3.24. Given: a directed acyclic graph \( G = (V, E) \)

Topologically sort \( G \)
(Gives an ordering \( v_1, v_2, \ldots, v_n \) of the nodes \( V \), where \( n = |V| \))

For \( i = 1 \) to \( n - 1 \):
  - if \( (v_i, v_{i+1}) \notin E \): output 'no'
  - Output 'yes'

A path touching all vertices is sometimes called a Hamilton path.

Suppose \( G \) has a Hamilton path. Any topological ordering must respect all the edges on this path, and must therefore put the vertices in exactly the same order as they appear on the path. The algorithm above will find this unique topological sort, and confirm the existence of the corresponding Hamilton path.

On the other hand, if \( G \) does not have a Hamilton path, the algorithm will find some topological ordering \( v_1, \ldots, v_n \), but one of the edges \( (v_i, v_{i+1}) \) will not be in \( E \).

The running time is the same as that of topological sort: \( O(|V| + |E|) \).

3.25. (a) *Give a linear-time algorithm which works for directed acyclic graphs.*

Suppose \( u \) has edges to nodes \( w_1, \ldots, w_k \). Then the nodes reachable from \( u \) are precisely: \( u \) itself, and the nodes reachable from all the \( w_i \). This gives us a simple recursive formula for \( \text{cost} \) values:

\[
\text{cost}[u] = \min\{p_u, \min_{(u,w) \in E} \text{cost}[w]\}.
\]

To make it iterative, it would help if we could make sure to compute all the \( \text{cost}[w] \) values before we get to \( \text{cost}[u] \). Well, for a dag this is easy: just handle vertices in reverse topological order! Here’s the algorithm, which is linear time because topological sorting is linear time.

Topologically sort the dag.
For each node \( u \in V \), in reverse topological order:
\[
\text{cost}[u] = \min\{p_u, \min_{(u,w) \in E} \text{cost}[w]\}.
\]

(b) *Extend this to a linear-time algorithm which works for all directed graphs.*

If two nodes \( u, v \) are in the same SCC, then the nodes reachable from \( u \) are identical to those reachable from \( v \), so \( \text{cost}[u] = \text{cost}[v] \). Therefore we do not need to distinguish between nodes in the same SCC, and we can pretty much work with the metagraph, which is a dag!

- Find the strongly connected components of \( G \).
- For each SCC \( C \): let the (meta-)price \( p^*_C \) for component \( C \) be the smallest price of its nodes, that is, \( p^*_C = \min_{u \in C} p_u \).
- Run the dag version of the algorithm (from part (a)) on the metagraph, using these metaprices \( p^*_C \). This returns component metacosts \( \text{cost}^*[C] \).
- For each SCC \( C \), and each node \( u \in C \): \( \text{cost}[u] = \text{cost}^*[C] \).

This takes linear time because the algorithm for strongly connected components and the algorithm from part (a) are both linear time.

4.1. *Dijkstra example (starting from node A).*
4.4. The problem is that the shortest cycle might involve more than one back edge. For instance, suppose DFS is run on the graph below; the dotted edges are the back edges it finds. The stated algorithm would return cycle $B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow B$ of length 5, whereas the shortest cycle is $A \rightarrow B \rightarrow F \rightarrow G \rightarrow A$ of length 4.

4.11. Given: a directed graph $G = (V, E)$ with edge lengths $l_e$.

shortest = $\infty$

For each node $u \in V$:

Run dijkstra($G, l, u$)

(Returns an array dist with distances from $u$ to all the other vertices)

For all vertices $v \neq u$ such that $(v, u) \in E$:

shortest = min{shortest, dist[v] + $l(v, u)$}

if shortest = $\infty$:

output ‘‘acyclic’’

else:

output shortest

Pick any edge $e = (v, u)$ in the shortest cycle. Then this cycle consists of the shortest path from $u$ to $v$ (which can be found by Dijkstra’s algorithm), followed by the single edge $e$. The procedure above iterates through all possibilities for $u$ in the outer loop, and all possibilities for $v$ in the inner loop.

The running time is $O(|V|)$ times the running time of Dijkstra’s algorithm, a total of $O(|V|^3)$ (if a linked list implementation of a priority queue is used).
4.17 (a) *Show how this can be done in time just $O(|V|W + |E|)$.*

Any path in the graph has length at most $(|V| - 1)W$. We will exploit this by designing a priority queue which only allows key values from 0 to $(|V| - 1)W$. Create an array $Q[0 \ldots (|V| - 1)W]$; the $k^{th}$ entry $Q[k]$ points to a doubly linked list of elements with key value $k$. In addition, maintain an index $k^*$, the current minimum key value.

- **To insert** a new element $x$ (with key value $k$), add $x$ to the front of the linked list at $Q[k]$.
- **To decreasekey** an element $x$ (with new key value $k$), remove it from the linked list it currently belongs to (deletions from a doubly-linked list are constant time) and add it, as before, to the list at $Q[k]$.
- **To deletemin**, check if there are any elements in the list at $k^*$. If not, increment $k^*$ until you encounter the first element. Return this element and remove it from its linked list.

Insert and decreasekey operations are $O(1)$. Deletemins are also $O(1)$, plus the time taken to advance $k^*$. This pointer starts at 0, and can only advance to the end of the array; therefore all these increments together take a total of $O(|V|W)$ time. The overall running time of Dijkstra’s algorithm is (as we saw in class)

$$O(|V| + |E| + (|V| \times \text{insert}) + (|V| \times \text{deletemin}) + (|E| \times \text{decreasekey})).$$

With our implementation, this works out to $O(|V|W + |E|)$.

(b) *Show that it can be done in time $O((|V| + |E|) \log W)$.*

Suppose we are in the middle of Dijkstra’s algorithm and the most recent node we chose (added to the “known region”) was at distance $d$. Then, everything in the priority queue is (1) at distance $\geq d$ and (2) connected by an edge to something at distance $\leq d$. Since the maximum edge length is $W$, everything in the priority queue has a key in the range $d, d + 1, \ldots, d + W$: only $W + 1$ possible values!

This suggests yet another priority queue implementation: use a binary heap in which *every node has a different key*, and contains a pointer to a doubly-linked list of elements with that key value. Therefore, at any given time the binary tree will contain at most $W + 1$ nodes, giving an insert/deletem in time of just $O(\log W)$. Well, almost: there’s one additional detail. When we have to insert a new element with key value $k$, we need to quickly check whether $k$ already has a node in the tree, and if so, where it is (we don’t have enough time to search the tree). A circular array of size $W + 1$ (containing pointers into the tree) works nicely for this.

The total running time is thus $O((|V| + |E|) \log W)$, plus $O(W)$ time for creating the array.