Satisfying Constraints - By Will Gibson

In a constraint satisfaction problem such as this, we need to look at the map of adjacencies and determine how to color this map so that no two colors are matching. The simplest way to do this is a greedy approach, though there are other algorithms which will accomplish the task in more time, but with less domains used.

MY APPROACH, Greedy Algorithm

The approach I used in my project is a greedy algorithm, which takes and one by one places each location, in order, in the first domain that can accept it. This approach will be incredibly fast, as you use the minimum number of iterations for each location on the map. It does not, however, reveal the smallest number of domains that we can use. It is the simplest algorithm to create, and the simplest to look at and understand. I went with it because of those reasons. While there isn't a time constraint to how fast this program needs to finish, not taking up a bunch of time to recursively check the whole map is nice. Keeping things simple also means that I am not spending massive amounts of time creating the algorithm either.

CHATPGT's APPROACH - Recursive Coloring

As any good computer scientist would, I decided to go to our AI overlords for some insight into possible methods of doing this project. It gave me a few different approaches, including some code I was able to incorporate into my own greedy algorithm, but the one that it gave when I asked it for the single most accurate approach to finding the least amount of domains, no matter the time cost, was an interesting little recursive algorithm.

This algorithm took a similar greedy approach to mine, and it added an extra element. When it found a color that worked for one location, it went back and redid the previous ones, to see if it could condense the number of domains down. It seemed to be checking every single valid combination of domains and colors, to find the one that used the least number of domains. This approach is probably the most accurate you will see, though when I tested it on my code, it gave nearly identical outputs, just with a slightly longer runtime.

THE FINAL APPROACH - Trickling Domains

Finally, the last approach I thought of, and the one that seems to be implied as the “preferred” way of doing this, is by limiting how many domains the computer has access to. This algorithm would take and start the search at one domain, which, assuming any of the locations bordered another location, would immediately fail. Then, you give it a second domain to work with. This would work for slightly longer, and as long as there are no “corners” where more than 2 locations border each other, this would find a solution. From there, you increase the domains to three, and so on, until a solution is found, and from there, you know the smallest number of domains. This approach of constraining the amount of domains you can split the locations into will give you the minimum number more often than the greedy algorithm will. This algorithm needs to be 100% sure there is no solution with the current number of domains before it moves on.

Heuristically speaking, you can help this algorithm along by starting it with a number of domains greater than one. For our project, and for most similar maps of the world, starting with 3 domains would work great, and would save vast amounts of time, because it won’t have to try to find a solution with 1 or 2 domains. If there is a minimum number you can start it with, where you know there won’t be a solution that is less than that number, it will help you greatly.