Sources

Diestel's Graph theory: https://diestel-graph-theory.com
Erickson's Algorithms: https://jeffe.cs.illinois.edu/teaching/algorithms/

Big-O notation

We would like to describe the complexity of a given algorithm as a function that maps the size of the input to the time the algorithm takes to solve the problem. Usually we care about bounds, since instances of inputs of the same size may not always take exactly the same time. It makes sense to ignore constant factors, since these are affected by implementation of the algorithm as well as the specific machine it is executed on. It is more informative to focus on how the run time grows as the size of the input grows. For this we use big-O notation.

Definition 1. *Let $f, g : \mathbb{N} \to \mathbb{R}$. We say that $f(n) \in O(g(n))$ if there exists a constant $c \in \mathbb{R}$ such that $c(f(n)) < g(n)$ for all n.*

Definition 2. *Let $f, g : \mathbb{N} \to \mathbb{R}$. We say that $f(n) \in o(g(n))$ if for every $c \in \mathbb{R}$ there exists a value $N \in \mathbb{N}$ such that $c(f(n)) < g(n)$ for all $n \geq N$. This is equivalent to $f(n)/g(n) \to 0$ as $n \to \infty$.*

Definition 3. *Let $f, g : \mathbb{N} \to \mathbb{R}$. We say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$.*

Definition 4. *Let $f, g : \mathbb{N} \to \mathbb{R}$. We say that $f(n) \in \Omega(g(n))$ if $g(n) \in O(f(n))$, and $f(n) \in \omega(g(n))$ if $g(n) \in o(f(n))$.*

Note that, often, we use a slight abuse of notation by replacing $\in$ with $=$, and say, e.g., $f(n) = O(g(n))$.

Exercise 1. *Show that there is an algorithm to check whether a graph is bipartite that runs in $O(n^2)$ time, and show that no algorithm exists that has run time $o(n^2)$.*

Directed graphs

We do a few more examples of graph algorithms, this time on directed graphs. A *tournament* is a complete graph in which every edge has one of two directions. A *Hamiltonian path* in a graph is a path that spans all the vertices. We prove the following claim constructively:

Claim 5. *Every tournament has a (directed) Hamiltonian path.*

In the context of *digraphs* (directed graphs) we can strengthen the notion of connectedness. We say a digraph is *strongly connected* if for every ordered pair of vertices x, y there is a directed x, y-path. Erickson proposes the following exercise:

Exercise 2. *Describe an algorithm to determine, given an undirected graph G as input, whether it is possible to direct each edge of G so that the resulting directed graph is strongly connected.*

In order to answer this question. We need a bit more understanding of graph connectedness. We say that a graph G is k-connected (or, k-vertex-connected) if $G \sim K_n$ or if G is connected and remains connected if the deletion of any set of at most $k-1$ vertices leaves G connected. In other words, disconnecting G requires the removal of at least k vertices. Note that this generalizes the definition of connectedness that you are already familiar with, which is really 1-vertex-connectedness. We will start with investigating the structure of 2-connected graphs. Let H be a subgraph of a graph G. We call a path P in G an *H-path* if the endpoints of P lie in H, but all other vertices of P are outside of H. We will also call this an *ear* on H. Now, we let an *ear decomposition* be a series of subgraphs of G: $C = H_0, H_1, \dots, H_k = G$, such that C is a cycle in G, and each H_i is obtained from H_{i-1} by attaching an ear.

Proposition 6. *Every 2-connected graph has an ear decomposition.* D. 3.1.1

We can find a similar result for edge-connectedness. We say that a graph G is k-edge-connected if G is connected and remains connected if the deletion of any set of at most $k-1$ edges leaves G connected.
We call a *closed ear* on H a cycle that has exactly one vertex in common with H.

Proposition 7. *Every 2-edge-connected graph has an ear decomposition that uses open and/or closed ears.*

Strongly connected components of digraphs

In the previous week, we discussed how DFS (or BFS) can be used to find the connected components (maximal connected subgraphs of G) of an undirected graph. We can define an equivalence relation on the vertex set $V(G)$ of a graph G as $v \sim w$ if there exists a v,w-path in G. This equivalence relation then gives rise to a partition whose classes represent the connected components of G. A single DFS starting at v will find the entire set of vertices in its connected component, so we can repeat on undiscovered vertices to find all components.

Similary, we can define the *strongly connected components* (maximal strongly connected subgraphs of G) of a digraph. Now, we use the equivalence relation defined by $v \sim w$ if there exists both a $v \to w$-path and a $w \to v$-path in G. For yourself, verify that this is an equivalence relation and that it gives us the strongly connected components. Note that we can use a single directed DFS (or BFS) from a vertex v to find the set of vertices w such that a $v \to w$-path exists. However, we then still need to check for paths in the opposite direction.

Exercise 3. *Describe the complexity of finding the connected components in an undirected graph. Then design an algorithm to find the strongly connected components in a directed graph and describe its complexity.*

Satisfiability

As a warm-up before we get deeper into complexity classes, we look at the classic satisfiability problem SAT, as well as an example of reduction of one problem to another. A *boolean literal* is a variable x that takes values True or False. We write $\overline{x}$ for its negation. We write formulas by combining literals with the operators $\wedge$ (OR), which is called a *disjunction*, and $\wedge$ (AND), which is called a *conjunction*. So, $x \wedge y$ evaluates to True if either x or y (or both) are True. And, $x \wedge y$ evaluates to True if x and y are both True.

A *clause* is a disjunction of a set of literals or their negations. A formula in *conjunctive normal form* (CNF) is a conjunction of a set of clauses.

Observation 8. *Every boolean formula can be written in CNF.*

We define the problem of k-SAT as the problem of deciding whether a formula in CNF has a solution (an assignment of T/F values to its literals such that the formula evaluates to True), given that its clauses have length at most k.

Claim 9. *Every instance of the 2-*SAT *problem can be solved in $O(n)$ time. Here, n is the length of the formula.*

Proof. We reduce the problem 2-SAT to the problem of checking the strongly connected components in a digraph. Let Ψ be a formula in CNF, with each clause of length 2, with n clauses on literals $x_1, \ldots, x_t$. Construct the digraph G as follows. Let $V(G) = \{v_1, \overline{v_1}, \ldots, v_t, \overline{v_t},$ representing the literals and their negations. For each clause $(a \wedge b)$, add the directed edges $\overline{a} \to b$ and $\overline{b} \to a$ to G.

Exercise 4. *Complete the proof.*

□

Intro to complexity classes

The problems which we have seen so far all had fairly efficient algorithms. More precisely, they had algorithms with polynomial tun time: $O(n^c)$ where c is any constant. We call this class of decision problems P. This class is part of a larger class called NP (nondeterministically polynomial). These are the decision problems whose solutions (if the answer is "yes") can be verified in polynomial time. Problems that lie in NP, but not in P, are not known to be impossible to solve in polynomial time: rather we just don't have algorithms. The question of whether $P = NP$ is a Millenium Prize problem with a million dollar prize.
Within the class of NP problems, there is a set of "hardest" problems, in the sense that every other problem in NP can be reduced to them. Stephen Cook proved in 1971 that the SAT problem was NP-complete, and in 1972 when Richard Karp published a set of 21 NP-complete problems, which includes proper 3-coloring of graphs. (Recall that we have seen that proper 2-coloring is in P.)

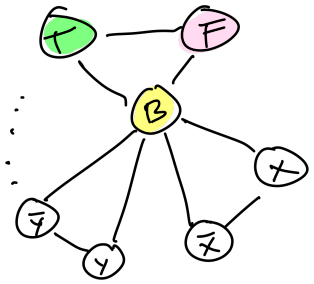
3-COL vs SAT

As an exercise, we will show reductions both ways, in order to show that these two problems are essentially equally difficult. First, we reduce graph 3-coloring to 3-SAT. Let G be an n-vertex graph. We will use $3n$ literals of the form v_i, for $v \in V(G)$ and $i \in \{1, 2, 3\}$, indicating that vertex v has color i. For each vertex v, we add a clause of the form $(v_1 \wedge v_2 \wedge v_3)$, to ensure that each vertex can be assigned at least one of the colors. For each edge vw and each color i, we add a clause of the form $(\overline{v_i} \wedge \overline{w_i})$, ensuring that no edge is monocolored. So, our final formula in CNF form is

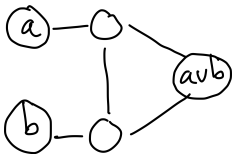
$$\left(\bigwedge_{v \in V(G)} (v_1 \wedge v_2 \wedge v_3) \right) \wedge \left(\bigwedge_{\substack{vw \in E(G), \\ i \in \{1,2,3\}}} (\overline{v_i} \wedge \overline{w_i}) \right),$$

and this formula is satisfiable if and only if G has a proper 3-coloring. Note that multiple v_is may be assigned to True for one vertex, but this just means that any of those colors is acceptable for v.

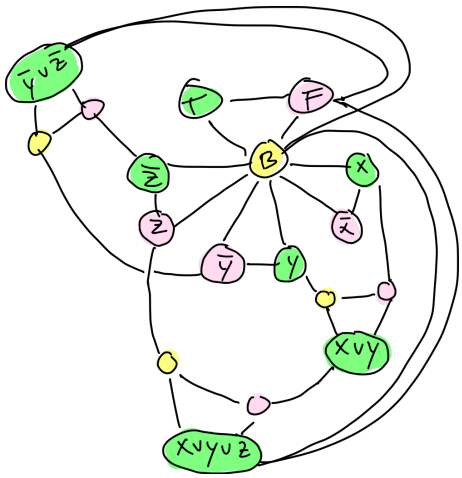
Now, we will reduce SAT to 3-COL. (I'm not entirely sure who to credit for this construction as I have seen it in various places). Given a formula in CNF, we start with the following construction which represents the literals and their negations, as well as a base triangle, which just assigns meanings to two of the three colors as True and False, and designates the third color as the base. In this graph, for each literal x we will either have x colored True and $\overline{x}$ colored False or vice versa.



Now, we need a gadget to capture an OR-gate. To capture $(a \wedge b)$, we need a vertex that is colored True if an only if a or b is colored True. This gate is drawn below. Note that the right-most vertex can only be True if at least one of the vertices a, b is True. Multiple of these gadgets can be combined to obtain gates for longer clauses.



In order to require that the final clause vertex for each clause to True, we add edges between each of them and the Base and False vertex in the original assignment triangle. Below is an example graph for the formula $(x \wedge y \wedge z) \wedge (\overline{y} \wedge \overline{z})$:



Exercise 5. *As in the example above, design a gadget for an AND-gate.*