# TCOM 501:
# Networking Theory & Fundamentals

Lecture 10

April 2003

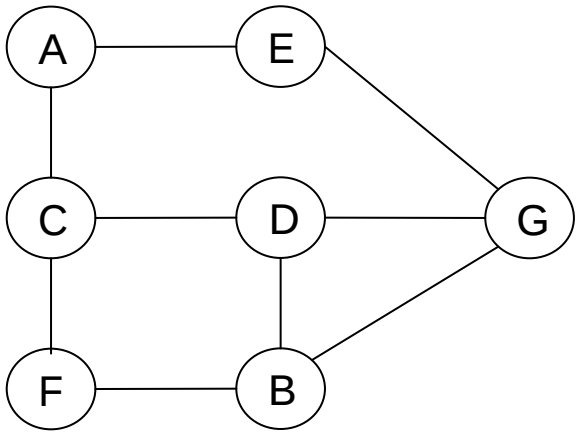
Prof. Yannis A. Korilis

# Topics

- Routing in Data Network
- Graph Representation of a Network
- Undirected Graphs
- Spanning Trees and Minimum Weight Spanning Trees
- Directed Graphs
- Shortest-Path Algorithms:
  - Bellman-Ford
  - Dijkstra
  - Floyd-Warshall
- Distributed Asynchronous Bellman-Ford Algorithm

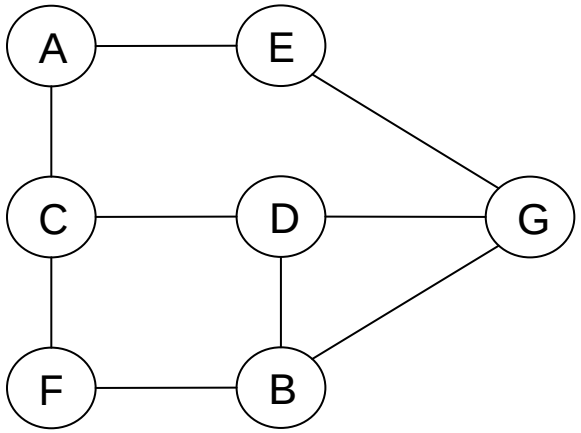
# Introduction to Routing

- What is routing?
  - The creation of (state) information in the network to enable *efficient delivery* of packets to their intended destination
- Two main components
  - Information acquisition: Topology, addresses
  - Information usage: Computing "good" paths to all destinations
- Questions
  - Where is B?
  - How to reach B?
  - How to *best* reach B?
  - How to best distribute all traffic (not only A to B)?



# Graph-Theoretic Concepts

- An Undirected Graph $G = (\mathcal{N}, \mathcal{A})$ consists of:
  - a finite nonempty set of nodes $\mathcal{N}$ and
  - a collection of "arcs" $\mathcal{A}$, interconnecting pairs of distinct nodes from $\mathcal{N}$.
- If $i$ and $j$ are nodes in $\mathcal{N}$ and $(i, j)$ is an arc in $\mathcal{A}$, the arc is said to be incident on $i$ and $j$
- Walk: sequence of nodes $(n_1, n_2, \ldots, n_k)$, where $(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$ are arcs
- Path: a walk with no repeated nodes
- Cycle: a walk $(n_1, n_2, \ldots, n_k)$, with $n_1 = n_k$ and no other repeated nodes
- Connected Graph: for all $i, j \in \mathcal{N}$, there exists a path $(n_1, n_2, \ldots, n_k)$, with $i = n_1, j = n_k$

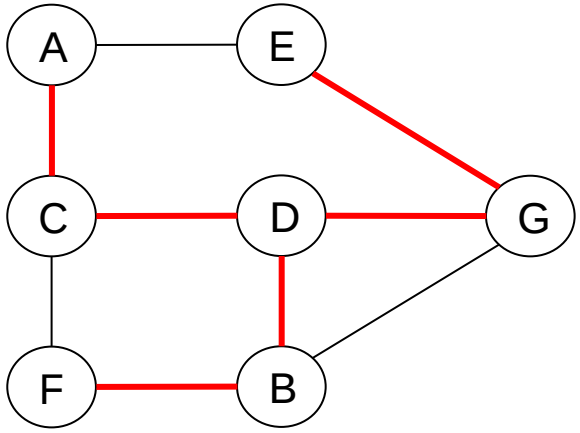



$$\mathcal{N} = \{A, B, C, D, E, F, G\}$$

$$\mathcal{A} = \{(A,E),(A,C),(C,D),(C,F),$$
$$(B,D),(B,G),(E,g)\}$$

# Spanning Tree

- A graph $G' = (\mathcal{N}', \mathcal{A}')$, with $\mathcal{N}' \subseteq \mathcal{N}$ and $\mathcal{A}' \subseteq \mathcal{A}$ is called a subgraph of $G$
- Tree: a connected graph that contains no cycles
- Spanning tree of a graph $G$: a subgraph of $G$, that contains all nodes of $G$, that is $\mathcal{N}' \subseteq \mathcal{N}$




- Lemma: Let G be a connected graph $G = (\mathcal{N}, \mathcal{A})$ and $S$ a nonempty strict subset of the set of nodes $\mathcal{N}$. Then, there exists at least one arc $(i, j)$ such that $i \in S$, and $j \notin S$.

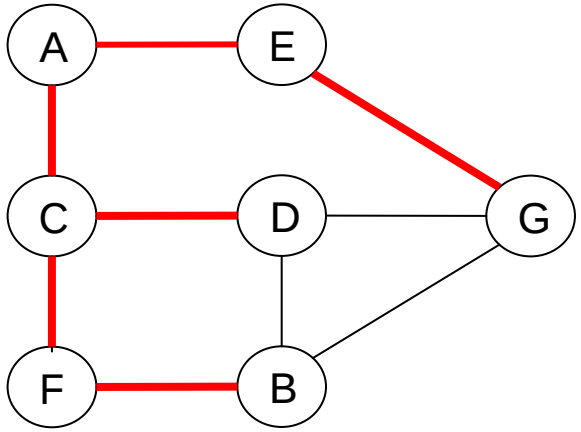
# Spanning Tree Algorithm

1. Select arbitrary node $n \in \mathcal{N}$, and initialize: $G' = (\mathcal{N}', \mathcal{A}')$,
$$\mathcal{N}' = \{n\}, \quad \mathcal{A}' = \varnothing$$
2. If $\mathcal{N}' = \mathcal{N}$, STOP: $G' = (\mathcal{N}', \mathcal{A}')$ is a spanning tree
ELSE: go to step 3
3. Let $(i, j) \in \mathcal{A}$ with $i \in \mathcal{N}'$ and $j \in \mathcal{N} - \mathcal{N}'$
   - Update:
   $\mathcal{N}' := \mathcal{N}' \cup \{j\}, \quad \mathcal{A}' := \mathcal{A}' \cup \{(i, j)\}$
   - Go to step 2

- Proof of correctness: Use induction to establish that after a new node i is added, $G$ remains connected and does not contain any cycles – therefore, it is a tree. After N-1 iterations

# Construction of a Spanning Tree

- $V' = \{A\}$; $E' = \varnothing$
- $V' = \{A,E\}$; $E' = \{(A,E)\}$
- $V' = \{A,E,C\}$;
  $E' = \{(A,E),(A,C)\}$
- $V' = \{A,E,C,D\}$;
  $E' = \{(A,E),(A,C),(CD)\}$
- $V' = \{A,E,C,D,F\}$;
  $E' = \{(A,E),(A,C),(CD),(CF)\}$
- $V' = \{A,E,C,D,F,B\}$;
  $E' = \{(A,E),(A,C),(CD),(CF),(F,B)\}$
- $V' = \{A,E,C,D,F,B,G\}$;
  $E' = \{(A,E),(A,C),(CD),(CF),(F,B),(E,G)\}$



# Spanning Trees

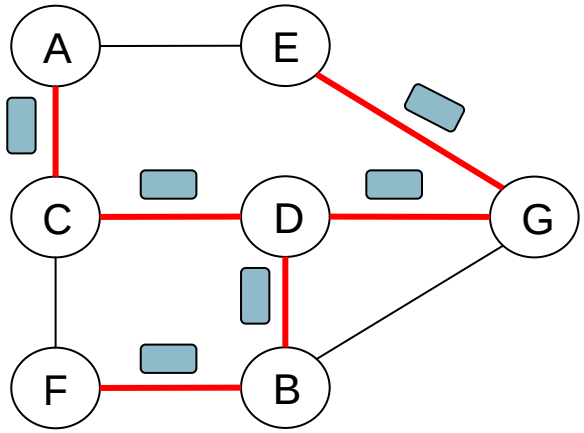
- **Proposition:** Let G be a connected graph with N nodes and A links

1. G contains a spanning tree
2. $A \geq N-1$
3. G is a tree if and only if A=N-1

- **Proof:** The spanning tree construction algorithm starts with a single node and at each iteration augments the tree by one node and one arc. Therefore, the tree is constructed after N-1 iterations and has N-1 links. Evidently $A \geq N-1$.
  - If A=N-1, the spanning tree includes all arcs of G, thus G is a tree.
  - If A>N-1, there is a link (i, j) that does not belong to the tree. Considering the path connecting nodes i and j along the spanning tree, that path and link (i, j) form a cycle, thus G is not a tree.

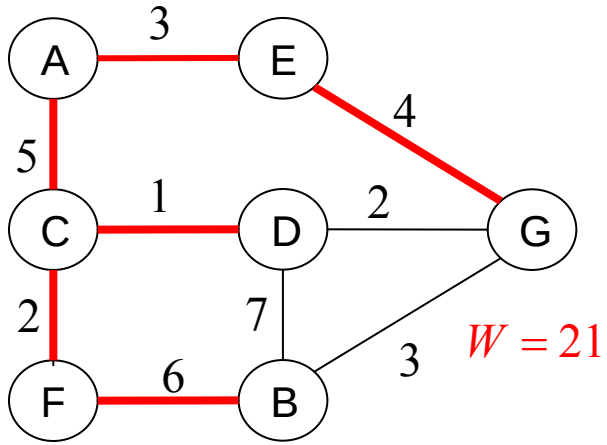
# Use of Spanning Trees

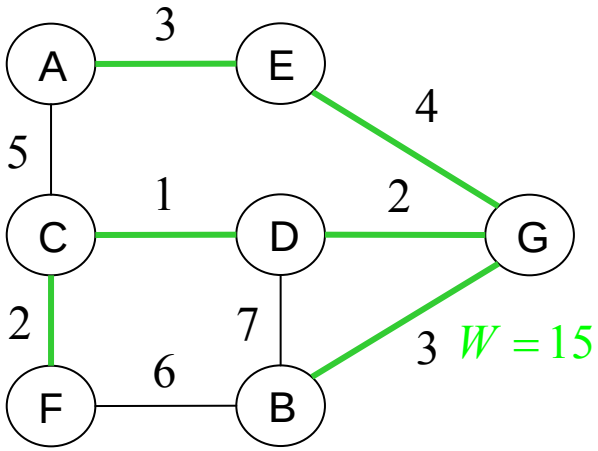
- Problem: how to distribute information to *all* nodes in a graph (network) – e.g., address and topology information
- Flooding: each node forwards information to all its neighbors. Simple, but multiple copies of the same packet traverse each link and are received by each node
- Spanning tree: nodes forward information only along links that belong to the spanning tree. More efficient:
  - Information reaches each node only once and traverses links at most once
  - Note that spanning tree is *bidirectional*



# Minimum Weight Spanning Tree

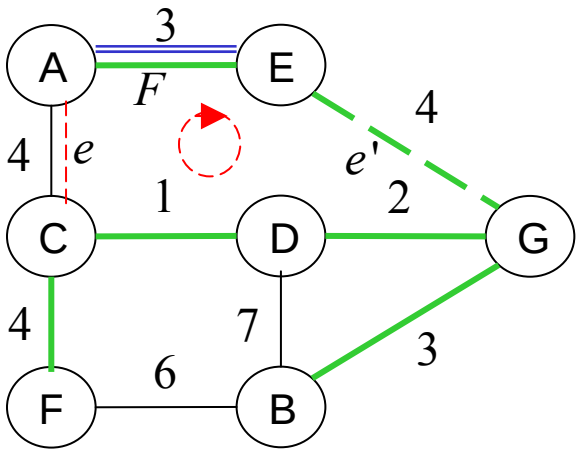
- Weight $w_{ij}$ is used to quantify the "cost" for using link $(i, j)$
- Examples: delay, offered load, distance, etc.
- Weight (cost) of a tree is the sum of the weights of all its links – packets traverse all tree links once
- Definition: *A Minimum Weight Spanning Tree (MST)* is a spanning tree with minimum sum of link weights
- Definition: A subtree of an MST is called a *fragment*. An arc with one node in a fragment and the other node not in this fragment is called an *outgoing arc* from the fragment.



# Minimum Spanning Trees

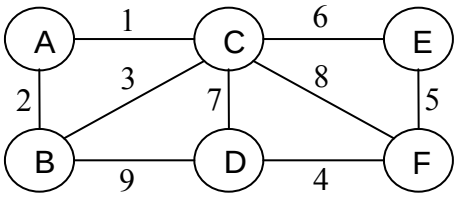
- Lemma: Given a fragment $F$, let $e=(i, j)$ be an outgoing arc with minimum weight, where $j \notin F$. Then $F$ extended by arc $e$ and node $j$ is a fragment.
- Proof:
  - Let $T$ be the MST that includes fragment $F$. If $e \in T$, we are done.
  - Assume $e \notin T$: then a cycle is formed by $e$ and the arcs of $T$
    - Since $j \notin F$. there is an arc $e' \neq e$ which belongs to the cycle and $T$, and is outgoing from $F$.
    - Let $T'=(T-\{e'\}) \cup \{e\}$. This is subgraph with N-1 arcs and no cycles, i.e., a spanning tree.
    - Since $w_e \leq w_{e'}$, the weight of $T'$ is less than or equal to the weight of $T$
    - Then $T'$ is an MST, and $F$ extended by arc $e$ and node $j$ is a fragment.

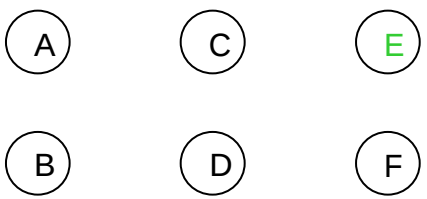


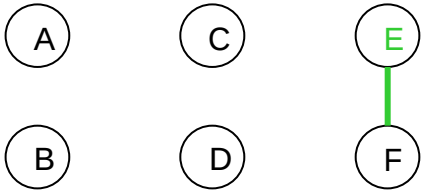
# Minimum Weight Spanning Tree Algorithms

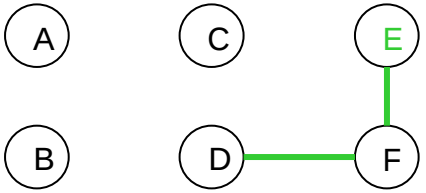
- Inductive algorithms based on the previous lemma
- Prim's Algorithm:
  - Start with an arbitrary node as the initial fragmant
  - Enlarge current fragment by successively adding minimum weight outgoing arcs
- Kruskal's Algorithm:
  - All vertices are initial fragments
  - Successively combine fragments using minimum weight arcs that do not introduce a cycle

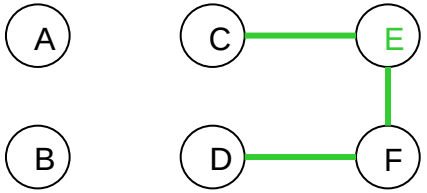
# Prim's Algorithm

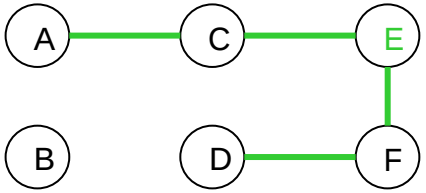


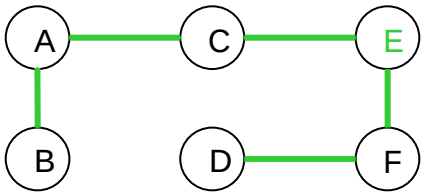



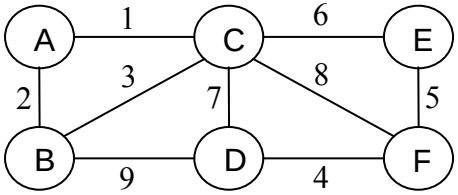



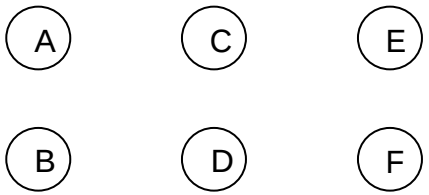



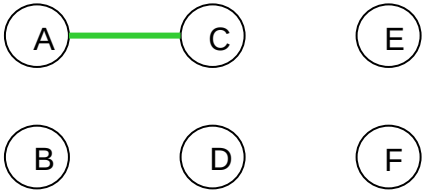



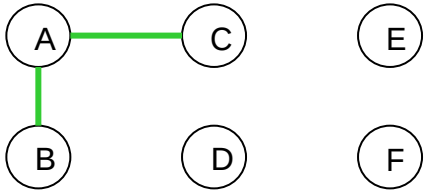



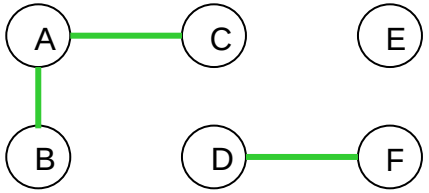
# Kruskall's Algorithm

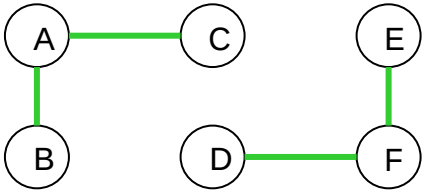


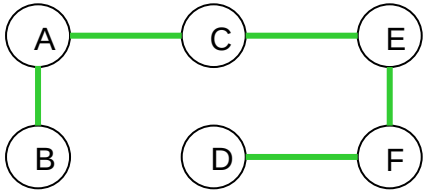



# Shortest Path Algorithms

- **Problem:** Given nodes A and B, find the "best" route for sending traffic from A to B
- "Best:" the one with minimum cost – where typically the cost of a path is equal to the sum of the costs of the links in the path
- Important problem in various networking applications
- Routing of traffic over network links $\Rightarrow$ need to distinguish direction of flow
- Appropriate network model: Directed Graph

# Directed Graphs

- A Directed Graph – or *Digraph* – $G = (\mathcal{N}, \mathcal{A})$ consists of:
  - a finite nonempty set of nodes $\mathcal{N}$ and
  - a collection of "directed arcs" $\mathcal{A}$, i.e., ordered pairs of distinct nodes from $\mathcal{N}$.
- Directed walks, directed paths and directed cycles can be defined to extend the corresponding definitions for undirected graphs
- Given a digraph $G = (\mathcal{N}, \mathcal{A})$, there is an associated undirected graph $G' = (\mathcal{N}', \mathcal{A}')$, with $\mathcal{N}' = \mathcal{N}$ and $(i, j) \in \mathcal{A}'$ if $(i, j) \in \mathcal{A}$, or $(j, i) \in \mathcal{A}$
- A digraph $G = (\mathcal{N}, \mathcal{A})$ is called connected if its associated undirected graph $G' = (\mathcal{N}', \mathcal{A}')$ is connected
- A digraph $G = (\mathcal{N}, \mathcal{A})$ is called *strongly connected* if for every $i, j \in \mathcal{N}$, there exists a *directed* path $(n_1, n_2, \ldots, n_k)$, with $i = n_1, j = n_k$

# Shortest Path Algorithms:Problem Formulation

- Consider an $N$ vertex graph $G = (V,E)$ with link lengths $d_{ij}$ for edge $(i,j)$ ($d_{ij} = \infty$ if $(i,j) \notin E$)
- Problem: Find minimum distance paths from all vertices in $V$ to vertex 1
  - Alternatively, find minimum weight paths from vertex 1 to all vertices in $V$
- General approach is again *iterative*

$$D_i^{(n+1)} = \min\{D_j^{(n)} + d_{ij}\}$$

  - Differences are in how the iterations proceed
  - Three main algorithms

# Bellman-Ford Algorithm

- Iterative step is over increasing *hop* count
- Define $D_i^h$ as a shortest walk from node $i$ to 1 that contains at most $h$ edges
  - $D_1^h = 0$ by definition for all $h$
- Bellman-Ford Algorithm
  - Define $D_i^{h+1} = \min_j \{D_j^h + d_{ij}\}, \quad \forall i \neq 1$
  - Set initial conditions to $D_i^0 = \infty$ for $i \neq 1$
  - a The scalars $D_i^h$ are equal to the shortest walk lengths with less than $h$ hops from node $i$ to node 1
  - b The algorithm terminates after at most $N$ iterations if there are no negative length cycles without node 1, and it yields the shortest path lengths to node 1

# Proof of Bellman-Ford (1)

- Proof is by induction on hop count $h$
  - For $h = 1$ we have
    - $D_i^1 = d_{i1}$ for all $i \neq 1$, so the result holds for $i = 1$
  - Assume that the result holds for all $k \leq h$. We need to show it still holds for $h+1$
- Two cases need to be considered

1 The shortest ($\leq h+1$) walk from $i$ to 1 has $\leq h$ edges
  - Its length is then $D_i^h$

2 The shortest ($\leq h+1$) walk from $i$ to 1 has ($h+1$) edges
  - It consists of the concatenation of an edge $(i,j)$ followed by a shortest $h$ hop walk from vertex $j$ to vertex 1
  - This second case is the one we will focus on

# Proof of Bellman-Ford (2)

- From Case 1 and Case 2, we have

$$\text{shortest } (\le h+1) \text{ walk length} = \min\left\{D_i^h, \min_{j \ne i}\left[D_j^h + d_{ij}\right]\right\}$$

- From induction hypothesis and initial conditions
  - $D_i^k \le D_i^{k-1}$ for all $k \le h$ so that

$$D_i^{h+1} = \min_j\left[D_j^h + d_{ij}\right] \le \min_j\left[D_j^{h-1} + d_{ij}\right] = D_i^h$$

  - $D_i^h \le D_i^1 = d_{i1} = d_{i1} + D_1^h$
- Combining the two gives

$$\text{shortest } (\le h+1) \text{ walk length} = \min\left\{D_i^h, \min_j\left[D_j^h + d_{ij}\right]\right\}$$

$$= \min\left\{D_i^h, D_i^{h+1}\right\} = D_i^{h+1}$$

which completes the proof of part a

# Proof of Bellman-Ford (3)

- Assume that the algorithm terminates after $h$ steps
  - This implies that $D_i^k = D_i^h$ for all $i$ and $k \geq h$
  - This means that adding more edges cannot decrease the length of any of those walks
  - Hence, there cannot be negative length cycles that do not include node 1
    - They could be repeated many times to make walk lengths arbitrarily small
  - Delete all such cycles
    - This yields paths of less than or equal length
    - So for all $i$ we have a path of $h$ hops or less to vertex 1 and with length $D_i^h$
    - Since those paths have no cycles they contain at most $N$-1 edges,and therefore $D_i^N = D_i^{N-1}$ for all $i$
- The algorithm terminates after at most $N$ steps, which proves part b

# Bellman-Ford Complexity

- Worst-case amount of computation to find shortest path lengths
  - Algorithm iterates up to $N$ times
  - Each iteration is done for $N$-1 nodes (all $i \neq 1$)
  - Minimization step requires considering up to $N$-1 alternatives

  $\Rightarrow$ Computational complexity is $O(N^3)$
  - More careful accounting yields a computational complexity of $O(hM)$

# Constructing Shortest Paths

- B-F algorithm yields shortest path *lengths*, but we are also interested in *actual* paths
- Start with B-F equation

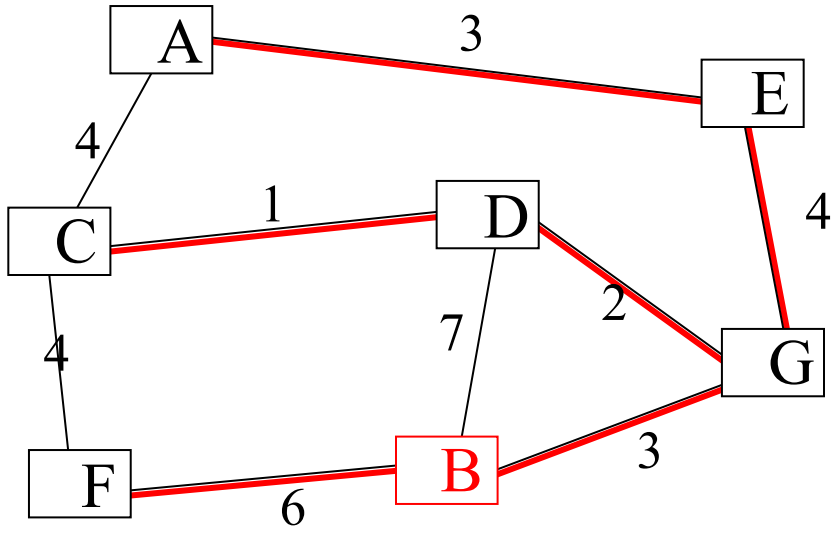
$$D_i = \min_{j \in G}\left[D_j + d_{ij}\right], \quad \forall i \neq 1, \quad \text{and} \quad D_1 = 0$$

- For all vertices $i \neq 1$ pick the edge $(i,j)$ that minimizes B-F equation
  - This generates a subgraph with $N$-1 edges (a tree)
  - For any vertex $i$ follow the edges from vertex $i$ along this subgraph until vertex 1 is reached
- Note: In graphs without zero or negative length cycles, B-F equation defines a system of $N$-1 equations with a *unique* solution

# Constructing Shortest Paths

- $D_A = 3 + D_E$
- $D_E = 4 + D_G$
- $D_G = 3 + D_B = 3$
- $D_D = 2 + D_G$
- $D_C = 1 + D_D$
- $D_F = 6 + D_B = 6$

- $AB =$ *A-E-G-B*



# Dijkstra's Algorithm (1)

- Different iteration criteria
  - Algorithm proceeds by increasing *path length* instead of increasing *hop count*
    - Start with "closest" vertex to destination, use it to find the next closest vertex, and so on
  - Successful iteration requires *non-negative* edge weights
- Dijkstra's algorithm maintains two sets of vertices
  - *L*: Labeled vertices (shortest path is known)
  - *C*: Candidate vertices (shortest path not yet known)
  - One vertex is moved from *C* to *L* in each iteration

# Dijkstra's Algorithm (2)

- Initialization
  - $L = \{1\}$ and $C = G - L$; (vertex 1 is the destination)
  - $D_1 = 0$ and $D_j = d_{j1}$ for $j \neq 1$
- Iteration steps

  1 Find the next *closest* vertex not in $L$, i.e.,
  - Find the vertex $i \notin L$ such that $D_i = \min_{j \notin L} D_j$
  - Update $C$ and $L$: $L = L \cup \{i\}$ and $C = C - \{i\}$

  2 Update lengths of paths of vertices remaining in $C$

$$D_j := \min[D_j, D_i + d_{ji}], \quad \forall j \in C$$

- Algorithm stops when $L = G$

# Proof of Dijkstra's Algorithm (1)

- At the beginning of each step 1, we have
  - a $D_i \le D_j$ for all $i \in L$ and $j \notin L$
  - b For any vertex $j$, $D_j$ is the shortest distance from $j$ to 1 for any path using vertices (except possibly $j$) only in $L$
- Proof of condition (a)
  - (a) is satisfied initially
  - We have $d_{ji} \ge 0$ and $D_i = \min_{j \notin L} D_j$ so that condition (a) is preserved by the update equation $D_j := \min[D_j, D_i + d_{ji}], \quad \forall j \in C$

non-negative edge weights

# Proof of Dijkstra's Algorithm (2)

- It remains to prove condition (b), which we proceed to do by induction
  - (b) is satisfied initially
  - Induction hypothesis (H)
    - Condition (b) holds at the beginning of some step 1, where $i$ is the vertex added to $L$ at that step
  - From (H), condition (b) holds for vertex $i$ as well as $\forall\, j \in L$ from (a)
  - Consider next a vertex $j \notin L \cup \{i\}$

# Proof of Dijkstra's Algorithm (3)

- Let $D'_j$ be the length of the shortest path from $j \notin L \cup \{i\}$ to 1 with all its vertices but $j$ in $L \cup \{i\}$
  - This path consists of an edge $(j,k)$, $k \in L \cup \{i\}$, concatenated with the shortest path from $k$ to 1 with vertices in $L \cup \{i\}$
    - From (H), $D_k$ is the length of this shortest path from $k$ to 1
  - We have
$$D'_j = \min_{k \in L \cup \{i\}} [D_k + d_{jk}] = \min \left[ \min_{k \in L} [D_k + d_{jk}], D_i + d_{ji} \right]$$
  - And from (H) we also have $D_j = \min_{k \in L} [D_k + d_{jk}]$
- Combining the two gives $D'_j = \min[D_j, D_i + d_{ji}] = D_j$
- Which completes the proof of the induction

# Computational Complexity of Dijkstra's Algorithm

- As with Bellman-Ford we have up to $N$ iterations, where $N = |G|$
- The major saving is that in each iteration we consider each node only *once*
  - Combination of step 1 and step 2 looks first at nodes in $L$ (step 1) and then $C = G\text{-}L$ (step 2)
  - Operations in each step have unit cost
- At most $N$ iterations and $N$ unit cost operations per iteration
  - Total cost of $O(N^2)$ (can be improved)

# Floyd-Warshall Algorithm (1)

- Targeted at computing shortest paths between *all pairs* of vertices
  - Can accommodate positive and negative edge weights but requires no negative length cycles
- Iteration step is on *set of vertices* allowed as intermediate vertices in a path
  - Initial condition is as for Bellman-Ford and Dijkstra, i.e., single edge paths for all vertices
  - Next iteration allows only vertex 1 as intermediate vertex
  - Algorithm stops when all vertices can be used as intermediate vertices

# Floyd-Warshall Algorithm (2)

- Initial conditions
  - $D_{ij}^{0} = d_{ij}, \quad \forall i, j \quad \text{and} \quad i \neq j$
- Let $D_{ij}^{n}$ be the shortest path length between vertices $i$ and $j$ when intermediate vertices on the path are limited to vertices 1,2,...,$n$
- Iteration step is defined as

$$D_{ij}^{n+1} = \min[D_{ij}^{n}, D_{i\,(n+1)}^{n} + D_{(n+1)\,j}^{n}], \quad \forall i \neq j$$

  - Check if distance between vertices $i$ and $j$ is improved by using vertex ($n$+1) as an intermediate vertex

# Proof of Floyd-Warshall

- Again by induction
- For $n$=0 the initial conditions clearly give the shortest paths without intermediate vertices
- Induction hypothesis (H)
  - For a given $n$, $D_{ij}^{n}$ gives the shortest path length from $i$ to $j$ with only vertices 1 to $n$ as intermediate vertices
- Shortest path from $i$ to $j$ with only vertices 1 to ($n$+1) as intermediate vertices either contain vertex ($n$+1) or does not
  - If it does, the length of the path is the 2nd term of the iteration step
  - If it does not, the length of the path is the 1st term of the iteration step

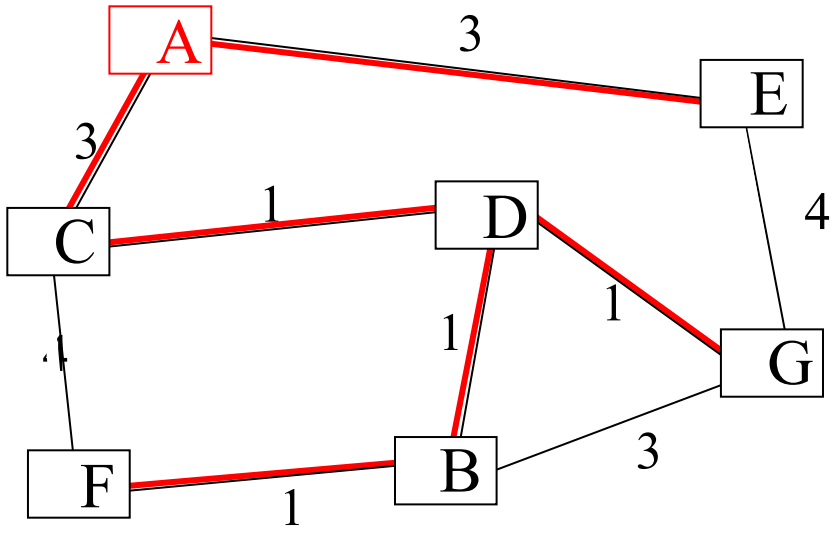
# Complexity of Floyd-Warshall

- $N$ iterations steps
  - One for each possible set of intermediate vertices
- Computations in each iteration step involve a comparison for each pair of vertices
  - There are $N(N\text{-}1)$ of them
- Total cost is $O(N^3)$
  - Equivalent to running $N$ instances of Dijkstra's algorithm
    - Each instance of Dijkstra gives shortest paths from all sources to one destination

# Source Based Shortest Paths

- Only difference is that we want paths from a source node to all destinations instead of all sources to a destination
- Basic approach remains the same
  - Minor differences in defining initialization conditions and iterative steps
  - We will consider the case of both Bellman-Ford and Dijkstra's algorithm

# Source Based Bellman-Ford




- Source is vertex A
- Initialization
  - $D_A{}^h = 0, \forall h$
  - $D_i^0 = \infty$ for $i \neq A$
- Define

$$D_i^{h+1} = \min_j \{D_j^h + d_{ji}\}, \quad \forall i \neq A$$

distance from node A to node $i$ in at most $h$+1 hops

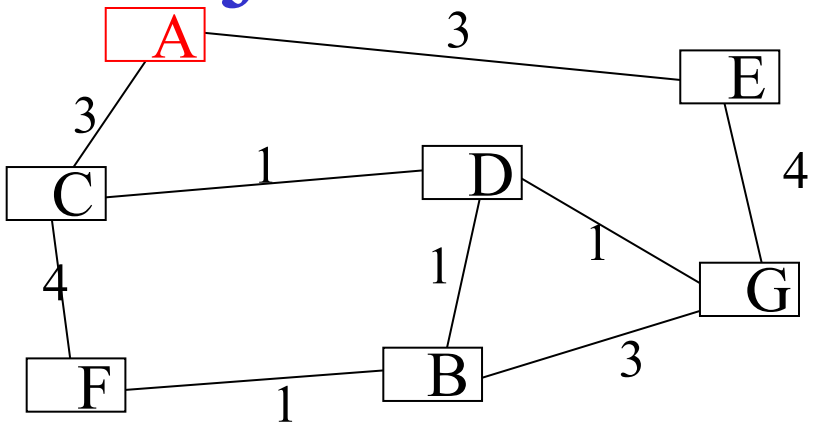
- As before, iterate until no more progress

- $D^1{}_E = 3, D^1{}_C = 3$
- $D^2{}_E = 3, D^2{}_C = 3, D^2{}_D = 4, D^2{}_G = 7, D^2{}_F = 7$
- $D^3{}_E = 3, D^3{}_C = 3, D^3{}_D = 4, D^3{}_G = 5, D^3{}_F = 7, D^3{}_B = 5$
- $D^4{}_E = 3, D^4{}_C = 3, D^4{}_D = 4, D^4{}_G = 5, D^4{}_F = 6, D^4{}_B = 5$

# Source Based Dijkstra

- Initialization
  - $L = \{A\}$ and $C = G - L$; (vertex A is the source)
  - $D_A = 0$ and $D_i = d_{Aj}$ for $j \neq A$
- Iteration steps
  1. Find next *closest* vertex in $C$
     - Find the vertex $i \notin L$ such that $D_i = \min_{j \notin L} D_j$
     - Update $C$ and $L$: $L = L \cup \{i\}$ and $C = C - \{i\}$
  2. Update lengths of paths of vertices remaining in $C$

$$D_j := \min[D_j, D_i + d_{ij}], \quad \forall j \in C$$




- {A}; $D_E$=3,$D_C$=3
- {A,E}; $D_E$=3; $D_C$=3,$D_G$=7
- {A,E,C}; $D_E$=3,$D_C$=3; $D_D$=4,$D_G$=7,$D_F$=7
- {A,E,C,D}; $D_E$=3,$D_C$=3,$D_D$=4; $D_G$=5,$D_F$=7,$D_B$=5
- {A,E,C,D,B}; $D_E$=3,$D_C$=3,$D_D$=4, $D_B$=5; $D_G$=5,$D_F$=6
- {A,E,C,D,B,G}; $D_E$=3,$D_C$=3, $D_D$=4,$D_B$=5,$D_G$=5; $D_F$=6
- {A,E,C,D,B,G,F}; $D_E$=3,$D_C$=3, $D_D$=4,$D_B$=5,$D_G$=5,$D_F$=6

# Tree of Shortest Paths

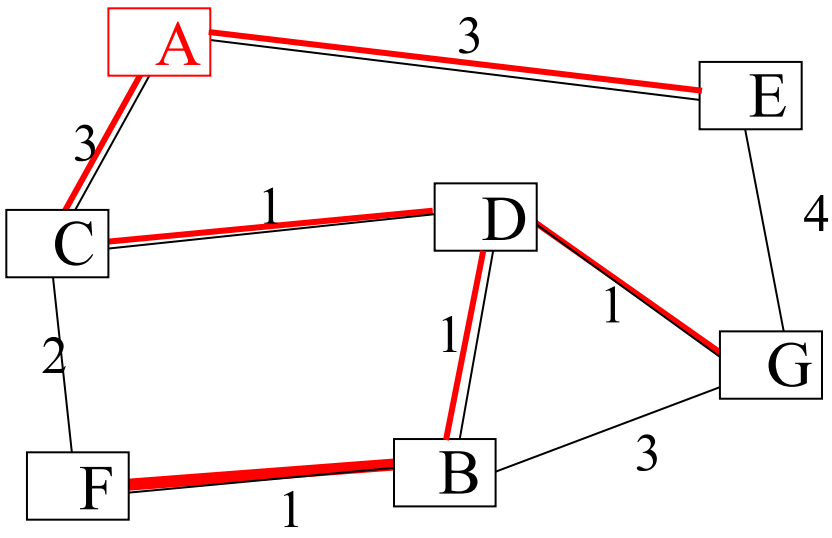
- For the additive distance function we have assumed
  - If the path $p=i_1,i_2,...,i_h$, with $i=i_1$ and $j= i_h$ is a shortest path from $i$ to $j$, then for every $k$, $0 \le k \le h-1$, the subpath $s=i_1,i_2,...,i_k$ is a shortest path from $i$ to $i_k$
    - If this did not hold, one could improve the shortest path between $i$ and $j$ by picking the shorter path that exists between $i$ and $i_k$
  - Let $D$ be the distance vector from vertex $i$, then a directed path $p$ from vertex $i$ to vertex $j$ is a shortest path iff $D_k = D_l + d_{kl}$ for all edges $(k,l) \in p$

$\Rightarrow$ We can construct a shortest path tree from vertex $i$ to all other vertices

# Shortest Path Tree vs Minimum Spanning Tree

- The two are **NOT** the same

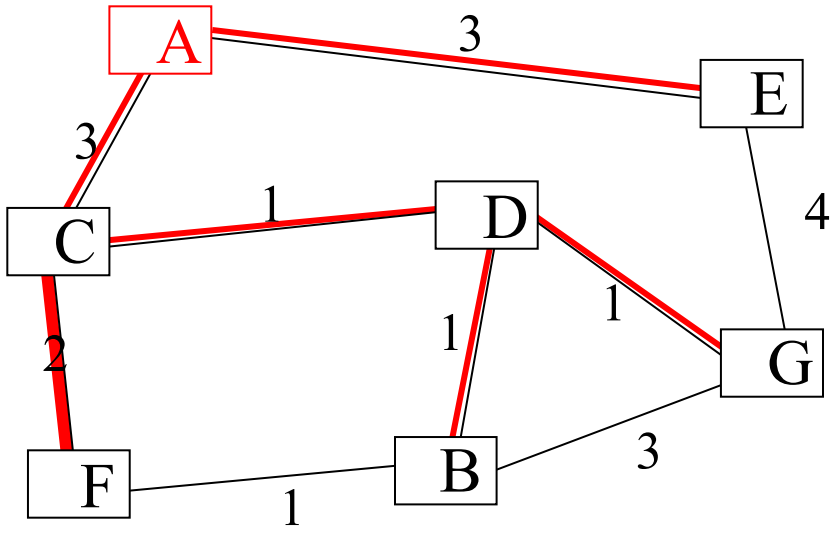
Minimum Spanning Tree




Total weight: 10

$D_{AF} = 6$

Shortest Path Tree




Total weight: 11

$D_{AF} = 5$