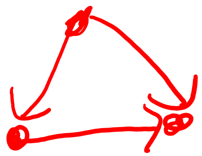
Graphs

Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph without cycles



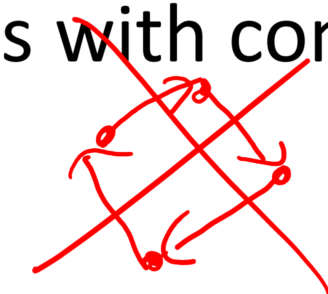
- How can you tell whether a directed graph has a cycle?

Directed Acyclic Graphs (DAGs)

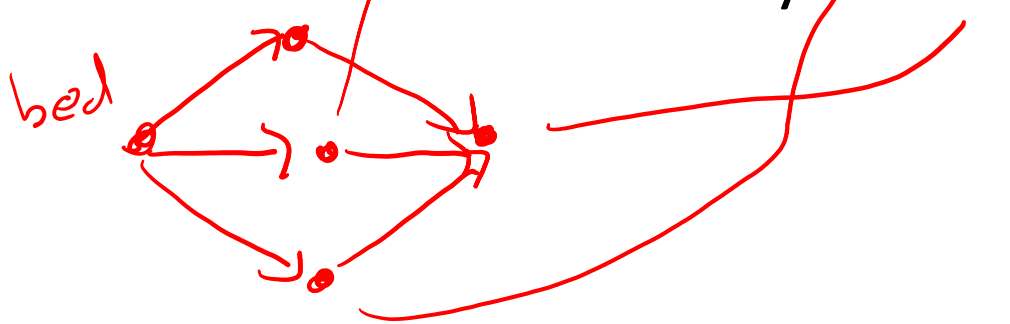
- A DAG is a directed graph without cycles
- How can you tell whether a directed graph has a cycle?
- Look for the presence of back edges!
  - Run DFS
  - Regardless of where you start, back edge $\Longleftrightarrow$ cycle

Directed Acyclic Graphs (DAGs)

- What good are DAGs?
- Often used to model situations with constraints

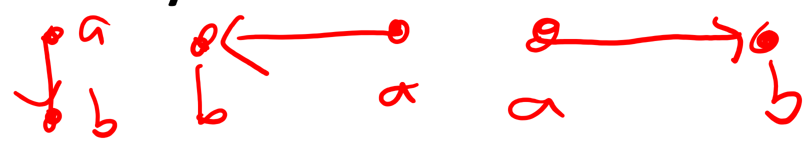


- Every day…
  - First you get out of bed
  - Then you could eat breakfast, brush your teeth, get dressed
  - All three of those have to be done before you leave the house
  - Etc.



Directed Acyclic Graphs (DAGs)

- A source is a node with no incoming edges
- A sink is a node with no outgoing edges
- A linearization is a sorting of the nodes so that every edge goes from an early node to a later node



  - Every DAG can be linearized!

In-Class Exercise

B D A C E F

source

- Find the sources, sinks, and all possible linearizations

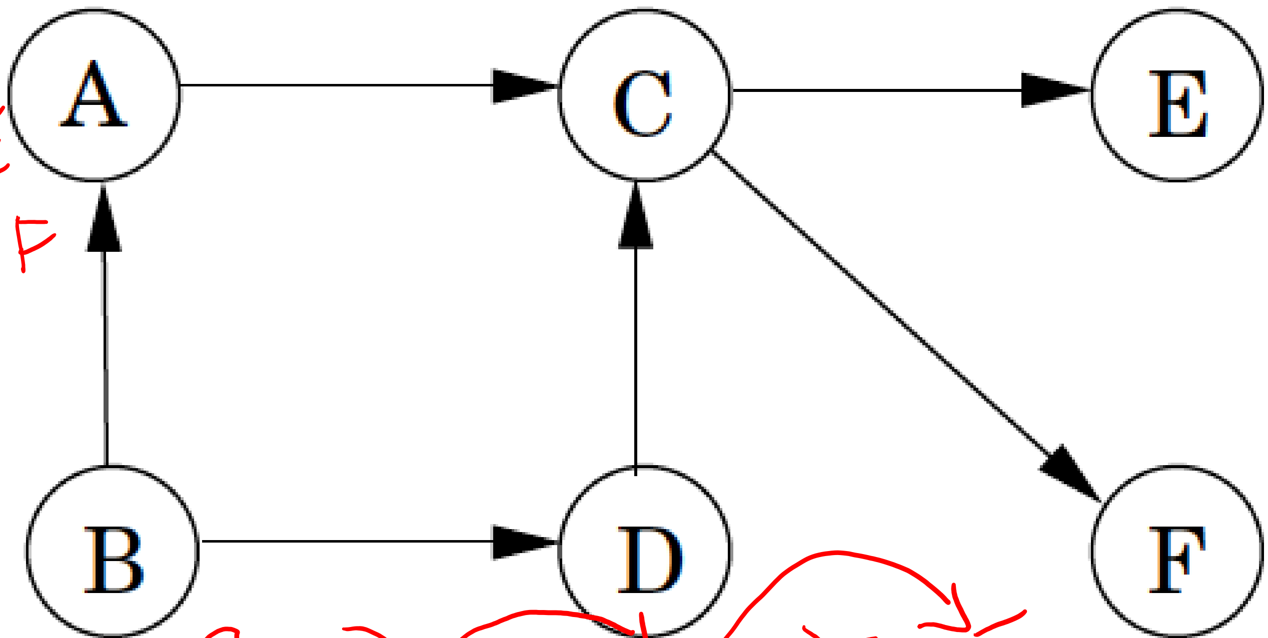
/B /E /F

4

B → A D C E F

BADCFE

BDACEF

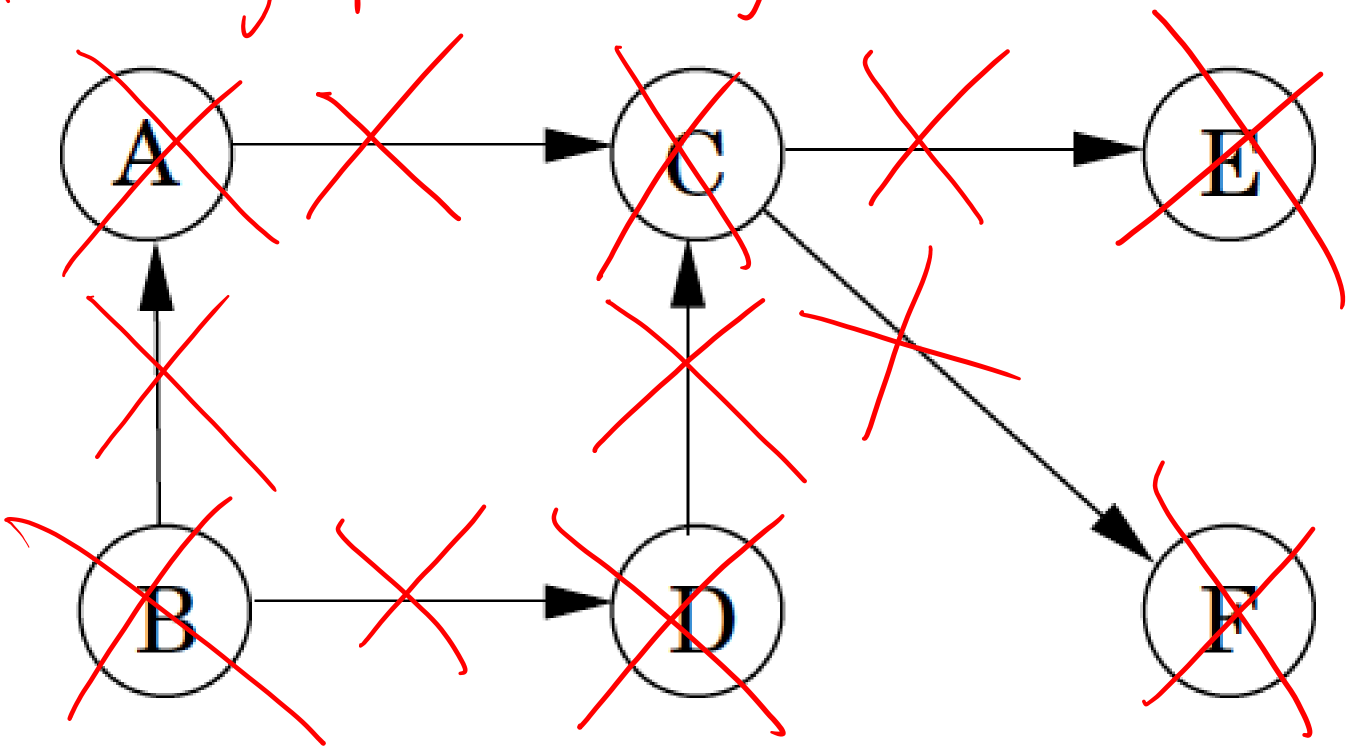


B → D A → C F E

In-Class Exercise

1. Find a source
2. Add ~~to front~~ to next pos.
3. Delete from graph
4. Repeat til graph is empty

B A D C F E

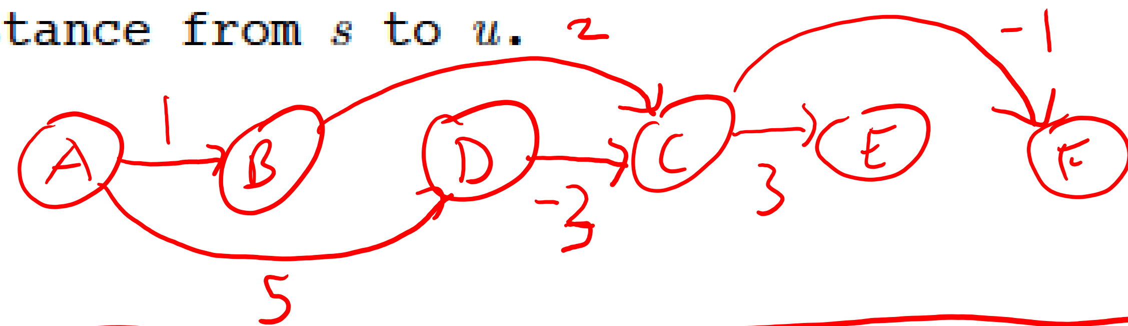


Shortest Paths on DAGS

```
procedure dag-shortest-paths(G, l, s)          s = A
Input:     Dag G = (V, E);
           edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:    For all vertices u reachable from s, dist(u) is set
           to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
    for all edges (u, v) ∈ E:
        update(u, v)
```



	A	B	D	C	E	F
dist	0	~~∞~~ 1	~~∞~~ 5	~~∞~~ ~~3~~ 2	~~∞~~ 5	~~∞~~ 1

~~if dist~~ $dist(v) = \min(dist(v), dist(u) + l(u,v))$

In-Class Exercise

procedure dag-shortest-paths(G, l, s)
Input: Dag $G = (V, E)$;
edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.

for all $u \in V$:
    dist$(u) = \cancel{\infty}$ $-\infty$
    prev$(u) =$ nil

dist$(s) = 0$
Linearize G
for each $u \in V$, in linearized order:
    for all edges $(u, v) \in E$:
        update(u, v)

change min → max

How can we modify this algorithm to find longest paths instead of shortest paths?

So When Can We Find Shortest Paths?

length of path = #edges

- If the graph is unweighted, use BFS - $O(V+E)$
- If graph is weighted, but all weights are non-negative, use Dijkstra's $O((V+E)\log V)$ - min-heap
  $O(E+V\log V)$ Fib. heap
- If the graph is weighted and has negative edges, but no negative cycles, use Bellman-Ford - slow! $O(V \cdot E)$
- If the graph has negative cycles, no shortest paths exist
- If graph is a DAG (no cycles, positive or negative), use the DAG shortest path algorithm - fastest!

Greedy Algorithms

Introduction to Greedy Algorithms

- For some algorithms, you need to look ahead or revise past decisions to get the best solution
- Greedy algorithms build a solution piece-by-piece, without revising past decisions or looking to future
  - What greedy algorithms have we seen?

BFS

Dijkstra's?

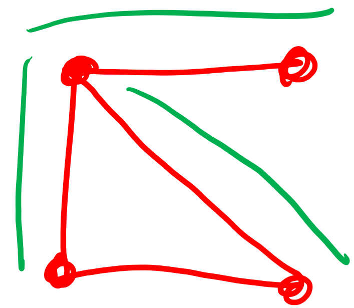
Spanning Trees

- We want to find spanning trees in an undirected graph

connected graph w/out cycles

- A spanning tree is a tree that touches every node
- How do we find spanning trees?

- DFS, return tree edges



Rules for Making Spanning Trees

1. Don't add an edge if it would create a cycle!
2. A tree on *n* nodes has $n-1$ edges
3. An undirected graph is a tree if and only if there is one unique path between every pair of nodes. (Why?)

These are all Equivalent

1. A graph is a tree
2. A graph has $n - 1$ edges and n nodes, and is connected
3. A graph is connected and acyclic

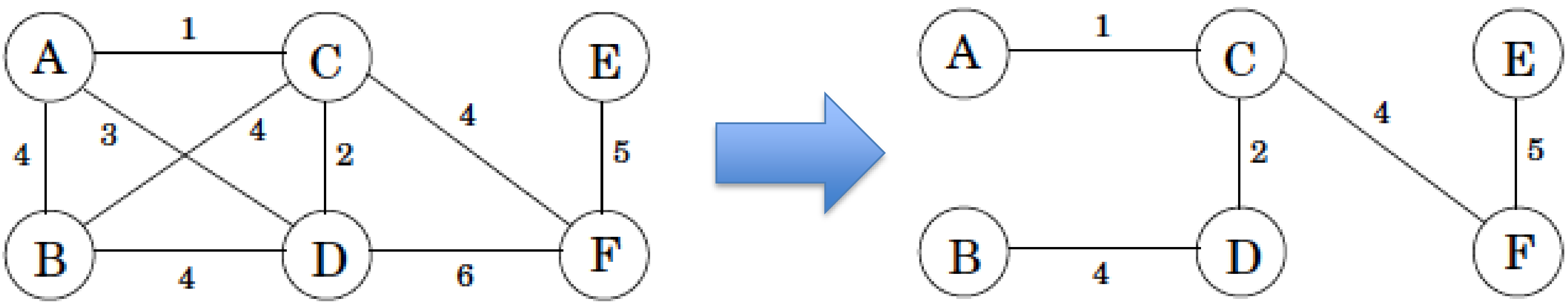
Rules for Making Spanning Trees

1. Don’t add an edge if it would create a cycle!
2. A tree on *n* nodes has *n* - 1 edges
3. An undirected graph is a tree if and only if there is one unique path between every pair of nodes.

With these rules, can you come up with a simple algorithm for making a spanning tree?

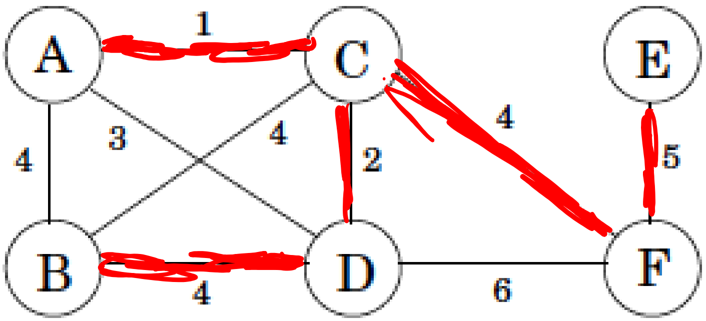
Minimum Spanning Trees

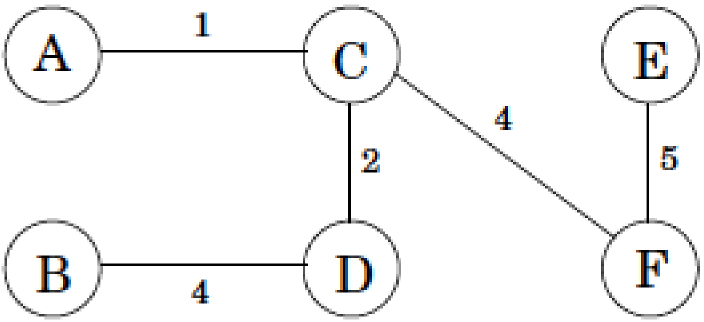
- A minimum spanning tree in an undirected, weighted graph is the spanning tree with minimum total weight



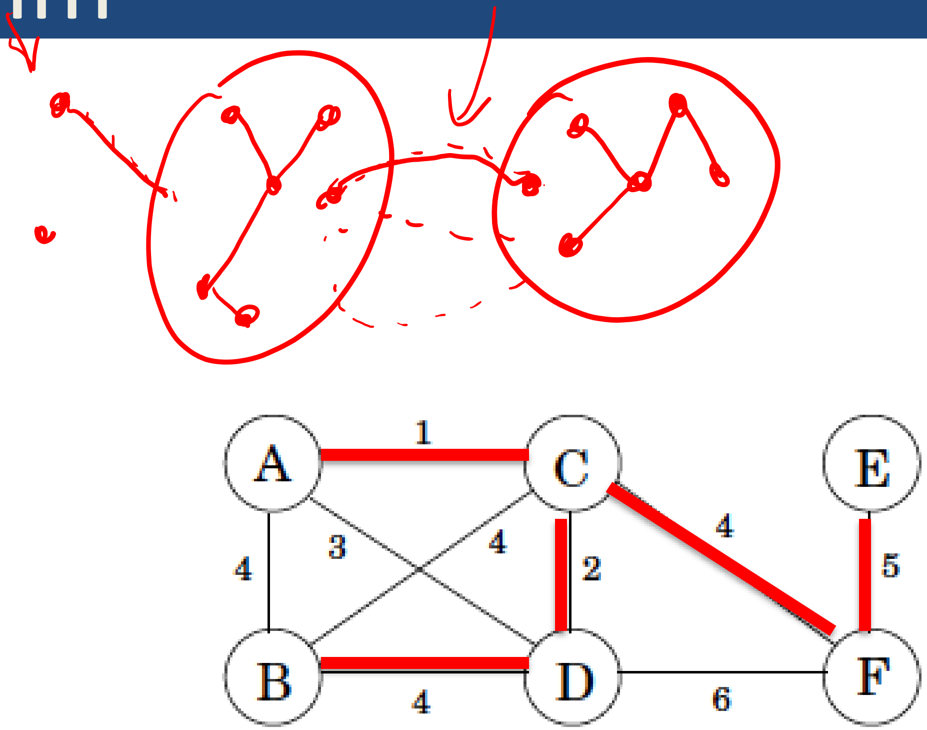
Finding Minimum Spanning Trees: Kruskal's Algorithm

- Kruskal's Algorithm:
  1. Sort the edges from least cost to greatest cost
  2. Add each edge in order to the spanning tree, unless it would make a cycle





Finding Minimum Spanning Trees: Kruskal's Algorithm



Proof of Correctness of Kruskal's

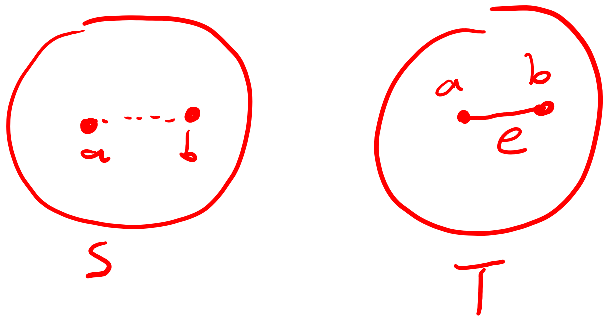
Claim: Kruskal's works! (give a MST)

Proof: First, we argue that Kruskal's returns a spanning tree. Let G be an undirected, weighted, distinct connected graph. Let T be what is returned by Kruskal's. T is acyclic because Kruskal's doesn't add an edge if it would create a cycle. T must be connected. If it had 2+ components, then there must be an edge connecting those comps in G. When Kruskal's encountered that edge, it would have added it, since it would not make a cycle.

Proof of Correctness of Kruskal's

Next, we argue ~~that~~ that Kruskal's returns a minimum ST. Suppose for a contradiction that it didn't. That means there is some other ST, S that has lighter total weight than T, and S is the true MST. Let e be the first (lightest) edge where S & T differ. It must be the case that $e \in T$ but $e \notin S$. This is because the only reason Kruskal's would skip e is if it made a cycle with what came before, but S & T agree on everything before, so then it couldn't be in S, either.

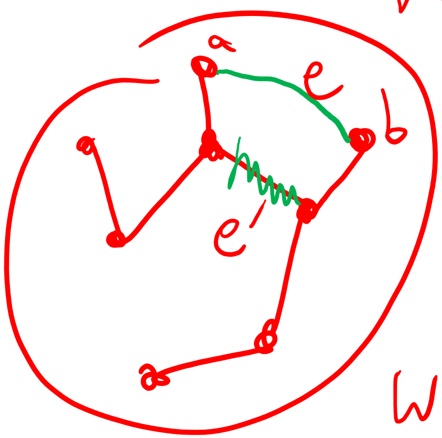
Proof of Correctness of Kruskal's



In S, because S is a ST, there is a unique path P from $a \to b$, e is not on that path. So there must be an edge $e' \in P$ that has weight greater than e. This is because if they all had weight $< w(e)$, Kruskal's would have added them to T; but if it had done that, adding e would make a cycle, but in actuality, it did add e. So now consider $S' = S + e - e'$. $w(S') < w(S)$, but S' is still a

Proof of Correctness of Kruskal's

spanning tree. The reason it's still a spanning tree because although paths in S were broken by removing e', adding e reconnects $a \leftrightarrow b$, so re-joins those two separate components. This is a contradiction, because now we've constructed a spanning tree lighter than S, but S was assumed to be the true MST. $\square$



In-Class Question:

- Suppose we find a minimum spanning tree *T* in a graph
- Suppose we then add 1 to the weight of every edge in the graph
- Is *T* still a minimum spanning tree? Why or why not? yes

edges = n - 1