A greedy graph-coloring algorithm

We present an algorithm to color the vertices of an undirected graph so that neighbors have different colors. It is an abstract algorithm, in the sense that we number the n vertices 0, 1, …, n-1 and assume we have n colors, also numbered 0, 1, …, n -1. In an OO situation, the vertices and colors will likely be objects.

We allow for n colors, for if the graph is complete (each vertex has n-1 neighbors), n colors are necessary.

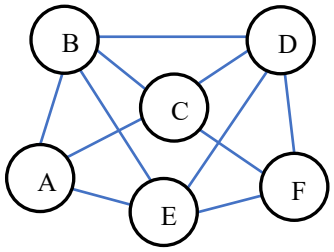
A very abstract algorithm

The abstract algorithm we now describe is how you should remember the algorithm. Knowing this version, you will be able to use it to color a given small graph by hand. You don't need to see code to do it.

We want to use as few colors as possible. Therefore: it makes sense always to use the smallest possible color when coloring a vertex. We call this a *greedy* choice. The notion of a *greedy algorithm* is covered in JavaHyperText. Our algorithm, then, is: Color the vertices one by one, as follows:

To color a vertex, choose the smallest color that is not already the color of a neighbor.

We use this algorithm to color the graph that appears to the right. For clarity, we use vertex names rather than integers.



1. Give A the color 0.
2. Give B the color 1, since its colored neighbors have the color 0.
3. Give C the color 2, since its colored neighbors have the colors 0 and 1.
4. Give D the color 0, since its colored neighbors have colors 1 and 2.
5. Give E the color 2, since its colored neighbors have colors 0, 1.
6. Give F the color 1, since its colored neighbors have colors 0, 2.

We want to be estimate the execution time and space requirements of this algorithm. That requires us to go into more detail on how we can determine the colors of the neighbors of a vertex. We proceed to do this.

The array to contain the coloring

To the right, we declare the array that will contain the resulting coloring: `color[v]` contains the color assigned to vertex `v`. If `v` has not been colored, then `color[v]` is bigger than any color number. This implementation of not being colored has been chosen just to make the presentation of the algorithm easier. We assume that, initially, all nodes are not colored.

```
/** color[v] is the color assigned
  * to vertex v. If v has no color
  * then color[v] = n */
int[] color= …;
```

Method findColorForVertex Read the specification of this method, given to the right. We step through the method.

```
/** Return the smallest color that is not
    the color of a neighbor of vertex v. */
int findColorForVertex(int v) {
  int d= number of neighbors of v;
  boolean[] usedC=  new boolean[d+1];

  for each neighbor w of v {
    if (color[w] <= d)
        usedC[color[w]]= true;
  }

  return the smallest c such that
        usedC[c] is false;
}
```

Vertex `v` has `d` neighbors. At the end of this method, local array `usedC` will contain values as follows: `usedC[c]` will be true iff some neighbor of `v` has color `c`. Initially, by the rules of Java, all elements of `usedC` are false.

When finished, we want `usedC` to contain at least one false element, representing a color that is not the color of a neighbor of `v`. Making the size of `usedC` one more than the number of neighbors guarantees that.

The loop looks at each neighbor in turn and, if its color is less than the size of the array, `usedC` is changed accordingly. Note that colors greater than `d` are ignored since we are interested only in finding the smallest color that is not the color of a neighbor. Also, uncolored vertices, which have color value `n`, are ignored.

Why don't we declare array `usedC` before doing any coloring, so it is created only once? We would then need a loop in the beginning of `findColorForVertex` to set its first `d+1` values to false. Further, we would need to define and explain `usedC` early, along with the explanation of array `color`. That change complicates things, with no measurable benefit. It doesn't change the time or space complexity of the algorithm. We prefer simplicity.

Coloring the graph

Now that we have method `findColorForVertex`, the simple algorithm to color the graph is given to the right.

```
/** Color the graph */
Initialize array color, as mentioned earlier;
for each node v of the graph {
   color[v]= findColorForVertex(v);
}
```

Time and space analayis

Assume the graph is given as an adjacency list in some form and that it takes constant time to get the outdegree of a vertex. The graph has `n` vertices and `e` edges. We determine its time and space complexity —the space needed beyond array `color`, which contains the result.

The algorithm has an outer loop that processes each vertex in turn. Each iteration colors one node. Thus, the worst case time is at least `O(n)`.

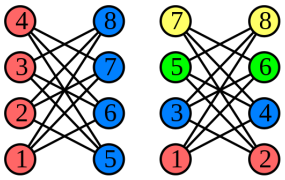
We investigate method `findColorForVertex`. It requires boolean array `usedC`, whose size is proportional to the number of neighbors of the vertex. Thus, the space complexity is `O(`maximum outdegree of all vertices`)`. Considering only dense graphs, that's `O(n)`. Considering only sparse graphs like a map of a town, it's `O(1)` —generally, the largest outdegree of any intersection on a map is 4 or 5, perhaps a bit more.

We now investigate the total time, over all calls of `findColorForVertex`, showing that it is `O(n+e)`. Both terms are necessary. For example, for graphs with 0 edges, it still takes time `O(n)`. Considering only sparse graphs, like maps of towns, `e` is at most linear in `n`, so it reduces to `O(n)`. Considering dense graphs, it is `O(n*n)`.

1. The assignment to `d` takes constant time, so it contributes time `O(n)`.
2. The assignment to `usedC` takes time proportional to the outdegree of the vertex. Over all calls, that's all edges (twice), so that's time `O(e)`.
3. In total, the loop over the neighbors contributes time `O(n+e)`. That loop is executed `n` times, once for each vertex —that's where the `n` comes from. In total, all `e` edges are processed, and the repetend of the loop takes constant time.
4. The return statement requires in the worst case all elements of array `usedC` to be referenced. The size of the array is `O(`outdegree of vertex `v`), so in total the time for this statement is `O(n+e)`.

How good is our greedy coloring algorithm?

It turns out that our greedy algorithm —greedy because at each step it chooses the smallest color that is not the color of a neighbor— does not always perform well. How well it performs depends on the order in which nodes are processed.



The image to the right[1] shows two versions of the same graph. Their vertices are numbered differently, each giving an order in which to process them during our coloring algorithm. This is a *bipartite* graph. The vertices of a bipartite graph can be split into two groups such that all neighbors of one group are in the other group.

Consider the leftmost graph.When processing vertices 1, 2, 3, 4, no neighbor has a color yet, so the first color, say red, is used. When processing vertices 5, 6, 7, 8, each neighbor is colored red, so the second color, say blue, is used. Obviously, a bipartite graph can be colored with two colors.

Consider coloring the rightmost graph. (1) Vertex 1 gets the color red, then vertex 2 gets the color red. (2) Vertex 3 gets the second color blue because its neighbor, 2, is red, and similarly for vertex 4. (3) Vertex 5 already has a red and a blue neighbor, and the same for vertex 6, so they get the third color, green. (4) Vertices 7 and 8 are given a fourth color, yellow. As can be seen four colors are needed. This is the worst coloring for this graph using our coloring algorithm.

In graph theory, the *Grundy number* of an undirected graph is the maximum number of colors that have to be used by our greedy coloring algorithm, considering all possible orderings of the vertices. Grundy numbers are named after P.M. Grundy, who used this concept for directed graphs in 1939. For more information, visit the webpage given in the footnote.

[1] Taken from https://en.wikipedia.org/wiki/Grundy_number#/media/File:Greedy_colourings.svg