Micah Brighton, James Bury, and Spencer Siegel

Comp 560: Assignment 1

In order to complete this assignment, we had to use two methodologies to color the maps we were given: backtracking search and local search. Each method required unique insights and strategies. When building our code, we first put together some code that would be required for both methodologies. We first wrote a class called “State.” In this class, we used methods that would allow us to read each state from the txt file and make each state its own class with a name, possible colors, a list of its neighboring states, and state-specific methods.

Backtracking was the first methodology we pursued. The first piece of completing the backtracking search was to add a new metric to each state: how many neighbors it was constraining. The number of neighbors that each state was constraining was equal to the number of its uncolored neighbors. Therefore, the number of constraining neighbors was initially equal to the number of neighbors, but each time a state was assigned a color, that state’s neighbors had it neighbors constrained variable decremented since there was one less neighbor whose color would be affected by a color assignment.

With each state now receiving a “constraining” variable that counts how many uncolored neighbors it constrains, the backtracking methodology had a key component it needed to determine what state to color. We made a loop that went through the states and checked for the most constrained variable, or the state with the least available color options. If we found multiple states tied for the most constrained variable, we then checked for the most constraining variable, which is the constraining variable outlined earlier. If there were multiple states tied for the most constrained variable and the most constraining variable, we randomly chose a state and colored it.

After selecting a state, we went to color it. In order to color the state, we looked through its domain to see its available colors. We then picked a random color from the available options

and colored the state that color. After assigning a state a color, we looped through the neighbors and did two things: 1) we implemented forward checking by removing the selected color from the neighboring states' available colors 2) we implemented arc consistency by checking to see if this removal of color from the neighbor's available colors left it with one remaining coloring option. If it did, then we recursively called this method to color this neighboring state and perform this same task. This forward checking did not count to our tracking of the number of steps.

In order to implement the backtracking portion, for each state, we iterate through the colors in its domain. We color the state and increment the number of steps. Then we find the next state to be colored and recall this method (execute). This recursive algorithm allows us to go back to the parent state and reassign it a different color if conflicts do not allow for a state to be assigned. The search ultimately ends when all states are colored. This algorithm worked with 3 steps on Australia and 42 steps on USA.

The next method we worked on was local search. We reused the same State class from above to store the information from the text file, then began our algorithm. The first step we took was randomly assigning a color to each state. Then, we began a timer for one minute to prevent the algorithm from running forever in case it got stuck on a local optimum. The next step we took was searching for the state that hurt our objective function the most. Since our objective function was minimizing the number connections between states of the same color, this meant we searched for the state that was connected to the most similarly colored states. In the case when there were multiple states that were tied for the most harmful to the objective function, we made a list of those states and randomly chose one of them.

From there, we decided which color that state should be to cause the fewest overall conflicts in the graph. We did this by assigning each available color to the state and seeing

which resulted in the fewest overall conflicts among all the states. Once again, if multiple colors tied in how few conflicts they resulted in, we put them in a list and randomly chose a color from them. At this point, we had assigned a color and thus completed a step. If the graph then had 0 connected states of the same color, we terminate the algorithm and present the results. If not, we decided whether we needed to restart by randomly assigning new colors to every state again.

To make this decision, we checked two things after every iteration. First, we created a deque at the beginning of the program with a maximum capacity equal to the number of possible colors + 1. After every step, the name of the chosen state is added to the deque, first removing the oldest one if the deque is already full. We decided that a good indication of whether we needed to randomly reassign all the state colors is if the algorithm is only changing two states back and forth for a consecutive series of steps. So to check for this, if the the deque was full but it only consisted of two unique states, we decided to randomly reassign all the states and effectively restart the algorithm. Additionally, after every step we check if there have been 1,000 steps since the last random reassign, in which case we also trigger a new random reassign. This entire process continues until it either reaches a solution with zero connected states of the same color, or 60 seconds pass. In our testing the algorithm arrived at a solution every time, however the amount of total steps and random reassigns varied between only a few steps and a few hundreds of steps.

Individual Contributions:

James Bury: Data processing, local search, and write-up

Micah Brighton: Backtracking search and write-up

Spencer Siegel: Backtracking search and write-up