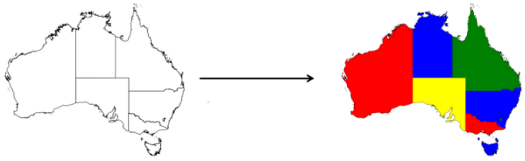
Map coloring using CSP algorithms

Amirhossein Foroughi
AmirKabir University, Tehran
Dr. Ghatee

Abstract

"Map-coloring" is a famous toy problem from cartography where we want to color a map in a way that two neighbouring states always have a different color. Interestingly, for a long time people knew that a minimum of four colors is required to solve this task. But there has not been a mathematical proof until the 1990. We'll first build a framework for CSPs that solves them using a simple recursive backtracking search. Then we'll use the framework to solve map coloring problem.



1 Introduction to CSP

We can break CSP problems including map coloring to smaller sections:

- *Variables*: the things we want to find acceptable values for them. In our case, these are the states and territories.
- *Domains*: these are the possible values that we can choose from for variables. In this case the domains of all the variables are the same.
- *Constraints*: these constraints are not to be violated by the variables. For example the colors of two states that have common borders can not have same color.

2 Building a constraint-satisfaction problem framework

Constraints are defined using a *Constraint* class. Each *Constraint* consists of the *variables* it constrains and a method that checks whether it's *satisfied()*. The determination of whether a constraint is satisfied is the main logic that goes into defining a specific constraint-satisfaction problem.

```python
from typing import Generic, TypeVar, Dict,
    List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V') # variable type
D = TypeVar('D') # domain type


# Base class for all constraints
class Constraint(Generic[V, D], ABC):
  # The variables that the constraint is
    between
  def __init__(self, variables: List[V]) ->
    None:
    self.variables = variables

  # Must be overridden by subclasses
  @abstractmethod
  def satisfied(self, assignment: Dict[V, D])
     -> bool:
```

The centerpiece of our constraint-satisfaction framework is a class called CSP. CSP is the gathering point for variables, domains, and constraints.

3 Backtracking implementation

Ok now we have design a framework we can apply it to every desirable problem. The idea here is to color a state. let us call it a node. We check all adjacent nodes to the node that we just colored; If we did not violate any of the constraints, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

Lets color this map.


```
colors = ['Red', 'Blue', 'Green']

states = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

neighbors = {}
neighbors['A'] = ['B', 'C']
neighbors['B'] = ['A', 'C', 'D']
neighbors['C'] = ['A', 'B', 'D', 'E', 'F']
neighbors['D'] = ['B', 'C', 'E']
neighbors['E'] = ['C', 'D','F']
neighbors['F'] = ['C', 'E']
neighbors['G'] = ['']

colors_of_states = {}
```

First we define variables that here is our states as *A, B, C, D, ...* and domain that here is the available colors. As 4-color theorem we can color every map with just 4 colors but here we can color this map with just three colors as *'Red', 'Blue', 'Green'*.

To define our adjacent matrice i used a python dictionary data structure.

In the last line we define an empty dictionary to store final results.

```
def promising(state, color):
  for neighbor in neighbors.get(state):
    color_of_neighbor = colors_of_states.get(
    neighbor)
    if color_of_neighbor == color:
    return False
  return True

def get_color_for_state(state):
  for color in colors:
    if promising(state, color):
      return color


if __name__ == "__main__":
  for state in states:
    colors_of_states[state] =
    get_color_for_state(state)

  print (colors_of_states)
```

Let us go through this part by part. In *main* we find color for each node one by one and use utility functions to check constraints.

In *get_color_for_state* we assign a color to a node, then we check this assignment by *promising* function. We check that node adjacents colors to check that they don't have same color; if so we return *False* but if all neighbors have different color we return *True*, so we can assign that color to that node.

But if *promising* returns *False* we check other colors. Otherwise, if we check all of the colors but there is no match that does not violate constraints it will returns *None*

Check this github repo for the code:

References

[1] XCSP3-core: A Format for Representing Constraint Satisfaction/Optimization Problems, Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, Cédric Piette, arXiv:2009.00514

[2] Scala-gopher: CSP-style programming techniques with idiomatic Scala, Ruslan Shevchenko, arXiv:1611.00602 [cs.PL]

[3] Sticky Brownian Rounding and its Applications to Constraint Satisfaction Problems, Sepehr Abbasi-Zadeh, Nikhil Bansal, Guru Guruganesh, Aleksandar Nikolov, Roy Schwartz, Mohit Singh, arXiv:1812.07769 [cs.DS]