1 Finding a DAG

- A DAG (Directed Acyclic Graph) can be detected by running a depth first search
- Name the type of edges encountered – tree edges, forward edges, back edges, cross edges
- Tree edges are edges that are used to visit unvisited nodes in DFS
- Forward edges are edges that are used to visit nodes already visited by nodes later in the DFS
- Back edges are edges that are used to visit nodes already visited by nodes earlier in the DFS
- Cross edges are edges that are used to visit nodes visited by nodes elsewhere in the DFS
- A graph is a DAG if and only if no back edges are found during DFS

2 Topological Sorting

- While doing a DFS, we can mark start and finish times on each node indicating when we started examining the node and its descendents and when we finished doing so
- We keep a counter, and for each unvisited node that we enter, we increase the counter, and for each node we leave (once visiting all descendents), we increase the counter
- After marking times on each node after a DFS, we print in order of decreasing finishing times, producing a topological sort
- To do the topological sort as we go, we can place each node on the front of a linked list as it is finished, producing a dependency-ordered list of nodes
- Useful in many places – dependency checking of all types (Makefile, PERT diagrams, etc...)

3 Strongly Connected Components

- We know that directed graphs can be strongly connected, but what if they are not? What other connectivity information can we get?
- We’d like to find the strongly connected components (SCCs) of a graph, that is, the subgraphs (sets of nodes) that themselves are strongly connected (there exists a path from every node in the set to every other)
- Run DFS, computing starting and finishing times
- Run DFS on the inverse graph, $G^T$ (that is, make all edges point in the opposite direction), and look at nodes in order of decreasing finishing times
- From this DFS run (on $G^T$), output each strongly connected component (when DFS runs out of nodes to visit, we have found one strongly connected component, and can move on to the next)
- Useful algorithm in many places – compiler optimization routines, network connectivity testing, etc...
4  Announcements

- Reading: Data Structures (Into Java) – 12.3.4, 12.3.6; Goodrich and Tamassia – 12.6

5  Greedy Algorithms

- Always choose what seems to be optimal at the current step
- In some cases this yields the actual optimal answer, but sometimes not
- Huffman coding and Dijkstra’s algorithm are greedy

6  Shortest Paths

- Problem: find the shortest path from a start node s to all nodes in a graph
- Plan: as we travel out from the start node, keep a running “estimate” of the distance from s to each node, improving the estimate as more information becomes available (never underestimate)

Dijkstra’s Algorithm:

Given a graph \( G = (V, E) \) and \( s \in V \), compute \( D \), which contains shortest path lengths \( s \rightarrow x, \forall x \in V \)
Also compute \( P \), which contains back pointers to construct the paths \( s \rightarrow x \)

\[
D[s] = 0 \\
\text{For all } u \in V, u \neq s \quad D[u] = \infty
\]

Create an empty back pointer array \( P \)
Construct a priority queue \( Q \) with the array \( D \) as keys for nodes

```
while (!Q.isEmpty()) {
    u = Q.removeMin()
    For each \( v, (u, v) \in E \)
    if \( (D[u] + w(u,v) < D[v]) \)
        \( D[v] = D[u] + w(u,v) \)
        Relax \( v \) in \( Q \) (decrease-key \( v \))
        \( P[v] = u \)
}
```

7  Question 1

What is/are the running time/s of Dijkstra’s Algorithm?

White: \( O(|V|) \)

Blue: \( O(|E|) \)

Yellow: \( O(|E|lg|E|) \)
Green: $O((|E| + |V|)lg|V|)$

Pink: $O((|E| + |V|)^2)$

Green and Pink are correct, with Green having the tighter bound. In the while loop of Dijkstra’s algorithm, the decrease key operation must be executed degree(u) times and the node must have been extracted from the priority queue at the beginning – assuming a binary heap priority queue implementation, each iteration must take $(\text{degree}(u) + 1)lg|V|$, since there are $|V|$ elements in the heap. Over all iterations, the degree(u) terms add up to $|E|$, giving us $(|E| + |V|)lg|V|$.

8 Question 2

Suppose we need to compute the shortest-path from a starting node s to a particular destination node t on an unweighted graph – what’s a faster way than executing Dijkstra’s Algorithm as we just saw?

White: There isn’t a faster way.

Blue: Compute the strongly-connected components of G, select the SCC that s resides in, and perform a DFS, counting the number of edges traversed from s to t

Yellow: Perform a BFS, counting the number of edges traversed from s to t

Green: Topologically sort G and take the absolute value of the difference in finishing times between s and t

Pink: Store G in an adjacency matrix M, and compute $M^k$ with increasing k values until s and t are connected by an edge in $M^k$, yielding k as the shortest-path

Blue assumes that there exist strongly-connected components in G – if there were, then it would work if s and t were in the the same SCC, but then computation of the SCCs would have been wasteful. Green will not give the right answer in most cases – it requires that G be a DAG and also that all nodes have an out-degree of 1.

Yellow and Pink will work, and it is clear that Yellow will work faster than Pink (matrix multiplication is slow).