DS 320 Algorithms for Data Science **Lecture #5**
Spring 2025 Prof. Kira Goldner and Jeffrey Considine

Application: Topological Sort

Definition 1. A *topological ordering* on the vertices is a total ordering assigning them numbers $1, \ldots, n$ such that only edges $(i, j) \in E$ where $i < j$ in the ordering.

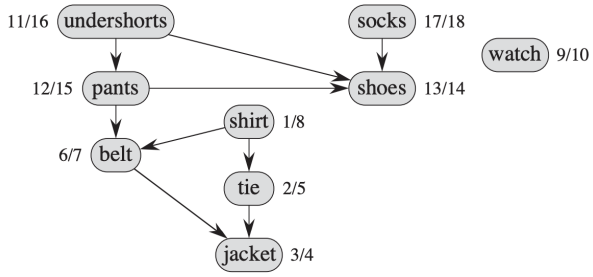

Figure 1: Top sort example graph from CLRS.

Theorem 1. *G has a topological order $\iff$ G is a DAG.*

Proof. ($\Rightarrow$) Topological ordering means only edges $(i, j) \in E$ where $i < j$. Consider the smallest i in the cycle. There exists an edge (j, i) in the cycle for $j > i$. Contradiction.

($\Leftarrow$) If there's a DAG, this implies that there exists a node with no incoming edges. Otherwise, one could backtrack, and after n steps, would find a cycle. □

Topological Sort Algorithm:

- Naive algorithm: Recursively remove a node with no incoming edges. $T(n) = O(n^2)$.
- Or, run DFS to assign postorder times, and then sort the DFS forest by decreasing postorder. $T(n) = O(n + m)$. (We can avoid the sorting runtime using a priority heap data structure.)

Theorem 2. *If the tasks are scheduled by decreasing postorder number, then all precedence constraints are satisfied.*

Proof. If G is acyclic then the DFS tree of G produces no back edges by Claim **??**. Therefore by Claim **??**, $(u, v) \in G$ implies $postorder(u) > postorder(v)$. So, if we process the tasks in decreasing order by postorder number, when task v is processed, all tasks with precedence constraints into v (and therefore higher postorder numbers) must already have been processed. □

Algorithm 1 BFS(G, s)

Input: Graph $G = (V, E)$ and starting vertex s.
initialize: (1) array *dist* of length n, (2) queue q, (3) linked list L of sets, (4) tree $T = (\{s\}, \emptyset)$
dist$[s] = 0$
$L[0] = \{s\}$
enqueue s to q
mark s as discovered and all other v as undiscovered
while size$(q) > 0$ **do**
  $v =$ dequeue(q)
  for $(v, w) \in E$ **do**
    if w is undiscovered **then**
      enqueue w in q
      mark w as discovered
      dist$(w) =$ dist$(v) + 1$
      add w to $L[dist(w)]$
      add (v, w) to T
    end if
  end for
end while
return T, L

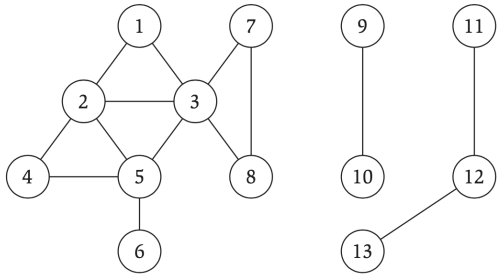

Figure 2: Example graph G. From Kleinberg Tardos.

Breadth-First Search

What happens when we run BFS(G, 1) where G is the graph below?
What is BFS doing? BFS labels each vertex with the distance from s, or the number of edges in the shortest path from s to the vertex. (**Exercise:** Prove this!)

Runtime: $O(|E|)$ (assuming that $|E| \geq |V|$). The reason is that BFS visits each edge exactly once, and does a constant amount of work per edge.

Claim 1. Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.

Proof. Suppose by way of contradiction that i and j differed by more than 1; in particular, suppose $i < j - 1$. Now consider the point in the BFS algorithm when the edges incident to x were being

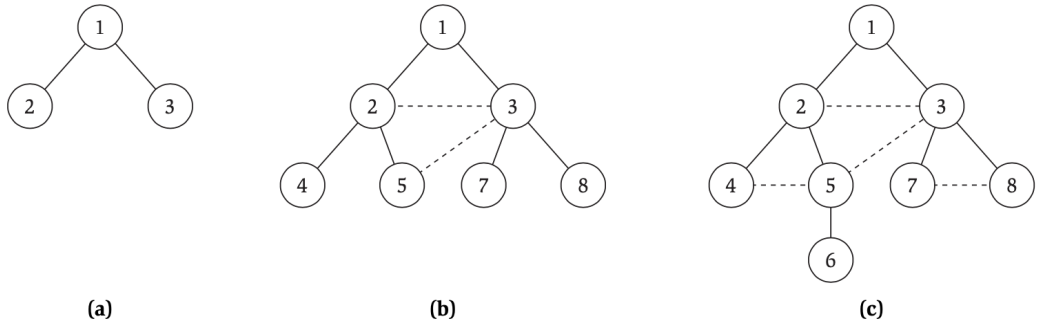

Figure 3: The BFS tree is shown in solid edges as constructed in stages (layers) by the above algorithm. The dashed edges are the edges that do not belong to the BFS tree but do belong to G. From Kleinberg Tardos.

examined. Since x belongs to layer L_i, the only nodes discovered from x belong to layers $L_i + 1$ and earlier; hence, if y is a neighbor of x, then it should have been discovered by this point at the latest and hence should belong to layer $L_i + 1$ or earlier. □

Definition 2. We say a graph $G = (V, E)$ is *bipartite* if the vertices can be partitioned into disjoint sets $V = A \sqcup B$ such that for every edge in E, one endpoint is in A and the other endpoint is in B. That is, there are no edges within A or within B.

We'll use another idea from graph theory: vertex coloring. We try to color the vertices with the *smallest* number of colors possible, but a vertex cannot be colored the same color as any of its neighbors. Then a bipartite graph is a graph that can be colored with two colors—all vertices in A can be colored red and all vertices in B can be colored blue.

First convince yourself of the following:

Claim 2. If a graph is bipartite, it cannot contain an odd cycle.

Then, use BFS to come up with a graph-coloring algorithm to determine whether the graph is bipartite.

Proof. Perform BFS, coloring s red, all vertices of distance 1 blue, all with distance 2 red, and so on, coloring odd-distanced vertices blue and even-distanced vertices red.

Check if any edge has both ends receive the same color, in which case return that it's not bipartite. Otherwise, return that it is. □

We now prove correctness of this algorithm, and start the proof as follows. Let G be a connected graph, and let $L_1, L_2, \ldots$ be the *layers*, or sets of vertices of distance 1, 2, and so on, produced by BFS starting at node s. Then exactly one of the following two things must hold.

a. Suppose G has no edges with both endpoints in the same layer. Argue that G must be a bipartite graph.

Hint: It may be useful to first prove claim below, and then add the idea of vertex coloring.

Proof. Claim 1 says that every edge of G joins nodes either in the same layer or in adjacent layers—this case then implies that all edges join two nodes in adjacent layers. Then each end of this (and every) edge will have different colors, hence G is bipartite. □

b. Suppose G has an edge with both endpoints in the same layer. Argue that in this case, G must contain an odd-length cycle and therefore not be bipartite.

Hint: It may again be useful to consider the BFS tree produced by the algorithm.

Proof. Suppose the edge joining vertices of the same layer is the edge $e = (x, y)$, with $x, y \in L_j$. Recall that L_0 ("layer 0") is the set consisting of just s. Now consider the BFS tree T produced by our algorithm, and let z be the node whose layer number is as large as possible, subject to the condition that z is an ancestor of both x and y in T; for obvious reasons, we can call z the lowest common ancestor of x and y. Suppose $z \in L_i$, where $i < j$.

We consider the cycle C defined by following the z-x path in T, then the edge e, and then the y-z path in T. The length of this cycle is $(j - i) + 1 + (j - i)$, adding the length of its three parts separately; this is equal to $2(j - i) + 1$, which is an odd number. □