HW 1

Group members: Anna Serenis, Kirsten Freeman

Backtracking Search: Implemented by Anna Serenius

Our implementation of backtracking search involves arc consistency and most constrained variable selection. The data structures used in our implementation include a node for each state, a priority queue of nodes named "states" that is organized using a custom comparator called "backtracking comparator," a set of arcs, and an initially empty set of nodes called "assigned." Each node has a set of arcs that represent its constraints, a set of colors for its domain, and a list of sets called "snapshots" that is used in the arc consistency part of the implementation. The backtracking comparator used to maintain the priority queue compares nodes based on the sizes of their domains so that when a node is removed from the priority queue it is the most constrained.

The search begins by removing a node from the priority queue and deep copying the state's domain so as to avoid an error being caused during arc consistency (w/o deep copying the domain, arc consistency would attempt to modify the state's domain while the domain was being iterated over within a for-loop). For each color in the node's domain the following is done:

1. A function called isSafe is called, which checks to see if the node can be assigned the color without violating any constraints. This check is done by iterating through the node's adjacent nodes and seeing if any of the adjacent nodes' domain are of size 1 and contain the same color.
2. The node is added to the set of assigned nodes.
3. A snapshot of the state's domain is taken, which creates a new set containing the colors in the node's domain before the domain is reduced to contain just the singular color of its assignment. This snapshot is taken in case arc consistency or the next recursive call of backtrack is unsuccessful.
4. The node's domain is reduced to contain just the one color to which it is being assigned.
5. A new set of nodes is created called "inferences," to which the node is added and to which any nodes affected during arc consistency are added as well.
6. Arc consistency is now performed:
    a. while a queue of arcs is not empty, an arc is removed.
    b. The function "revise" is called on the two nodes of the arc:
        i. A snapshot is taken of node 1's domain.
        ii. For each color in node 1's domain, if there is not a safe color in node 2's domain, the color is removed from node 1's domain.

iii. If node 1's domain was revised, node 1 is added to the set of nodes "inferences," and true is returned for the call of "revise," otherwise false is returned.

c. If "revise" returns true, the size of node 1's domain is checked: a domain of size 0 means that node 1 has no safe color assignments, and so arc consistency returns false. Otherwise, all of the arc involving node 1 are added to the queue, and the while loop is iterated again.

d. If "revise" is successful for every arc, arc consistency returns true.

7. If arc consistency is successful, backtrack recurses.
8. If arc consistency is unsuccessful, the node is removed from the set "assigned" and the function undoInferences is called, which for every node in the set inferences, resets the node's domain to what it was in the snapshot last taken.
9. Backtrack returns false if there is no color in the state's domain that is safe. Backtrack returns true if all the nodes have been added successfully to the "assigned" set.

Local Search: Implemented by Kirsten Freeman

For this part of the project, we decided to implement Local Search by using Steepest-Ascent Hill-Climbing with Restarts (whenever the current state reaches a local maximum).

According to the definition in the book, in order to implement Steepest-Ascent Hill-Climbing, we should start by generating a random state. Then, at each step of hill climbing, the current state is replaced by the best neighboring state, or the algorithm will terminate if we found a solution.

So, in order to implement this algorithm, we started by using methods in the java.util.Random library to choose random colors for each node on the map (implemented in the generateRandomStart() method). After this, we check to see if the randomly-generated start state is a solution; if it is, then we return the set of states and if not, we enter a while loop.

In the while loop, a couple different methods are called, and we also keep track of the number of times we've been looping. The first method (whichNeighborNext()) called will choose the node that has the highest number of neighbors with the same color. If there are multiple nodes with the same number of matching-colored neighbors, then this method will choose the one that has the smallest number of total neighbors. However, if this method chooses the same node it chose on the last iteration of the while loop (if this isn't our first time through the while loop of course) then another method (chooseAlternateNode(Node node)) is called, which uses the same logic as the previous method, except it doesn't allow us to choose the same node as last time. After we have chosen the node we want to "work on," a method is called that allows

us to change the color of the node (called chooseNextColor()). In this method, we keep track of what the effect will be if we change the color of the node to each of the different colors available. In the end, we choose to change the node to the color that matches the least amount of neighboring nodes. If there are two colors that result in the same number of matching neighboring colors, we'll choose the color that matches that of the neighboring node with the least amount of edges. This resulting state should be the best possible state we could move to (in one step) from the previous state. After this is done, isSolution() is called, and if the current state isn't a solution, we keep looping.

Lastly, if we get stuck in a local maxima and can't make any further progress, then we restart our search in order to ensure that we will eventually find a solution.