Seamus Holland & Michael Quint

HW1

Our program relies on a basic backtrack search in addition to a local search that uses Simulated Annealing to all but guarantee an efficient solution.

Input Read

The user begins by executing our main file, `a1.py`. This file first reads the input from the `input.txt` file in three phases: insertion of colors, insertion of adjacency list keys, and insertion of adjacency list values. The `colors` list simply contains all available colors, while the `adj_list` dictionary represents the US map as an adjacency list. The program then initiates our backtracking search, which can be found in `BacktrackingSearch.py`.

Backtracking Search

Our Backtracking Search acts like a standard recursive depth-first search with extra checks. At each level of recursion – that is, at each state node – we check whether or not there are any colors available to that node, based on the colors of the adjacent nodes that have already been colored.

If there are no colors available for that node, then we remove it from our list answer dictionary, `state_color_dict`. Then, we backtrack to the previous node by returning `False`. At the previous level of recursion and previous node, such a result indicates that the search failed with this node's current color. We thus change this node's color by incrementing its `pointer` variable, which refers to the current position in the list of available colors. If there are no more available colors, we must backtrack further, again by returning `False`. All backtracks, as well as all color assignments, are recorded in the output.

Once the last adjacent node in the search has been colored successfully, the search has produced a solution for that set of nodes. All of this logic, found in the `bt_utility` function, is applied to every distinct set of connected nodes. Then, the full set of colorings is printed.

We felt that as this search only backtracks twice with the given input, further improvements were not necessary.

Local Search

Our Local Search first randomly assigns a color to every state in the set. Then, a neighboring candidate solution is identified by randomly selecting one node and randomly changing its color.

A cost function is established which simply counts each edge connecting two states of the same color as 1, then sums them together. If a candidate solution has a lower cost than the current solution, that candidate is *accepted*, and becomes the current solution.

With this approach alone, the algorithm almost never produced a correct solution within the time constraint. In order to remedy this, we implemented Simulated Annealing to avoid arriving at local minima. This optimization heuristic allows candidates with higher costs to be occasionally be accepted in order to fully explore the problem space. Higher cost solutions are accepted according to the following function:

$$e^{-\Delta D/T} > R\,(0,\ 1),$$

Where *ΔD* is the change in cost (negative indicating a "good trade"), *T* is an arbitrary "temperature" variable, and *R(0,1)* is a random decimal between 0 and 1. We set our initial *T* to the standard 1. Every 100 runs, *T* goes through a "cooling" period, where it is reduced by a factor of .9. Thus, after every cooling period, it becomes harder to accept higher cost solutions, as the focus of the algorithm turns from exploring the problem space to finding the globally optimum solution – which it generally does very quickly.

There is also an asynchronous timer that ensures that local search does not run for longer than 60 seconds, although it never hit that constraint once we implemented Simulated Annealing.

Individual Contributions

Aside from some small commits, Michael and Seamus used pair programming to complete this project, with Seamus doing the actual typing. Thus, the work was evenly split between us, and we essentially each tackled every component together.