CREATURE. *Hold on, you are confusing me.*
How do the graph shelves get their data, then?

COLLEAGUE. *You really ask the weirdest things. I guess*
They send some hunter-gatherers to catch
Or pick the graphs they find out in the wild.

CRE. *You make it sound like graphs exist, for real.*
But are they not defined by their observers?

COL. *Who are you? Not the Spanish Inquisition?*
All graphs have nodes and edges, that's what matters.
Sometimes they come with weights or attributes.
Semantics—God, who cares?—graphs are abstractions,
And abstract data is our working truth.

— Corinna Coupette, Jilles Vreeken, and Bastian Rieck,
"All the world's a (hyper)graph: A data drama" (2022)

1 Graph Modeling

Eventually this note will contain simpler examples of graph modeling and reductions, but i'm focusing on layering for now.

1.1 Graph Layering

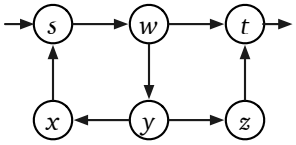
Graph layering is a technique for designing graph algorithms to find paths or walks *that satisfy certain constraints* in a given input graph G. Rather than modify an existing algorithm to directly address those additional constraints, a graph-layering algorithm builds a new graph H by carefully connecting several copies of G, and then invokes a textbook algorithm on H. The connections between the copies of G (informally called "layers") enforce the required constraints.

Recall that a ***walk*** in a directed graph G is a sequence of vertices $v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ connected by edges; $v_{i-1} \to v_i$ is an edge in G, for every index i. Similarly, a walk in an undirected graph is a sequence of vertices $v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ such that $v_{i-1} v_i$ is an edge in G, for every index i. A walk is a ***simple path*** if all vertices are distinct. Unfortunately, the word "path" often informally refers to either walks or simple paths, depending on context, so I will avoid it in these notes.

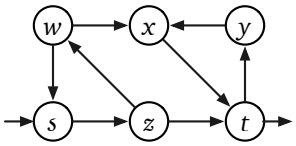
1.2 First Example: Length Divisible by 3

Problem 1. *Suppose we are given a directed graph G, and two vertices s and t. Describe an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is a multiple of* 3*.*

For example, given the following graph as input, your algorithm should return TRUE, because (for example) the walk $s \to w \to y \to x \to s \to w \to t$ has length 6. Notice that this walk visits vertices s and w twice and thus is not a simple path. The only simple paths from s to t in this graph are $s \to w \to t$ and $s \to w \to y \to z \to t$, neither of which have length divisible by 3.



On the other hand, given the following graph as input, your algorithm should return FALSE. There are infinitely many walks from s to t in the second graph, including two simple paths, but none of them have length divisible by 3. (This claim should not be obvious, but we'll prove it shortly.)



This smells like a reachability problem; we want to know if it's possible to get from one vertex to another in the graph G. However, because of the length constraint, it isn't the *standard* reachability problem. So we can't solve it by applying the standard reachability algorithm to G. Hmmm.

Before we start thinking about algorithms, let's first imagine wandering around the graph G, starting at vertex s, looking for a valid walk to t. What information do we need to maintain? We obviously have to keep track of where we are in the graph, but that isn't enough. Whenever we wander into t, we also need to know whether the number of edges we've traversed so far is a multiple of 3. So we need to keep track of how many edges we've traversed, but only modulo 3. When we start at s, we initialize our counter to 0, and every time we traverse an edge we update the counter from 0 to 1, or from 1 to 2, or from 2 to 0. If we reach t with a counter at 0, we've found a valid walk!

Thus, at all times, we need to maintain two pieces of information: A vertex v and a mod-3-counter counter i. We can formalize this intuition as follows. Given the original graph $G = (V, E)$ as input, we construct a new layered graph $G' = (V', E')$ as follows:

- $V' = \big\{(v, i) \;\big|\; v \in V \text{ and } i \in \{0, 1, 2\}\big\}$
- $E' = \big\{(u, i) \to (v, i + 1 \bmod 3) \;\big|\; u \to v \in E\big\}$

Less formally, G' contains three copies of every vertex and every edge of G; specifically, each copy of each edge $u \to v$ connects one copy of u to the *next* copy of v (modulo 3). Each possible value i of our mod-3-counter defines a "layer" of vertices (v, i).

There is a direct correspondence between walks in G and walks in G' that start at a vertex in "layer 0". Specifically, for any walk $s \to v_1 \to v_2 \to v_3 \to v_4 \to \cdots \to v_\ell$ in G, there is a corresponding walk

$$(v_0, 0) \to (v_1, 1) \to (v_2, 2) \to (v_3, 0) \to (v_4, 1) \to \cdots \to (v_\ell, \ell \bmod 3).$$

with the same length in G'. Crucially, the layer of each vertex in this walk is the length of the walk up to that vertex, modulo 3. Conversely, for any walk in G', we can recover the corresponding walk in G with the same length by ignoring the layer information. Corresponding walks in G and G' have equal lengths. It follows that G' has the following crucial structural property:

> **G contains a walk from s to t whose length is a multiple of 3**
> **if and only if**
> **G' contains a walk from $(s, 0)$ to $(t, 0)$.**

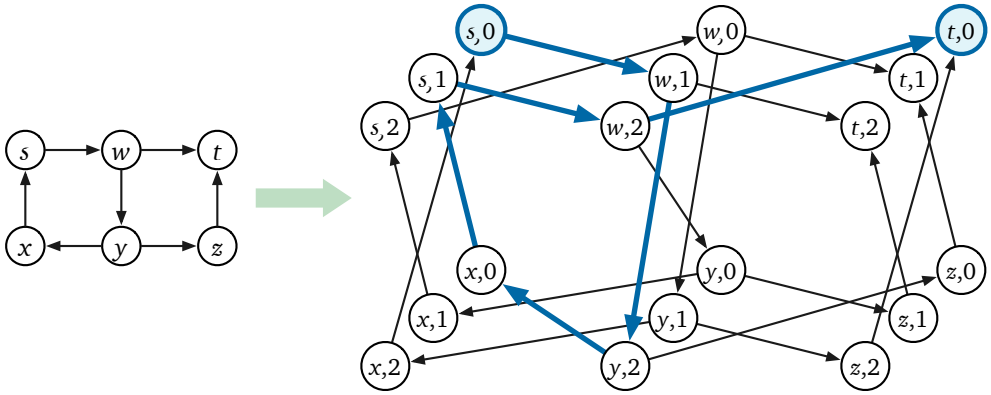
Intuitively, wandering around *G with a mod-3 counter* is the same as wandering around G' with nothing else. The structure of G' is keeping track of the counter for us!

Now let's start thinking about algorithms! Our construction reduces our constrained reachability problem in G to a standard reachability problem in G'.

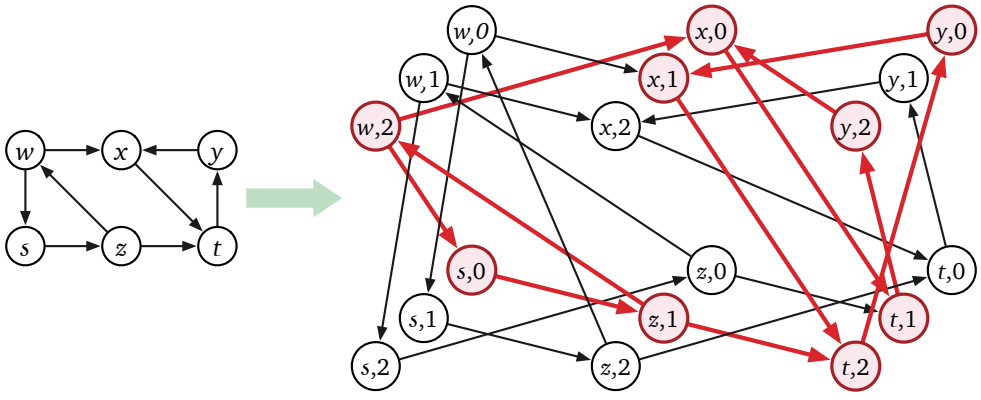
- Construct the layered graph G'.
- Perform a whatever-first search in G' starting at $(s, 0)$.
- If the whatever-first search reaches $(t, 0)$, return TRUE; otherwise, return FALSE.

The layered graph G' has $3V$ vertices and $3E$ edges, so we can construct it by brute force in $O(3V + 3E) = O(V + E)$ time. Running whatever-first search also takes $O(3V + 3E) = O(V + E)$ time. So the entire algorithm runs in $O(V + E)$ time.

Let's consider our earlier input examples. Our reduction transforms the first example graph G into the 18-node layered graph G' shown below. The bold blue path from $(s, 0)$ to $(t, 0)$ in G' corresponds to the walk $s \to w \to y \to x \to s \to w \to t$ in G, which has length 6.



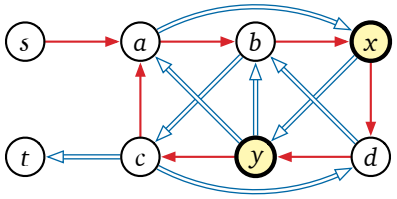
Our reduction transforms the second example graph G is transformed into the following 18-node layered graph G'. Here the bold red vertices and edges show the subgraph of G' that is reachable from $(s, 0)$. There is no walk in G' from $(s, 0)$ to $(t, 0)$, and therefore there is no walk in G from s to t whose length is a multiple of 3.



1.3 Second Example: The City and The City[1]

Problem 2. *Suppose you are given a directed graph G in which every edge is either red or blue, and some vertices are marked* special*, along with two vertices s and t. Describe an algorithm that either computes the length of the shortest walk in G from s to t that changes color only at special vertices, or correctly reports that no such walk exists.*

For example, suppose we are given the following input graph, where single arrows indicate red edges, double arrows indicate blue edges, and special vertices x and y are bold. Then our algorithm should return the integer 8, which is the length of the shortest valid walk $s{\to}a{\to}b{\to}\boldsymbol{x}{\Rightarrow}y{\Rightarrow}b{\Rightarrow}c{\Rightarrow}t$; this walk is valid because the edge color changes only at the special vertex x. The shorter walk $s{\to}a{\to}b{\Rightarrow}c{\Rightarrow}t$ is not valid, because the edge color changes at b even though b is not special.



This smells like a shortest-path problem in an unweighted graph, but the weird color rules means it's not a *standard* shortest-path problem, so we can't just run breadth-first search on G. Instead, we can use graph layering to enforce the color constraint.

Call a walk in G *valid* if the edge colors change only at special vertices. As before, let's begin by imagining actually wandering around G, trying to find a valid walk from s to t. If we enter a non-special vertex v through a red edge, we must leave v through a red edge; if we enter a non-special vertex through a blue edge, we must leave through a blue edge. Thus, in addition to our current location in G, we must remember the color of the edge we traversed to get here.[2]

This intuition suggests the following construction. Given the input graph $G = (V, E)$, we construct a new directed graph $G' = (V', E')$ as follows:

- $V' = V \times \{\mathsf{red}, \mathsf{blue}\}$
- E' is the union of four sets:
  - normal red edges: $\{(v, \mathsf{red}){\to}(w, \mathsf{red}) \mid v{\to}w \text{ is red}\}$
  - normal blue edges: $\{(v, \mathsf{blue}){\to}(w, \mathsf{blue}) \mid v{\to}w \text{ is blue}\}$
  - special blue edges: $\{(v, \mathsf{red}){\to}(w, \mathsf{blue}) \mid v \text{ is special and } v{\to}w \text{ is blue}\}$
  - special red edges: $\{(v, \mathsf{blue}){\to}(w, \mathsf{red}) \mid v \text{ is special and } v{\to}w \text{ is red}\}$

As in our previous example, there is a direct correspondence between *valid* walks in the original graph G and *arbitrary* walks in the layered graph G'. The structure of G' enforces the rules for us! Moreover, corresponding walks in G and G' have the same length. However, the correspondence between *endpoints* of these walks is not so direct; a valid walk in G from s to t

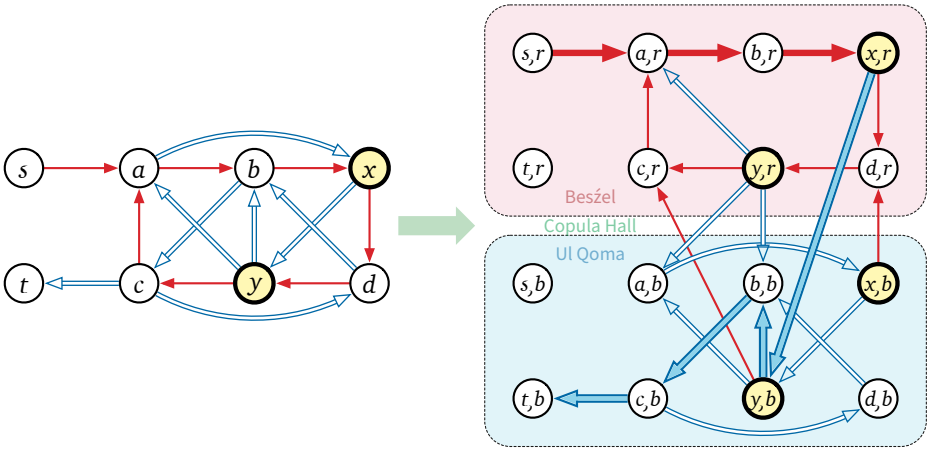
[1] If you've read the excellent China Miéville novel, you'll recognize this problem immediately. If you haven't, I won't spoil it!

[2] In principle, we could forget the incoming edge color when we're at a special vertex, but trying to optimize would only make our solution more complicated, so we won't bother.

can correspond to a walk in G' from either (s, red) or (s, blue) to either (t, red) or (t, blue). So our key structural property is slightly more complicated:

The shortest valid walk in G from s to t corresponds to the shortest walk in G' from $(s, *)$ to $(t, *)$.

For example, our earlier eight-node graph G is transformed in to the following 16-node layered graph G'. Red edges in G appear in the red layer of G'; blue edges in G appear in the blue layer of G', and additional copies of edges leaving special vertices of G cross between the layers of G'. The shortest valid walk in G from s to t corresponds to the shortest path in G' from (s, red) to (t, blue).



Now that we've nailed down our layered graph construction, we're finally ready to describe and analyze our algorithm.

- Construct the layered graph G'.
- Compute the shortest-path distances in G' from (s, red) to both (t, red) and (t, blue) using breadth-first search.
- Compute the shortest-path distances in G' from (s, blue) to both (t, red) and (t, blue) using breadth-first search.
- Return the minimum of those four distances.

The layered graph G' has $2V$ vertices and *at most* $2E$ edges, so we can construct it by brute force in $O(2V + 2E) = O(V + E)$ time. Each invocation of breadth-first search in G' also takes $O(2V + 2E) = O(V + E)$ time. So the entire algorithm runs in $O(V + E)$ time.

1.4 The General Pattern

The previous examples demonstrate a recommend a general strategy for reducing algorithmic problems to problems solved by standard textbook graph algorithms. The usual high-level goal is to construct a graph G such that walks in G represent valid sequences of "decisions" or "events" or "moves" or "transitions" in the given problem. For optimization problems, was also typically want the *length* of any walk in G to equal some value associated with the corresponding sequence of decisions in the stated problem.

- **What kind of graph problem are you aiming for?** Reachability? Shortest paths? Topological ordering? Maximum flow? This is mostly a smell text, but it does inform the graph construction, and it does suggest which standard algorithm(s) you will eventually need to apply.

- **What are the vertices of the graph? What does each vertex represent?** For most graph layering problems, each vertex of the *layered* graph represents one vertex of the *input* graph plus some additional problem-specific information.

  If you're familiar with finite-state machines, vertices correspond to *states* (and graph layering often resembles a *product construction*). If you're familiar with dynamic programming, vertices correspond to *subproblems*.

- **What are the edges of the graph? Are they directed or undirected?** For graph layering problems: Does the additional information restrict which edges you can traverse in the input graph? How does that information change when you traverse an edge? The layered graph may require directed edges even when the input graph is undirected (and vice versa).

  If you're familiar with finite-state machines, edges correspond to *transitions*. If you're familiar with dynamic programming, edges correspond to *dependencies* or *recursive calls*.

- Depending on the problem you are aiming for, some additional steps are important:

  - **What values, if any, are associated with the edges of your graph?** This question is crucial if you are trying to reduce to a shortest-path problem.
  - **Is your graph actually a dag?** This question is crucial if you are trying to reduce to a topological sorting problem. Answering "yes" requires a proof!

- **What is the precise relationship between your constructed graph and the stated problem?** This is by far the most important step in the design process. I strongly recommend actually writing out this relationship as a complete English sentence. This description plays the same role as the English function specification in a dynamic-programming algorithm.

- **Apply a standard graph algorithm to solve the problem.** For example, if this is a reachability problem, use whatever-first search. If you need to compute shortest paths in an unweighted graph, use breadth-first search. If you've constructed a directed acyclic graph, use topological sort. If this is a shortest-path problem and your graph has weighted edges, use Dijkstra or Bellman-Ford. Don't just *name* the algorithm; describe exactly how you would invoke it, and exactly how you would use its output.

- **What is the running time of your algorithm?** Don't forget to include the time to construct the graph. Remember to express the running time as a function of the original input parameters, not just in terms of the vertices and edges of your graph.

1.5 Common Mistakes

To close this note I want to describe several common errors that beginners make when designing graph algorithms. I'll describe these mistakes in the context of finding a *shortest path* meeting some additional problem-specific constraints, but beginners make variants of these mistakes for other types of graph-modeling and graph-layering problems.

- **Misreading the question.** Consider our first example problem: Design an algorithm to find the shortest walk from s to t whose length is a multiple of 3. Many beginners proposed the following incorrect algorithm: *First compute the shortest walk w from s to t. Then, if the length of w is a multiple of 3, return w.* Sometimes the proposed algorithm raise and exception or return an error code if the length of w is not a multiple of 3, but more often, the behavior in that case is left unspecified.

  These beginners are apparently interpreting the problem description as a *promise* that the shortest walk has length divisible by 3. But that reading is incorrect; the problem is making no such promise. Instead, the problem is asking us to find a walk whose length is a multiple of 5, and specifically, the shortest such walk.

  This confusion is especially common among non-native English speakers.[3] It is also one of the reasons why I try to include at least one example in every algorithm-design problem statement.

- **Misunderstanding standard algorithms.** Another common mistake is based on a misunderstanding of what problems standard algorithm solve. For example, breadth-first search finds *one* path of minimum length from the start vertex s to every other vertex in the graph. Nevertheless, beginners sometimes propose solutions with sentences like the following:

  - **If the shortest path doesn't have length divisible by 3, ***just*** run breadth-first search again.**
  - ****Just*** keep running breadth-first search until it finds a path whose length is a multiple of 3.**

  Breadth-first search is a deterministic algorithm; running it repeatedly yields the same shortest paths every time. And once any algorithm returns a result, it's done; there is nothing for it to "keep doing".

  This mistake is frequently accompanied by the word "just", which is *always* a red flag!

- **Considering all possible walks.** Anther common incorrect approach is to consider *all possible walks* from s to t, and return the shortest one that satisfies the necessary constraints. This approach reveals correct interpretation of the *problem*, but it does not lead to a correct and efficient *algorithm*.

  First, beginners who submit this solution rarely explain *how* to enumerate all walks from s to t. Some add a rider like "using breadth-first search" or "using depth-first search", but that's not what those algorithms do. A defining feature of breadth-, depth-, and other variants of breadth-first search is that they return ***exactly one*** walk from the start vertex s to every vertex reachable from s, and that walk is always a simple path. Whatever-first search systematically explores the *vertices* of its input graph, not the space of all possible *walks* (or even simple paths) in that graph.

[3] Arguably, this issue can be blamed on subtle rules of English grammar and punctuation. The directive "Find the shortest walk**,** whose length is a multiple of 5", with a comma after "walk", ***does*** promise that the shortest walk has length divisible by 5. The comma makes the clause "whose length is a multiple of 5" nonrestrictive, which means it can be deleted without changing the meaning of the noun ("shortest walk") that it modifies.

It is possible to enumerate all walks from s to t using a different algorithm called *iterative deepening search*, but this algorithm is both tricky to describe correctly and incredibly inefficient.[4] Using techniques well beyond the scope of this class, it is possible to enumerate (implicit representations of) walks from s to t *in order of increasing length*, stopping when we find a walk that satisfies our additional constraints, in $O(E + V + k \log k)$ time, where k is the number of walks considered.[5] However, even using this complex algorithm, the parameter k could be exponential in the size of the graph.

Second, unless the input graph is acyclic, there can be *infinitely* many walks between any two vertices, so enumerating them *all* would literally take forever. In principle, it is possible to enumerate walks from s to t in increasing order of length (using iterative deepening, for example), stopping when we find a walk that satisfies our additional conditions. However, we might need to consider an exponential number of walks before we find one that we like. Moreover, if the input graph doesn't contain a walk satisfying the extra conditions, algorithms that systematically examines all walks (including iterative deepening) will never terminate.

- **Modifying the algorithm instead of the input.** More generally, beginners are often tempted to modify a standard textbook algorithm to handle additional constraints, or even to develop their own algorithm from scratch, rather than modifying the input data so that the problem can be solved using a textbook algorithm as a black box. This approach isn't *necessarily* wrong, but it is more difficult to follow correctly. In particular, *correctly* modifying a textbook algorithm usually requires exactly the same logic as correctly reducing to a standard graph problem.

  Consider our first example problem: Find a shortest walk from s to t whose length is divisible by 3. There is a variant of breadth-first search that correctly solves this problem, but instead of maintaining a queue of vertices, it maintains a queue of (vertex, length mod 3) pairs, and it requires each vertex to store three different distances and three different predecessor pointers.

  A telling feature of most beginners' attempts to adapt whatever-first search to additional constrains is that the resulting algorithm can only return simple paths, even for problems (like our first example) where the required walk may not or cannot be simple.

- **Inappropriate dynamic programming.** Finally, a common knee-jerk response to any new optimization problem is to attempt to solve it by dynamic programming. This is especially tempting for problems where the input contains an array or grid, and in algorithms courses like mine that spend several weeks discussing dynamic programming before starting on graph algorithms. Again, this approach is not *necessarily* incorrect, but it must be considered with care. Beginners often try to design dynamic programming algorithms around "recurrences" whose dependency graphs contain cycles. Evaluating such a recurrence directly would lead to an infinite loop; equivalently, dependency cycles

[4] Iterative deepening, introduced by Richard Korf in 1985, is an algorithm for exploring implicitly defined graphs that are too large to fit into memory or even infinite. Variants of iterative deepening are standard tools in (classical) artificial intelligence; nevertheless, at least one canonical AI textbook describes the algorithm incorrectly. The worst-case running time of iterative deepening is exponential in the size of the graph.

[5] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing* 28(2):652–673, 1998.

rule out every iterative evaluation order. Fortunately, it is usually easy to recover from this stumble; the incorrect recurrence already defines a dependency graph!

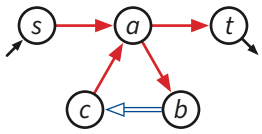
On the other hand, if a problem can be formulated as a recursive function with an acyclic dependency graph, that problem can usually be solved either using dynamic programming or using block-box algorithms for directed acyclic graphs. Under the hood, these are actually the same algorithms—Black-box algorithms for dags ***are*** dynamic programming!—but the two approaches exercise different types of intuition. Which of those two approaches is "better" depends almost entirely on context, convenience, and personal preference.

1.6 More Solved Problems

The next several pages contain more examples of graph-layering algorithms. For each problem, we are given a directed graph G, each of whose edges are either red or blue, and we are asked to find a shortest walk between two specified vertices that satisfies certain color-related constraints. Different problems have different constraints, and therefore require different layering constructions. As usual, none of the solutions described below are unique.

Problem 3. *Let G be a directed graph, each of whose edges is either red or blue, and let s and t be two vertices in G. Describe an algorithm to compute the shortest walk from s to t that traverses* at least three *blue edges.*

For example, given the following graph as input, your algorithm should return the walk $s{\to}a{\to}b{\Rightarrow}c{\to}a{\to}b{\Rightarrow}c{\to}a{\to}b{\Rightarrow}c{\to}a{\to}t$.



Solution: We construct a new graph $G' = (V', E')$ defined as follows:

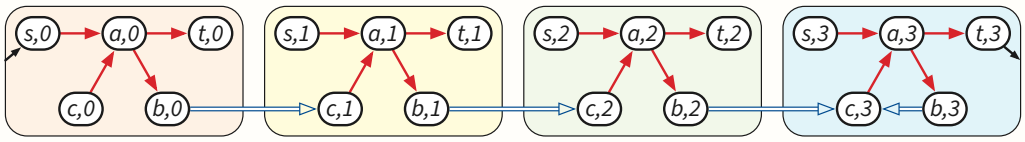
- $V' = V \times \{0,1,2,3\}$. Each vertex (v,i) with $i < 3$ denotes being located at v after traversing i blue edges. Each vertex $(v,3)$ denotes being located at v after traversing *at least* 3 blue edges.
- E' is the union of two sets of edges:

$$\{(u,i){\to}(v,i) \mid u{\to}v \in E \text{ and } (u{\to}v \text{ is red or } i = 3)\}$$
$$\{(u,i){\to}(v,i+1) \mid u{\to}v \in E \text{ and } u{\to}v \text{ is blue and } i \neq 3\}$$

We can construct G' by brute force in $O(V+E)$ time. Every walk in G' from $(s,0)$ to $(t,3)$ corresponds to a walk in G from s to t that traverses at least three blue edges, and vice versa. In particular, the *shortest* path in G' from $(s,0)$ to $(t,3)$ corresponds to the *shortest* walk in G from s to t that traverses at least three blue edges.

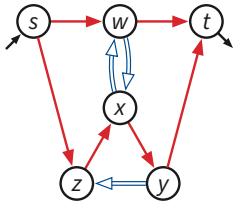
After constructing G', we compute the shortest path in G' from $(s,0)$ to $(t,3)$ in $O(V'+E') = O(V+E)$ time using breadth-first search. ■

For example, if G is the example graph shown above, the algorithm would construct the following layered graph G; the colored boxes show each individual layer.



Problem 4. *Let G be a directed graph, each of whose edges is either red or blue, and let s and t be two vertices in G. Describe an algorithm to compute the shortest walk from s to t whose edges alternate red and blue.*

For example, given the following graph as input, your algorithm should return the walk $s \to w \Rightarrow x \to y \Rightarrow z \to w \Rightarrow t$.



Solution: We construct a new graph $G' = (V', E')$ as follows:

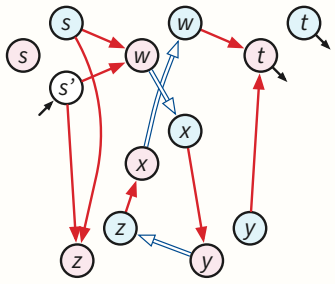
- $V' = V \times \{\text{red}, \text{blue}\} \cup \{s'\}$. Each vertex (v, c) represents being located at v immediately after traversing an edge with color c. The special start vertex $s' \in V'$ represents being located at the start vertex $s \in V$ before traversing any edges.
- E' is the union of four sets of edges:

$$\begin{aligned}
\big\{(u, \text{red}) \to (v, \text{blue}) \;\big|\; u \to v \in E \text{ and } u \to v \text{ is blue}\big\} \\
\big\{(u, \text{blue}) \to (v, \text{red}) \;\big|\; u \to v \in E \text{ and } u \to v \text{ is red}\big\} \\
\big\{s' \to (v, \text{blue}) \;\big|\; s \to v \in E \text{ and } s \to v \text{ is blue}\big\} \\
\big\{s' \to (v, \text{red}) \;\big|\; s \to v \in E \text{ and } s \to v \text{ is red}\big\}
\end{aligned}$$

We can construct G' by brute force in $O(V + E)$ time. Every walk in G' from s' to (t, red) corresponds to an alternating-color walk in G from s to t whose last edge is red, and vice versa; similarly, every walk in G' from s' to (t, blue) corresponds to an alternating-color walk in G from s to t whose last edge is blue red, and vice versa. In particular, the *shortest* path in G' from s' to (t, red) or (t, blue) corresponds to the *shortest* alternating-color walk in G from s to t.

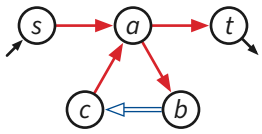
After constructing G', we compute the shortest paths in G' from s' to (t, red) and (t, blue) in $O(V' + E') = O(V + E)$ time using breadth-first search, and then return the shorter of those two paths. ■

For example, if G is the example graph shown above, the algorithm would construct the following layered graph G:



Problem 5. *Let G be a directed graph, each of whose edges is either red or blue, and let s and t be two vertices in G. Describe an algorithm to compute the shortest walk from s to t that contains* at least one edge of each color.

For example, given the following graph as input, your algorithm should return the walk $s \to a \to b \Rightarrow c \to a \to t$.



Solution: We construct a new graph $G' = (V', E')$ defined as follows:

- $V' = V \times \{\text{TRUE}, \text{FALSE}\} \times \{\text{TRUE}, \text{FALSE}\}$. Each node (v, b, r) represents reaching v through a walk that includes at least one red edge if and only if $r = \text{TRUE}$ and includes at least one blue edge if and only if $b = \text{TRUE}$.
- E' is the union of the following sets of edges:

$$\{(u, r, b) \to (v, b, \text{TRUE}) \mid u \to v \text{ is red}\}$$
$$\{(u, r, b) \to (v, \text{TRUE}, r) \mid u \to v \text{ is blue}\}$$

We can construct G' by brute force in $O(V + E)$ time.

Every walk in G' from $(s, \text{FALSE}, \text{FALSE})$ to $(t, \text{TRUE}, \text{TRUE})$ corresponds to a walk in G from s to t that traverses at least one red edge and at least one blue edge. In particular, the *shortest path* in G' from $(s, \text{FALSE}, \text{FALSE})$ to $(t, \text{TRUE}, \text{TRUE})$ corresponds to the *shortest* walk in G from s to t that traverses both colors. We can compute this shortest path in $O(V' + E') = O(V + E)$ time using breadth-first search. ■

Here are two different drawings of the layered graph G' constructed from the example graph G shown above. The first drawing keeps all nodes with the same red and blue bits together; the second keeps all copies of each vertex of G together.