Map Coloring as a Constraint Satisfaction Problem: Detailed Explanation and Step-by-Step Code

August 20, 2025

Abstract

This document explains the map coloring problem as a Constraint Satisfaction Problem (CSP) and provides a detailed, step-by-step Python implementation of a solver. It covers core concepts, heuristics (MRV, Degree, LCV), inference techniques (Forward Checking, AC-3), and a runnable, modular codebase. Each code block is explained line-by-line and provided as ready-to-run Python.

Contents

1 Introduction **2**

2 CSP model: variables, domains, neighbors **2**
2.1 CSP class: design and purpose 2

3 Heuristics: choosing variables and values **3**
3.1 Minimum Remaining Values (MRV) and Degree heuristic 3
3.2 Least Constraining Value (LCV) 3

4 Inference: Forward Checking and AC-3 **4**
4.1 Forward Checking (FC) 4
4.2 AC-3 (Arc Consistency) 4

5 Backtracking search (full algorithm) **5**

6 Example graphs and usage **6**

7 Running examples (step-by-step) **7**

8 Practical tips and common pitfalls **7**

9 Appendix: Full source (concise listing) **7**

1 Introduction

Map coloring is a canonical CSP: given regions and adjacency constraints, assign colors so that adjacent regions differ. This document will teach you:

- CSP formulation (variables, domains, constraints).
- Basic backtracking search.
- Heuristics to reduce search: MRV, Degree, LCV.
- Inference techniques: Forward Checking and AC-3 (arc consistency).
- Full, modular Python code with explanations for each part.

2 CSP model: variables, domains, neighbors

Variables Each region in the map (e.g., "WA", "NT") is a variable.

Domains The domain of a variable is the set of colors available (e.g., {Red, Green, Blue, Yellow}).

Constraints For every pair of adjacent regions (an edge), enforce that their colors are different:
$\text{Color}(A) \neq \text{Color}(B)$.

2.1 CSP class: design and purpose

The CSP class stores variables, their domains, adjacency (neighbors), and small counters for assignments and backtracks (useful for benchmarking). Below is the implementation; afterwards each component is explained.

Listing 1: CSP core class

```
from collections import deque
from typing import Dict, List, Set

class CSP:
    def __init__(self, variables: List[str], domains: Dict[str, List[str]],
        ↪ neighbors: Dict[str, Set[str]]):
        # Keep variables and a (copyable) domain dictionary.
        self.variables = list(variables)
        self.domains = {v: list(domains[v]) for v in self.variables}
        # Neighbors: adjacency list where neighbors[v] = set of adjacent variables.
        self.neighbors = {v: set(neighbors.get(v, set())) for v in self.variables}
        # Metrics for diagnostics
        self.n_assigns = 0
        self.n_backtracks = 0

    def consistent(self, var: str, value: str, assignment: Dict[str, str]) -> bool:
        '''Check the binary inequality constraint: neighbors can't have same value.
            ↪ '''
        for n in self.neighbors[var]:
            if assignment.get(n) == value:
                return False
        return True
```

```
21
22      def assign(self, var: str, value: str, assignment: Dict[str, str]):
23          assignment[var] = value
24          self.n_assigns += 1
25
26      def unassign(self, var: str, assignment: Dict[str, str]):
27          if var in assignment:
28              del assignment[var]
```

Notes:

- `self.domains` is copied so we can maintain a local domain copy per search instance.
- `consistent()` checks only binary constraints with already assigned neighbors; it does not mutate domains.
- `assign()` and `unassign()` manage the assignment dictionary and update counters.

3 Heuristics: choosing variables and values

3.1 Minimum Remaining Values (MRV) and Degree heuristic

MRV: choose the unassigned variable with the fewest available legal values (smallest domain). If there is a tie, Degree heuristic breaks ties by choosing the variable with the largest number of unassigned neighbors.

Listing 2: MRV with optional Degree tie-break

```
1  def select_unassigned_variable(csp: CSP, assignment: Dict[str, str], local_domains:
       ↪ Dict[str, List[str]], heuristic: str):
2      unassigned = [v for v in csp.variables if v not in assignment]
3      if heuristic == 'basic':
4          return unassigned[0]
5
6      # MRV: choose variable with the smallest domain size
7      m = min(len(local_domains[v]) for v in unassigned)
8      candidates = [v for v in unassigned if len(local_domains[v]) == m]
9
10     # Degree tie-break: pick the variable with most unassigned neighbors
11     if heuristic == 'mrv+deg' and len(candidates) > 1:
12         def degree_unassigned(v):
13             return sum(1 for n in csp.neighbors[v] if n not in assignment)
14         maxdeg = max(degree_unassigned(v) for v in candidates)
15         candidates = [v for v in candidates if degree_unassigned(v) == maxdeg]
16
17     return candidates[0]
```

3.2 Least Constraining Value (LCV)

LCV orders values so that the chosen value eliminates the fewest choices in neighbor domains.

Listing 3: LCV value ordering

```
1  def order_domain_values(csp: CSP, var: str, assignment: Dict[str, str],
       ↪ local_domains: Dict[str, List[str]], heuristic: str):
2      if heuristic != 'lcv':
3          return list(local_domains[var])
4
5      counts = []
6      for val in local_domains[var]:
7          eliminated = 0
```

```
        for n in csp.neighbors[var]:
            if n not in assignment:
                eliminated += sum(1 for v in local_domains[n] if v == val)
        counts.append((eliminated, val))
    counts.sort()  # fewest eliminations first
    return [val for _, val in counts]
```

4 Inference: Forward Checking and AC-3

Inference prunes domains and detects contradictions earlier.

4.1 Forward Checking (FC)

After assigning `var=value`, FC removes `value` from each unassigned neighbor's domain. If any neighbor's domain becomes empty, a failure is detected immediately.

Listing 4: Forward checking

```
def forward_checking(csp: CSP, var: str, value: str, assignment: Dict[str, str],
    ↪ local_domains: Dict[str, List[str]]):
    pruned = []
    for n in csp.neighbors[var]:
        if n in assignment:
            continue
        if value in local_domains[n]:
            local_domains[n].remove(value)
            pruned.append((n, value))
            if not local_domains[n]:
                return False, pruned
    return True, pruned
```

4.2 AC-3 (Arc Consistency)

AC-3 enforces arc consistency, which for binary constraints requires that for every value in a variable's domain there is some compatible value in each neighbor's domain. For inequality constraints, revise removes values that have no different value in the neighbor's domain.

Listing 5: AC-3 algorithm (arc consistency)

```
from collections import deque

def AC3(csp: CSP, local_domains: Dict[str, List[str]], arcs=None):
    queue = deque()
    if arcs is None:
        for Xi in csp.variables:
            for Xj in csp.neighbors[Xi]:
                queue.append((Xi, Xj))
    else:
        for arc in arcs:
            queue.append(arc)

    pruned = []

    def revise(Xi, Xj):
        revised = False
        to_remove = []
        for x in list(local_domains[Xi]):
            if not any(x != y for y in local_domains[Xj]):
                to_remove.append(x)
        for x in to_remove:
```

```
            local_domains[Xi].remove(x)
            pruned.append((Xi, x))
            revised = True
        return revised

    while queue:
        Xi, Xj = queue.popleft()
        if revise(Xi, Xj):
            if not local_domains[Xi]:
                return False, pruned
            for Xk in csp.neighbors[Xi] - {Xj}:
                queue.append((Xk, Xi))
    return True, pruned
```

5 Backtracking search (full algorithm)

Combine the heuristics and inference in a backtracking search loop. This implementation supports:

- `inference` = { `'none'`, `'fc'`, `'ac3'` }
- variable heuristics = { `'basic'`, `'mrv'`, `'mrv+deg'` }
- value heuristics = { `'basic'`, `'lcv'` }
- `use_ac3_at_start` boolean - run AC-3 before search

Listing 6: Backtracking search with pluggable heuristics and inference

```
import time

def backtracking_search(csp: CSP, inference='none', var_heuristic='basic',
    ↪ val_heuristic='basic', use_ac3_at_start=False, time_limit=10.0):
    start = time.time()
    assignment = {}
    csp.n_assigns = 0
    csp.n_backtracks = 0

    local_domains = {v: list(csp.domains[v]) for v in csp.variables}

    initial_pruned = []
    if use_ac3_at_start:
        ok, pruned0 = AC3(csp, local_domains)
        initial_pruned = pruned0
        if not ok:
            return None, {'success': False, 'time_s': time.time() - start, 'assigns
                ↪ ': csp.n_assigns, 'backtracks': csp.n_backtracks, 'initial_pruned
                ↪ ': len(initial_pruned)}

    def backtrack():
        if len(assignment) == len(csp.variables):
            return dict(assignment)

        var = select_unassigned_variable(csp, assignment, local_domains,
            ↪ var_heuristic)
        for value in order_domain_values(csp, var, assignment, local_domains,
            ↪ val_heuristic):
            if not csp.consistent(var, value, assignment):
                continue

            csp.assign(var, value, assignment)
```

```
28
29              pruned = []
30              ok = True
31              if inference == 'fc':
32                  ok, pruned = forward_checking(csp, var, value, assignment,
                        ↪ local_domains)
33              elif inference == 'ac3':
34                  arcs = [(n, var) for n in csp.neighbors[var]]
35                  ok, pruned = AC3(csp, local_domains, arcs=arcs)
36
37              if ok:
38                  result = backtrack()
39                  if result is not None:
40                      return result
41
42              # Undo assignment and prunings
43              csp.unassign(var, assignment)
44              for (v, val) in pruned:
45                  if val not in local_domains[v]:
46                      local_domains[v].append(val)
47
48          csp.n_backtracks += 1
49          return None
50
51      sol = backtrack()
52      metrics = {'success': sol is not None, 'time_s': time.time() - start, 'assigns'
            ↪ : csp.n_assigns, 'backtracks': csp.n_backtracks, 'initial_pruned': len(
            ↪ initial_pruned)}
53      return sol, metrics
```

6 Example graphs and usage

Below are helper functions for example maps (Australia and a simple cross graph). Use these to instantiate the CSP and run the solver.

Listing 7: Example graphs

```
1  def australia_graph():
2      regions = ["WA","NT","SA","Q","NSW","V","T"]
3      edges = {
4          ("WA","NT"),("WA","SA"),("NT","SA"),("NT","Q"),
5          ("SA","Q"),("SA","NSW"),("SA","V"),
6          ("Q","NSW"),("NSW","V")
7      }
8      neighbors = {r:set() for r in regions}
9      for a,b in edges:
10         neighbors[a].add(b); neighbors[b].add(a)
11     return regions, neighbors
12
13 def square_cross_graph():
14     regions = ["A","B","C","D","E"]
15     edges = {
16         ("A","B"),("B","C"),("C","D"),("D","A"),
17         ("E","A"),("E","B"),("E","C"),("E","D")
18     }
19     neighbors = {r:set() for r in regions}
20     for a,b in edges:
21         neighbors[a].add(b); neighbors[b].add(a)
22     return regions, neighbors
```

7 Running examples (step-by-step)

This section shows how to run the solver for Australia with different configurations, and how to interpret metrics.

Listing 8: Run examples and compare configs

```
# Build CSP for Australia
regions, neighbors = australia_graph()
colors = ["Red","Green","Blue","Yellow"]
domains = {r:list(colors) for r in regions}
csp = CSP(regions, domains, neighbors)

# Example: strong setting
sol, metrics = backtracking_search(csp, inference='fc', var_heuristic='mrv+deg',
    ↪ val_heuristic='lcv', use_ac3_at_start=True)
print("Solution:", sol)
print("Metrics:", metrics)
```

8 Practical tips and common pitfalls

- **Corner vs edge adjacency:** Represent an edge only when regions share a full boundary (not just a corner), or else you'll over-constrain.
- **Restoring pruned values:** Always store pruned pairs `(var, value)` and restore them on backtrack to preserve correct domain state.
- **MRV ties:** If MRV picks are frequently tied, the Degree tie-breaker yields significant benefit.
- **AC-3 tradeoffs:** Running AC-3 often reduces backtracking but costs CPU time up front. For large instances, try AC-3 as preprocessing, and use FC during search.
- **Symmetry:** Fix one region's color to eliminate symmetric solutions and reduce search (e.g., force the first variable to Red).

9 Appendix: Full source (concise listing)

For convenience, the major functions are listed again in one place (omitted here to avoid repeating the full content). Use previous sections to copy the exact implementations.

End of document.