Exponential time algorithms for graph coloring

Uriel Feige

Lecture notes, March 14, 2011

1 Introduction

Let $[n]$ denote the set $\{1, \ldots, k\}$. A k-labeling of vertices of a graph $G(V, E)$ is a function $V \longrightarrow [k]$. Given a k-labeling, an edge is monochromatic if both its endpoints are assigned the same label. A k-labeling is a k-coloring if no edge is monochromatic. The chromatic number of a graph is the minimum k for which a k-coloring exists. Observe that in a k-coloring, every color class is an independent set (no two vertices induce an edge of G), and hence a k-coloring is a partitioning of V into k independent sets, and also a covering V by k independent sets.

Deciding whether a graph has a k-coloring for $k \geq 3$ is NP-hard. Hence one would not expect to find a polynomial time algorithm for it. The trivial algorithm will try all possible k-labeling, and would take time k^n. The purpose of this write-up is to show that despite NP-hardness, there is plenty of room for designing algorithms that are substantially more efficient than the trivial algorithm.

Throughout we assume that all graphs are connected, since otherwise each connected component can be colored separately. In analyzing running times of algorithms, we shall omit factors that are polynomial in n and shall only be concerned with the exponential terms. For example, a running time of $n^3 2^n$ will be denoted as $O^*(2^n)$.

2 Algorithms for 2-coloring

2-coloring, or equivalently, determining whether a graph is bipartite, can be solved in polynomial time. A polynomial time algorithm for 2-coloring can assign an arbitrary color to an arbitrary vertex, and thereafter iteratively pick a yet uncolored vertex that has a colored neighbor and color it with the color not assigned to its neighbor. This algorithm must produce a 2-coloring if it exists, and must run into a monochromatic edge otherwise.

An optimization problem associated with 2-coloring is to find a 2-labeling with fewest monochromatic edges, or equivalently, with the largest number of properly colored edges. This last problem is known as *max-cut* and is NP-hard. We remark here that this is a prototypical example of what we shall later in the course call a *unique game*.

Take home message. 2-coloring is in P. Its optimization version max-cut is NP-hard.

3 Algorithms for 3-coloring

3.1 Spanning trees

Observe that a tree on n-vertices has exactly $3 \cdot 2^{n-1}$ different 3-colorings: color one vertex arbitrarily, and then iteratively every uncolored vertex with a colored neighbor has two possible colors.

Hence to 3-color a graph, pick an arbitrary spanning tree and check whether any of the 3-coloring of the spanning tree is a 3-coloring of the graph.

Running time $O^*(2^n)$.

3.2 Deterministic reduction to 2-coloring

Recall that 3-coloring is the same as partitioning V into three independent sets. Once one color class is given, the remaining graph needs to be 2-colored, which can be done in polynomial time. The smallest color class has size at most $n/3$. Finding it by exhaustive search involves checking $\sum_{i=1}^{n/3} \binom{n}{i}$ sets. As we ignore polynomial factors, this last term can be replaced by $\binom{n}{n/3} = \frac{n!}{(n/3)!(2n/3)!} \simeq 3^{n/3}(3/2)^{2n/3} = (27/4)^{n/3}$.

Running time $O^*\left((27/4)^{n/3}\right) \leq O^*(1.89^n)$.

Take home message. To design exponential time algorithms, one needs to know polynomial time algorithms.

3.3 Randomized reduction to 2-SAT

Associate with vertex v_i a Boolean variable x_i. Decide on a random interpretation as colors for the Boolean values of the variables. That is, for every v_i independently, pick a random color to correspond to $x_i = 1$, a different random color to correspond to $x_i = 0$, and discard the remaining color. Given this interpretation, set up a 2CNF formula where each edge (v_i, v_j) contributes either one or two clauses involving the respective x_i, x_j, and expressing that the values of its endpoints correspond to a legal coloring of the edge. There is a 3-coloring consistent with the random interpretation if and only if the corresponding 2CNF formula is satisfiable. As 2SAT can be solved in polynomial time (if you do not know how, please look this up), this gives a polynomial time algorithm for 3-coloring provided that the random interpretation did not discard the correct color of any vertex (correct with respect to some arbitrary 3-coloring). The probability of not discarding the correct color of a vertex is 2/3, and by independence, the probability of this happening for all vertices is $(2/3)^n$. Hence running the above algorithm multiple times, each time with fresh randomness, the expected number of tries needed in order to find a 3-coloring (if it exists) is at most $(3/2)^n$ (and perhaps smaller as the graph may have several 3-colorings).

An alternative view of the above algorithm is through the notion of *list coloring*. In a list coloring problem, every vertex is given a list of permissible colors, and one is required to obtain a legal coloring by selecting one color from each list. The randomized algorithm above takes a list coloring problem with lists of size 3 (in our case, initially all vertices have identical lists, but the algorithm applies more generally), discards at random one color from each list, and obtains a list coloring problem with lists of size 2. List coloring problems with lists of size 2 are solvable in polynomial time, by using a variation on the algorithm for 2-coloring (or by viewing them as 2SAT instances).

We note that the above algorithm is randomized. If stopped after any fixed number of iterations it might fail to find a 3-coloring even if one exists. This is one reason for preferring deterministic algorithms. There are various techniques for converting randomized algorithms into deterministic ones, and the term commonly used in this context is *derandomization*. However, this topic is beyond the scope of the current course, and we shall consider randomized algorithms to be essentially as "legitimate" as deterministic ones.

Expected running time $O(1.5^n)$.

Take home message. Randomized algorithms are useful.

3.4 Tighter reduction to 2-coloring

The homework assignment presents an approach that combines Section 3.2 with bounds on the number of *maximal* independent sets [5], and gives a 3-coloring algorithm with running time $O^*(3^{n/3}) \leq O^*(1.45^n)$.

3.5 The exponential time hypothesis ETH

For 3-coloring we have seen a sequence of algorithms which improves the base of the exponent for the running time from 3 (the trivial algorithm) to around 1.45 (the homework assignment). Even better bounds, below 1.33, are known [1]. It is a major open question whether for every $b > 1$, there is an algorithm for 3-coloring running in time $O^*(b^n)$. A question of a similar nature can be asked for other NP-hard questions, such as 3SAT. The *exponential time hypothesis* ETH [3] conjectures that for 3SAT, the answer is negative. If true, this would imply (by an appropriate reduction from 3SAT to 3-coloring) that the answer for 3-coloring is negative as well.

Take home message. For 3-coloring algorithms, determining the best running time (base of exponent) is open. Reductions among NP-hard problems and the ETH hypothesis tie between this question and similar questions for other NP-hard problems.

4 Algorithms for k-coloring

4.1 Dynamic programming

Dynamic programming is an algorithmic technique with many applications. Here is a sketch of how it can be used in the context of k-coloring.

Use exhaustive search to find and list all sets that are 1-colorable (independent sets). Now proceed in an inductive manner. Given all sets that are 1-colorable and all sets that are $(j-1)$-colorable, find all sets that are j-colorable. For each candidate set, try all ways of partitioning it in two and checking (using previously computed and stored steps) whether one part is 1-colorable and the other is $(j-1)$-colorable. Hence the total number of checks is $\sum_{S \subset V} 2^{|S|} = 3^n$. (A ternary vector can represent a check. The 0 entries are those vertices not in S. the 1 entries are those vertices in the independent set. The 2 entries are those in the $(j-1)$-colorable set.)

Running time $O^*(3^n)$. The space of this algorithm is also exponential, $O^*(2^n)$.

Remark. By considering only maximal independent sets (similar to Section 3.4), the dynamic programming based algorithm can be modified to have an improved running time of $O^*(2.45^n)$ (see [4]).

Take home message. Dynamic programming is a method that avoids redoing computations over and over again in exhaustive search. This leads to significant savings in running times, typically at the expense of requiring more space to store previously computed values.

4.2 Inclusion-exclusion

Rather than find a k-coloring, let us consider the more difficult task of counting how many different k-colorings there are. Given the number of k-coloring, we can check if this number is at least 1, and by this solve the decision problem. As with essentially all NP-complete problems, the search problem of k-coloring is reducible to the decision problem. For example, try to add edges to G and check if it remains k-colorable. When no more edges can be added, a simple greedy algorithm that assigns to each vertex the least color not yet assigned to any of its neighbors will k-color the graph.

For the counting version, it would be convenient to think of k-colorings as coverings of V by k independent sets, without requiring that the independent sets are disjoint. In fact, independent sets in the cover are allowed to be identical to each other. Moreover, the k independent sets are assumed to be ordered, and k-coverings that differ from each other only in the order of (nonidentical) independent sets are considered as different coverings. Still, under these conventions, the number of k-covering is positive if and only if the number of k-colorings is positive, and this suffices for using the counting algorithm as a subroutine that k-colors the graph.

Let $c_k(G)$ denote the number of coverings of V by k independent sets in G. The main idea to get a relatively efficient counting algorithm for $c_k(G)$ is to use the inclusion-exclusion formula. (A similar idea was used by Ryser to compute the *permanent*, which implies counting perfect matchings. See algorithms course from Spring 2010.) Given a set $X \subset V$, let $a(X)$ denote the number of independent sets in the subgraph induced on X.

Proposition 1 *With definitions as given above,*

$$c_k(G) = \sum_X (-1)^{n-|X|}(a(X))^k$$

where X ranges over nonempty subsets of V (including V).

Proof. Observe that $(a(X))^k$ counts the number of ordered k-tuples of independent sets in the subgraph induced on X.

To prove the equality, consider any ordered collection of (possibly overlapping) independent sets $I_1, \ldots, I_k$. Denote their union by U. If $U = V$ they should contribute 1 to the left hand side. Indeed, they contribute 1 to the right hand side (only the term $X = V$ counts them, and with coefficient $+1$). If $U \neq V$ then they should not contribute to the left hand side. Indeed, their contribution to the left hand side is 0, as there are as many even sized sets X as odd sets X sandwiched between Y and V (including Y and V). ■

The expression for $c_k(G)$ in the proposition involves the terms $a(X)$ for all $X \subset V$ (except for the empty set). We can compute all of them simultaneously in time $O^*(2^n)$ using dynamic programming. For the empty set $\emptyset$ we have $a(\emptyset) = 0$. Having computed and stored $a(X)$ for all X with $|X| \leq s$, we can compute $a(X)$ for a set X with $|X| = s + 1$ by the following inductive approach. Let v be an arbitrary vertex in X, and let $N_X(v)$ be the set of neighbors of v that are contained in X. Then $a(X) = 1 + a(X - v) + a(X - v - N_X(v))$. (The first term is the independent set containing only v, the second counts independent sets not containing v, and the third counts independent sets containing v and at least one more vertex.) Having, computed and stored all $a(X)$, using Proposition 1 one can compute $c_k(G)$ in time $O^*(2^n)$.

The above algorithm is taken from [2], which proves additional results using similar techniques.

Running time $O^*(2^n)$. The algorithm also uses space $O^*(2^n)$.

Take home message. Counting problems are at least as difficult as the corresponding decision problems. Nevertheless, sometimes the fastest known algorithms for a decision problem go through its counting version. Typically, in these cases the counting algorithms use algebraic expressions that involve cancelation effects.

4.3 Open questions

It is not known whether there is a k-coloring algorithm that runs in time $O^*(b^n)$ with $b < 2$. Likewise, it is not known whether there is an $O^*(2^n)$ algorithm for k-coloring that only uses polynomial space, though polynomial space algorithms that use somewhat more time (say, $O^*(2.3^n)$) are known.

References

[1] Richard Beigel, David Eppstein: 3-coloring in time $O(1.3289^n)$. J. Algorithms 54(2): 168–204 (2005).

[2] Andreas Bjorklund, Thore Husfeldt, Mikko Koivisto: Set Partitioning via Inclusion-Exclusion. SIAM J. Comput. 39(2): 546–563 (2009).

[3] Russell Impagliazzo, Ramamohan Paturi: On the Complexity of k-SAT. J. Comput. Syst. Sci. (JCSS) 62(2):367–375 (2001).

[4] Eugene L. Lawler: A Note on the Complexity of the Chromatic Number Problem. Inf. Process. Lett. (IPL) 5(3):66–67 (1976).

[5] J. W. Moon, L. Moser: On cliques in graphs. Israel Journal of Mathematics 3: 23-28 (1965).