Programming Lab 5: Chromatic Number of a Graph

This assignment was based around graph theory, specifically he concepts of breadth-first search and chromatic numbers. Using a political map of South America and an abstracted graph representing the same, we were to:

1. Generate an adjacency matrix;
2. Create a Graph class that can use this as its internal representation and perform breadth-first searching;
3. Derive a Map class which can read in the list of countries and colors, and use those to calculate the chromatic number of the graph.

Structure

The Adjacency Matrix

An Adjacency Matrix is a mathematical construct used to represent graphs as a matrix of booleans representing connections. Each 1 in the matrix is a connecting edge, and each 0 is a non-connection. To check if two arbitrary nodes, m and n, are adjacent, we check position (n, m) in the adjacency matrix; a 1 will represent a connection, and a 0 no connection.

The political map of a continent is a simple graph; countries do not connect to themselves, and borders are by nature two-way. This simplifies the adjacency matrix considerably, and the resulting matrix is mirrored across the diagonal:

$$\begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}$$

The Graph Class

The Graph class exists to contain the Adjacency Matrix and the array of nodes which represent countries, and to enable breadth-first searching on using those structures. Since it is a matrix of nothing but 1s and 0s, the adjacency matrix is best representing in code as a two-dimensional array of booleans. If there is an adjacency between nodes n and m, then the value at `adjacencies[n][m]` will be `true`, and if there is not, it will be `false`.

To maintain the order of nodes, they are stored in a parallel array which corresponds to the matrix by index. This allows the encapsulation of Node information; each node only contains information about itself, and no pointers to other nodes. The Graph can read the information in its aggregated nodes to structure and search, but the nodes do not know anything about each other. In this way I hybridized the adjacency matrix and adjacency list approaches.

The only methods in the class are loading the adjacencies and nodes, and performing the breadth-first search. The loading is straight-forward; I used vectors as opposed to arrays to facilitate use of this class for future adaptations. The Breadth-first searching is built to return a queue in the correct order for some other use, rather than writing out the results. It takes one argument, that of a starting node, and then references the adjacency matrix and array of nodes to return the correct order.

The Map Class

The Chromatic Map class inherits all of the functionality of the Graph class, and includes the necessary methods to include colors, and find the smallest number of colors that can be used to color in the graph such that no adjacent nodes are the same color. The allowed colors can be read from a file and stored in a map, which is then used as a printable enumerator.

The largest piece of the Chromatic Map class is the calculation of the chromatic number. I have programmed it to do this by brute force.

Analysis

Adjacency Matrix vs. Adjacency Lists

An adjacency matrix uses one monolithic data structure to record all of the edges in a graph as simple booleans. It is implemented directly within the Graph class of this project.

The adjacency matrix is a good choice for graphs that may need to be modified frequently, as adding or removing an edge is a single operation of setting a boolean value to true or false. It is also reasonable if you just need to look up if there is a connection between two specific points at any given time, as this is also a single operation, reading the same boolean.

It is inefficient if the entire graph must be checked for edges, as this will always require traversing across n lists of length$n$, resulting in this *always* taking n^2 operations to complete. Likewise, this arrangement stores a boolean for every possible edge, giving us n^2 space complexity in all cases.

The other option is adjacency lists. In adjacency lists, only the current existing edges are recorded, with no entry if there is no edge. I have actually implemented this by including the vectors of adjacencies in my Nodes, and am using it during the calculation of the chromatic number instead of the adjacency matrix.

The advantages of the list over the matrix are in the best-case time and space complexities. While the worst-case remains n^2 for both, removing all non-adjacencies from the records can greatly reduce it in the best and average cases. In the best case, each node has a small number of adjacencies: if each node were adjacent to $log_{10}(n)$ nodes, accessing all edges will be in $log_{10}(n)$ time, and only require $log_{10}(n)$ booleans in memory, dramatically reducing both the time and space complexities.

As a downside, changing the matrix or checking for a single edge is more time-consuming. Rather than a random-access index, a list must be iterated to check for the edge in question. If the adjacency list used to check is full, this will require n operations to find. Adding or removing an edge is similarly more complex. In the matrix, it is an indexed access and assignment; in the list

format, two lists must be accessed and modified, requiring the search for an edge, and then insertion methods that can be more complex if the list is sorted.

Breadth-First Search

Breadth-first searching is where, from an arbitrary starting node, the graph is searched for some information one layer at a time. First checking the starting node, then all of the nodes connected to that one, then all of the nodes connected to those nodes, and so on until the entire graph (or connected portion) has been searched. Like all searching algorithms, the best case is that you find what you want in the first node checked, and the worst case is that you don't find it at all after traversing the entire data set. This gives us $\Omega(1)$ and $O(n)$ time complexities, respectively. As is typical with $O(n)$ algorithms, the average case is that the desired node is found halfway through, giving us $\Theta(\frac{n}{2})$, which simplifies to $\Theta(n)$.

In the case we have, we are doing more of a breadth-first comparison than a search. Since we have to compare each node to its neighbors every time we reach a new node, the worst case actually expands to the case where each node is compared to every other node, resulting in a time complexity of $O(n^2)$.