, v, colour, c):

for i in range(*self*.V):

if *self*.graph[v][i] == 1 and colour[i] == c:

return False

return True

def **retrieve\_neighbors**(*self*, state):

neighbours = []

for i in range(*self*.V):

if *self*.graph[state][i] == 1:

neighbours.append(i)

return neighbours

def **check\_if\_coloured**(*self*, colors):

totalvertex = 0

for color in colors:

if color != 0:

totalvertex = totalvertex + 1

if totalvertex == 50:

return True

else:

return False

def **graph\_color**(*self*, m, colour, v):

global counterForBacktrack

try:

if *self*.check\_if\_coloured(colour):

return True

if v == *self*.V:

return True

if not domain\_dictionary[v+1]:

return False

for c in domain\_dictionary[v+1]:

if *self*.check\_safe(v, colour, c) == True:

colour[v] = c

neighbors = *self*.retrieve\_neighbors(v)

for neighbor in neighbors:

if c in domain\_dictionary[neighbor+1]:

domain\_dictionary[neighbor+1].remove(c)

next\_state = *self*.get\_next\_state(domain\_dictionary, colour)

if next\_state != -1:

if *self*.graph\_color(m, colour, next\_state) == True:

return True

else:

if *self*.graph\_color(m, colour, v+1) == True:

return True

for neighbor in neighbors:

a = neighbor+1

if c not in domain\_dictionary[a]:

domain\_dictionary[a].append(c)

domain\_dictionary[a].sort()

colour[v] = 0

else:

counterForBacktrack+=1

except Exception as e:

print(*"something wrong"*, e)

def **graph\_colouring**(*self*, m):

colour = [0] \* *self*.V

if *self*.graph\_color(m, colour, 0) == False:

return False

print(*"Solution exist and Following are the colours that have been assigned:"*)

for idx, val in enumerate(colour):

result\_dictionary[state\_dictionary[str(idx+1)]] = color\_dictionary[str(val)]

return True

def **dictionary\_of\_domain**(n):

for key, value in enumerate(state\_dictionary):

integers = list(range(1,n))

domain\_dictionary[key+1] = integers

if country == *'America'*:

change\_position=[[9,10,24,42],[],[5,6,28,31,44],[18,24,25,36,42,43],[3,28,37],[3,16,27,31,34,36,50],[21,32,39],[20,30,38],[1,10],[1,9,33,40,42],[],[26,28,37,44,47,50],[14,15,17,22,25,49],[13,17,22,35],[13,23,25,27,41,49],[6,25,27,36],[13,14,25,35,42,46,48],[4,24,43],[29],[8,38,46,48],[7,29,32,39,45],[13,14,23,35,49],[15,22,34,41,49],[1,18,42,43],[4,13,15,16,17,27,36,42],[12,34,41,50],[6,15,16,25,41,50],[3,5,12,37,44],[19,21,45],[8,32,38],[3,6,36,43,44],[7,21,30,38,39,45],[10,23,26,40,41,42,46],[23,26,41],[14,17,22,38,48],[4,6,16,25,31,43],[5,12,28,47],[8,20,30,32,35,48],[7,21,32],[10,33],[15,23,26,27,34,50],[1,4,10,17,24,25,33],[4,18,31,36],[3,6,12,28,31,50],[21,29,32],[17,20,33,42,48],[12,37],[17,20,35,38,46],[13,15,20,22,23],[6,12,26,27,41,44]]

positions = []

for i in range(0,50) :

individual=[]

for j in range(0,50) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary={}

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

state\_dictionary[*"1"*] = *"AL"*

state\_dictionary[*"2"*] = *"AK"*

state\_dictionary[*"3"*] = *"AZ"*

state\_dictionary[*"4"*] = *"AR"*

state\_dictionary[*"6"*] = *"CO"*

state\_dictionary[*"5"*] = *"CA"*

state\_dictionary[*"7"*] = *"CT"*

state\_dictionary[*"8"*] = *"DE"*

state\_dictionary[*"9"*] = *"FL"*

state\_dictionary[*"10"*] = *"GA"*

state\_dictionary[*"11"*] = *"HI"*

state\_dictionary[*"12"*] = *"ID"*

state\_dictionary[*"13"*] = *"IL"*

state\_dictionary[*"14"*] = *"IN"*

state\_dictionary[*"15"*] = *"IA"*

state\_dictionary[*"16"*] = *"KS"*

state\_dictionary[*"17"*] = *"KY"*

state\_dictionary[*"18"*] = *"LA"*

state\_dictionary[*"19"*] = *"ME"*

state\_dictionary[*"20"*] = *"MD"*

state\_dictionary[*"21"*] = *"MA"*

state\_dictionary[*"22"*] = *"MI"*

state\_dictionary[*"23"*] = *"MN"*

state\_dictionary[*"24"*] = *"MS"*

state\_dictionary[*"25"*] = *"MO"*

state\_dictionary[*"26"*] = *"MT"*

state\_dictionary[*"27"*] = *"NE"*

state\_dictionary[*"28"*] = *"NV"*

state\_dictionary[*"29"*] = *"NH"*

state\_dictionary[*"30"*] = *"NJ"*

state\_dictionary[*"31"*] = *"NM"*

state\_dictionary[*"32"*] = *"NY"*

state\_dictionary[*"33"*] = *"NC"*

state\_dictionary[*"34"*] = *"ND"*

state\_dictionary[*"35"*] = *"OH"*

state\_dictionary[*"36"*] = *"OK"*

state\_dictionary[*"37"*] = *"OR"*

state\_dictionary[*"38"*] = *"PA"*

state\_dictionary[*"39"*] = *"RI"*

state\_dictionary[*"40"*] = *"SC"*

state\_dictionary[*"41"*] = *"SD"*

state\_dictionary[*"42"*] = *"TN"*

state\_dictionary[*"43"*] = *"TX"*

state\_dictionary[*"44"*] = *"UT"*

state\_dictionary[*"45"*] = *"VT"*

state\_dictionary[*"46"*] = *"VA"*

state\_dictionary[*"47"*] = *"WA"*

state\_dictionary[*"48"*] = *"WV"*

state\_dictionary[*"49"*] = *"WI"*

state\_dictionary[*"50"*] = *"WY"*

color\_dictionary = {*"1"*:*"red"*, *"2"*:*"green"*, *"3"*:*"yellow"*, *"4"*:*"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(5)

g = dfs\_forwardchecking\_heuristic(50)

g.graph = positions

m=4 ## chromataic number

if country == *'Australia'*:

change\_position = [[3,4,6],[3,4,7],[1,2,4],[1,2,3,4,6,7],[],[1,4],[2,4]]

positions = []

for i in range(0,7) :

individual=[]

for j in range(0,7) :

individual.append(0)

for j in change\_position[i] :

individual[j-1] =1

positions.append(individual)

state\_dictionary = {}

state\_dictionary[*"1"*] = *"New South Wales"*

state\_dictionary[*"2"*] = *"Northern Territory"*

state\_dictionary[*"3"*] = *"Queensland"*

state\_dictionary[*"4"*] = *"South Australia"*

state\_dictionary[*"5"*] = *"Tasmania"*

state\_dictionary[*"6"*] = *"Victoria"*

state\_dictionary[*"7"*] = *"Western Australia"*

color\_dictionary = {*"1"*: *"red"*, *"2"*: *"green"*, *"3"*: *"blue"*}

result\_dictionary = {}

domain\_dictionary = {}

dictionary\_of\_domain(4)

# Driver Code

g = dfs\_forwardchecking\_heuristic(7) #number of states

g.graph = positions

m=3 ## chromataic number

g.graph\_colouring(m)

ending\_time = datetime.now()

difference = ending\_time - starting\_time

print(*"THE TOTAL TIME TOOK FOR EXECUTION ----------------"*,str(difference.total\_seconds()))

print(*"The number of backtracking steps happened:"*+str(counterForBacktrack))

for key, value in result\_dictionary.items():

print(*"{} => {}"*.format(key,value))

**SIMULATION RESULTS FOR USA:**

**for Backtracking**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | Time taken |
| 1 | 85364006 | 285.335 seconds |
| 2 | 85364006 | 288.238 seconds |
| 3 | 85364006 | 280.345 seconds |

**RESULT SAMPLE:**

D:\uncc\intelligent\_systems\export\IS\_Project03\withoutHeuristics>python backtracking.py

Please Enter the Country

America / Australia

America

Solution exist and Following are the assigned colours:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '285.335')

The number of backtracking steps happened:85364006

WA ==> green

DE ==> red

WI ==> blue

WV ==> blue

HI ==> red

FL ==> green

WY ==> yellow

NH ==> green

NJ ==> green

NM ==> yellow

TX ==> green

LA ==> yellow

NC ==> green

ND ==> yellow

NE ==> blue

TN ==> blue

NY ==> blue

PA ==> yellow

CA ==> red

NV ==> green

VA ==> red

CO ==> green

AK ==> red

AL ==> red

AR ==> red

VT ==> yellow

IL ==> red

GA ==> yellow

IN ==> green

IA ==> yellow

OK ==> blue

AZ ==> red

ID ==> red

CT ==> green

ME ==> red

MD ==> green

MA ==> red

OH ==> red

UT ==> blue

MO ==> green

MN ==> red

MI ==> yellow

RI ==> yellow

KS ==> red

MT ==> blue

MS ==> green

SC ==> red

KY ==> yellow

OR ==> yellow

SD ==> green

**Backtracking with forward checking**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | Time taken |
| 1 | 31422458 | 363.937 seconds |
| 2 | 31422458 | 389.456 seconds |
| 3 | 31422458 | 386.235 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withoutHeuristics>python forwardchecking.py

Please Enter the Country

America / Australia

America

Solution exist and Following are the colours that we have assigned:

('THE TOTAL TIME TOOK FOR EXEC ----------------', '363.937')

The number of backtracking steps happened:31422458

WA => green

DE => red

WI => blue

WV => blue

HI => red

FL => green

WY => yellow

NH => yellow

NJ => green

NM => yellow

TX => green

LA => blue

NC => green

ND => yellow

NE => blue

TN => blue

NY => blue

PA => yellow

CA => green

NV => yellow

VA => red

CO => green

AK => red

AL => red

AR => red

VT => red

IL => red

GA => yellow

IN => green

IA => yellow

OK => blue

AZ => red

ID => red

CT => red

ME => red

MD => green

MA => green

OH => red

UT => blue

MO => green

MN => red

MI => yellow

RI => yellow

KS => red

MT => blue

MS => yellow

SC => red

KY => yellow

OR => blue

SD => green

**Backtracking with forward checking and singleton**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | Time taken |
| 1 | 31422458 | 383.225 seconds |
| 2 | 31422458 | 378.237 seconds |
| 3 | 31422458 | 389.247 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withoutHeuristics>python singleton.py

Please Enter the Country

America / Australia

America

Solution exist and Following are the assigned colours:

('THE TOTAL TIME TAKEN FOR EXECUTION ----------------', '383.225')

The number of backtracking steps happened:31422458

WA ==> green

DE ==> red

WI ==> blue

WV ==> blue

HI ==> red

FL ==> green

WY ==> yellow

NH ==> yellow

NJ ==> green

NM ==> yellow

TX ==> green

LA ==> blue

NC ==> green

ND ==> yellow

NE ==> blue

TN ==> blue

NY ==> blue

PA ==> yellow

CA ==> green

NV ==> yellow

VA ==> red

CO ==> green

AK ==> red

AL ==> red

AR ==> red

VT ==> red

IL ==> red

GA ==> yellow

IN ==> green

IA ==> yellow

OK ==> blue

AZ ==> red

ID ==> red

CT ==> red

ME ==> red

MD ==> green

MA ==> green

OH ==> red

UT ==> blue

MO ==> green

MN ==> red

MI ==> yellow

RI ==> yellow

KS ==> red

MT ==> blue

MS ==> yellow

SC ==> red

KY ==> yellow

OR ==> blue

SD ==> green

**Backtracking with heuristic:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 66 | 3.062 seconds |
| 2 | 66 | 3.456 seconds |
| 3 | 66 | 2.978 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withHeuristics>python backtrackingheuristic.py

Please Enter the Country

America / Australia

America

Solution exist and Following are the assigned colours:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '3.062')

The number of backtracking steps happened:66

WA ==> yellow

DE ==> red

WI ==> blue

WV ==> red

HI ==> red

FL ==> yellow

WY ==> blue

NH ==> red

NJ ==> yellow

NM ==> blue

TX ==> red

LA ==> yellow

NC ==> red

ND ==> blue

NE ==> yellow

TN ==> yellow

NY ==> red

PA ==> green

CA ==> red

NV ==> blue

VA ==> blue

CO ==> red

AK ==> red

AL ==> red

AR ==> green

VT ==> yellow

IL ==> yellow

GA ==> green

IN ==> blue

IA ==> green

OK ==> yellow

AZ ==> yellow

ID ==> red

CT ==> yellow

ME ==> green

MD ==> yellow

MA ==> green

OH ==> yellow

UT ==> green

MO ==> red

MN ==> yellow

MI ==> red

RI ==> blue

KS ==> green

MT ==> green

MS ==> green

SC ==> yellow

KY ==> green

OR ==> green

SD ==> red

**Backtracking with forward checking using heuristic:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 1 | 4.693 seconds |
| 2 | 1 | 3.456 seconds |
| 3 | 1 | 2.978 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withHeuristics>python forwardcheckingheuristic.py

Please Enter the Country

America / Australia

America

Solution exist and Following are the colours that have been assigned:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '4.693')

The number of backtracking steps happened:1

WA => yellow

DE => red

WI => blue

WV => red

HI => red

FL => yellow

WY => green

NH => red

NJ => yellow

NM => blue

TX => red

LA => yellow

NC => red

ND => green

NE => yellow

TN => yellow

NY => red

PA => green

CA => red

NV => blue

VA => blue

CO => red

AK => red

AL => red

AR => green

VT => yellow

IL => yellow

GA => green

IN => blue

IA => green

OK => yellow

AZ => green

ID => red

CT => yellow

ME => green

MD => yellow

MA => green

OH => yellow

UT => yellow

MO => red

MN => yellow

MI => red

RI => blue

KS => green

MT => yellow

MS => blue

SC => yellow

KY => green

OR => green

SD => blue

**Backtracking with forward checking and Singleton using Heuristic:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 1 | 3.263 seconds |
| 2 | 1 | 4.245 seconds |
| 3 | 1 | 3.876 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withHeuristics>python singletonheuristic.py

Please Enter the Country

America / Australia

America

Solution exist and Following are the colours that have been assigned:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '3.263')

total number of backtracks taken:1

WA => yellow

DE => red

WI => blue

WV => red

HI => red

FL => yellow

WY => green

NH => red

NJ => yellow

NM => blue

TX => red

LA => yellow

NC => red

ND => green

NE => yellow

TN => yellow

NY => red

PA => green

CA => red

NV => blue

VA => blue

CO => red

AK => red

AL => red

AR => green

VT => yellow

IL => yellow

GA => green

IN => blue

IA => green

OK => yellow

AZ => green

ID => red

CT => yellow

ME => green

MD => yellow

MA => green

OH => yellow

UT => yellow

MO => red

MN => yellow

MI => red

RI => blue

KS => green

MT => yellow

MS => blue

SC => yellow

KY => green

OR => green

SD => blue

**TABULATION FOR AUSTRALIA:**

**Backtracking:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 5 | 6.376 seconds |
| 2 | 5 | 4.863 seconds |
| 3 | 5 | 5.986 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withoutHeuristics>python backtracking.py

Please Enter the Country

America / Australia

Australia

Solution exist and Following are the assigned colours:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '6.376')

The number of backtracking steps happened:5

Northern Territory ==> red

Western Australia ==> green

Victoria ==> green

New South Wales ==> red

Queensland ==> green

Tasmania ==> red

South Australia ==> blue

**Backtracking with forward checking:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 0 | 5.122 seconds |
| 2 | 0 | 6.017 seconds |
| 3 | 0 | 6.345 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withoutHeuristics>python forwardchecking.py

Please Enter the Country

America / Australia

Australia

Solution exist and Following are the colours that we have assigned:

('THE TOTAL TIME TOOK FOR EXEC ----------------', '5.122')

The number of backtracking steps happened:0

Northern Territory => red

Western Australia => green

Victoria => green

New South Wales => red

Queensland => green

Tasmania => red

South Australia => blue

**Backtracking with forward checking and singleton:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 0 | 0.004 seconds |
| 2 | 0 | 0.065 seconds |
| 3 | 0 | 0.003 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withoutHeuristics>python singleton.py

Please Enter the Country

America / Australia

Australia

Solution exist and Following are the assigned colours:

('THE TOTAL TIME TAKEN FOR EXECUTION ----------------', '0.004')

The number of backtracking steps happened:0

Northern Territory ==> red

Western Australia ==> green

Victoria ==> green

New South Wales ==> red

Queensland ==> green

Tasmania ==> red

South Australia ==> blue

**Backtracking with heuristic:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 11 | 4.394 seconds |
| 2 | 11 | 5.345 seconds |
| 3 | 11 | 4.236 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withHeuristics>python backtrackingheuristic.py

Please Enter the Country

America / Australia

Australia

Solution exist and Following are the assigned colours:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '4.394')

The number of backtracking steps happened:11

Northern Territory ==> red

Western Australia ==> blue

Victoria ==> blue

New South Wales ==> red

Queensland ==> blue

Tasmania ==> red

South Australia ==> green

**Backtracking with forward checking using heuristic:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 0 | 3.064 seconds |
| 2 | 0 | 3.645 seconds |
| 3 | 0 | 4.013 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withHeuristics>python forwardcheckingheuristic.py

Please Enter the Country

America / Australia

Australia

Solution exist and Following are the colours that have been assigned:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '3.064')

The number of backtracking steps happened:0

Northern Territory => red

Western Australia => blue

Victoria => blue

New South Wales => red

Queensland => blue

Tasmania => red

South Australia => green

**Backtracking with forward checking and singleton using heuristic:**

|  |  |  |
| --- | --- | --- |
| Number of run | Number of backtrack | **Time taken** |
| 1 | 0 | 1.976 seconds |
| 2 | 0 | 2.035 seconds |
| 3 | 0 | 1.567 seconds |

RESULT SAMPLE:

D:\uncc\intelligent\_systems\export\IS\_Project03\withHeuristics>python singletonheuristic.py

Please Enter the Country

America / Australia

Australia

Solution exist and Following are the colours that have been assigned:

('THE TOTAL TIME TOOK FOR EXECUTION ----------------', '1.976')

total number of backtracks taken:0

Northern Territory => red

Western Australia => blue

Victoria => blue

New South Wales => red

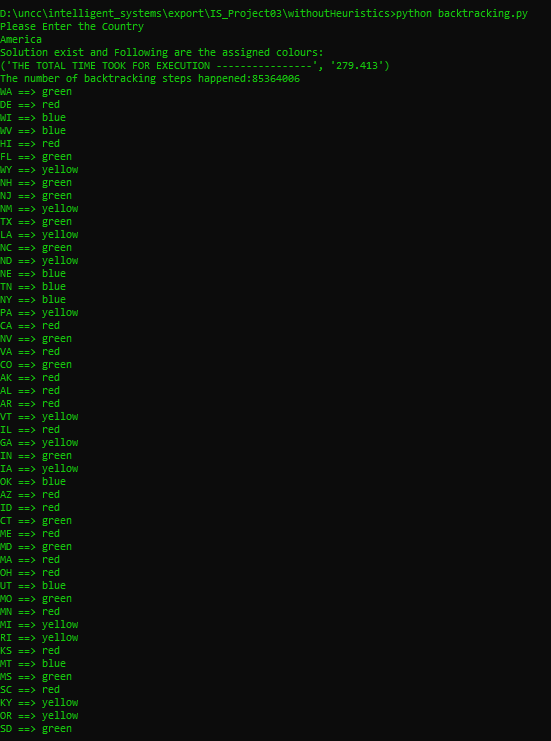
Queensland => blue

Tasmania => red

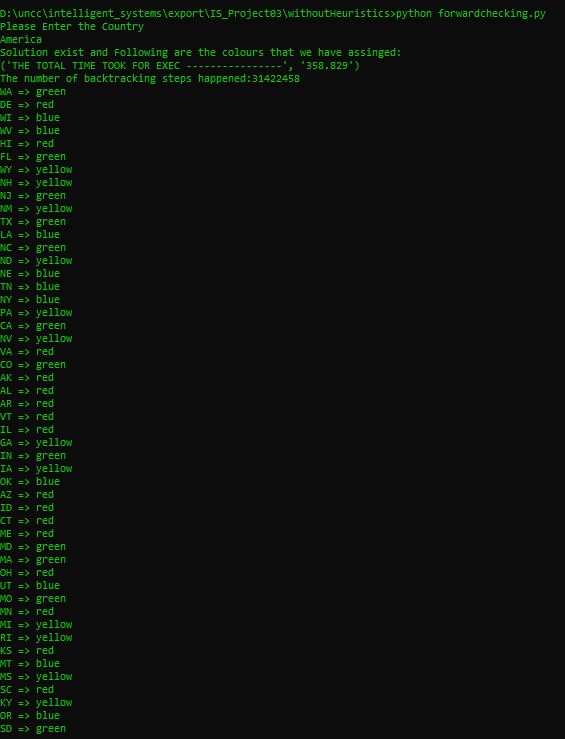
South Australia => green

**OUTPUT Screenshots for America Map Coloring Solution:**

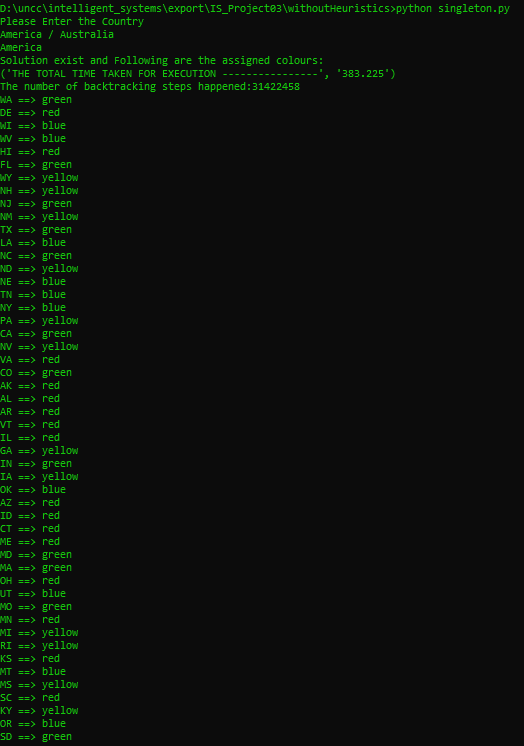
1. **Backtracking:**



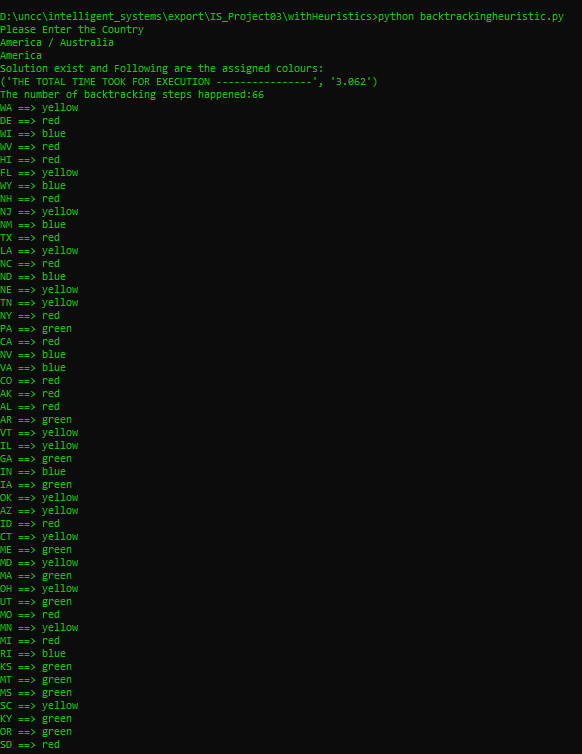
1. **Backtracking with Forward Checking:**



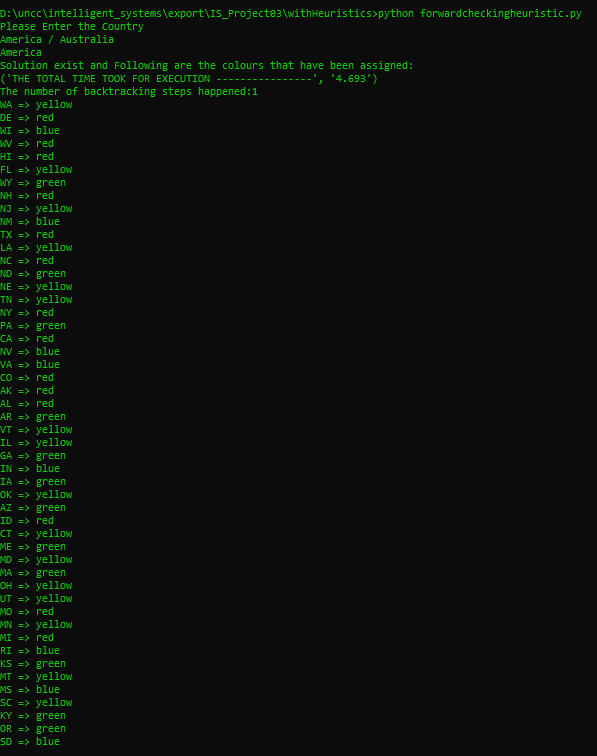
1. **Backtracking with Forward Checking and singleton:**



1. **Backtracking using heuristic:**



1. **Backtracking with Forward Checking using heuristic:**



1. **Backtracking with Forward Checking and singleton using Heuristic:**

