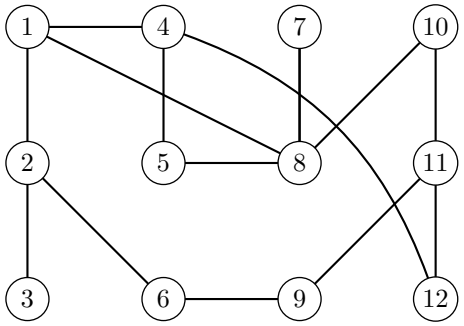
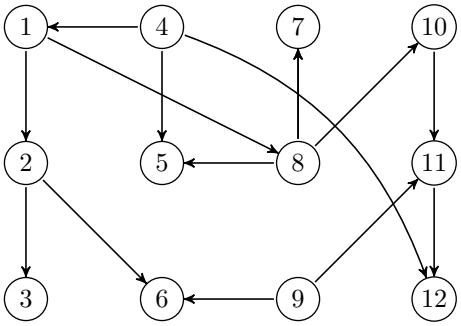
ASSIGNMENT 2: Solutions CMSC 451 (Spring 2016)

1. *BFS and DFS on undirected and directed graphs.*

   Consider the following two graphs:



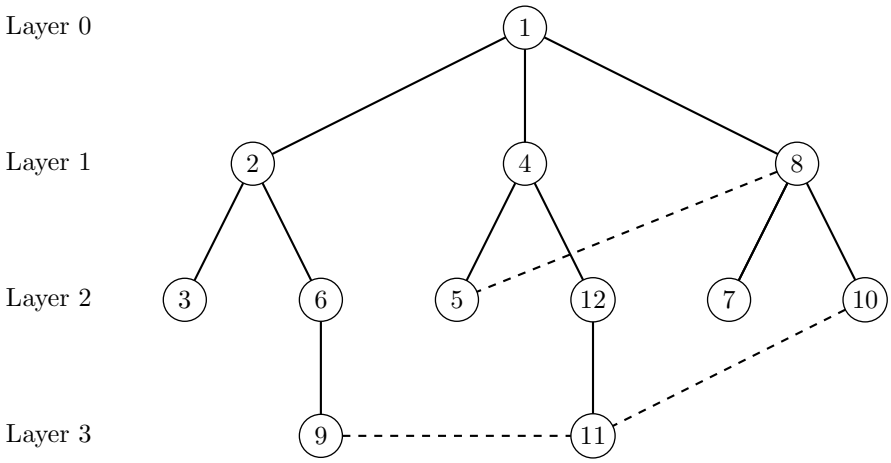
G_1



G_2

   (a) Run the breadth-first search algorithm on the undirected graph G_1, starting from vertex 1, and show its final output. When you have to choose which vertex to process next (and that choice is not otherwise specified by the BFS algorithm), use the one with the smallest label. Draw the tree edges with solid lines and the non-tree edges with dashed lines. Indicate the layers of the BFS tree.

   Solution:

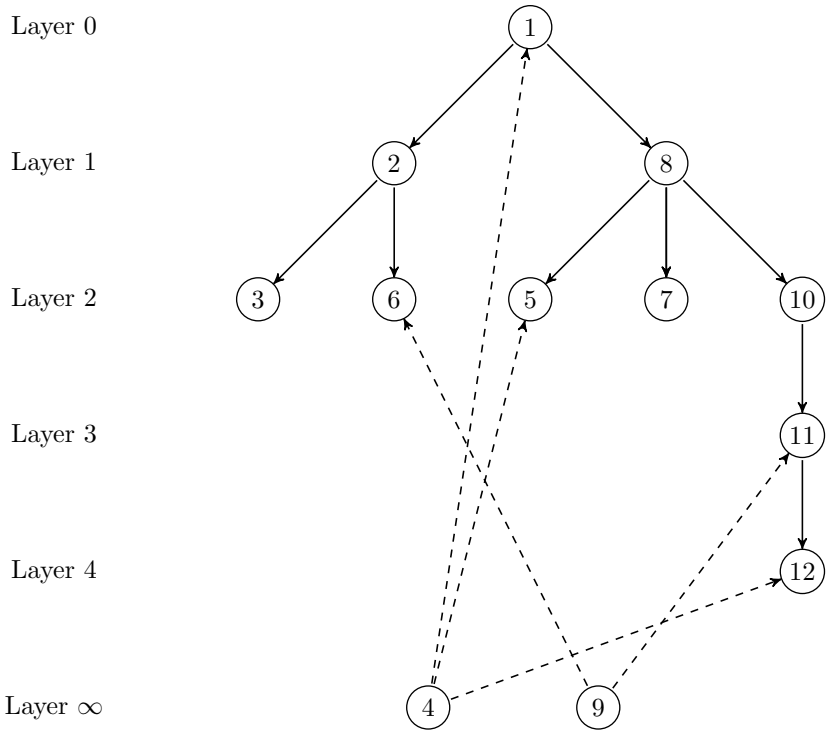


   (b) Is G_1 bipartite? Justify your answer with reference to the BFS tree.

   Solution: No. There is an edge $(9, 11)$ between two nodes in the same level of the BFS tree. As a result, the graph G_1 contains an odd-length cycle (9,6,2,1,4,12,11,9).

   (c) Repeat part (a) on the directed graph G_2.

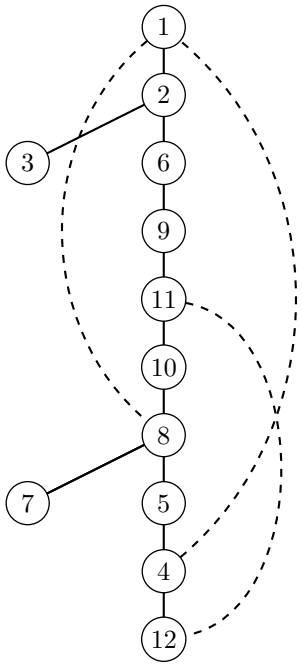
   Solution:



(Here we have also included the disconnected vertices in a "Layer ∞", but it would be acceptable to omit these.)

(d) Run the depth-first search algorithm on the undirected graph G_1, starting from vertex 1, and show its final output. When you have to choose which vertex to process next (and that choice is not otherwise specified by the DFS algorithm), use the one with the smallest label.

Solution:



(e) Repeat part (d) on the directed graph G_2.

Solution: The output turns out to be the same as in part (c).

(f) Is G_2 strongly connected? Justify your answer with reference to a search tree (or trees).

Solution: No. The nodes 4 and 9 are not reachable starting from node 1. This can be seen in both the BFS and DFS traversals starting at node 1.

2. *When are BFS and DFS the same?*

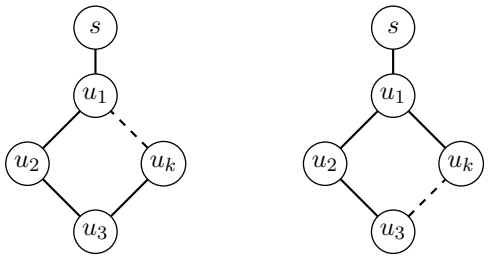
Characterize the set of undirected, connected graphs G with the following property: there exists a vertex s of G such that some BFS tree rooted at s is identical to some DFS tree rooted at s. Prove that your characterization is correct.

Solution: G is a tree.

$\Rightarrow$ Suppose G is a tree. Since BFS and DFS both output a spanning tree of G and the only spanning tree of a tree is itself, we have that BFS = DFS.

$\Leftarrow$ Suppose that for a connected graph G, the BFS starting at vertex s is identical to the DFS starting at s. Let us assume for the sake of contradiction that G is not a tree. Let C be a cycle in the graph G. Without loss of generality, let the cycle be $C = (u_1, u_2, \ldots, u_k, u_1)$ where u_1 is the first vertex explored by the DFS starting at s. By the definition of DFS, the algorithm will traverse the path $(u_1, u_2, \ldots, u_k)$ and then encounter $\{u_k, u_1\}$ as a non-tree edge.

On the other hand, consider the BFS tree rooted at s. Once the vertex u_1 is discovered by the BFS, both its incident edges $\{u_1, u_2\}$ and $\{u_1, u_k\}$ will be added to the BFS tree. But this is a contradiction since the trees must have been identical. Hence, G must be a tree.



3. *DAG algorithms.*

(a) Design an algorithm that, given a directed graph as input, determines whether it is a directed acyclic graph (DAG).

Solution: A directed graph G is a DAG if and only if it has a topological ordering. One solution to determine whether a given directed graph G is acyclic is to try to obtain a topological ordering—if we succeed, then G is a DAG and if we fail, then it is not.

Algorithm:

While there is a vertex v with no incoming edge
    Delete v (and all associated edges) from G
Endwhile
If G is empty then
    Return true (G is a DAG)
Else
    Return false (G is not a DAG)
Endif

Correctness: As shown in Kleinberg-Tardos (3.19), in every DAG G, there must be a vertex v with no incoming edges. Since the graph obtained by deleting a vertex from an acyclic graph is itself acyclic, we are guaranteed that if G is acyclic, then we must always be able to find a vertex with no incoming edges until the graph is empty. Conversely, if the graph is not acyclic then it cannot admit a topological ordering. Hence, we must have an iteration when the graph is not empty and yet there is no vertex with no incoming edges.

Running Time: A naïve implementation of the above algorithm takes $O(n^2)$ time: it takes $O(n)$ time to identify a vertex with no incoming edges and $O(n)$ such iterations. However, we can obtain an improved running time of $O(m+n)$ by maintaining appropriate state variables (as discussed in the implementation of topological sorting presented in class). Suppose we declare a node to be "active" if it has not yet been deleted. Let $\text{in}(w)$ denote the number of incoming edges that node w has from active nodes and let S denote the set of active nodes with no incoming edges from active nodes. We can initialize both $\text{in}(w)$ and S at the start of the algorithm in $O(n)$ time. In each iteration of the algorithm, we pick v to be an arbitrary node from S (if S is non-empty, otherwise return false). After deleting v, we update $\text{in}(w)$ for each out neighbor w of v and add any neighbors to S if they satisfy $\text{in}(w) = 0$. Thus in each iteration, we perform $O(\text{outdegree}(v))$ operations, leading to $O(m)$ work over all the iterations. Thus we have a total time complexity of $O(m+n)$.

(b) If we view a DAG as representing precedence constraints on a set of operations at the vertices (i.e., an operation cannot be performed until the operations at the heads of all incoming edges have been performed), the length of a longest path in the DAG represents the amount of time required to perform all the operations, assuming operations can be performed in parallel (subject to the precedence constraints). Design an algorithm that, given a DAG as input, determines the length of a longest path. (Hint: one possible approach is based on depth-first search; another is based on topological sorting. Any correct algorithm is acceptable.)

Solution: Assume that we have obtained a topological ordering of the precedence graph G. This can be achieved in time $O(m+n)$. Assume without loss of generality that the vertices of G are named $v_1, v_2, \ldots v_n$ where the subscript denotes the position of the vertex in the topological ordering. Given such a topological ordering, we can now compute the length of the longest path in G as follows.

Let $d[i]$ denote the length of the longest path that ends at vertex v_i. The following recurrence relation expresses $d[i]$ for every i:

$$d[i] = \begin{cases} 0 & \text{if } v_i \text{ has no incoming edges} \\ \max\limits_{\substack{j<i \\ (v_j,v_i)\in E}} d[j]+1 & \text{otherwise.} \end{cases} \quad (1)$$

We can compute $d[i]$ for every i in time $O(m)$ since at each step of the recurrence, we only need to perform one lookup for each edge. Once we have computed $d[i]$ for all i, the final answer is simply $\max_i d[i]$, which can be computed in time $O(n)$.

(c) Again viewing a DAG as encoding precedence constraints on operations that may be performed in parallel, the *earliest start time* of a vertex is the earliest time at which that operation can be performed subject to the precedence constraints (where time starts at 0 and each step of the execution takes unit time). Design an algorithm that, given a DAG as input, computes the earliest start time of every vertex.

Solution: For any vertex v_i, the *earliest start time* of v_i is equal to the length of the longest path that ends at vertex v_i. This is because, as each operation takes unit time and time starts at 0, a path of length l indicates that l operations need to be performed before v_i. Hence, the $d[i]$ values computed in the solution to part (b) above are exactly the earliest start times of each vertex v_i.

In each part, prove the correctness of your algorithm and analyze its running time. For full credit, all your algorithms should run in time $O(n + m)$, where the input graph has n vertices and m edges.

4. *Route planning with variable travel times.*

Suppose you would like to navigate a system of roads, represented by a directed graph $G = (V, E)$. Because of variable traffic and weather conditions, the time required to travel along an edge varies with time. Fortunately, you are given access to an oracle that can reliably predict how long it will take to travel along an edge at any chosen starting time. Given an edge $e = (u, v) \in E$ and a starting time t, the oracle returns the amount of time $r_e(t) \geq 0$ that it will take to travel from vertex u to vertex v along edge e, assuming you leave u at time t. The travel times have the property that if $t' > t$, then $t' + r_e(t') > t + r_e(t)$ (i.e., if you leave later, you will arrive later), but are otherwise arbitrary. Each query to the oracle can be performed in time $O(1)$. Design a polynomial-time algorithm to find the fastest way to get from a given starting vertex to a given destination vertex, starting at time 0. Prove that your algorithm is correct and analyze its running time.

Solution: Dijkstra's algorithm can be modified slightly to handle time-varying edge lengths $r_e(t)$. Let s denote the designated starting vertex. Algorithm 1 shows the complete algorithm.

Initially $S = \{s\}$ and $d(s) = 0$ and $\text{prev}(s) = \varnothing$;
while $S \neq V$ **do**
  Select a node $v \notin S$ for which $d'(v) = \min_{e=(u,v):\, u \in S} d(u) + r_e(d(u))$;
  Add v to S;
  Set $d(v) = d'(v)$;
  Set $\text{prev}(v) = u$ where u is a neighbor of v achieving the minimum $d'(v)$;
Let w be the destination vertex and $P^* = \varnothing$;
while $w \neq s$ **do**
  $P^* = \text{prev}(w) + P^*$;
  $w = \text{prev}(w)$;
Return P^*;

Algorithm 1: Modified Dijkstra's Algorithm

Correctness:

Claim 1. *At any point in the algorithm's execution, for any vertex $u \in S$, $d(u)$ denotes the earliest time required to reach u from s.*

Proof. We prove this by induction on the size of S. The base case of $|S| = 1$ is easy as we have $S = \{s\}$ and $d(s) = 0$. Let us assume that the claim holds when $|S| = k$ for some value of $k \geq 1$. We now grow S to size $k + 1$ by adding the node v and let (u, v) be the final edge considered, i.e. $u = \text{prev}(v)$.

By the induction hypothesis, $d(u)$ is the earliest time at which we can reach u from s. Since the variable edge lengths satisfy the property that if $t' > t$, then $t' + r_e(t') > t + r_e(t)$, we are guaranteed that $d(v) = d(u) + r_e(d(u))$ is the earliest we can reach v via a path that contains edge (u, v). Now consider any other path P from s to v; we need to show that following any such path, we cannot reach v before $d(v)$. Let $y \notin S$ be the first vertex on P that is not in S and let $x \in S$ be the node just before y. In the current iteration, since we chose to add vertex v to S instead of y, we know that $d(x) + r_{(x,y)}(d(x)) \geq d'(y) \geq d(v) = d(u) + r_{(u,v)}(d(u))$. Since we have $r_e(t) \geq 0$ for all edges e and times t, we are guaranteed that following path P, the time required to reach v is at least $d'(y) \geq d(v)$. This completes the proof by induction. □

The actual path that needs to be taken to reach the destination by the earliest time is obtained by traversing back through the edges chosen by the algorithm. P^* denotes such an optimal path.

Running Time: Just as for Dijkstra's algorithm as described in Kleinberg and Tardos (4.15), the above algorithm can be implemented to run in $O(m \log n)$ time using a priority queue.