More Graph Algorithms

Weiss ch. 14

Exercise: MST idea from yesterday

- Alternative minimum spanning tree algorithm idea
- Idea: Look at smallest edge not yet examined. Select it so long as it does not create a cycle with edges selected so far.

```
toDo ← all edges in graph
while (num selected < |V|-1 && toDo not empty) {
   c ← find and remove smallest edge from toDo
   if (selected plus c has no cycles)
         selected.insert(c)
```

What data structures are needed?

What is running time?

Kruskal's Algorithm

- Can sort edges in toDo set first
  - O(|E| log |E|) for sort
  - O(1) for finding and removing minimum (at most |E| times)
  - O(|E| log |E|) overall contribution to running time
- Alternatively, put them in a heap
  - O(|E|) for building the heap
  - O(log |E|) for finding and removing minimum (at most |E| times)
  - O(|E| log |E|) overall contribution to running time
- In practice?
  - How many edges are examined? → influences choice of heap vs. sorting

Kruskal's Algorithm: Checking for Cycles

- At each step, we have a set of selected edges, and one new edge. (Say there are n edges together.)

- Need to check if together they form any cycles.

- How?

Checking for Cycles

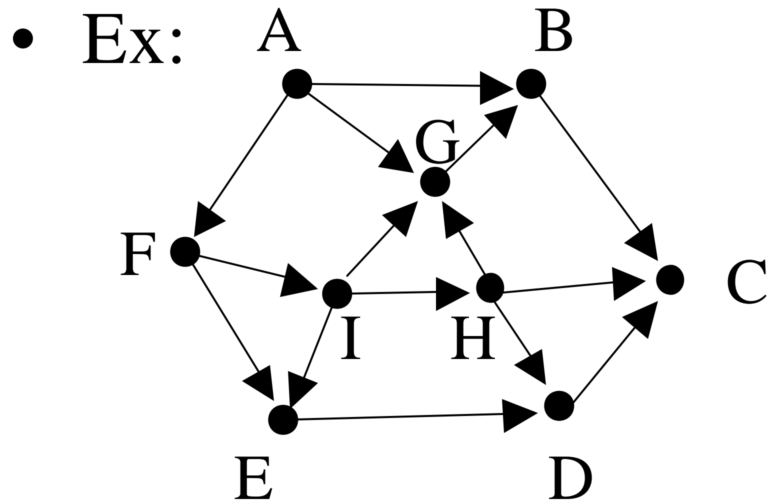
- Adding edge (u, v)
- Do a graph search using edges selected already
  - Look for a path from u to v
  - Slow – O(|E|) worst case at every step → O(|E| * |E|)
- Better: keep track of which "component" each node is in – can we do it in O(log |E|) ? → O(|E| log |E|)
  - Iff u and v in same component, then cycle would be formed
  - Update "component" labels whenever new edge added
    - component of u and component of v merged into a single component
  - There is a data structure for doing this union/find
    - details beyond cs211

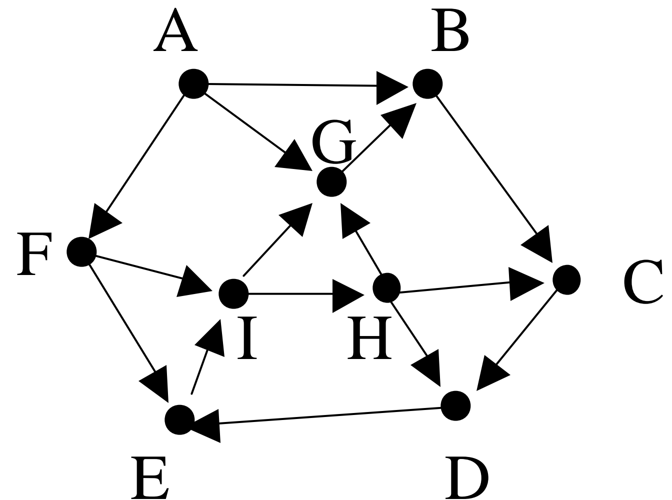
Graph Sort Orders

- A *sort order* is just some ordering of the nodes in a graph.
- We have seen:
  - BFS ordering – in order of (some) BFS traversal from A
  - DFS ordering – in order of (some) DFS traversal from A
  - Minimum path ordering – in order of min distance to A
- A move flexible ordering is possible for directed graphs…

Topological Sort

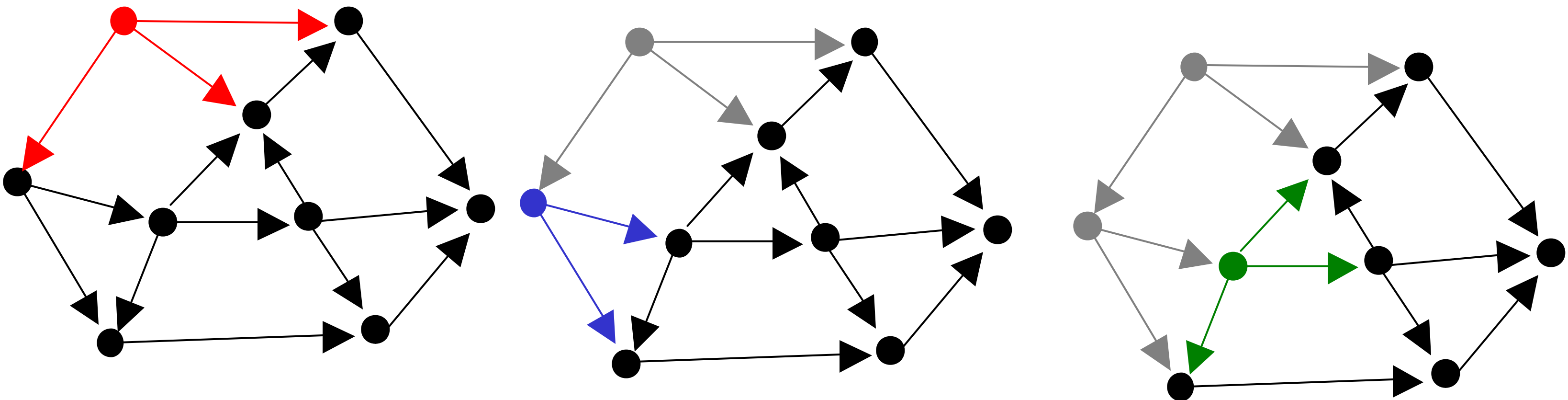
- Directed graph G.
- Rule: if there is an edge u → v, then u must come before v.
- Ex:





Intuition

- Cycles make topological sort impossible.
- Select any node with no in-edges
  – print it
  – delete it
  – and delete all the edges leaving it
- Repeat
- What if there are some nodes left over?
- Implementation? Efficiency?



Implementation

- Start with a list of nodes with in-degree = 0
- Select any edge from list
  - mark as deleted
  - mark all outgoing edges as deleted
  - update in-degree of the destinations of those edges
    - If any drops below zero, add to the list
- Running time?

Implementation

- Start with a list of nodes with in-degree = 0
- Select any edge from list
  - mark as deleted
  - mark all outgoing edges as deleted
  - update in-degree of the destinations of those edges
    - If any drops below zero, add to the list
- Running time? In all, algorithm does work:
  - O(|V|) to construct initial list
  - Every edge marked “deleted” at most once: O(|E|) total
  - Every node marked “deleted” at most once: O(|V|) total
  - So linear time overall (in |E| and |V|)

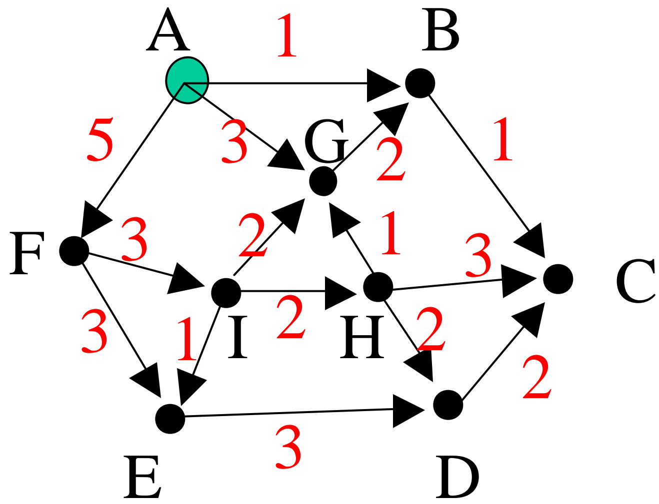
Why should we care?

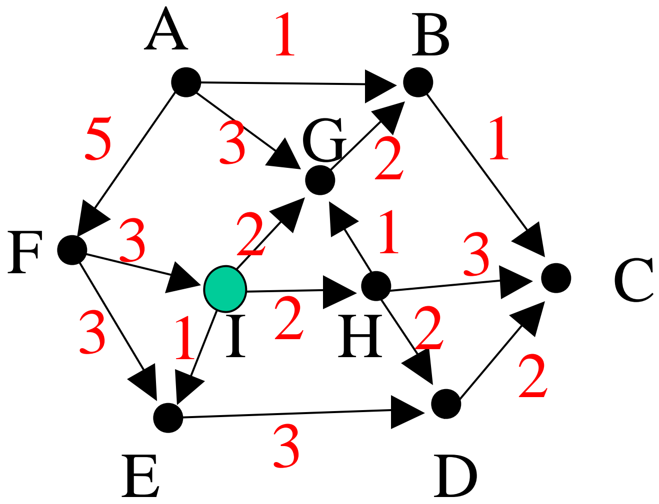
- Shortest path problem in directed, acyclic graph
  - Called a DAG for short
- General problem:
  - Given a DAG input G with weights on edges
  - Find shortest paths from source A to every other vertex

Naïve Solution

- Just do Dijkstra’s algorithm, with modification to support directed edges
  - O(|E| log |E|)
- We can do better. (much better).

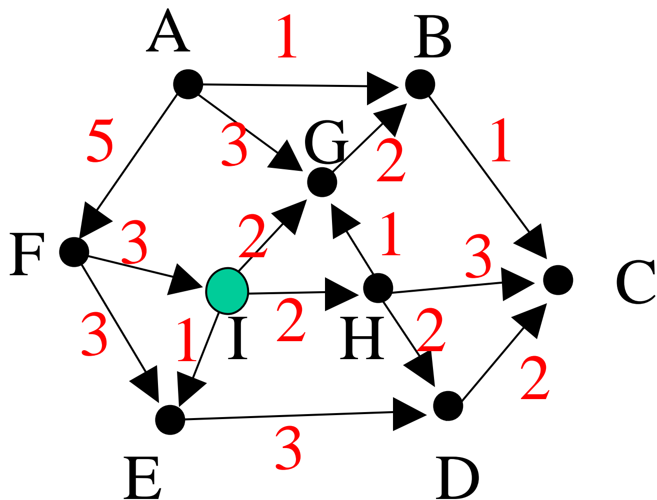
Example





Algorithm

- Given a DAG G, and a source vertex A
- Do topological sort on G, and visit nodes in that order
- Nodes before A get distance negative infinity
- Node A gets distance 0
- Each node u after A computes distance based on incoming edges (all of which have sources before u in the list, and so are already computed).
- Does it work for negative weights?



One sort order: A F I E H G D B C

A = -inf

F = -inf

I = 0

E = 3 + F or 1 + I = 1

H = 2 + I = 2

G = 3 + A or 2 + I or 1 + H = 2

…

Algorithmic Complexity

- Our algorithm on a directed acyclic graph (possibly with negative weights)
  - Clearly linear time in |V| + |E|
  - Topological sort is linear time
  - Then examine each node and edge once more
- Bellman-Ford on possibly cyclic directed graph with possibly negative weights
  - Similar to our algorithm, but no topological sort
  - Needs to examine every node & edge on every pass
    - When examinning node v, update estimate by looking at every edge u→v
  - O(|V| |E|)
  - But: can detect and handle cycles
    - Each node v examined at most |V| times… why?