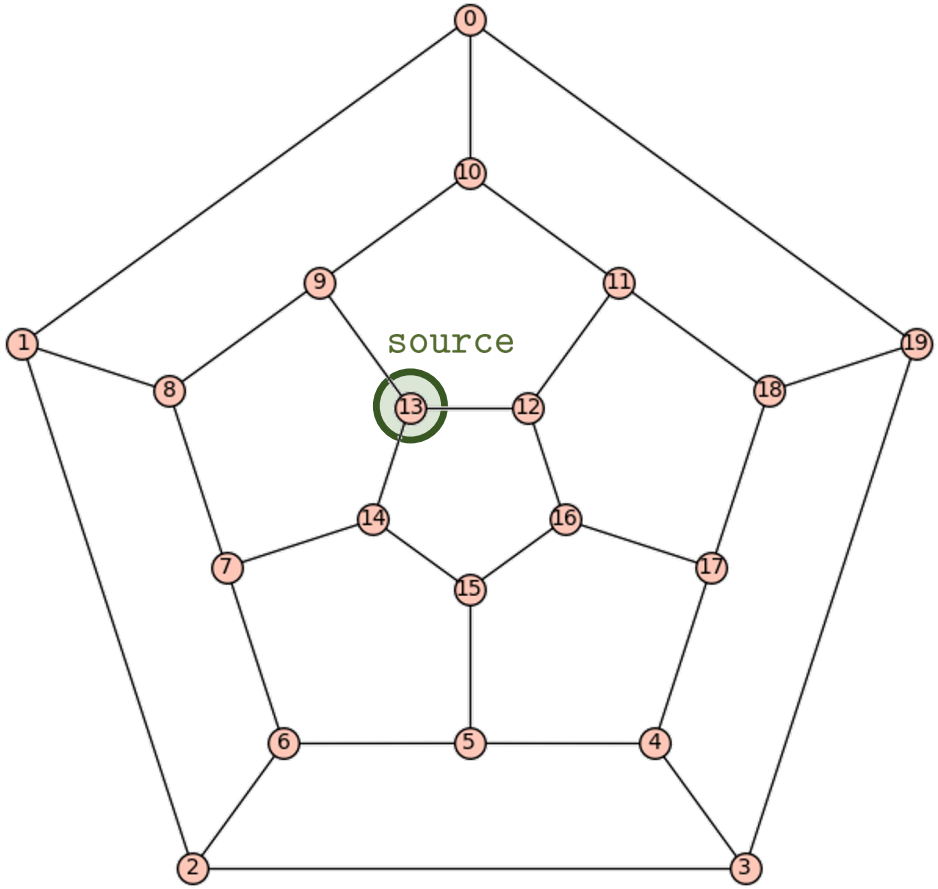
Algorithms

Single Source Shortest Paths (SSSP)

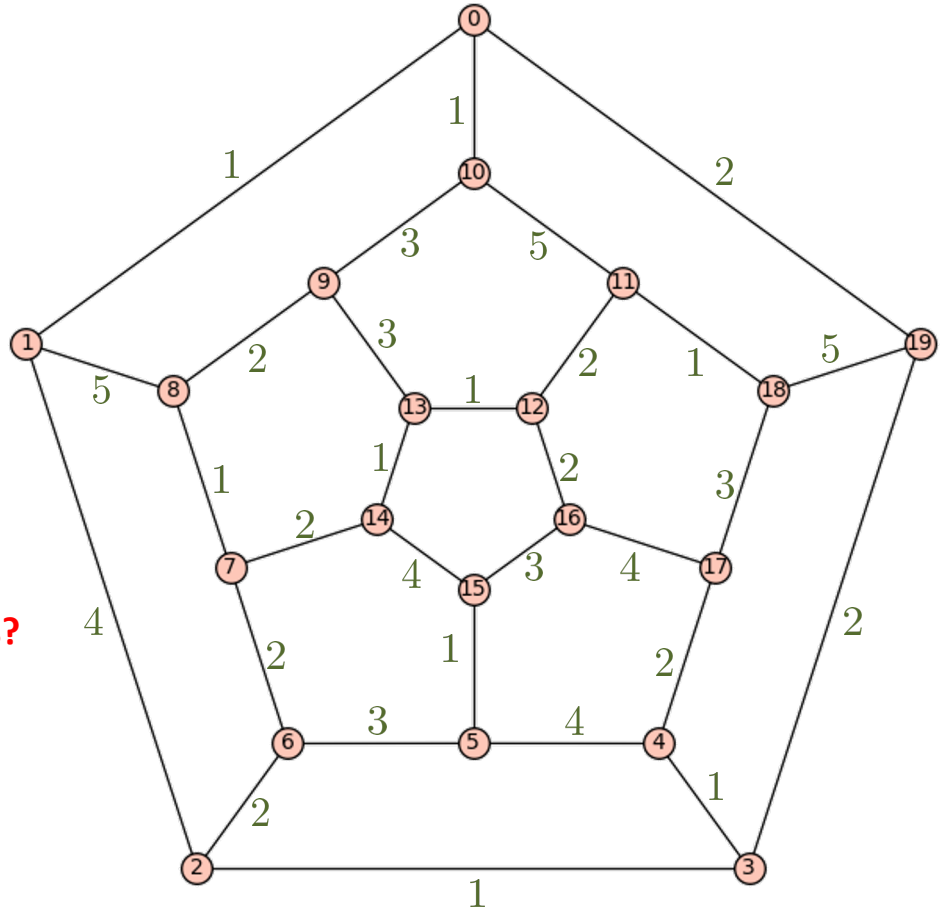
How do we find the shortest path from a given node in a graph to all other nodes?

BFS!



Single Source Shortest Paths (SSSP)

What if we had non-negative weights on the edges indicating its "length"?



If numbers represent time, then.:
What is the earliest time to reach each of the vertices?

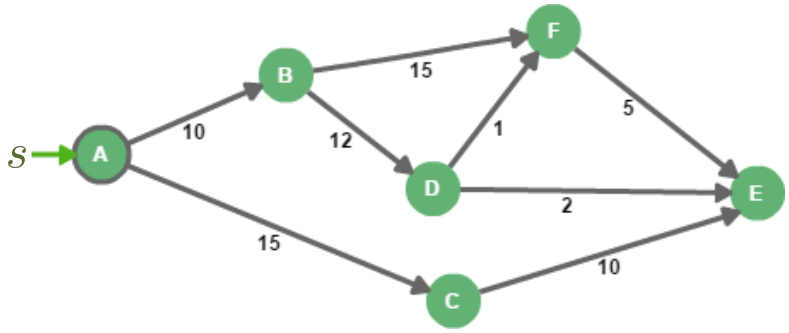
Single Source Shortest Paths (SSSP)

Observation. It suffices to find an algorithm for directed graphs.

An algorithm for directed graph can also be used for undirected graphs.

We are given a directed graph in which each edge e has a positive length $\ell(e)$.

We are also specified a *source* vertex s in the graph.



Goal. Find the shortest directed path from s to all other vertices in the graph.

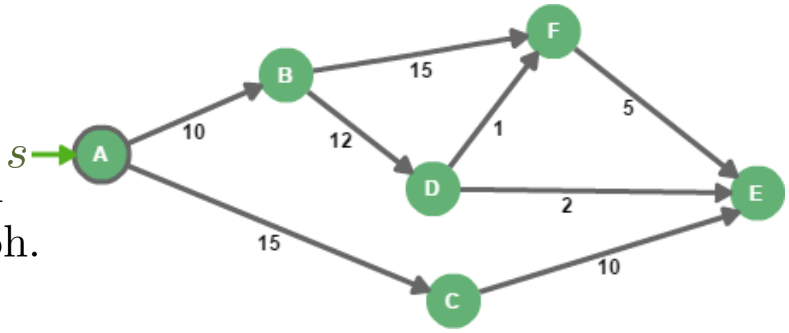
Would minimizing the next step work?

Single Source Shortest Paths (SSSP)

Observation. It suffices to find an algorithm for directed graphs.

An algorithm for directed graph can also be used for undirected graphs.

Goal. Find the shortest directed path from s to all other vertices in the graph.



Dijkstra's Algorithm for SSSP

At any point in time we maintain a set of vertices S for which we have already computed the shortest path distance from s.

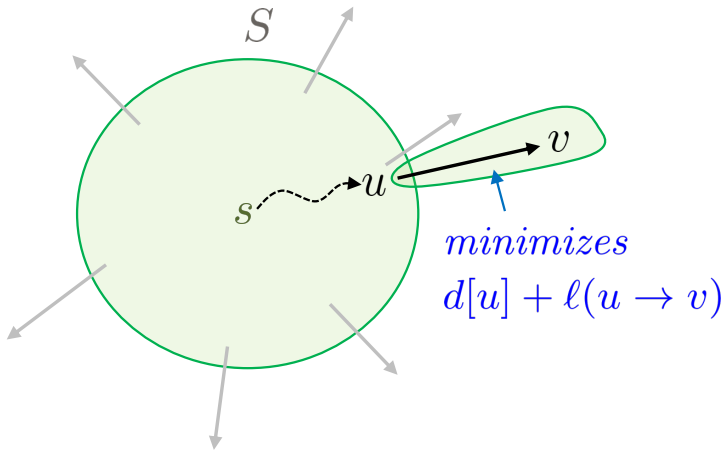
Initially $S = \{s\}$ and $d[s] = 0$.

$v \in N(S)$

We find an edge $u \to v$ from a vertex $u \in S$ to a vertex $v \notin S$ which *minimizes* $d[u] + \ell(u \to v)$.

We set $d[v] = d[u] + \ell(u \to v)$, and $S = S \cup \{v\}$.

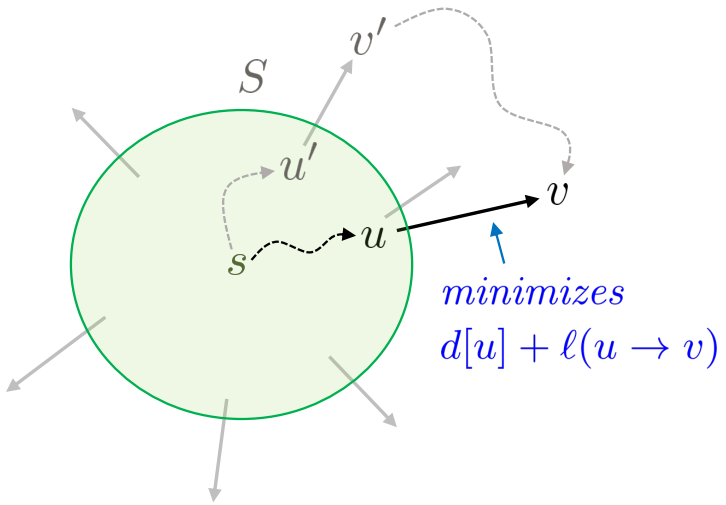
We repeat until all vertices belong to S.



Claim. The shortest $s \rightsquigarrow u$ path followed by $u \to v$ is **a** shortest $s \rightsquigarrow v$ path.

Dijkstra's Algorithm for SSSP

Claim. The shortest $s \rightsquigarrow u$ path followed by $u \to v$ is **a** shortest $s \rightsquigarrow v$ path.



Proof. Assume otherwise.

The shortest $s \rightsquigarrow v$ path leaves the set S via some edge $u' \to v'$.

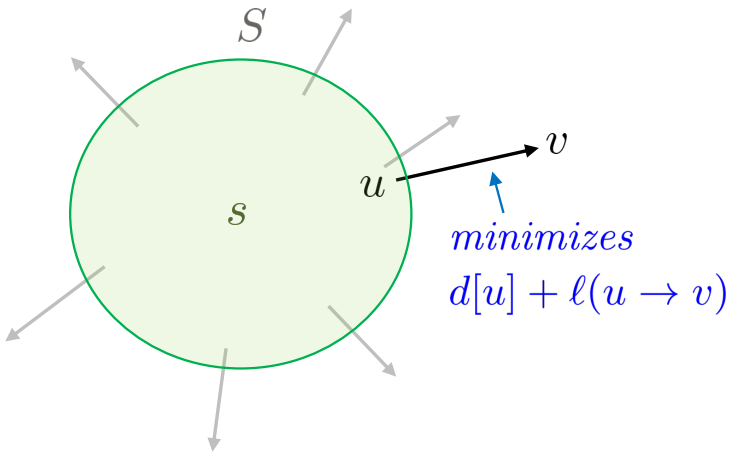
Since the $s \rightsquigarrow v$ path that consists of the shortest $s \rightsquigarrow u$ path followed by $u \to v$ has length $d[u] + \ell(u \to v)$, the shortest $s \rightsquigarrow v$ path must have a smaller length.

This implies that $d[u'] + \ell(u' \to v') < d[u] + \ell(u \to v)$,
which contradicts the assumption that $u \to v$ minimizes $d[u] + \ell(u \to v)$. ■

Dijkstra's Algorithm for SSSP

How do we implement this algorithm?

In each iteration, we could check all the edges going out of the set S and find the edge $u \to v$ minimizing $d[u] + \ell(u \to v)$ but this is inefficient.



Note that we don't really care about the edge $u \to v$, we only need to know the next vertex v to add to the set S.

So, we need the vertex $v \notin S$ that minimizes $d[v] := \min_{u \to v \,:\, u \in S} \{d[u] + \ell(u \to v)\}$

"tentative distance" of v from s.

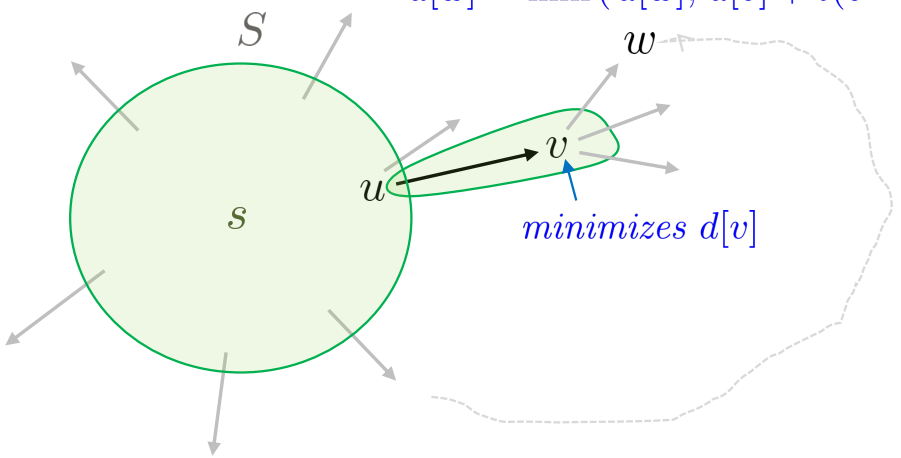
Note that we use "tentative distance" only for vertices not in S.

For any $x \in S$, $d[x]$ is the length of the shortest $s \rightsquigarrow x$ path, as before.

Dijkstra's Algorithm for SSSP

$d[w] = \min\,(\,d[w],\, d[v] + \ell(v \to w)\,)$

Maintaining $d[x]$.



Initially:

$S = \emptyset$

$d[s] = 0$ and $d[x] = \infty$ for all $x \neq s$.

Whenever a new vertex v is included in S, we go over all edges $v \to w$ and update $d[w]$ as follows: $d[w] = \min\,(\,d[w],\, d[v] + \ell(v \to w)\,)$.

Meaning of $d[x]$.

For any $x \in S$, $d[x]$ is the length of the shortest $s \rightsquigarrow x$ path. *actual distance*

For any $x \notin S$, $d[x]$ is the length of the shortest $s \rightsquigarrow x$ path that apart from x, only contains vertices in S. *tentative distance*

Dijkstra's Algorithm for SSSP

Pseudocode.

```
d:  array storing tentative distances

d[s] = 0
for each vertex v ≠ s: d[v] = ∞

S = ∅

while S ≠ V:        V is the set of all vertices

    v = arg min { d[x] :  x ∈ V }      vertex with the smallest tentative distance

    S = S ∪ {v}

    for each edge v → w:

        d[w] = min( d[w], d[v] + ℓ(v → w) )      update tentative distances to all out-neighbors of v
```

How do we do this efficiently?

Dijkstra's Algorithm for SSSP

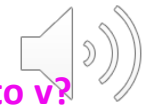
Questions raised in class

```
d:  array storing tentative distances

d[s] = 0
for each vertex v ≠ s: d[v] = ∞

S = ∅

while S ≠ V:        V is the set of all vertices

    v = arg min { d[x] :  x ∈ V }      vertex with the smallest tentative distance

    S = S ∪ {v}

    for each edge v → w:

        d[w] = min( d[w], d[v] + ℓ(v → w) )      update tentative distances to all out-neighbors of v
```

In every round, the vertex v with min d[x] is added to the set S. Then its neighbors are checked and their tentative distances updated.

How do we do this efficiently?

Exercise: why the vertex v with the minimum d[v] is added to the set S? i.e. why is d[v] the minimal path from s to v?



Dijkstra's Algorithm for SSSP

We will use a data structure called a ***priority queue*** which supports the following operations:

$Insert(x, p)$: insert the element x with priority p

$ExtractMin()$: remove and return the element x with the smallest priority

$DecreaseKey(x, \Delta)$: decrease the priority of the element x by $\Delta > 0$.

$Empty()$: returns whether the priority queue is empty.

In a simple implementation using binary heaps each of these operations takes $O(\log n)$ time where n is the number of items in the data structure.

In our algorithm, we first insert all the vertices with their initial tentative distances as priorities.

We use $ExtractMin$ to get the vertex with the minimum tentative distance.

We use $DecreaseKey$ when we need to update the tentative distances.

Dijkstra's Algorithm for SSSP

Pseudocode.

```
d:  array storing tentative distances
d[s] = 0
for each vertex v ≠ s: d[v] = ∞
Q = empty priority queue
for each vertex v: Q.Insert(v, d[v])
while !Q.Empty():
    v = Q.ExtractMin()
    for each edge v → w:
        Δ = d[w] - (d[v] + ℓ(v → w))
        if Δ > 0: Q.DecreaseKey(w, Δ)
```

*Q stores the vertices **not** in S*

How many of operations of each kind do we use?

Insert, Empty and ExtractMin are called $O(n)$ times, DecreaseKey is called $O(m)$ times

Overall running time?

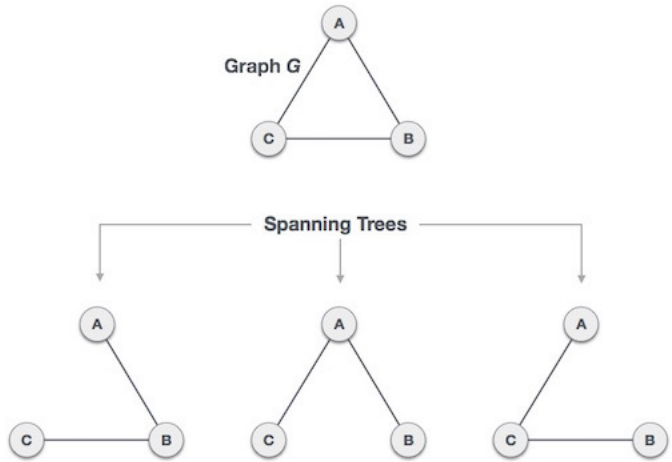
$n = \#vertices, m = \#edges$

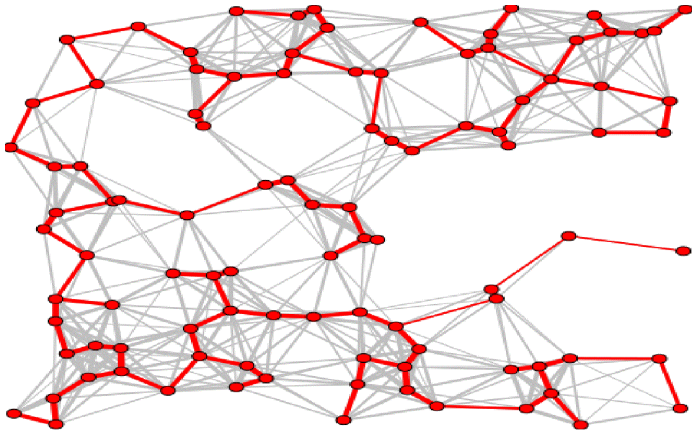
$O((m+n)\log n)$

Exercise: why is every edge visited once?

Spanning Tree

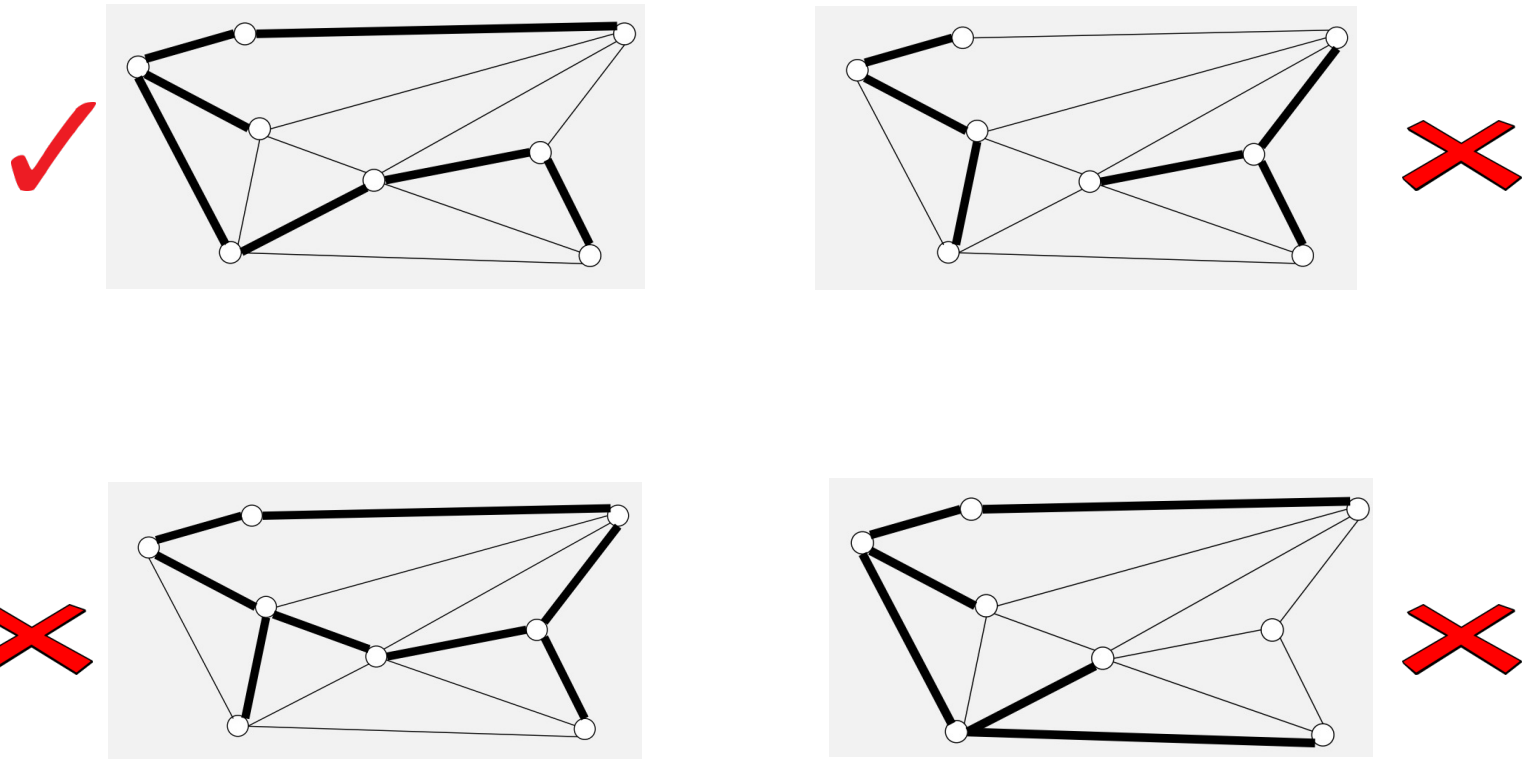
Given a connected graph G, a **spanning tree** of G is a subgraph of G which includes all vertices and forms a tree.





Spanning Tree

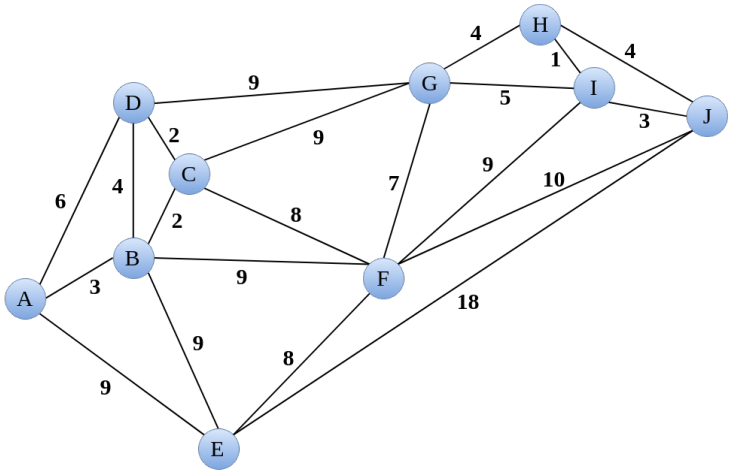
Which of the following show a spanning tree?



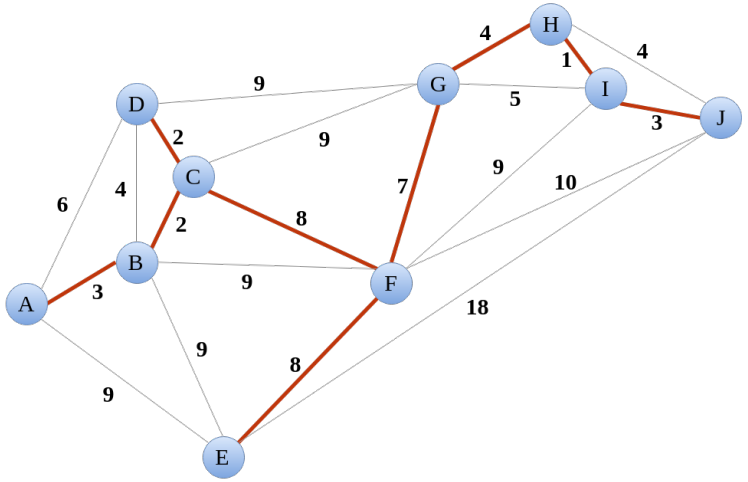
Minimum Spanning Tree

Let G be a connected undirected graph with +**ve** weights on the edges.

A **minimum spanning tree** of G is a spanning tree of G that minimizes the total weight of the edges in the tree.



G



MST of G

Many applications including network design, cluster analysis, image processing, error correction etc.

See https://en.wikipedia.org/wiki/Minimum_spanning_tree

Reading

Mandatory reading.

1. ***Priority queues***
2. ***Minimum spanning tree***