COMP 560 – A1: Constraint Satisfaction

Sunnie Kwak

Backtracking Search

First, the data read in from the text file are put into 4 main different data structures: a list of colors, an adjacency dictionary, which has the states as keys and all its adjacent states in list form as the values, the solution dictionary, which has the states as keys and their assigned colors as the values, and a list of states, which is sorted by highest number of adjacent states first. The list of states is used to easily reference states by index, and it is sorted to reduce the number of steps (I found that starting with the most constrained variable optimized the program just a bit).

My implementation starts with a color (taken from the color list) and looks to see if a state (taken from the state list) can be assigned that color without violating the constraints, as in that no other adjacent state has that same color, with the `valid` method. This constraint is checked by retrieving all adjacent states from the `adj` dictionary, then checking the current assigned color of those adjacent states by referencing the `sol` dictionary. If a color can be assigned without violating any constraints, the `sol` dictionary will be updated so that the state has a newly assigned color. If the color cannot be assigned because it violates the constraint, `valid` will return `False`, and the next color in the color array will be tried in the same way. If all colors in the array have been tried, but no color has been assigned to the current state, meaning that there was a violation for all colors, the program will backtrack to the last recursive call and try the next color in the array for the previous state it has backtracked to.

This continues until all states have been assigned a color and stops when the index is no longer less than the number of states. The `sol` dictionary containing the states and the assigned colors will be printed, along with the number of steps it took. If all possibilities have been exhausted and not all states have been assigned a color, the system will print out “no solution”.

Local Search

Similar to the Backtracking Search code, I use the `colors` array, `adj` dictionary, and `sol` dictionary. I also use a `violations` dictionary, which holds the number of violations (number of adjacent states that have the same color) for each state, and a `maxviolstates` list that will hold only the states with the maximum number of violations for each iteration.

During set up, I first assign each state a random color (this is updated in the `sol` dictionary). Then, in the infinite while loop, which only breaks once the entire map has no violations, I count up the number of violations for each state and record that in the `violations` dictionary. Using that dictionary, I find the highest number of violations and put the state(s) with that number of violations in to the `maxviolstates` list. I do this because I found that it is more efficient to start with states that have more violations than randomly choosing any state with a violation. If the highest number of violations is 0, I break out of the loop and return `True` because that means the solution has been found. Within the `maxviolstates` list, I choose one random state. I do a random choice rather than just taking the first element of the list because choosing the first element every time risks putting the code into an infinite loop. I change the color of this chosen state and update the `sol` dictionary with the chosen color. At the end of every iteration of my while loop, I check to see if the program has been running for more than a minute. If it has, the

program times out and returns `False` (no solution has been found). This loop continues until the program times out or the max number of violations for all the states equals 0.

All work done by me (Sunnie Kwak)