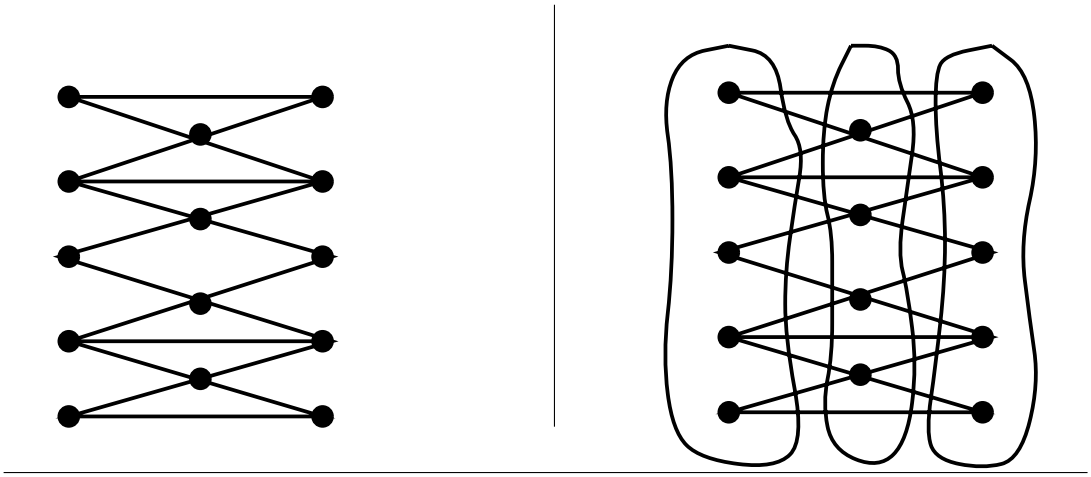

INPUT OUTPUT

16.7 Vertex Coloring

Input description: A graph $G = (V, E)$.

Problem description: Color the vertices of V using the minimum number of colors such that i and j have different colors for all $(i, j) \in E$.

Discussion: Vertex coloring arises in many scheduling and clustering applications. Register allocation in compiler optimization is a canonical application of coloring. Each variable in a given program fragment has a range of times during which its value must be kept intact, in particular after it is initialized and before its final use. Any two variables whose life spans intersect cannot be placed in the same register. Construct a graph where each vertex corresponds to a variable, with an edge between any two vertices whose variable life spans intersect. Since none of the variables assigned the same color clash, they all can be assigned to the same register.

No conflicts will occur if each vertex is colored using a distinct color. But computers have a limited number of registers, so we seek a coloring using the fewest colors. The smallest number of colors sufficient to vertex-color a graph is its *chromatic number*.

Several special cases of interest arise in practice:

- *Can I color the graph using only two colors?* – An important special case is testing whether a graph is *bipartite*, meaning it can be colored using only two different colors. Bipartite graphs arise naturally in such applications as mapping workers to possible jobs. Fast, simple algorithms exist for problems

such as matching (see Section 15.6 (page 498)) when restricted to bipartite graphs.

Testing whether a graph is bipartite is easy. Color the first vertex blue, and then do a depth-first search of the graph. Whenever we discover a new, uncolored vertex, color it opposite of its parent, since the same color would cause a clash. The graph cannot be bipartite if we ever find an edge (x, y) where both x and y have been colored identically. Otherwise, the final coloring will be a 2-coloring, constructed in $O(n + m)$ time. An implementation of this algorithm is given in Section 5.7.2 (page 167).

- *Is the graph planar, or are all vertices of low degree?* – The famous four-color theorem states that every planar graph can be vertex colored using at most four distinct colors. Efficient algorithms for finding a four-coloring on planar graphs are known, although it is NP-complete to decide whether a given planar graph is three-colorable.

  There is a very simple algorithm to find a vertex coloring of a planar graph using at most six colors. In any planar graph, there exists a vertex of at most five degree. Delete this vertex and recursively color the graph. This vertex has at most five neighbors, which means that it can always be colored using one of the six colors that does not appear as a neighbor. This works because deleting a vertex from a planar graph leaves a planar graph, meaning that it must also have a low-degree vertex to delete. The same idea can be used to color any graph of maximum degree Δ using $\leq \Delta + 1$ colors in $O(n\Delta)$ time.

- *Is this an edge-coloring problem?* – Certain vertex coloring problems can be modeled as *edge coloring*, where we seek to color the edges of a graph G such that no two edges are colored the same if they have a vertex in common. The payoff is that there is an efficient algorithm that always returns a near-optimal edge coloring. Algorithms for edge coloring are the focus of Section 16.8 (page 548).

Computing the chromatic number of a graph is NP-complete, so if you need an exact solution you must resort to backtracking, which can be surprisingly effective in coloring certain random graphs. It remains hard to compute a good approximation to the optimal coloring, so expect no guarantees.

Incremental methods are the heuristic of choice for vertex coloring. As in the previously-mentioned algorithm for planar graphs, vertices are colored sequentially, with the colors chosen in response to colors already assigned in the vertex's neighborhood. These methods vary in how the next vertex is selected and how it is assigned a color. Experience suggests inserting the vertices in nonincreasing order of degree, since high-degree vertices have more color constraints and so are most likely to require an additional color if inserted late. Brèlaz's heuristic [Brè79] dynamically selected the uncolored vertex of highest *color degree* (i.e. , adjacent to the most different colors), and colors it with the lowest-numbered unused color.

Incremental methods can be further improved by using *color interchange*. Taking a properly colored graph and exchanging two of the colors (painting the red vertices blue and the blue vertices red) leaves a proper vertex coloring. Now suppose we take a properly colored graph and delete all but the red and blue vertices. We can repaint one or more of the resulting connected components, again leaving a proper coloring. After such a recoloring, some vertex v previously adjacent to both red and blue vertices might now be only adjacent to blue vertices, thus freeing v to be colored red.

Color interchange is a win in terms of producing better colorings, at a cost of increased time and implementation complexity. Implementations are described next. Simulated annealing algorithms that incorporate color interchange to move from state to state are likely to be even more effective.

Implementations: Graph coloring has been blessed with two useful Web resources. Culberson's graph coloring page, *http://web.cs.ualberta.ca/~joe/Coloring/*, provides an extensive bibliography and programs to generate and solve hard graph coloring instances. Trick's page, *http://mat.gsia.cmu.edu/COLOR/color.html*, provides a nice overview of graph coloring applications, an annotated bibliography, and a collection of over 70 graph-coloring instances arising in applications such as register allocation and printed circuit board testing. Both contain a C language implementation of the DSATUR coloring algorithm.

Programs for the closely related problems of finding cliques and vertex coloring graphs were sought for at the Second DIMACS Implementation Challenge [JT96], held in October 1993. Programs and data from the challenge are available by anonymous FTP from *dimacs.rutgers.edu*. Source codes are available under *pub/challenge/graph* and test data under *pub/djs*, including a simple "semi-exhaustive greedy" scheme used in the graph-coloring algorithm XRLF [JAMS91].

GraphCol (*http://code.google.com/p/graphcol/*) contains tabu search and simulated annealing heuristics for constructing colorings in C.

The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) contains an implementation of greedy incremental vertex coloring heuristics. GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for vertex coloring.

Pascal implementations of backtracking algorithms for vertex coloring and several heuristics, including largest-first and smallest-last incremental orderings and color interchange, appear in [SDK83]. See Section 19.1.10 (page 662).

Nijenhuis and Wilf [NW78] provide an efficient Fortran implementation of chromatic polynomials and vertex coloring by backtracking. See Section 19.1.10 (page 661).

Combinatorica [PS03] provides Mathematica implementations of bipartite graph testing, heuristic colorings, chromatic polynomials, and vertex coloring by backtracking. See Section 19.1.9 (page 661).

Notes: An old but excellent source on vertex coloring heuristics is Syslo, Deo, and Kowalik [SDK83], which includes experimental results. Classical heuristics for vertex coloring include [Brè79, MMI72, Tur88]; see [GH06, HDD03] for more recent results.

Wilf [Wil84] proved that backtracking to test whether a random graph has chromatic number k runs in *constant time*, dependent on k but independent of n. This is not as interesting as it sounds, because only a vanishingly small fraction of such graphs are indeed k-colorable. A number of provably efficient (but still exponential) algorithms for vertex coloring are known. See [Woe03] for a survey.

Paschos [Pas03] reviews what is known about provably good approximation algorithms for vertex coloring. On one hand, it is provably hard to approximate within a polynomial factor [BGS95]. On the other hand, there are heuristics that offer some nontrivial guarantees in terms of various parameters, such as Wigderson's [Wig83] factor of $n^{1-1/(\chi(G)-1)}$ approximation algorithm, where $\chi(G)$ is the chromatic number of G.

Brook's theorem states that the chromatic number $\chi(G) \leq \Delta(G) + 1$, where $\Delta(G)$ is the maximum degree of a vertex of G. Equality holds only for odd-length cycles (which have chromatic number 3) and complete graphs.

The most famous problem in the history of graph theory is the four-color problem, first posed in 1852 and finally settled in 1976 by Appel and Haken using a proof involving extensive computation. Any planar graph can be five-colored using a variation of the color interchange heuristic. Despite the four-color theorem, it is NP-complete to test whether a particular planar graph requires four colors or if three suffice. See [SK86] for an exposition on the history of the four-color problem and the proof. An efficient algorithm to four-color a graph is presented in [RSST96].

Related Problems: Independent set (see page 528), edge coloring (see page 548).