

Lecture 3

*Lecturer: Ronitt Rubinfeld**Scribe: Rinoa Oliver***Topics covered.**

1. Graph coloring and the $(\Delta + 1)$ -coloring problem
2. The greedy $(\Delta + 1)$ -coloring algorithm
3. A randomized $(\Delta + 1)$ -coloring algorithm that checks color classes instead of neighborhoods (statement and example; the analysis will be covered next lecture)

1 Graph Coloring

Throughout this lecture, $G = (V, E)$ is an undirected graph with $n = |V|$ vertices, $m = |E|$ edges, and maximum degree Δ . For a vertex v , we write $N(v)$ for the set of neighbors of v .

Definition 1 A proper k -coloring of G assigns a color $c_v \in \{1, \dots, k\}$ to each vertex $v \in V$ such that $c_u \neq c_v$ for every edge $(u, v) \in E$.

In general, deciding whether G has a proper k -coloring is NP-complete. However, there is an important special case that can be solved in linear time: $k = \Delta + 1$. We refer to the task of finding a proper $(\Delta + 1)$ -coloring of G as the $(\Delta + 1)$ -coloring problem.

1.1 The greedy algorithm

The following simple algorithm always finds a proper $(\Delta + 1)$ -coloring.

Algorithm 1 Greedy $(\Delta + 1)$ -coloring

For each $v \in V$ (in any order), assign v a color $c_v \in \{1, \dots, \Delta + 1\}$ that is different from c_u for every already-colored neighbor $u \in N(v)$.

Such a color always exists: v has at most Δ neighbors, so its neighbors use at most Δ of the $\Delta + 1$ available colors, and at least one color is left over for v .

To color v , the algorithm looks at the colors of its at most Δ neighbors, so the total running time is $O(n\Delta)$. (More precisely, vertex v requires $O(\deg(v) + 1)$ work, for a total of $O(n + m)$, which is linear in the size of the graph.)

Can we do better?

2 A Faster Algorithm for $(\Delta + 1)$ -Coloring

It turns out that we can.

Theorem 1 (Assadi, Chen, Khanna) *There is a randomized algorithm that finds a proper $(\Delta + 1)$ -coloring of any n -vertex graph in $\tilde{O}(n\sqrt{n})$ time.*

Here $\tilde{O}$ hides polylogarithmic factors. Note that when $\Delta = \Theta(n)$, the graph may have $\Theta(n^2)$ edges, so this running time is sublinear in the size of the input: the algorithm does not even look at most of the edges.

In this lecture we present a simplification of the algorithm of Assadi, Chen, and Khanna due to Assadi and Yazdanyar. It is slightly less efficient, losing some logarithmic factors.

When is greedy already enough? If $\Delta = \tilde{O}(\sqrt{n})$, then the greedy algorithm runs in time $O(n\Delta) = \tilde{O}(n\sqrt{n})$, which already achieves the bound in the theorem. So the question is what to do when $\Delta = \Omega(\sqrt{n})$.

A new idea. Instead of having each vertex v look at all of its neighbors, let v pick a random color $c \in \{1, \dots, \Delta + 1\}$ and go over all of the vertices currently colored c , checking whether any of them is a neighbor of v . If none of them is, then c is a valid color for v .

How many vertices do we have to check? As a thought experiment, suppose the vertices were colored uniformly at random (which they are not). Then we would expect each color to be used by about $n/(\Delta + 1)$ vertices, which is $O(n/\sqrt{n}) = O(\sqrt{n})$ since we are assuming $\Delta = \Omega(\sqrt{n})$.

The two approaches are complementary:

- The greedy algorithm checks, for each v , all of the neighbors of v to see which colors are free. It is fast when vertices have “few” neighbors.
- The new algorithm checks, for each v , all of the vertices with color c to see whether any of them is a neighbor of v . It is fast when there are “few” vertices of each color, or equivalently, when there are lots of colors.

2.1 The algorithm

Input representation. We assume that G is given as an adjacency matrix, so that a query of the form “is $(u, v) \in E$?” can be answered in $O(1)$ time. This is needed for Step 2b below.

The algorithm maintains a set $\mathcal{C}_c$ for each color c , with the invariant that $\mathcal{C}_c$ is exactly the set of vertices that have been colored c so far.

Algorithm 2 Randomized $(\Delta + 1)$ -coloring

1. Initialize $\mathcal{C}_1, \dots, \mathcal{C}_{\Delta+1} \leftarrow \emptyset$.
 2. Let $v_1, \dots, v_n$ be the vertices of G in a uniformly *random* order. For $i = 1, \dots, n$:
 - (a) Sample a color c uniformly at random from $\{1, \dots, \Delta + 1\}$.
 - (b) For every $u \in \mathcal{C}_c$, check whether u is a neighbor of v_i .
 - (c) If no vertex in $\mathcal{C}_c$ is a neighbor of v_i , color v_i with c , update $\mathcal{C}_c \leftarrow \mathcal{C}_c \cup \{v_i\}$, and move on to the next vertex. Otherwise, go back to Step 2a and try again.
-

Note that the algorithm uses randomness in two places: the order in which the vertices are processed, and the colors sampled in Step 2a. The random ordering is important, and its role will become clear in the analysis.

2.2 An example

Suppose the random ordering processes the vertices in the order $1, 2, \dots, 7$, and consider the step at which vertex 6 is colored, as shown in Figure 1. At this point the color classes are

$$\mathcal{C}_{\text{green}} = \{1\}, \quad \mathcal{C}_{\text{purple}} = \{2\}, \quad \mathcal{C}_{\text{blue}} = \{3\}, \quad \mathcal{C}_{\text{red}} = \{4, 5\},$$

and any other color classes are still empty.

Say Step 2a samples red. In Step 2b we check vertices 4 and 5, and since 5 is a neighbor of 6, red is not available and we go back to Step 2a. Now say we sample blue. The only blue vertex is 3, which is a neighbor of 6, so we sample again. Finally, say we sample green. The only green vertex is 1, which is not a neighbor of 6, so we color vertex 6 green and update $\mathcal{C}_{\text{green}} \leftarrow \{1, 6\}$.

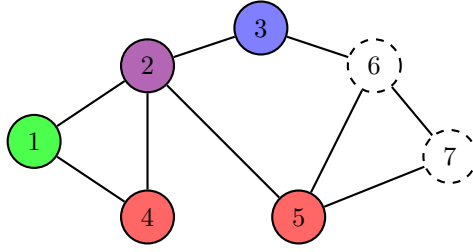


Figure 1: The state of Algorithm 2 just before vertex 6 is colored. Vertex 1 is green, vertex 2 is purple, vertex 3 is blue, and vertices 4 and 5 are red. Vertices 6 and 7 (dashed) have not been colored yet.

A note on the running time. A student asked the following question: since colors are sampled with replacement, what if we keep sampling a color c for which $\mathcal{C}_c$ contains a neighbor of the vertex we are trying to color? In the example above, we could in principle sample red over and over again. This is indeed possible, so there is no bound on the worst-case running time of the algorithm. However, such long runs are unlikely, and the quantity we will analyze is the *expected* running time.