Danielle du Preez
Maina Edula
Colin Kilmartin

Assignment 1: Constraint Satisfaction

Map

We created classes Country, State, and Color to represent a map. Country is basically a graph class, and State is a node class, except their parameters and functions are specific to the assignment. The functions for the local and backtracking searches are found in the Country class.

Each State object encapsulates a list of it's "forward neighbors" and "backwards neighbors". Adding a neighbor to the forward or backward neighbor list is based on the order of the states in the adjacency section of the text input file. Each Country object encapsulates a list of States and Colors.

Backtracking

We implemented our backtracking search by checking arc consistency for each node searched, and used the "most constraining variable" heuristic to determine which node to search next. After reading in the inputs, a priority queue is created based on the number of neighbors each state has, effectively how many other states' domains it constrains. Then, a recursive function is called to execute the backtracking search.

At each recursive call, we copy the current priority queue and domain set and feed it to a checker function to check for arc consistency if we were to assign a particular node a specific color. For each arc consistency check, we forward-check repeatedly until no more domains can be changed. We do this by adding a state to a forward-checking queue each time its domain is trimmed in the process of forward checking. When a domain becomes size zero, the arc consistency check returns false. When a domain becomes size one, the state is removed from the copied priority queue because if the current path leads to a valid solution, that state will no longer need to be checked in the main search.

Finally, if arc consistency returns true within the recursive call, the priority queue and domain set copies are passed up to the previous call and become the new local priority queue and domain set. If the priority queue is empty, a valid solution was reached and the local domain set becomes permanent domain set.

Local Search

Our initial local search algorithm was as follows: A temporary array is used to keep track of the colors assigned for each state. At the start, each state is assigned a random color. The steps counter is first incremented here. Then, a while loop starts that will keep running under the condition that the solution is not complete. This is checked by a helper method that iterates through every state and checks if it is valid, aka if it does not have the same color as any of its neighbors.

A for loop starts that will run 10 times, and each iteration, it picks a random state to start with. A helper method checkConstraints is used to calculate the number of violated constraints, or how many neighboring states the current state's color would conflict with. If the current number of conflicts is not 0, then it enters a loop where for each possible color, the loop calculates what the number of conflicts would be if the state changed color.

If a given color results in a lower number of conflicts than the previous color, the temporary array's spot for that state is updated to be that new color, and the loop continues and keeps the color with the lowest number of conflicts. In other words, the loop tests all the colors and keeps the least problematic color. This ends the while loop, so if the solution is not complete, the search will run again with 10 random starting positions.

The results with this method were not consistently successful, so the search was then optimized:

First, the states are colored randomly. Then, a start state is randomly chosen, and its neighbors are assessed to see whether or not there are conflicts. A global array keeps track of the number of conflicts caused by each of the states. We change the color of the state that has historically caused the most conflicts to the color that most reduces the overall number of conflicts, and then clear that state's "conflict history" count. If the same state has the highest overall conflicts twice in a row, we know we are "stuck", so we clear the entire conflict histories and start re-checking at a random node.

We observed that in the few successful runs we had before implementing random resets that it would take hundreds of thousands of steps to reach a solution. So, we approximated that if we reached a million steps, we could be confident that we were “stuck” and had to reset. We added a local step counter that, when it reached a million, would reset all the domains to random colors again. With this method, we were able to consistently get a solution from the local search.

Contributions

Danielle - made the base classes, wrote backtracking search, optimized local search
Maina - wrote initial version of local search
Collin - debugged initial design