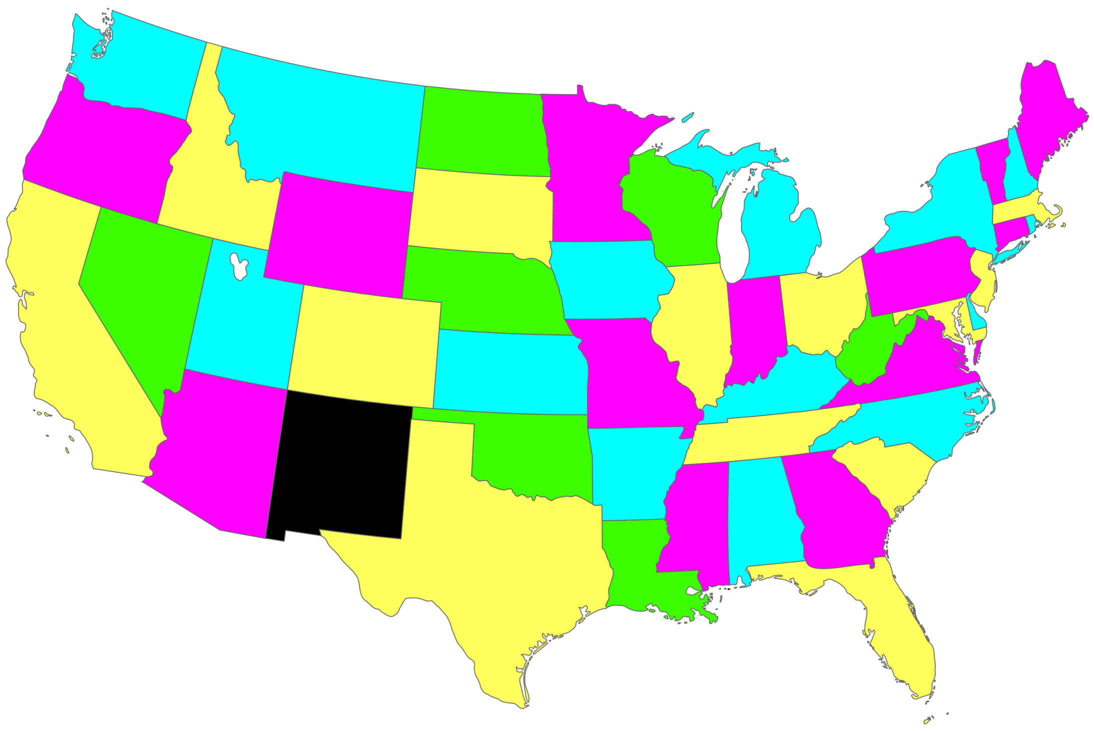
Data Structures for Problem Solving vcss242 Backtracking Lecture

12/03/2012

1 Problem

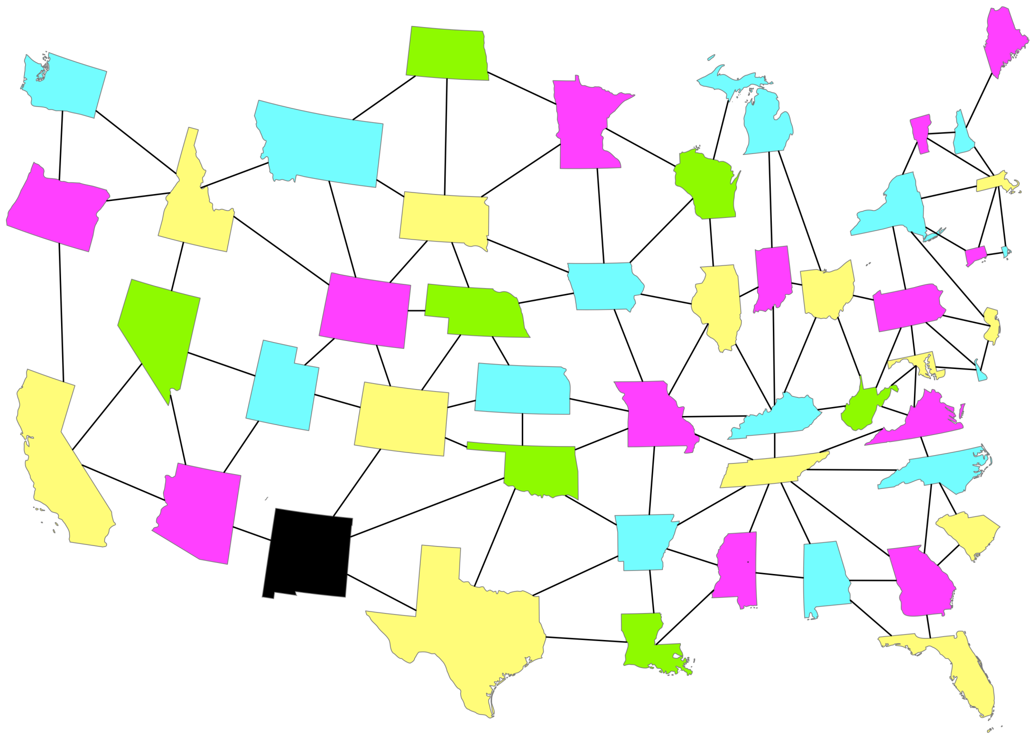
You are a cartographer. Your latest project is to develop a map for the "lower 48" states of the United States of America. Each state in the map should be made easily distinguishable from its neighbors through the use of distinct colors. However, the more colors you use, the more your costs go up. What is the minimum number of colors needed to ensure that each state can be colored a different color than any of its neighbors?



2 Analysis and Solution Design

2.1 Data Representation

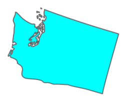
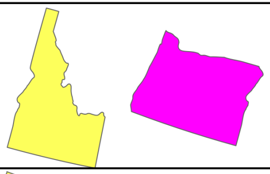
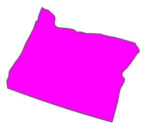
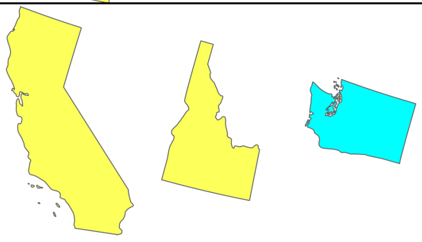
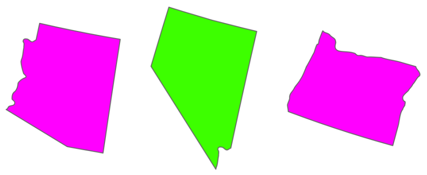
When one thinks of maps one often thinks of coordinates, latitude, and longitude. This in turn leads to a representation in terms of a *grid*, where states have coordinates. Unfortunately, this solution does not lend itself to the primary need of knowing which states are neighbors, since states come in all shapes and sizes, with differing numbers of neighbors.



Looking at the above figure's "exploded" view, it is actually more natural to represent the situation with an *adjacency list*.

- The collection of states will be represented by a collection of nodes.
- Each node will have a string name and (eventually) a color.
- Each node object will also have a slot that holds a list of adjacent nodes.

Below you will see a graphic representation of the adjacency list for a small part of the map of the U.S.A. — the west coast.

STATE	NEIGHBORING STATES
	
	
	

Again, this can be represented in code as a set of instances of a node class containing a state name (or other identifier) and a list of adjacent nodes. Since the colors of the states will be constantly changing, we will move those colorings into a separate data structure for more efficient manipulation.

2.2 A Map Coloring Algorithm

Before the algorithm is described, a slight modification will be made. The new problem statement is the following.

Can a map M be colored with no more than C colors?

If the more general question, what is the minimum number of colors needed, is required, one can simply repeat the algorithm with a smaller and smaller C until the answer is "no". (Although there are better ways.)

The algorithm for coloring interconnected nodes, e.g., a map, can be viewed as a tree search, but the "nodes" of that tree may not be what you expect. Each tree "node" is a possible coloring of the map. Assume that the root node represents a map with no states colored yet. A coloring $c2$ is a child of the coloring $c1$ if one of the states that was uncolored in $c1$ has had its color assigned in $c2$. A complete coloring for this problem is a set of 48 colors. For that many states and 5 colors (plus the uncolored state), there could be 6^{48} or 2.2×10^{37} different colorings!

Hopefully we can avoid having to check all these colorings. In fact, we are not even going to build the tree data structure explicitly. Here is a very rough description of the approach that will be followed.

> Color all nodes as being uncolored (the *unassigned* color).
> Go to the first node.
> Assign a color, and make note of what other colors haven't been tried.
> Go to the next node, and repeat the process.
> ⋮
> Eventually we will either discover a valid complete coloring or an invalid coloring.
> If invalid, back up to an earlier node where there are still some choices left and make a different choice. Proceed forward as before.

This approach, which must still be expressed in proper algorithmic form, is called *backtracking* because we are allowed to undo some decisions to go back to a point where a choice was made, then make a different choice and proceed again.

Let's add one more trick that will help us evaluate far fewer colorings. Even if some of the nodes are not yet colored, we should give up if two neighboring states are already colored the same. This technique is called *pruning*. It comes from the metaphor of the search space being a tree. Abandoning certain paths appears like cutting branches from the tree.

Here is the pseudo-code for this algorithm. Recall that the coloring is separated out into its own data structure. The map comes along for the ride, simply representing the unchanging connections between the nodes (states).

```
define find_coloring(current_coloring, map)
   observe the current_coloring of the map.
   if the coloring is complete (no uncolored states)
      return the current_coloring
   otherwise
      choose an as yet uncolored node.
      generate new colorings based on that node getting
                                 every possible color.
      for each new coloring c
          if the coloring is valid (no adjacent states have the same color)
              solution = find_coloring(c, map)
              if the solution indicates success
                 return the solution
          # Repeating the loop body is the "backtrack"
          # that will try the other colorings.
      return Failure
```

We assume that, each time `find_coloring` is called, including the first time, the current-coloring is valid. If we assume `find_coloring` is first called with all nodes (states) uncolored, we will be fine.

This algorithm does backtracking because of the loop. Imagine entering the loop for the first time. You try a valid coloring *c* and call your function recursively. Maybe it is recursively called several more times. Finally, a point is reached where coloring fails. Eventually, these calls return back to yours, and you try the next valid coloring, which means that the current node gets a color it has not gotten before. Since your activation, or instance, of the function remembers what has and has not been tried, it is realizing the informal backtracking strategy mentioned previously.

Consult the file `map_coloring.py` for the Python implementation. One important thing to note is that all the activations of the `find_coloring` function cannot share a coloring structure. Each time a new coloring is made (a new node in the search tree), it must be generated by modifying a copy of the old coloring.

2.3 A More General Algorithm

Can you detect a more general search strategy in the algorithm? It could be applied to other problems that, on the surface, appear to be quite different. (Perhaps your laboratory assignment will be one of them!) In the Python code below, the value `None` is returned when the coloring tested fails.

```
def solve(configuration):                          # (i)
   """solve: Config -> Config or None"""
   if isGoal(configuration):                       # (ii)
      return configuration
   else:
      for child in successors(configuration): # (iii)
         if isValid(child):                        # (iv)
            solution = solve(child)
            if solution != None:
               return solution
      return None
```

The parts of the above algorithm that need to be defined for the specific problem at hand have been marked with Roman numeral comments. Specifically, it is necessary to (*i*) define the configuration structure, (*ii*) define what it means for a configuration to be a goal configuration, (*iii*) define how a list of successor configurations is generated, and (*iv*) define what it means for a configuration to be valid[1]. Additional arguments to the `solve` function may or may not be needed depending whether or not additional *solve features* are desired (e.g. watching the search).

2.4 Complexity

The coloring problem for a general set of interconnected nodes is a well-known problem in computer science. Below is a more precise mathematical description of it. We can describe the problem with a tuple of two sets, N, the set of nodes, and C, the set of connections between them.

1. Instead of using `isValid` in the definition of `solve`, it is possible to use it only in `successors`.

For a rough analysis of our `find_coloring`, consider a less efficient version that does not perform pruning and instead checks the validity of a coloring only when a coloring is complete. In the worst case, the algorithm must check the validity of every possible complete coloring. Since there are k ways to color each node and $|N|$ nodes, there are $k^{|N|}$ complete colorings of the map. For each complete coloring, the validation check must examine the pair of adjacent nodes corresponding to each member of the set C. Therefore, complexity of this non-pruning version of `find_coloring` is $O(|C| \times k^{|N|})$.

There is no simple characterization of the complexity of the pruning version of `find_coloring`; the effectiveness of pruning depends upon the specific problem type and the order in which nodes are colored. Nonetheless, the pruning version of `find_coloring` can be no worse than the worst-case of the non-pruning version of `find_coloring`. In practice, a pruning backtracking implementation is usually *much* better than $O(|C| \times k^{|N|})$.

3 Testing

How many colors do you think are needed for the continental United States?

We have provided the following sets of test data.

- US — the entire continental
- USA West — the five westernmost states
- NE — the northeast states
- NEng — the New England states
- One — just one state, no connections

What is the largest subregion of the US you can find that requires only 3 colors? 2 colors?

Because of the pruning that is part of the algorithm, there is a significant variation in execution time depending on the ordering of the nodes in the adjacency list structure. Therefore the code we provide that is based on that algorithm could take an hour or a few seconds. As provided, the adjacency file puts states with the most neighbors near the front of the list. If the ordering is in our favor, we will quickly find the impossible situations where there are not enough colors for a set of adjacent states. If the file is reversed (see the "USRev" files), things really slow down.

Other things that affect the running time are the following.

1. Representing state and color names as strings instead of integers: comparison of strings requires character-by-character comparison. If the average size of a string is s characters, much of the program slows down by a factor of s.
2. The size of the set of colors: This is a "Goldilocks" situation! If the set is too small, an impossible situation is found very early and the program quickly halts in failure. If too large, it is more likely that an early color choice for each state will just work, so there is less backtracking.

4 Further Reading

The coloring problem is one of a set of very famous problems called $\mathcal{NP}$-complete problems whose known solutions are intractable (the number of cases to test grows exponentially with the input), but for which there is as yet no proof that it must take that much time.

Maps represent a very special interconnected node problem. The word used is *planar* because the nodes can be arranged so that none of the connections cross over each other, i.e., two dimensions suffice. If you look at the exploded view of the US map, you will see that it is planar[2]. It has been proved that all planar node networks can be colored using a maximum of four colors. The proof itself is interesting. It was the first to be proved by a computer program, because the number of cases to examine was deemed too large to be done by hand[3]. Just because four colors suffice does not imply that the complexity of the algorithm is diminished; success and speed are two different things! But in this case it is true. The complexity for the best algorithm that colors planar node networks with four colors runs in quadratic ($O(n^2)$) time[4].

2. However, if you accept that Utah, Arizona, New Mexico, and Colorado all touch each other at the Four Corners point, the network of states is no longer planar. Planar node networks result from defining neighboring states as sharing a boundary line, not a single point.

3. See `http://unhmagazine.unh.edu/sp02/mathpioneers.html`

4. Robertson, N., Sanders, D. P., Seymour, P., and Thomas, R. 1996. Efficiently four-coloring planar graphs. 1996. DOI = `http://doi.acm.org/10.1145/237814.238005`