Benjamin Levine
Daniel Su

COMP 560 HW 1

Background Description:

To represent the states in our map, we map each state to a Node object that stores information about the name, color, and adjacency list of neighbors. Once all possible nodes are added to our graph, we initialize each respective node's color to empty string to indicate the node color is uninitialized. Once our Python script reads in our node pairs, we represent a connection between the nodes by adding one another to the neighboring node representation. Since our graph is undirected (if two nodes are neighbors, they are both next to each other), we update both node's adjacency list.

Backtrack description:

The bulk of the code for the backtrack implementation can be found in the functions backtrack_search and assign_colors_recursive. It is a fairly standard recursive implementation of backtrack search which does a depth first search through the graph, assigning colors to nodes as it goes. We maintain arc consistency as we search by verifying that when assigning a color to a node, it will not leave any of the node's neighbors with no available colors. This is accomplished by the function get_available_colors which not only looks at which colors are immediately available for a given node, but also rules out colors that will leave any of its neighbors with no available colors. If we find that a node has no available colors, we begin backtracking immediately because we know the current configuration will not work. We also use the most constrained variable technique when deciding which values to assign. This is accomplished by slightly modifying the normal depth first search to make it such that when looking through the list of neighbors that each node has in order to recursively call on whichever ones have not yet been assigned a color, we first sort the neighbors from least to greatest in terms of the amount of available colors they have. We then make the recursive call on the neighbor which has the fewest available colors (being the most constrained) and we repeat this process until all of the neighbors have been assigned a color.

Local search description:

We represented our Local Search with our LocalSearch class. Once the run method is called, we verify to ensure our graph is not by default valid. If it's already valid, we return the colors and our solution. Otherwise, we preprocess our nodes with random colors from our graph's color set and run our traverse_graph_change_colors which attempts to randomly iterate through our graph and individually change each node color to a valid one. Note that we did NOT use recursion for LocalSearch because Python by default cannot handle a recursive depth of more than 1,000 without a "Maximum recursive depth" error. Because it's possible for us to reset our graph more than 1,000 times, we opted for an iterative approach. If we cannot assign our node a valid color, our approach restarts the board with random colors and tries the traverse_graph_change_colors again. Note that we attempted a solution which continues to search the board when this condition occurs. However, we found that this tends to take longer since in theory a solution may not be found in time. Yet, for our approach, frequently resetting

the board nets us the opportunity to not expend too much effort exploring “bad” random color configurations and “cut our losses” sooner. In the case when we’ve searched past a predetermined threshold (in our case we believed 1,000,000 node searches would be large enough), we’ll stop our search and conclude no solution could be found. We also added an additional variable which keeps track of the number of board resets we’ve performed. If we’ve exceeded that constant variable (in our case 10,000) we also conclude no solution could be found.

Individual Contributions:

Ben: Wrote the backtrack search code.

Daniel: Wrote the local search code.

Group effort: We both reviewed each other’s code and tested each implementation