CS 3510 — Algorithms, Spring 2022

Lecture 21: Graph Coloring

April 19, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

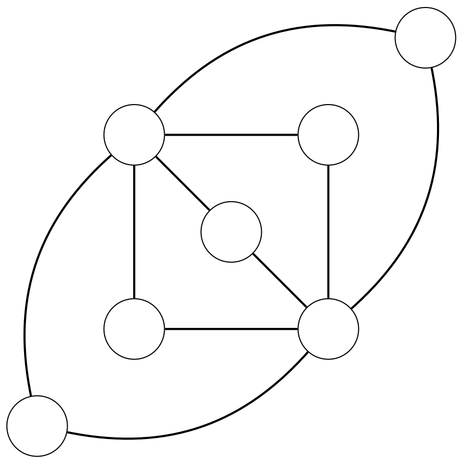
1 Quick Note on Knapsack

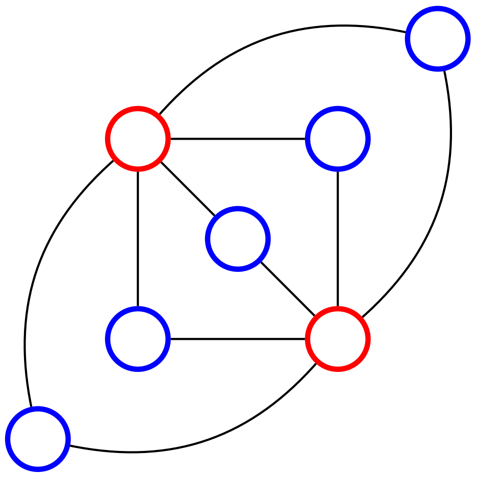
From the previous lecture, we may remember that we could reduce 3SAT to Subset-Sum to show that Subset-Sum was NP-Hard. Then, we showed Subset-Sum reduces to Knapsack. But, didn't we have a polynomial time algorithm for Knapsack? If so, this would imply $P = NP$!

However, that is not the case (no millionaires here!). The running time for Knapsack was $O(nB)$ but B is not just the input size, it's some weight. We can rewrite B as 10^k where k is the number of digits in B, and therefore the runtime of Knapsack would be $O(n \cdot 10^k)$ which is definitely not polynomial.

2 Graph Coloring

Given a graph G and a number of colors $\leq k$, can you assign a color to each vertex of the graph such that no adjacent vertices have the same color?





As an example, the graph above can be colorable with 2 colors (notice that no connected vertices have the same color on the right graph).

Theorem. *2-Coloring is in P.*

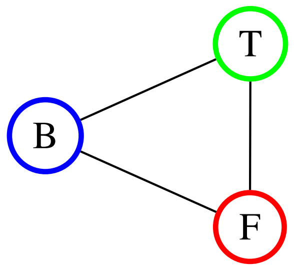
Proof. Pick some vertex to start with and color it some color, the next vertex that is connected to this must be of another color. Keep doing this and determine if it is possible. This is polynomial as there is only one combination of colors to try (since coloring one vertex determines the colors for the rest of the graph).

Theorem 2.1. *3-Coloring is in NP.*

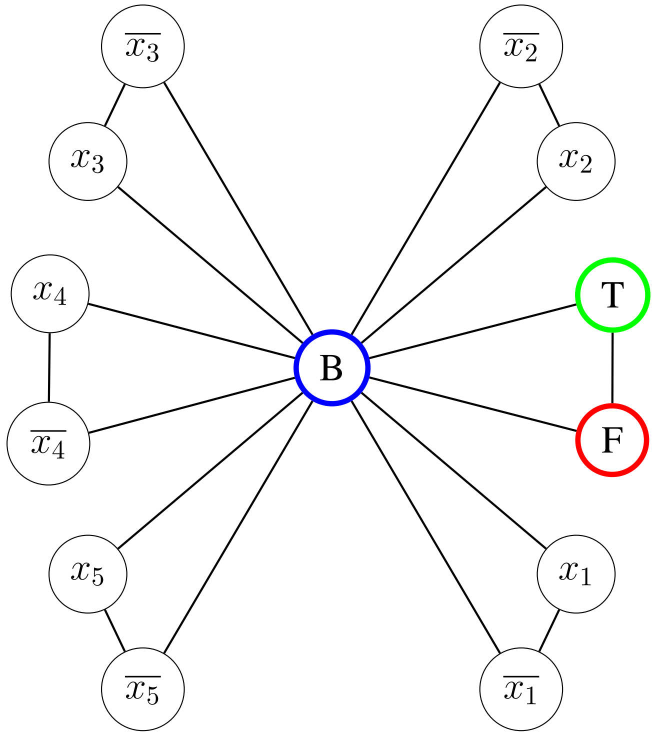
Proof. First, we can verify a candidate solution in polynomial time by simply traversing through the graph and ensuring that no two vertices that are connected to each other have the same color. Hence, 3-Coloring is in NP.

Next, we want to pick a known NP-Complete problem to 3-Coloring. We can choose the 3-SAT problem and construct a graph G such that if the input F to 3-SAT is satisfiable, then G has a 3-coloring (and vice a versa).

We want to construct a graph G, first we will become by constructing part of the graph that we can call the *flower* (no, this is not a technical term, it just looks like a flower). We begin this construction like so:

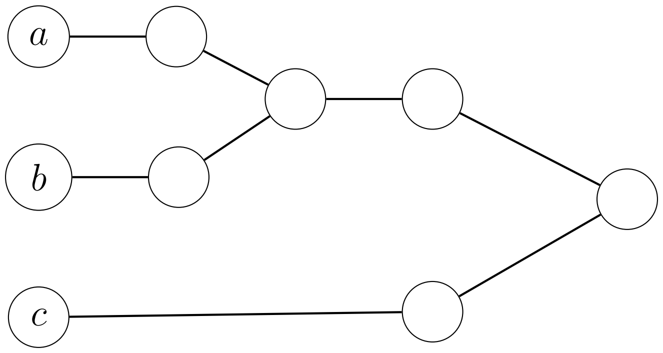


This is only part of the flower (one of the pedals). We start by having the center of the flower called B which is fixed to blue. Since all pedals will connect to this, the only colors for the pedals will be green or red (which corresponds with true or false). We can continue constructing the flower by having vertices for each literal.

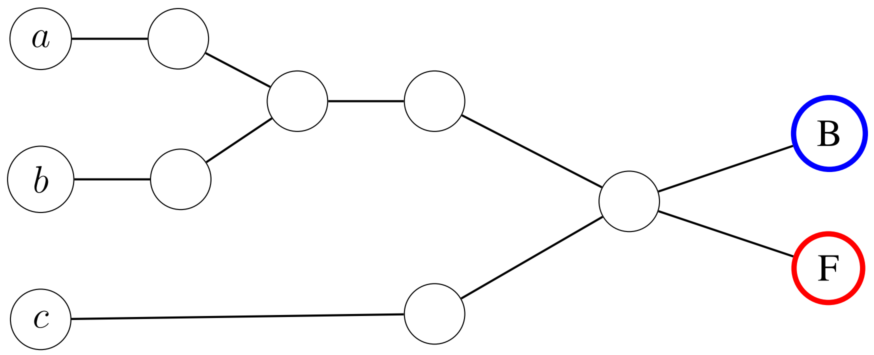


If we ran a 3-coloring algorithm on this, then some of these vertices will be colored red and some will be colored green (none will be colored blue, except the vertex labeled B). Now, we want to restrict the colorings based on the clauses, so we will define a series of gadgets that correspond with each clause.

The gadget that we use for each clause may remind you of a circuit with a bunch of OR gates. Let us construct a gadget for an example clause such as $(a \vee b \vee c)$.



The goal with this gadget is that if all of the literals a, b, c then then the final node at the end should be false as well. Further, if one of a, b, c is true, then it should be possible for the end node to be true. How do we force it to be true? We can simply attach the end to the original B and F vertices such that it must be T.



The other question is how do we connect literals together? Deceptively, the a, b, c vertices on the left of this gadget actually are not new vertices, but rather vertices from the original flower. You should convince yourself that no matter what permutation of values for a, b, or c if one is true, the right most vertex is true and otherwise it is false.

Hence, by creating a gadget for all of the clauses (connected to the original flower), it should only be 3-colorable if all these clauses are true (as otherwise you could not color them since they are connected to B and F vertices). Now, we must prove our reduction. We claim that F is satisfiable if and only if G is 3-colorable.

($\implies$) F is satisfiable implies that G is 3-Colorable. Since F is satisfiable, then we can assign any color to T, F, and B and then assign x_i, $\neg x_i$ according to the assignment of F and then all clause nodes get the T color. The only nodes left are the nodes within each gadget, but since each clause has at least one true variable implies that each gadget has at least one T-colored node on left (which can be colored consistently).

($\impliedby$) G is 3-Colorable implies that F is satisfiable. The proof for this claim is essentially the same as before, in the reverse direction.

3 Challenges

1. Show that 4-Coloring is NP-Complete.

   Partial Proof. Only a quick proof outline is given. First, we can check 4-Coloring is in NP by simply iterating over edges and ensuring that we are not using more than 4 colors and no two vertices have the same color.

   To show NP-Hardness, We can reduce 3-Coloring to 4-Coloring. For some input G to the 3-Coloring problem, add a dummy vertex and create an edge from this dummy vertex to every vertex in G. Since this is connected to every other vertex, it will force an additional color. If the original graph was 3-Colorable, this modified graph is 4-Colorable.

2. Show that Clique-and-IS is NP-Complete. The problem that should be true if an input graph G has a Clique and Independent Set of size k.

   Partial Proof. We can show this problem is in NP by modifying the verifiers for Clique and the verifier for IS.

   We can show this is NP-Hard by reducing Clique. The input to the Clique problem is G, g, we can add g additional vertices with no edges and run this through Clique-and-IS. If the original graph G had a clique, the modified graph must have a Clique and an IS (as we added g disconnected vertices).

3. Show that 2-IS is NP-Complete. The problem where we want to pick a set of k vertices such that there is no path of length 2 between vertices.

   Partial Proof. We can show problem is in NP by adapting the verifier for IS but taking into account the path of length 2 restriction.

   We can show this is NP-Hard by reducing IS. For some input G, k to the IS problem, we can split each edge into 2 edges with some dummy vertex. However, it is possible that these dummy vertices are included in the 2-IS. We prevent this by creating a clique of all the added edge vertices such that they could not be included in the 2-IS and therefore we will return an IS in the original graph.