6.852: Distributed Algorithms
Fall, 2009

Class 3

Today's plan

- Algorithms in general synchronous networks (continued):
  - Shortest paths spanning tree
  - Minimum-weight spanning tree
  - Maximal independent set
- Reading: Sections 4.3-4.5

- Next:
  - Distributed consensus
  - Reading: Sections 5.1, 6.1-6.3

Last time

- Lower bound on number of messages for comparison-based leader election in a ring.
- Leader election in general synchronous networks:
  - Flooding algorithm
  - Reducing message complexity
  - Simulation relation proof
- Breadth-first search in general synchronous networks:
  - Marking algorithm
  - Applications:
    - Broadcast, convergecast
    - Data aggregation (computation in networks)
    - Leader election in unknown networks
    - Determining the diameter

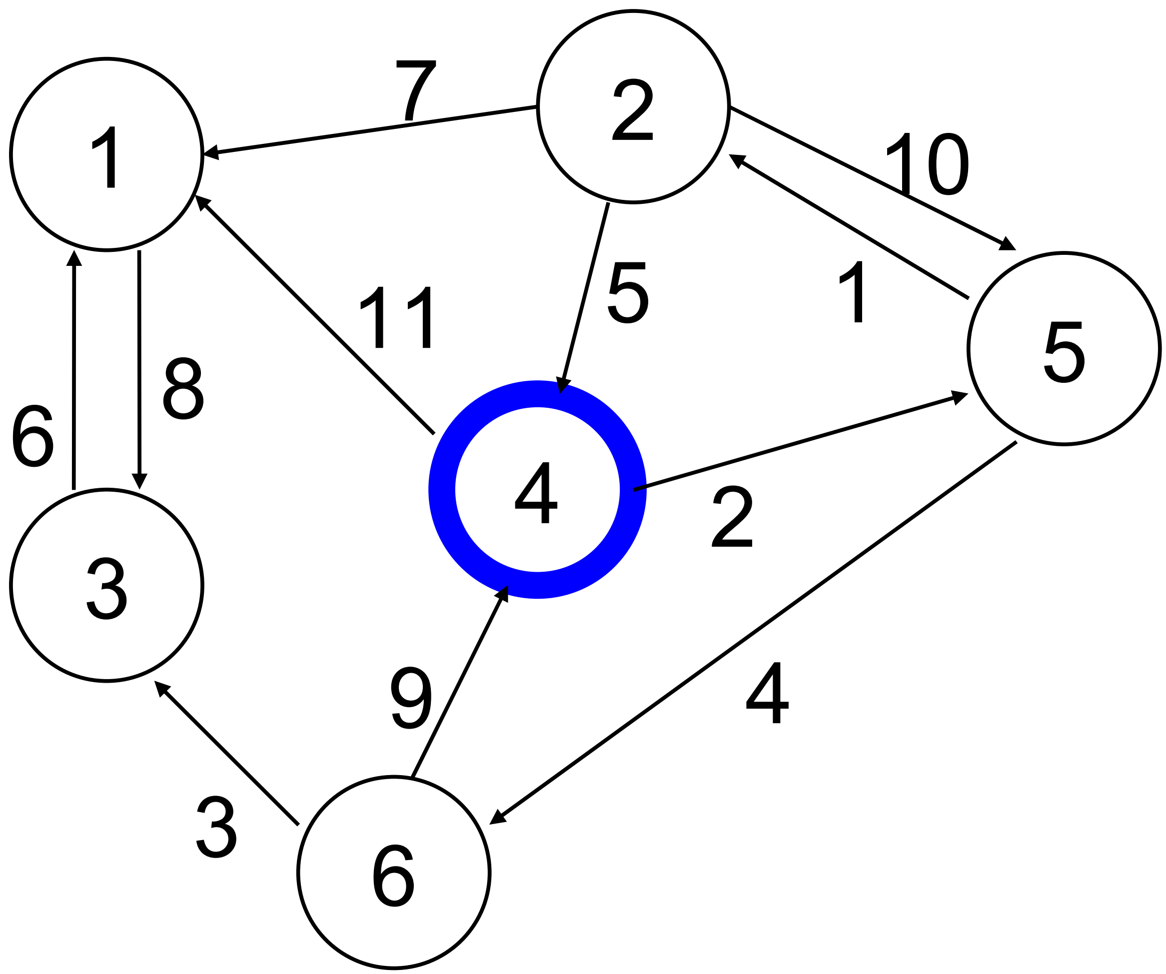
Termination for BFS

- Suppose i_0 wants to know when the BFS tree is completed.
- Assume each search message receives a response, parent or non-parent.
  - Easy if edges are bidirectional, harder if unidirectional.
- After a node has received responses to all its search messages, it knows who its children are, and knows they are all marked.
- Leaves of the tree discover who they are (receive all non-parent responses).
- Starting from the leaves, fan in complete messages to i_0.
- Node can send complete message after:
  - It has receives responses to all its search messages (so it knows who its children are), and
  - It has received complete messages from all its children.

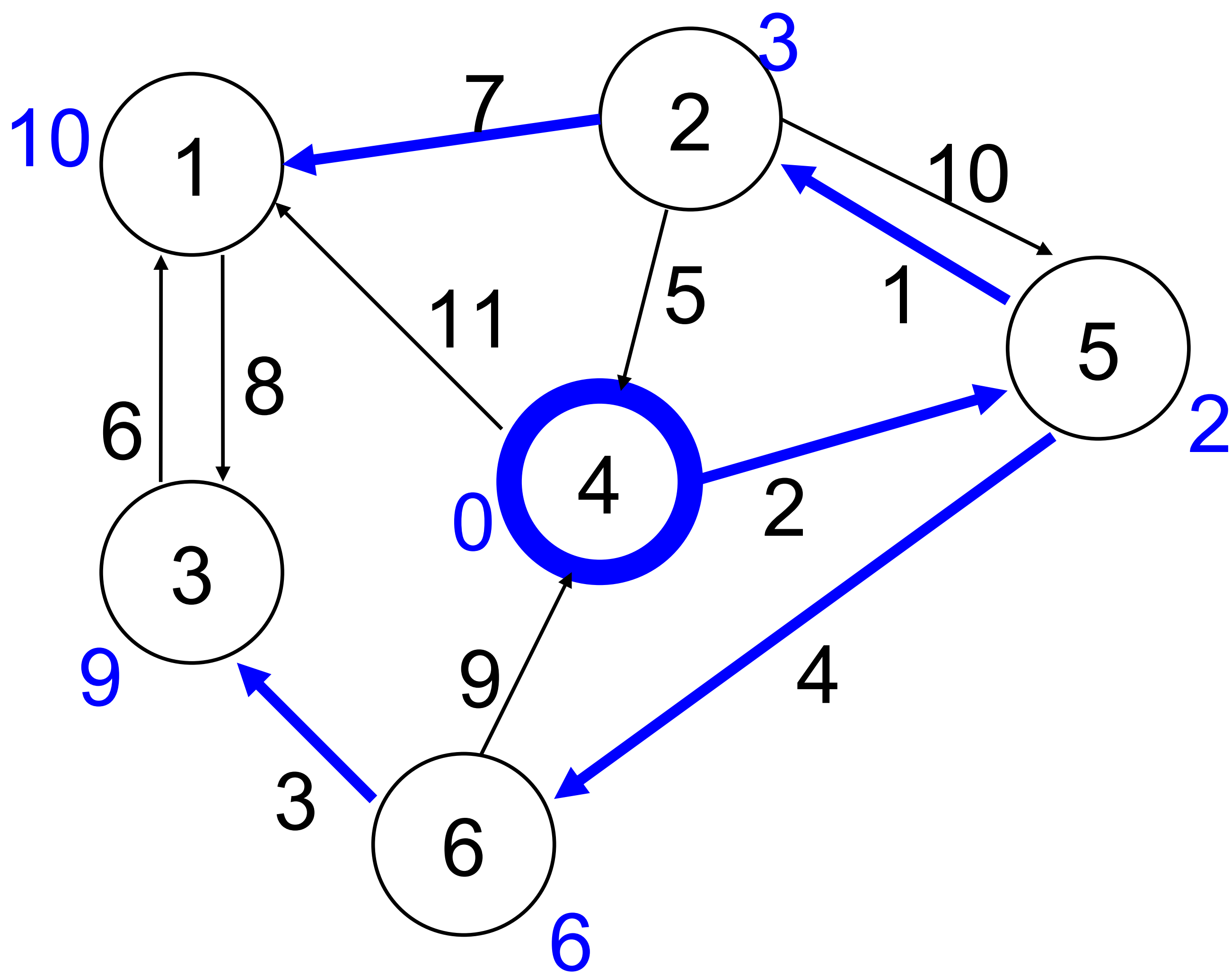
Shortest paths

- Motivation: Establish structure for efficient communication.
  - Generalization of Breadth-First Search.
  - Now edges have associated costs (weights).
- Assume:
  - Strongly connected digraph, root i_0.
  - Weights (nonnegative reals) on edges.
    - Weights represent some communication cost, e.g. latency.
  - UIDs.
  - Nodes know weights of incident edges.
  - Nodes know n (need for termination).
- Required:
  - Shortest-paths tree, giving shortest paths from i_0 to every other node.
  - Shortest path = path with minimum total weight.
  - Each node should output parent, “distance” from root (by weight).

Shortest paths



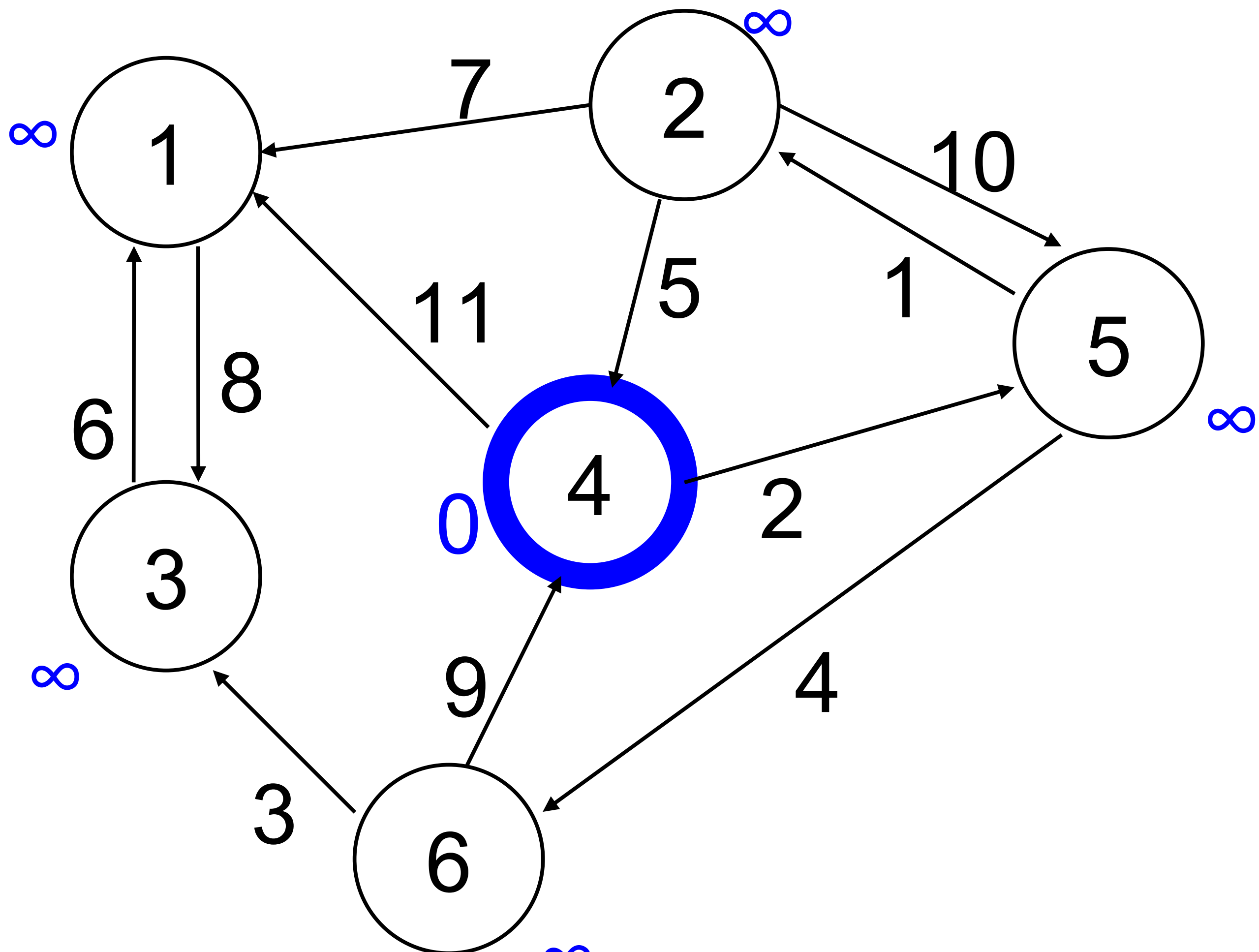
Shortest paths



Shortest paths algorithm

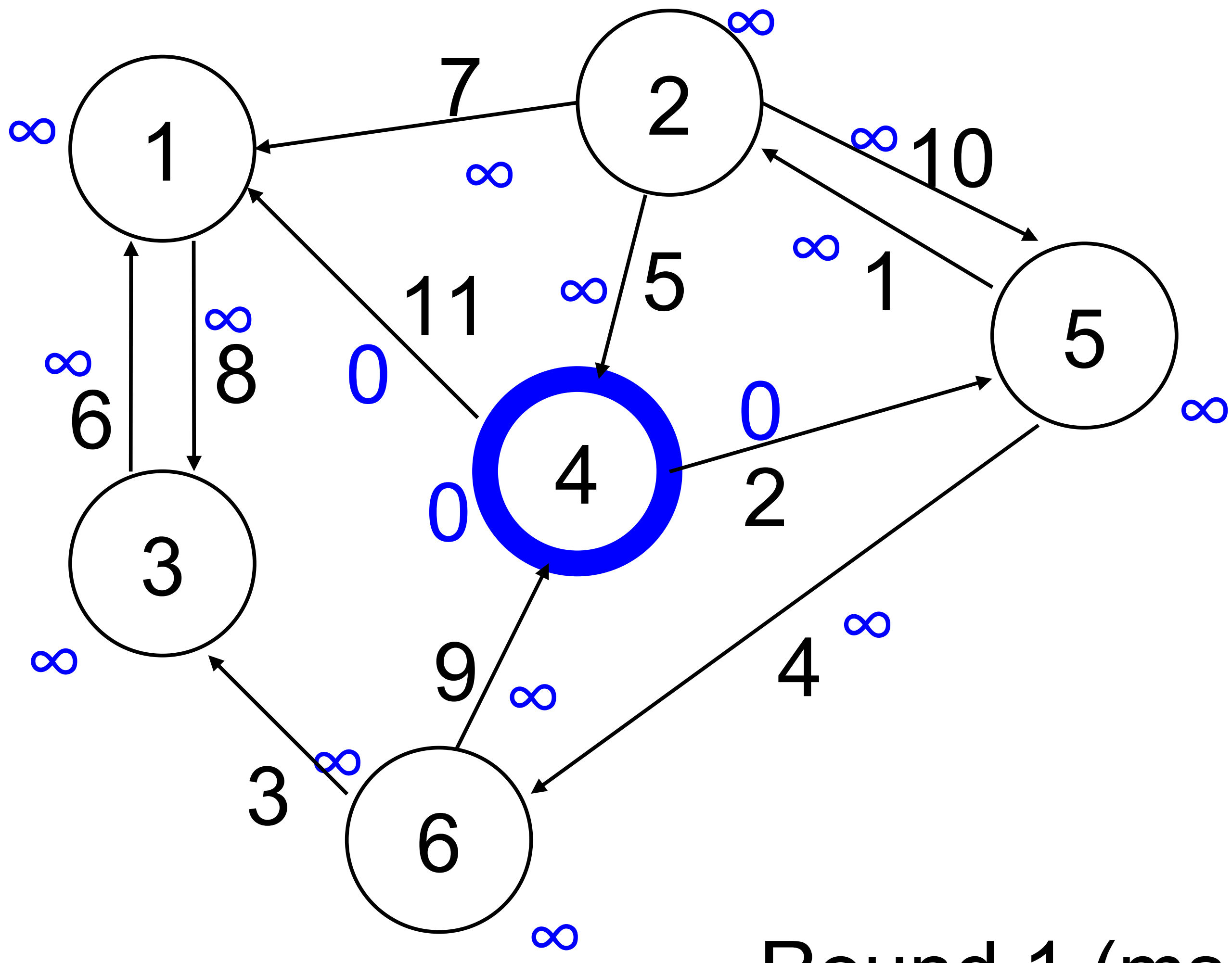
- Bellman-Ford (adapted from sequential algorithm)
- "Relaxation algorithm"
- Each node maintains:
  - dist, shortest distance it knows about so far, from i_0
  - parent, its parent in some path with total weight = dist
  - round number
- Initially i_0 has dist 0, all others ∞; parents all null
- At each round, each node:
  - Send dist to all out-nbrs
  - Relaxation step:
    - Compute new dist = min(dist, $\min_j(d_j + w_{ji})$).
    - Update parent if dist changes.
- Stop after n-1 rounds
- Then (claim) dist contains shortest distance, parent contains parent in a shortest-paths tree.

Shortest paths



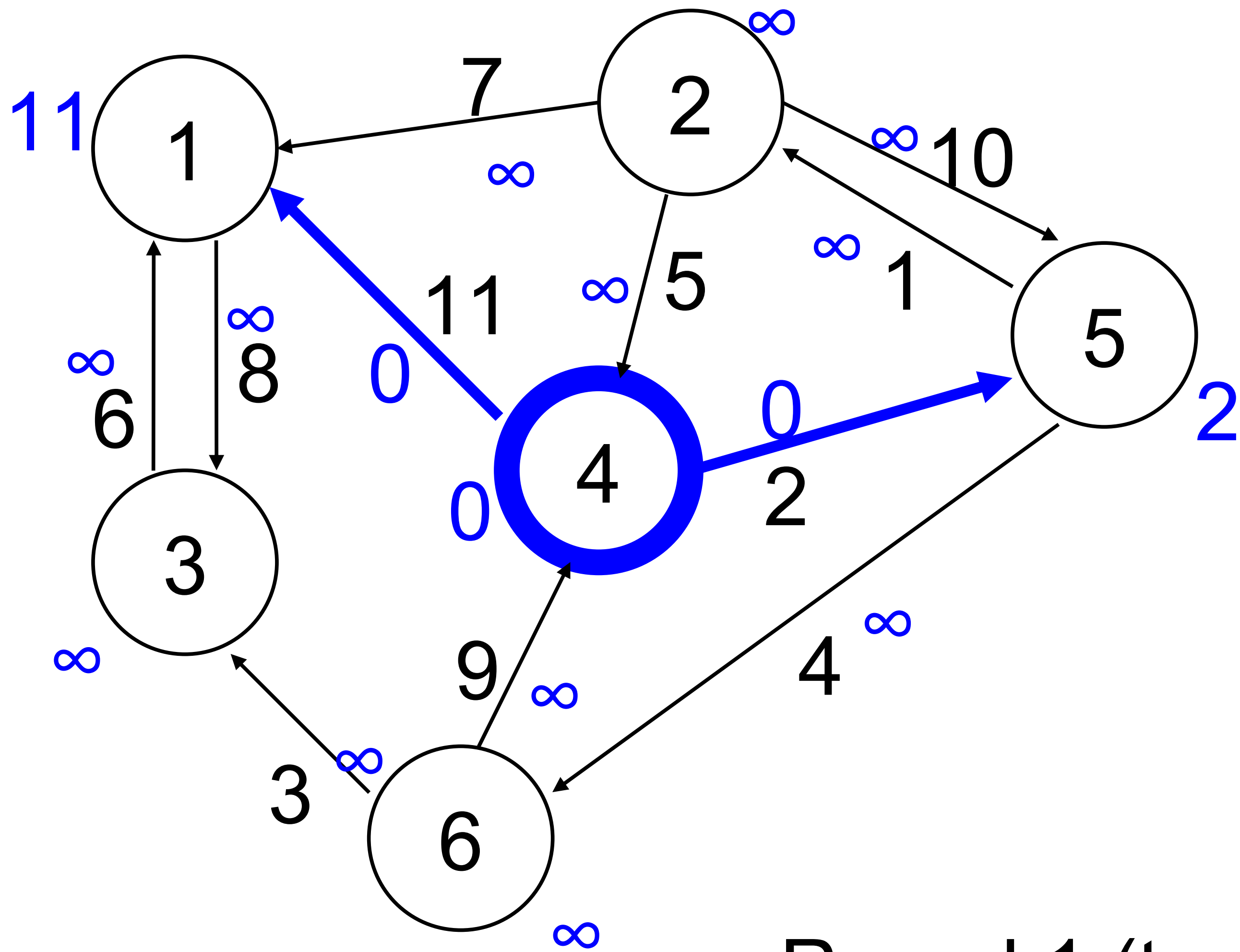
Round 1 (start)

Shortest paths



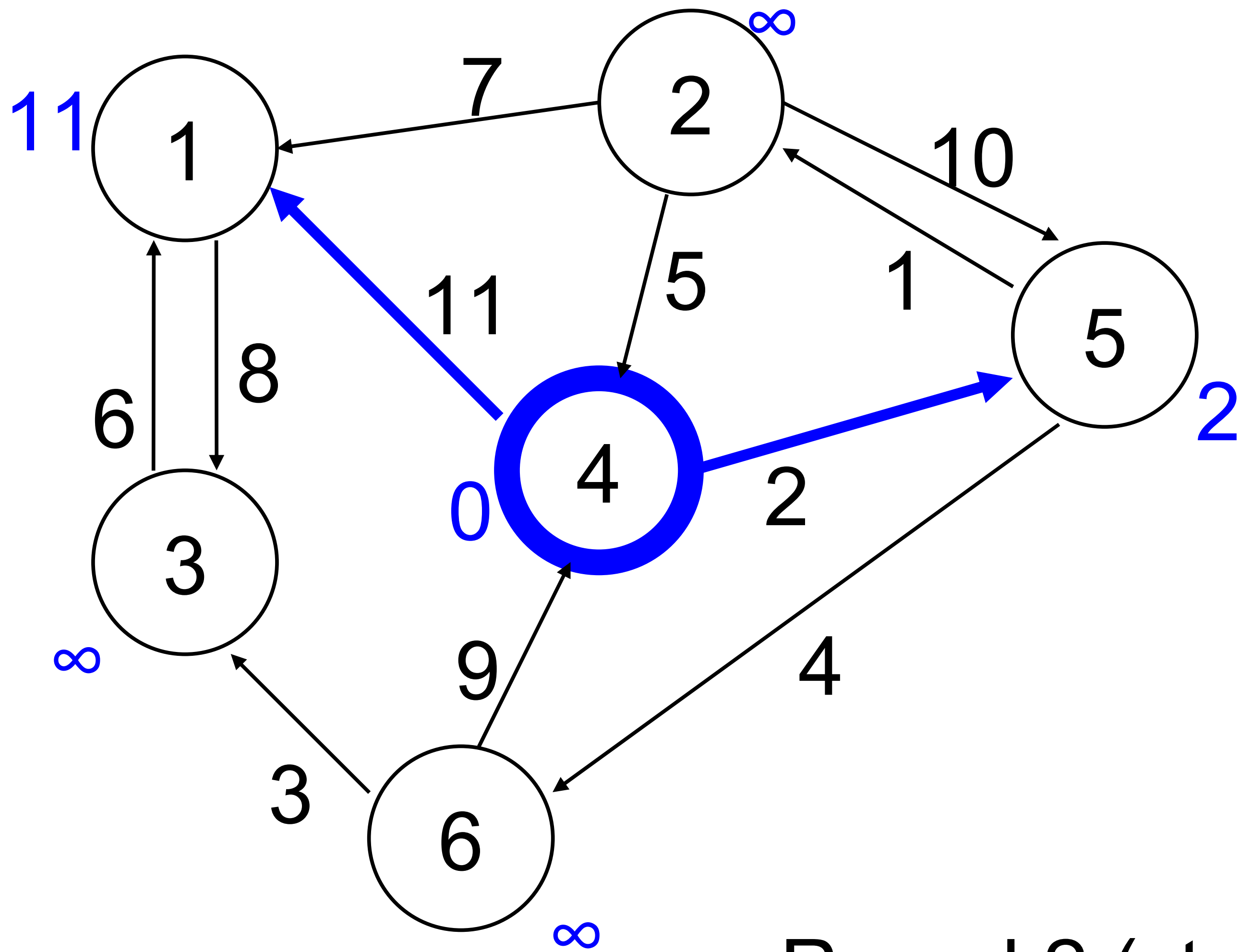
Round 1 (msgs)

Shortest paths



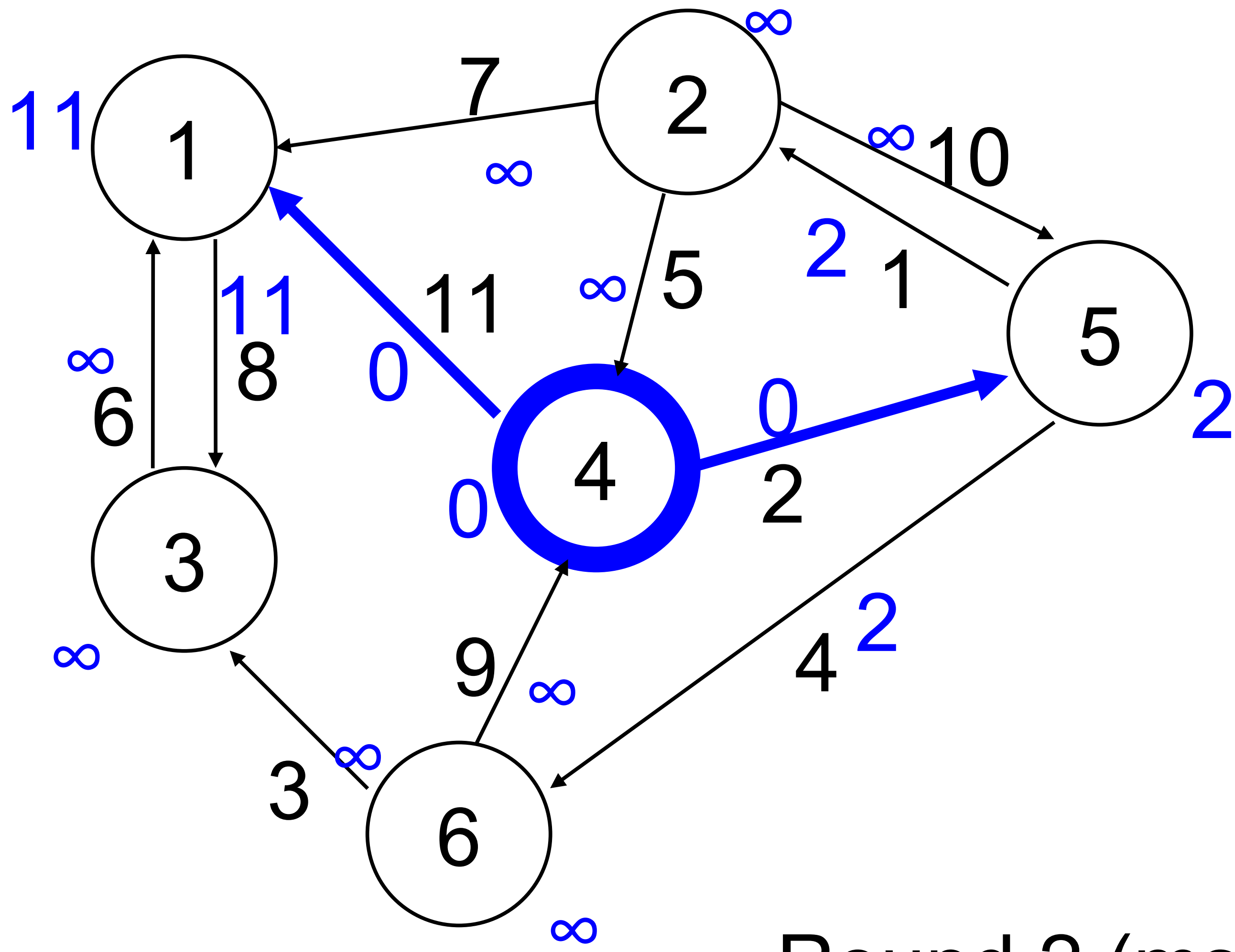
Round 1 (trans)

Shortest paths



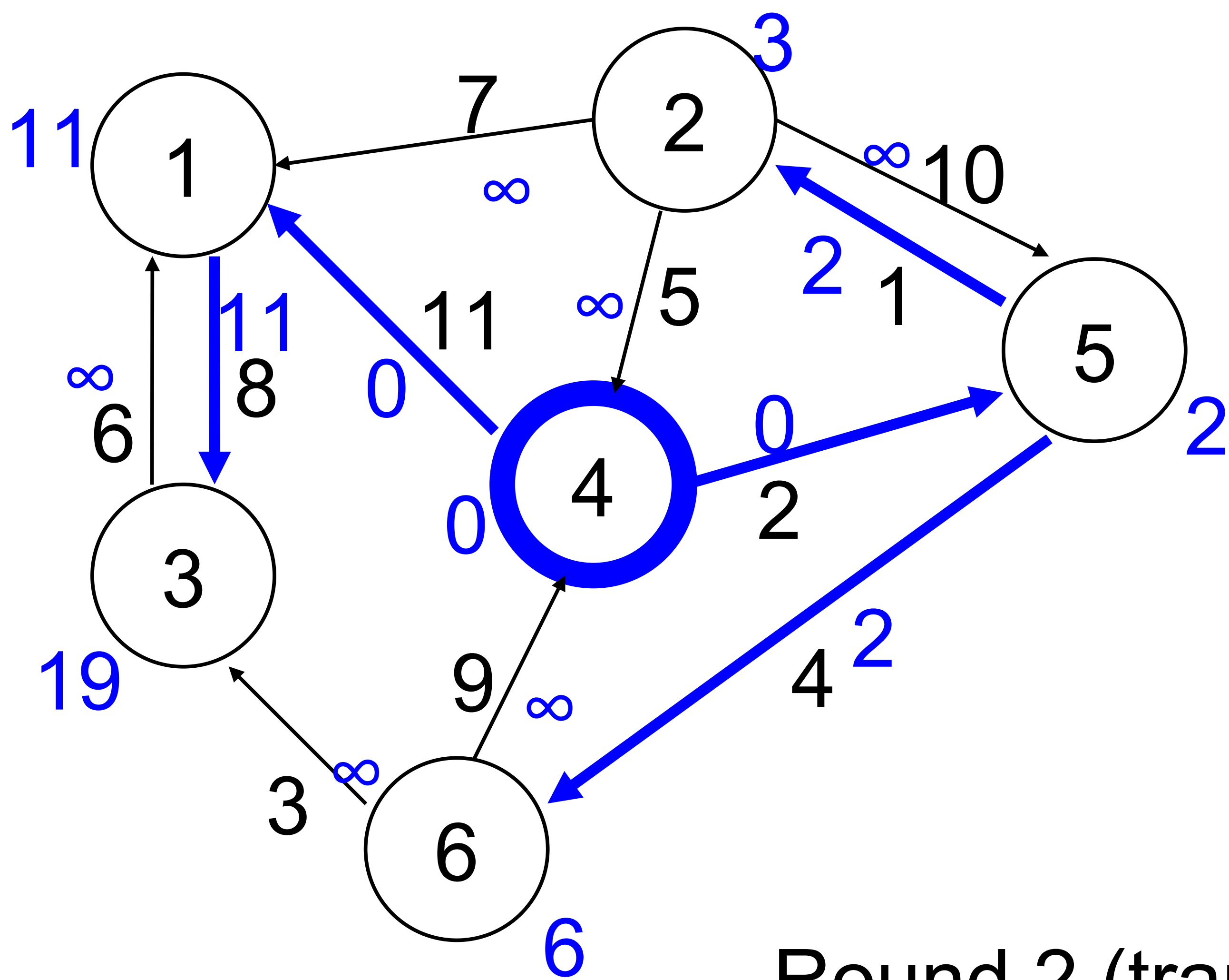
Round 2 (start)

Shortest paths



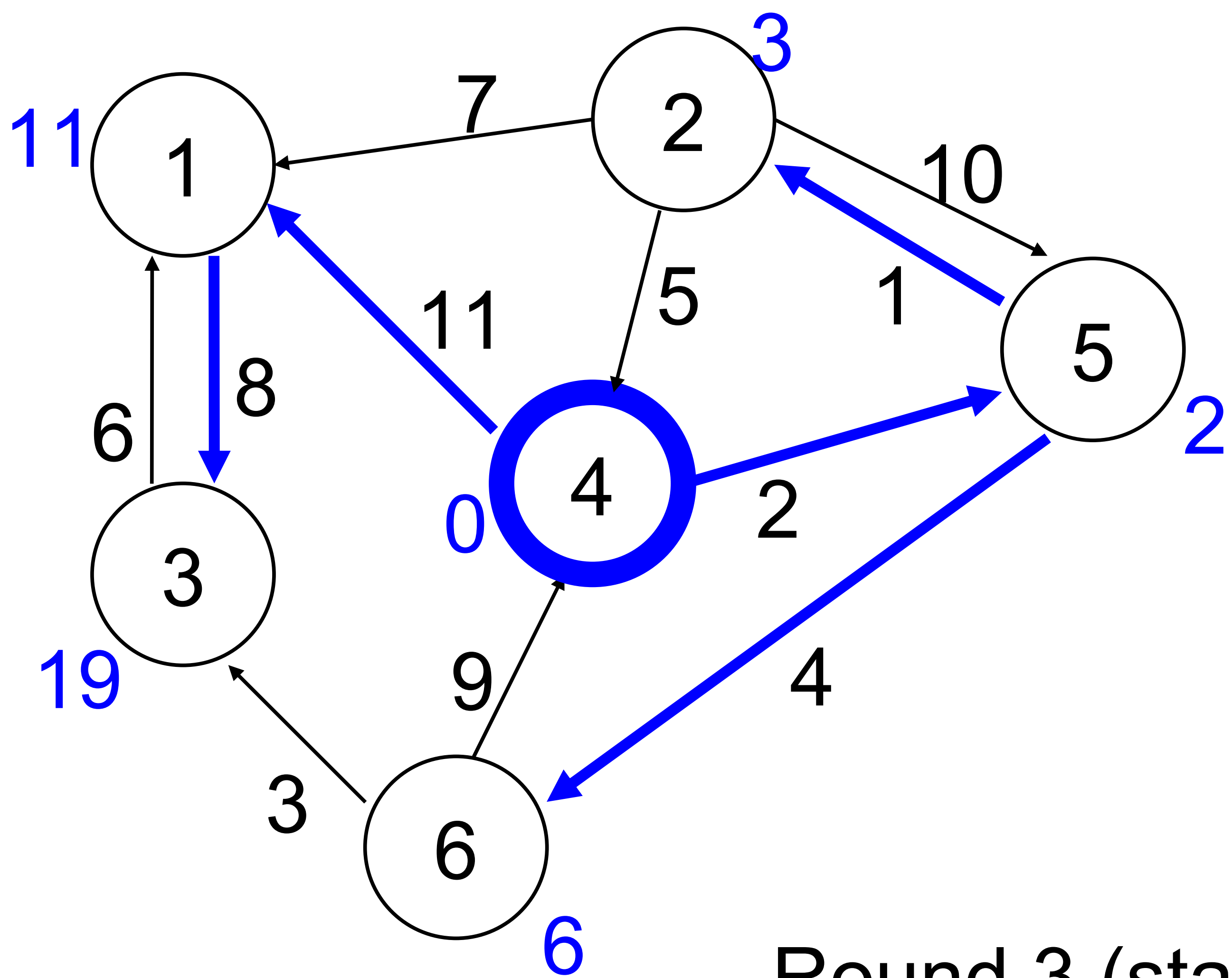
Round 2 (msgs)

Shortest paths



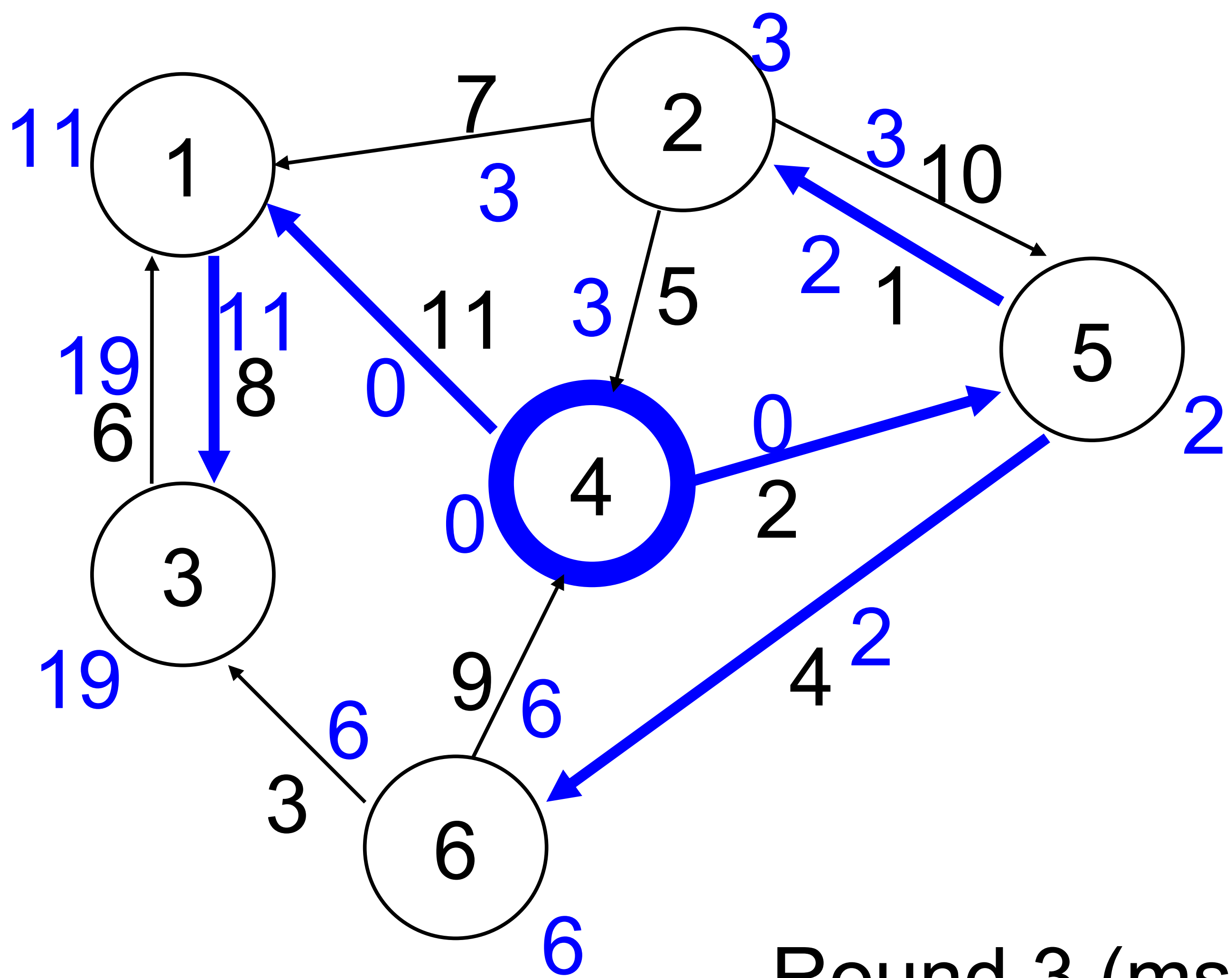
Round 2 (trans)

Shortest paths



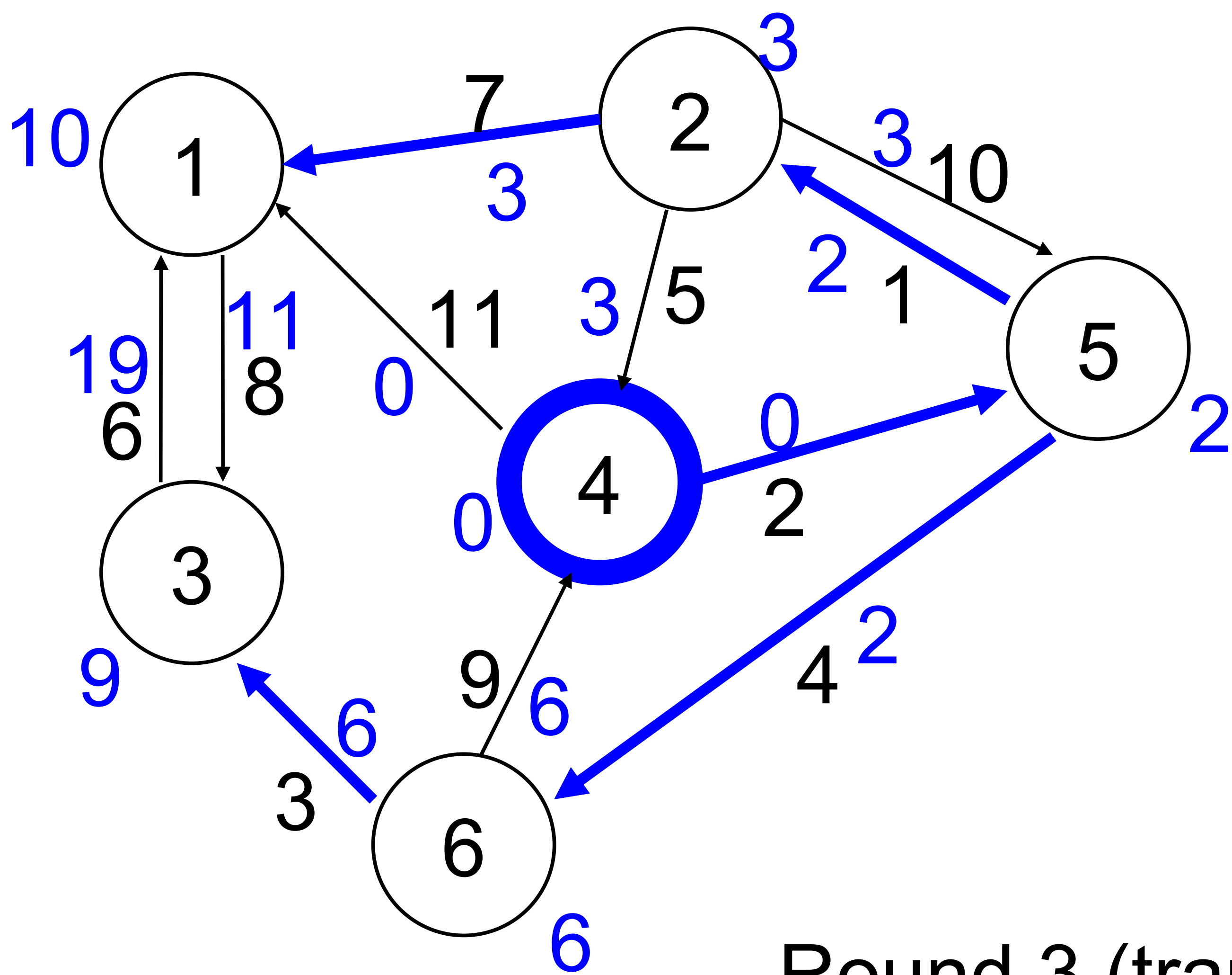
Round 3 (start)

Shortest paths



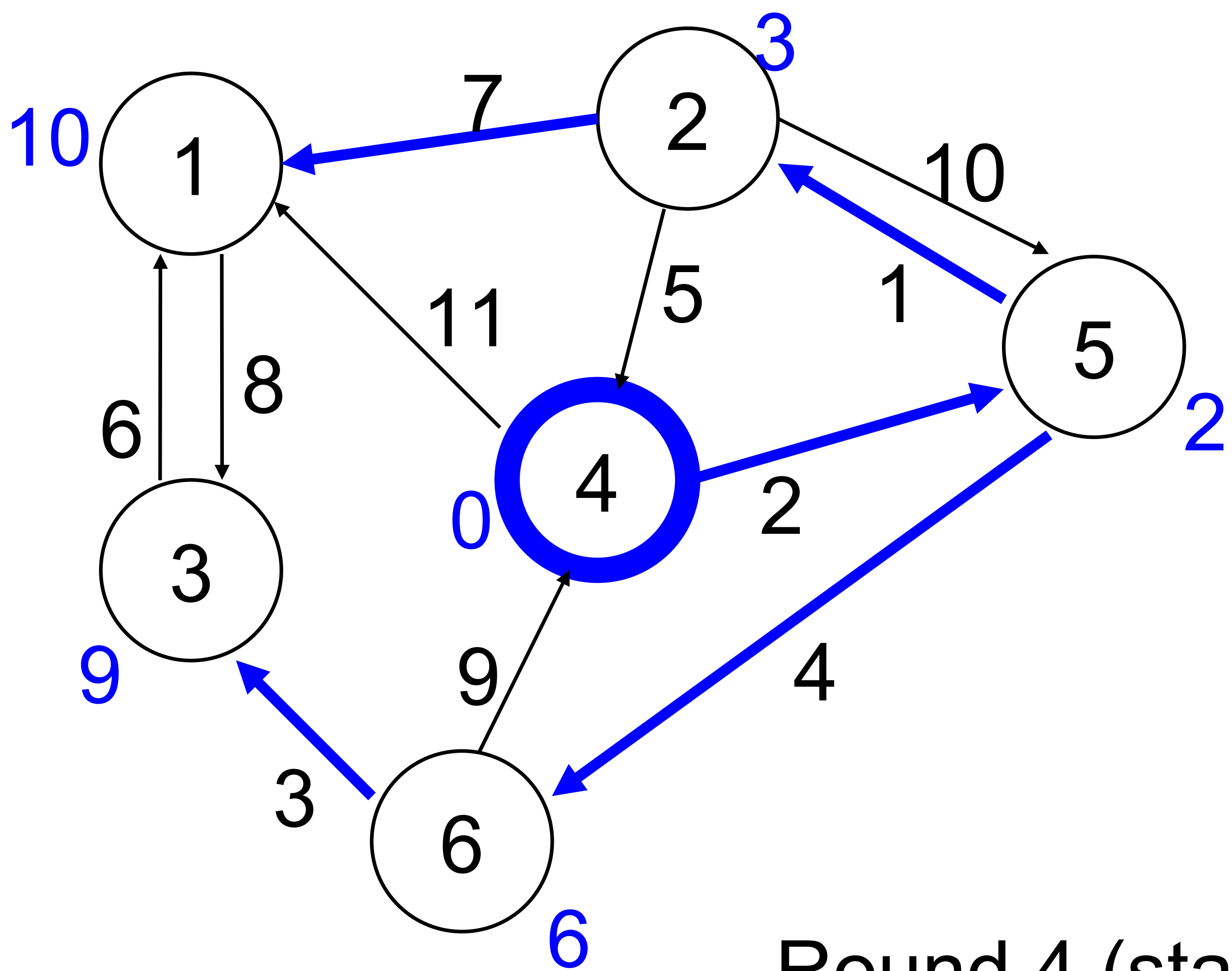
Round 3 (msgs)

Shortest paths



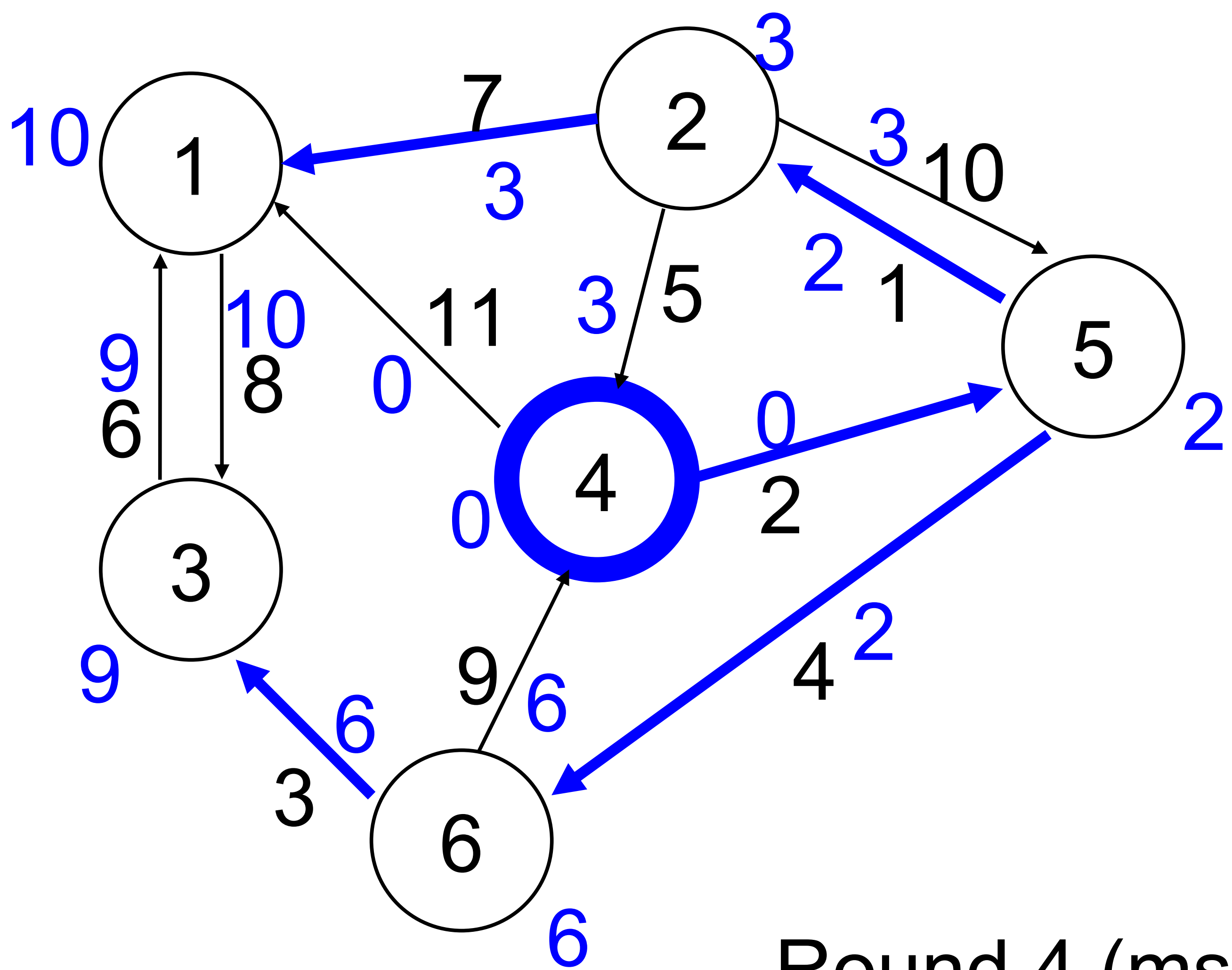
Round 3 (trans)

Shortest paths



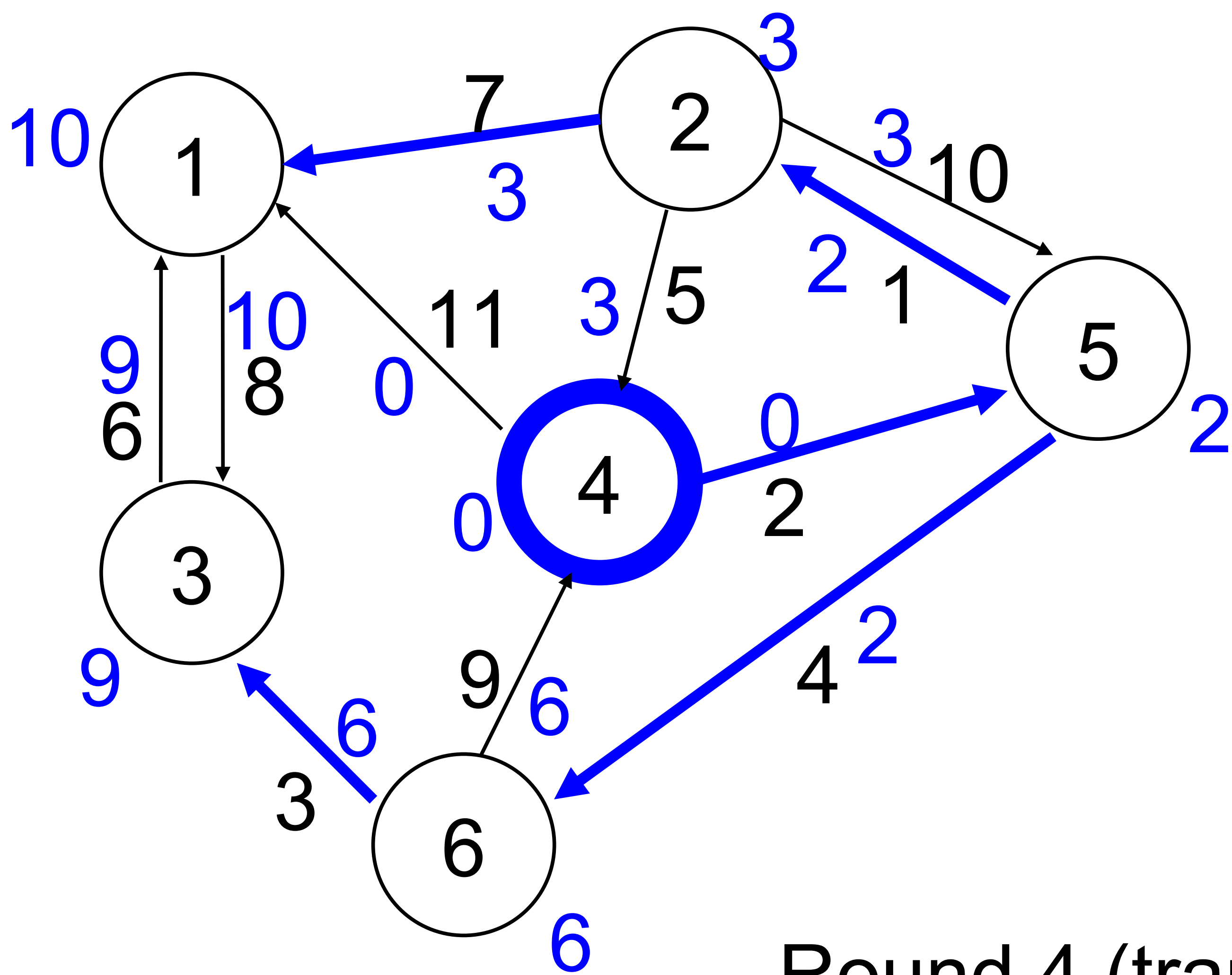
Round 4 (start)

Shortest paths



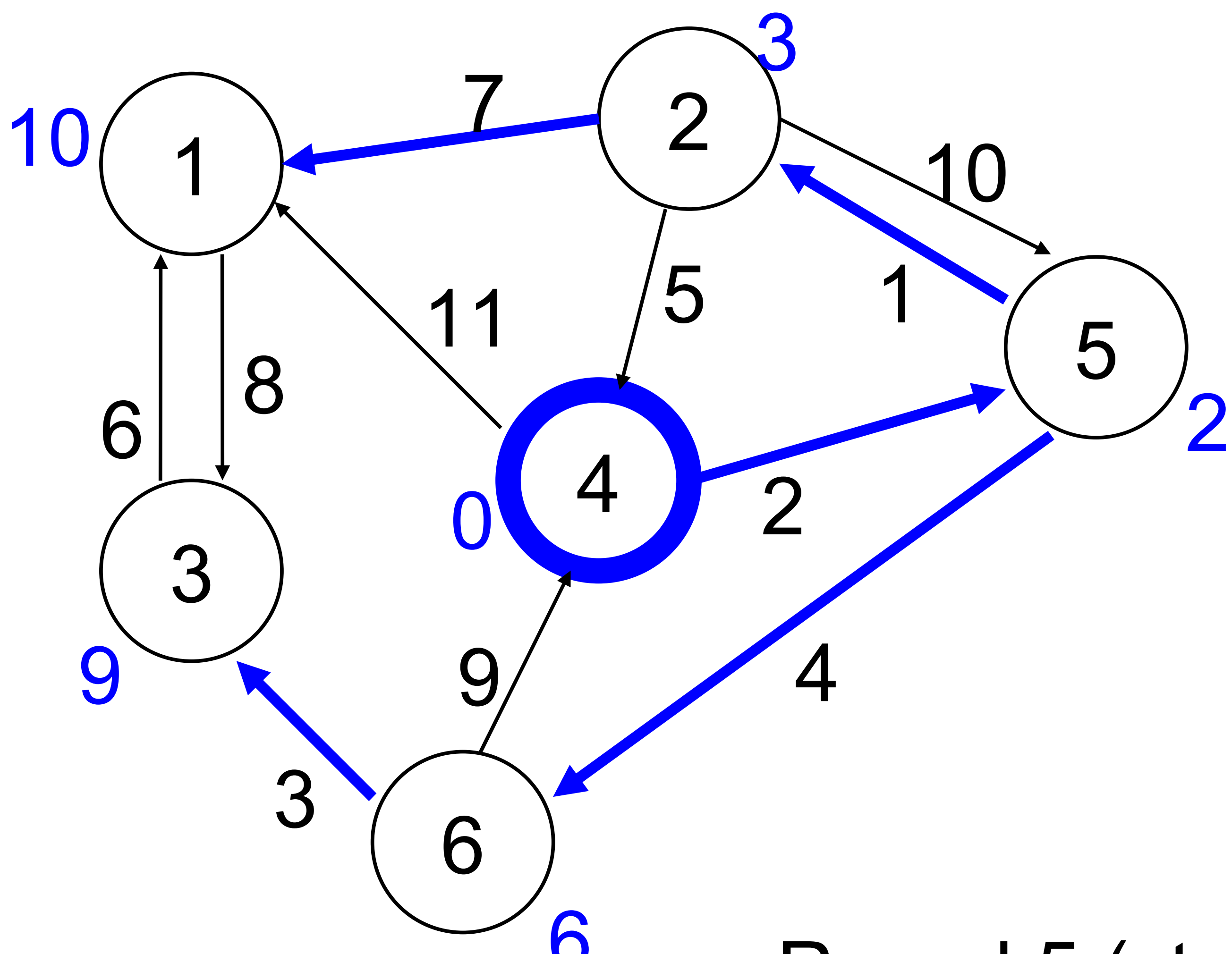
Round 4 (msgs)

Shortest paths



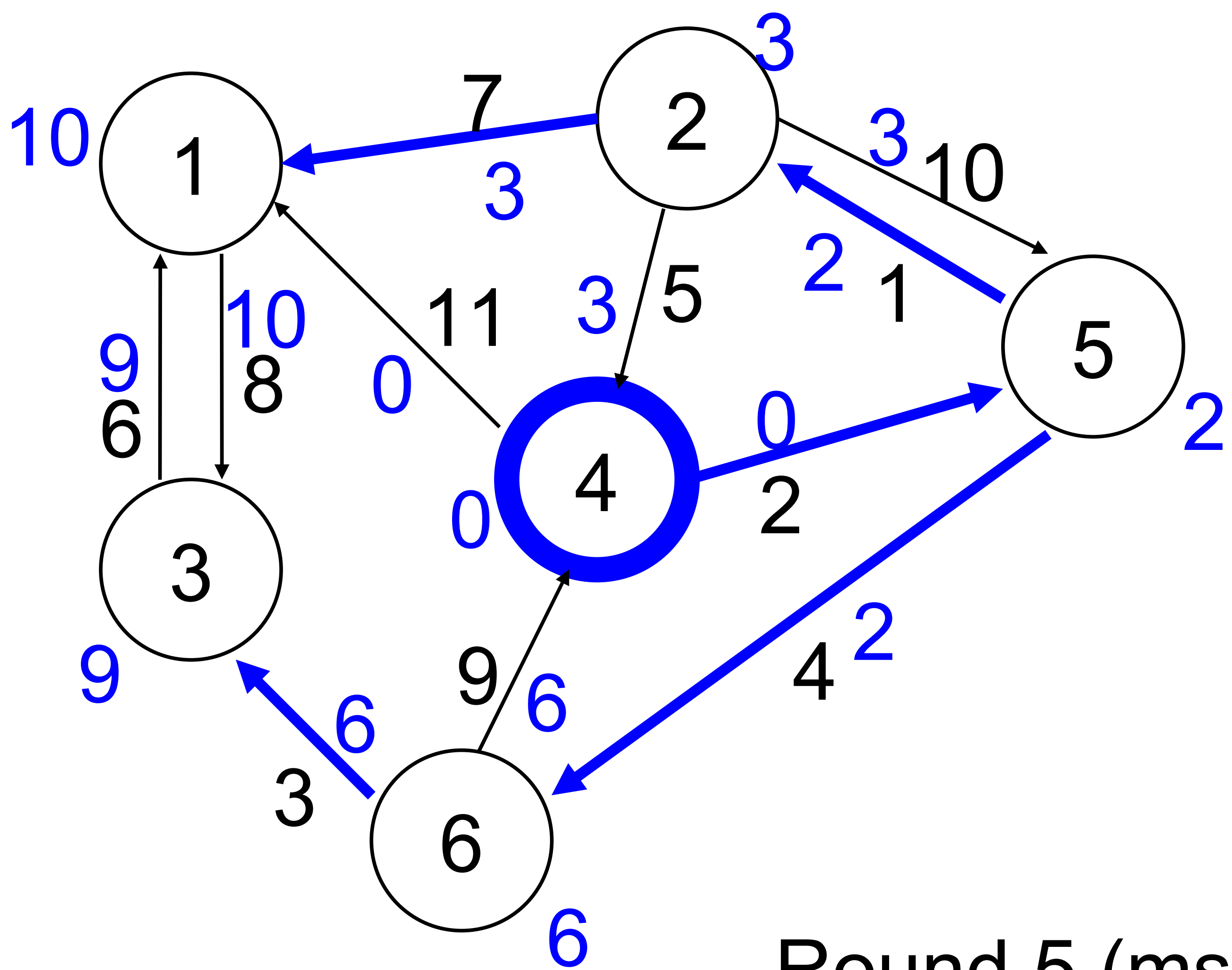
Round 4 (trans)

Shortest paths



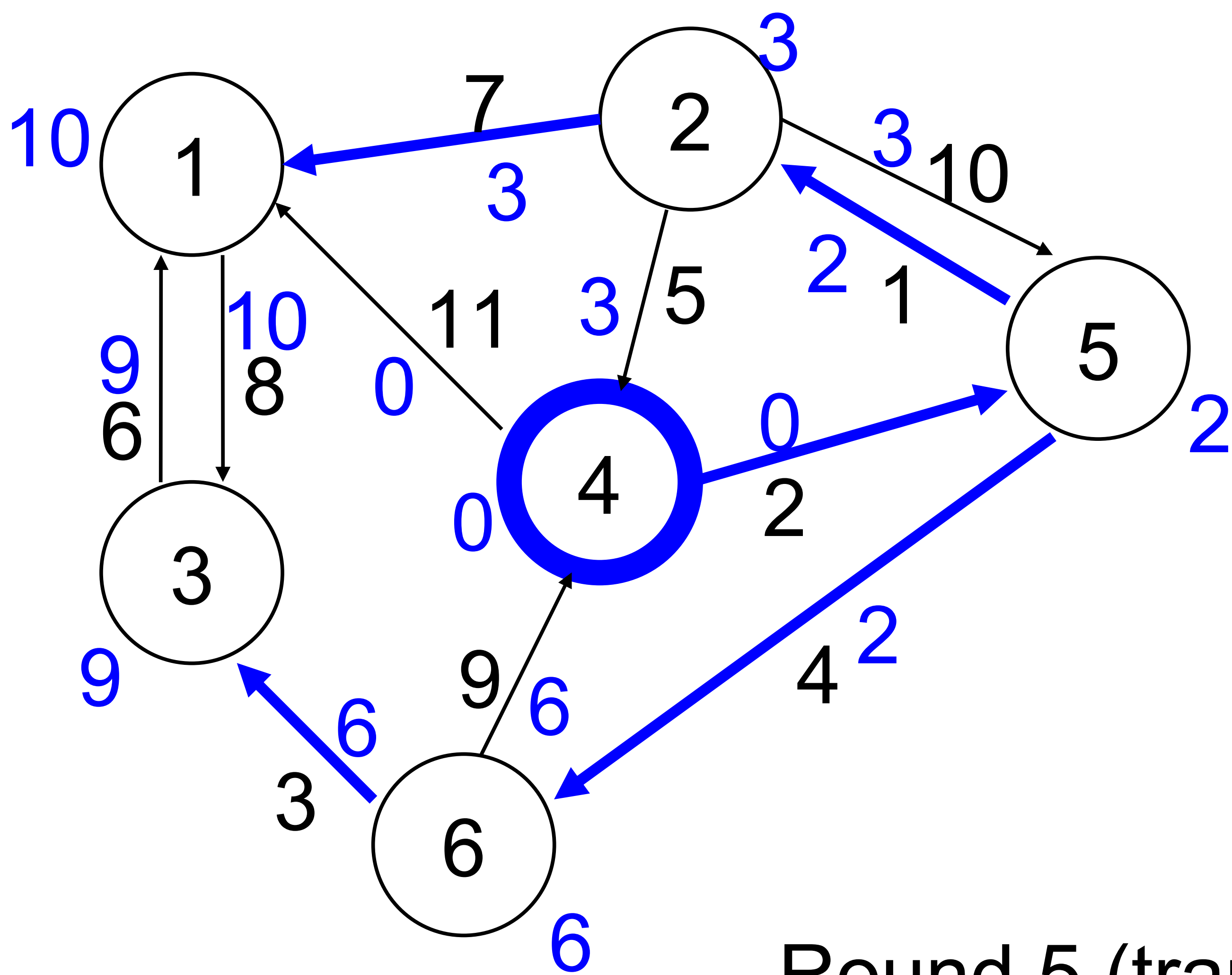
Round 5 (start)

Shortest paths



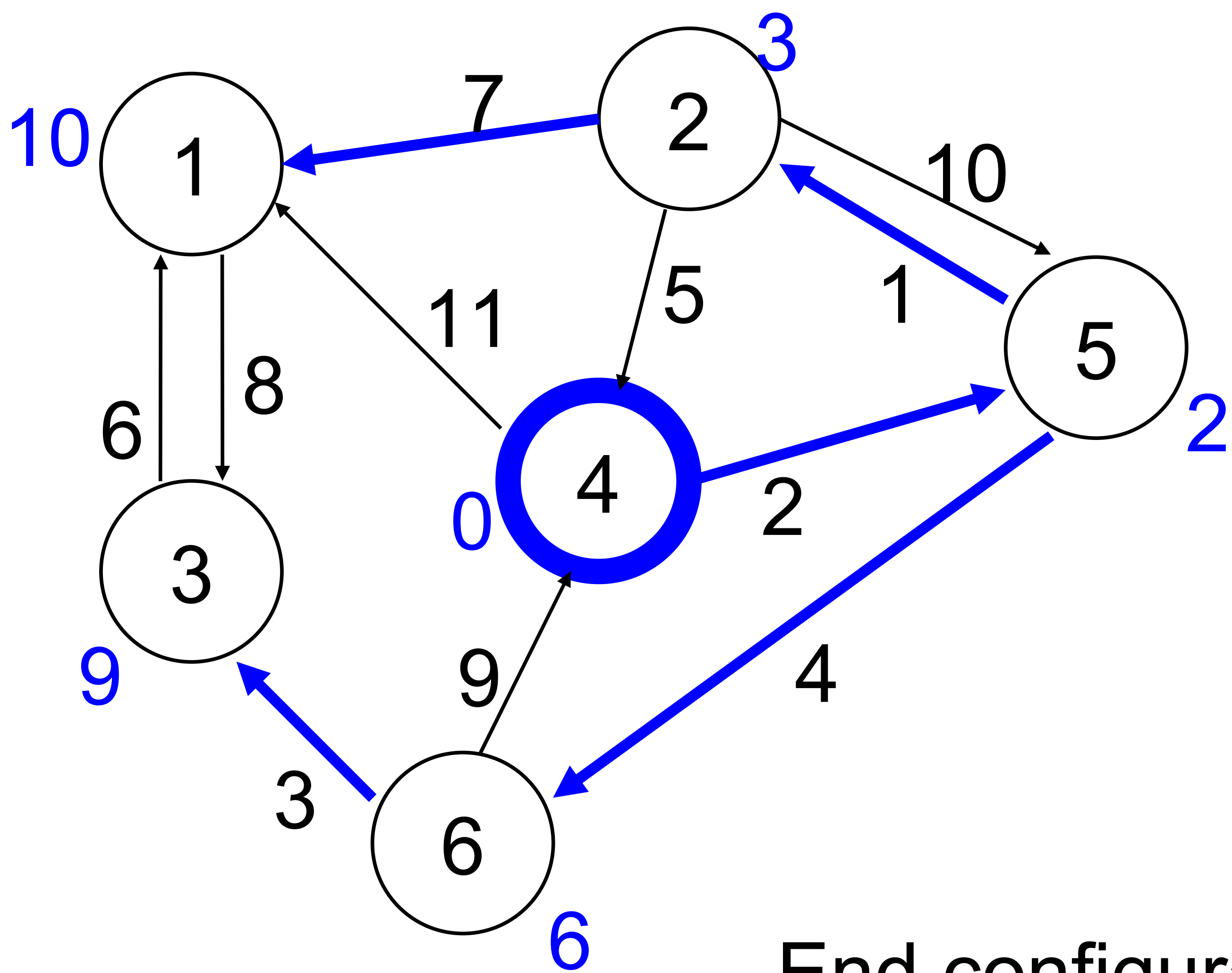
Round 5 (msgs)

Shortest paths



Round 5 (trans)

Shortest paths



End configuration

Correctness

- Need to show that, after round n-1, for each process i:
  - $dist_i$ = shortest distance from i_0
  - $parent_i$ = predecessor on shortest path from i_0
- Proof:
  - Induction on the number r of rounds.
  - But, what statement should we prove about the situation after r rounds?

Correctness

- Key invariant: After r rounds:
  - Every process i has its dist and parent corresponding to a shortest path from i_0 to i among those paths that consist of at most r hops (edges).
  - If there is no such path, then dist = ∞ and parent = null.

- Proof (sketch):
  - By induction on the number r of rounds.
  - Base: r = 0: Immediate from initializations.
  - Inductive step: Assume for r-1, show for r.
    - Fix i; must show that, after round r, $dist_i$ and $parent_i$ correspond to a shortest at-most-r-hop path.
    - First, show that, if $dist_i$ is finite, then it really is the distance on some at-most-r-hop path to i, and parent is its parent on such a path.
    - LTTR---easy use of inductive hypothesis.
    - But we must still argue that $dist_i$ and $parent_i$ correspond to a shortest at-most-r-hop path.

Correctness

- Key invariant: After r rounds:
  - Every process i has its dist and parent corresponding to a shortest path from i_0 to i among those paths that consist of at most r hops (edges).
  - If there is no such path, then dist = ∞ and parent = null.

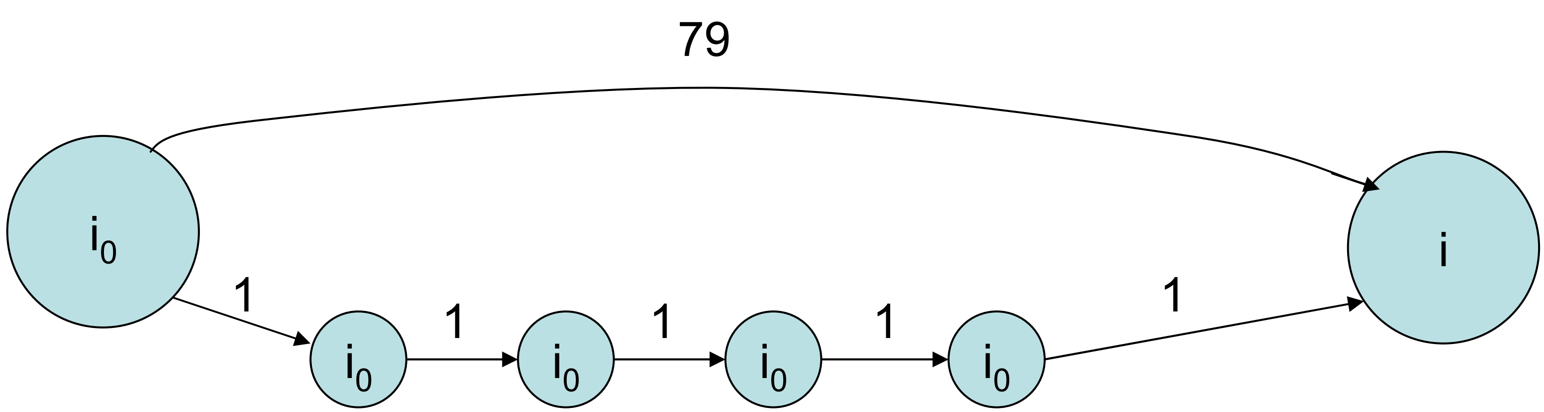
- Proof, inductive step:
  - Assume for r-1, show for r.
  - Fix i; must show that, after round r, $dist_i$ and $parent_i$ correspond to a shortest at-most-r-hop path.
  - If $dist_i$ is finite, then it really is the distance on some at-most-r-hop path to i, and parent is its parent on such a path.
  - Claim that $dist_i$ and $parent_i$ correspond to a shortest at-most-r-hop path.
  - Any shortest at-most-r-hop path from i_0 to i, when cut off at i's predecessor j on the path, yields a shortest (r-1)-hop path from i_0 to j.
  - By inductive hypothesis, after round r-1, for every such j, $dist_j$ and $parent_j$ correspond to a shortest at-most-(r-1)-hop path from i_0 to j.
  - At round r, all such j send i their info about their shortest at-most-(r-1)-hop paths, and process i takes this into account in calculating $dist_i$.
  - So after round r, $dist_i$ and $parent_i$ correspond to a shortest at-most-r-hop path.

Complexity

- Complexity:
  - Time: n-1 rounds
  - Messages: (n-1) |E|

- Worse that BFS, which has:
  - Time: diam rounds
  - Messages: |E|

- Q: Does the time bound really depend on n, or is it O(diam)?
- A: It's really n, since "shortest path" can be over a path with more links.
- Example:



Bellman-Ford Shortest-Paths Algorithm

- Will revisit Bellman-Ford shortly in asynchronous networks.
- Gets even more expensive there.
- Similar to old Arpanet routing algorithm.

Minimum spanning tree

- Another classical problem.
- Many sequential algorithms.
- Construct a spanning tree, minimizing the total weight of all edges in the tree.
- Assume:
  - Weighted undirected graph (bidirectional communication).
    - Weights are nonnegative reals.
    - Each node knows weights of incident edges.
  - Processes have UIDs.
  - Nodes know (a good upper bound on) n.
- Required:
  - Each process should decide which of its incident edges are in MST and which are not.

Minimum spanning tree theory

- Graph theory definitions (for undirected graphs)
  - Tree: Connected acyclic graph
  - Forest: An acyclic graph (not necessarily connected)
  - Spanning subgraph of a graph G: Subgraph that includes all nodes of G.
    - Spanning tree, spanning forest.
  - Component of a graph: A maximal connected subgraph.

- Common strategy for computing MST:
  - Start with trivial spanning forest, n isolated nodes.
  - Repeat (n-1 times):
    - Merge two components along an edge that connects them.
    - Specifically, add the minimum-weight outgoing edge (MWOE) of some component to the edge set of the current forest.

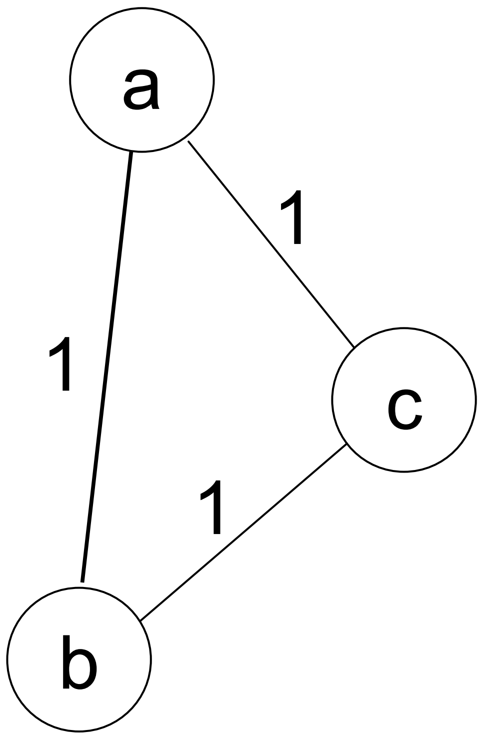
Why this works:

- Similar argument to sequential case.
- Lemma 1: Let { T_i : $1 \leq i \leq k$ } be a spanning forest of G. Fix any j, $1 \leq j \leq k$. Let e be a minimum weight outgoing edge of T_j. Then there is a spanning tree for G that includes all the T_is and e, and has minimum weight among all spanning trees for G that include all the T_is.
- Proof:
  - Suppose not---there's some spanning tree T for G that includes all the T_is and does not include e, and whose total weight is strictly less than that of any spanning tree that includes all the T_is and e.
  - Construct a new graph T′ (not a tree) by adding e to T.
  - Contains a cycle, which must contain another outgoing edge, e′, of T_j.
  - weight(e′) ≥ weight(e), by choice of e (smallest weight).
  - Construct a new tree T″ by removing e′ from T′.
  - Then T″ is a spanning tree, contains all the T_is and e.
  - weight(T″) ≤ weight(T).
  - Contradicts assumed properties of T.

Minimum spanning tree algorithms

- General strategy:
  - Start with n isolated nodes.
  - Repeat (n-1 times):
    - Choose some component i.
    - Add the minimum-weight outgoing edge (MWOE) of component i.
- Sequential MST algorithms follow (special cases of) this strategy:
  - Dijkstra/Prim: Grows one big component by adding one more node at each step.
  - Kruskal: Always add min weight edge globally.
- Distributed?
  - All components can choose simultaneously.
  - But there is a problem…

Can get cycles:



Minimum spanning tree

- Avoid this problem by assuming that all weights are distinct.
- Not a serious restriction---could break ties with UIDs.
- Lemma 2: If all weights are distinct, then the MST is unique.
- Proof: Another cycle argument (LTTR).

- Justifies the following concurrent strategy:
  - At each stage, suppose (inductively) that the current forest contains only edges from the unique MST.
  - Now several components choose MWOEs concurrently.
  - Each of these edges is in the unique MST, by Lemma 1.
  - So OK to add them all (no cycles, since all are in the same MST).

- GHS (Gallager, Humblet, Spira) algorithm
  - Very influential (Dijkstra prize).
  - Designed for asynchronous setting, but simplified here.
  - We will revisit it in asynchronous networks.

GHS distributed MST algorithm

- Proceeds in phases (levels), each with O(n) rounds.
  - Length of phases is fixed, and known to everyone.
  - This is all that n is used for.
  - We'll remove use of n for asynchronous algorithm.

- For each $k \geq 0$, level k components form a spanning forest that is a subgraph of the unique MST.
- Each component is a tree rooted at a leader node.
  - Component identified by UID of leader.
  - Nodes in the component know which incident edges are in the tree.
- Each level k component has at least 2^k nodes.
- Every level k+1 component is constructed from two or more level k components.

- Level 0 components: Single nodes.
- Level k $\rightarrow$ level k+1:

Level k $\rightarrow$ Level k+1

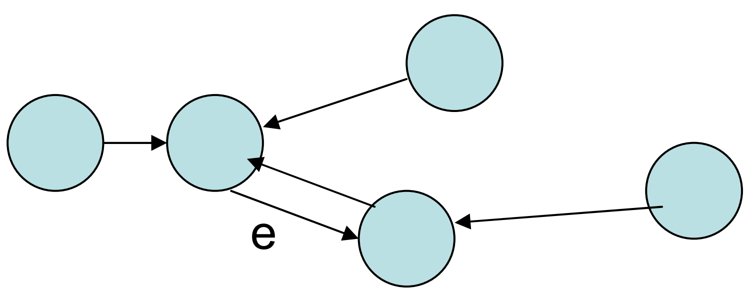
- Each level-k component leader finds MWOE of its component:
  - Broadcasts search (via tree edges).
  - Each process finds the mwoe among its own incident edges.
    - Sends test messages along non-tree edges, asking if node at the other end is in the same component (compare component ids).
  - Convergecast the min back to the leader (via tree edges).
  - Leader determines MWOE.

- Combine level-k components using MWOEs, to obtain level-(k+1) components:
  - Wait long enough for all components to find MWOEs.
  - Leader of each level k component tells endpoint nodes of its MWOE to add the edge for level k+1.
  - Each new component has $\geq 2^{k+1}$ nodes, as claimed.

Level k → Level k+1, cont'd

- Each level-k component leader finds MWOE of its component.
- Combine level-k components using MWOEs, to obtain level-(k+1) components.

- Choose new leaders:
  - For each new, level k+1 component, there is a unique edge e that is the MWOE of two level k sub-components:



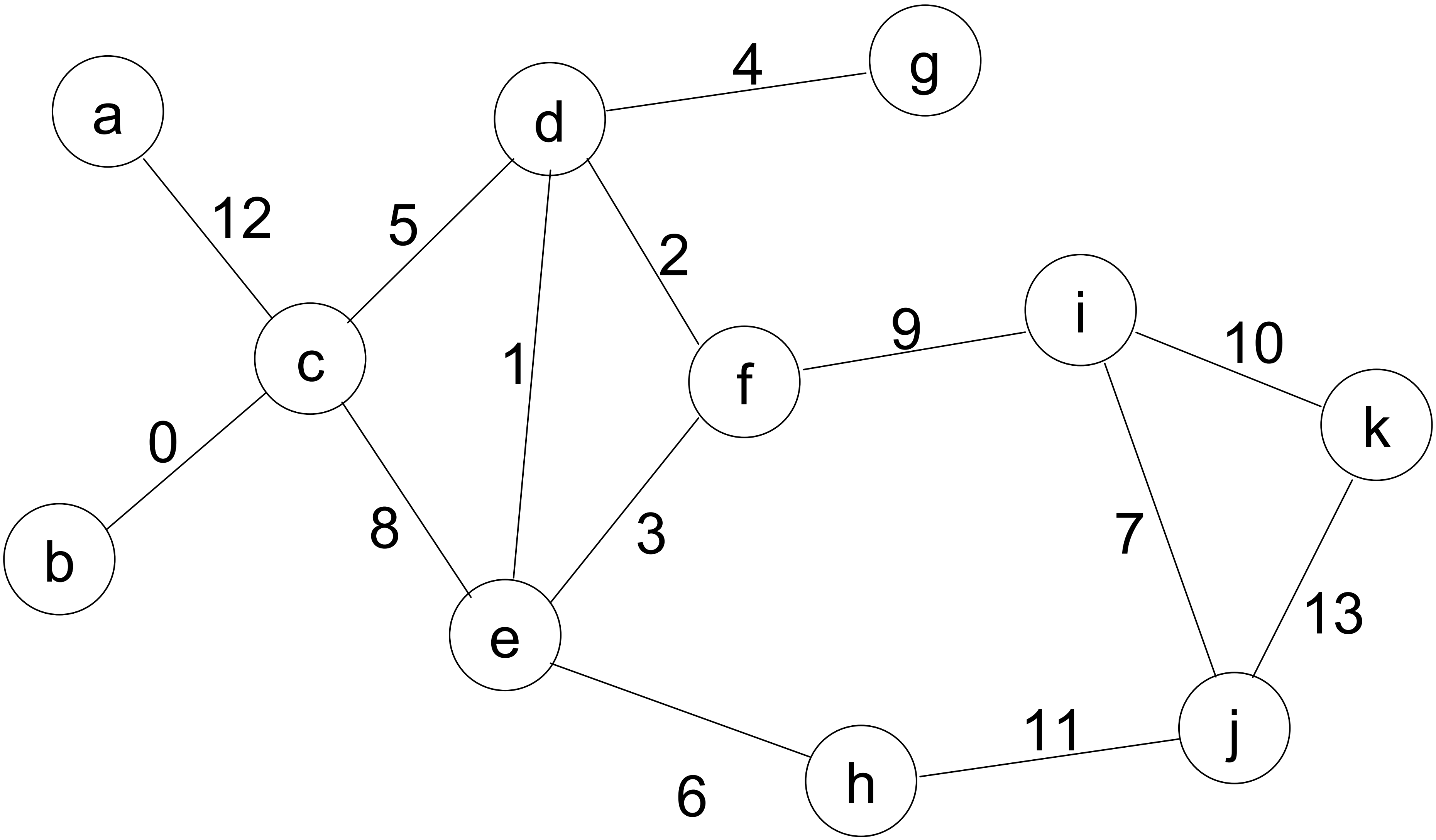
n edges, must have a cycle. Cycle can't have length > 2, because weights of different edges on the cycle must decrease around the cycle.

  - Choose new leader to be the endpoint of e with the larger UID.
  - Broadcast leader UID to new (merged) component.
- GHS terminates when there are no more outgoing edges.

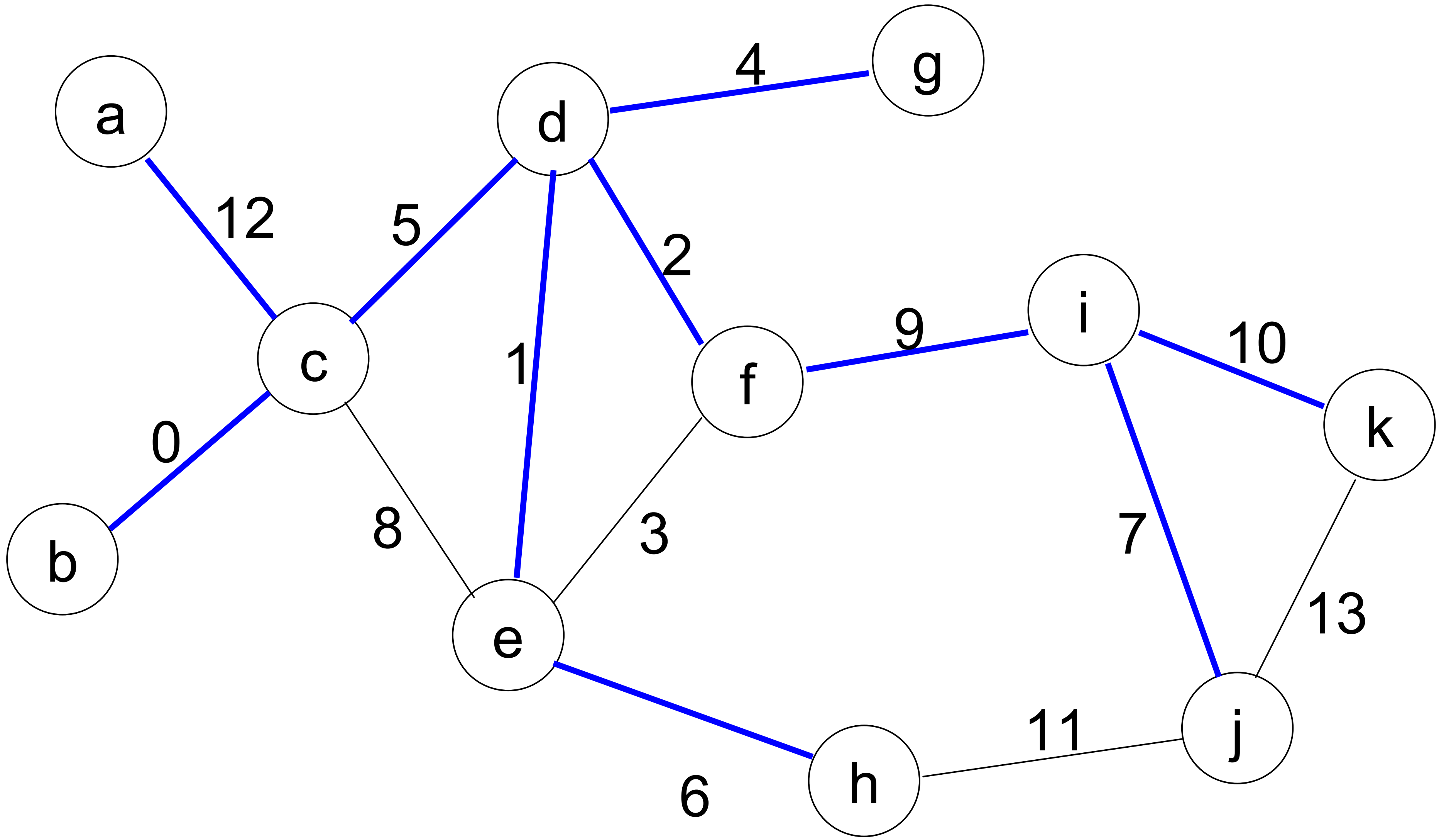
Note on synchronization

- This simplified version of GHS is designed to work with component levels synchronized.
- Difficulties can arise when they get out of synch (as we'll see).
- In particular, test messages are supposed to compare leader UIDs to determine whether endpoints are in the same component.
- Requires that the node being queried has up-to-date UID information.

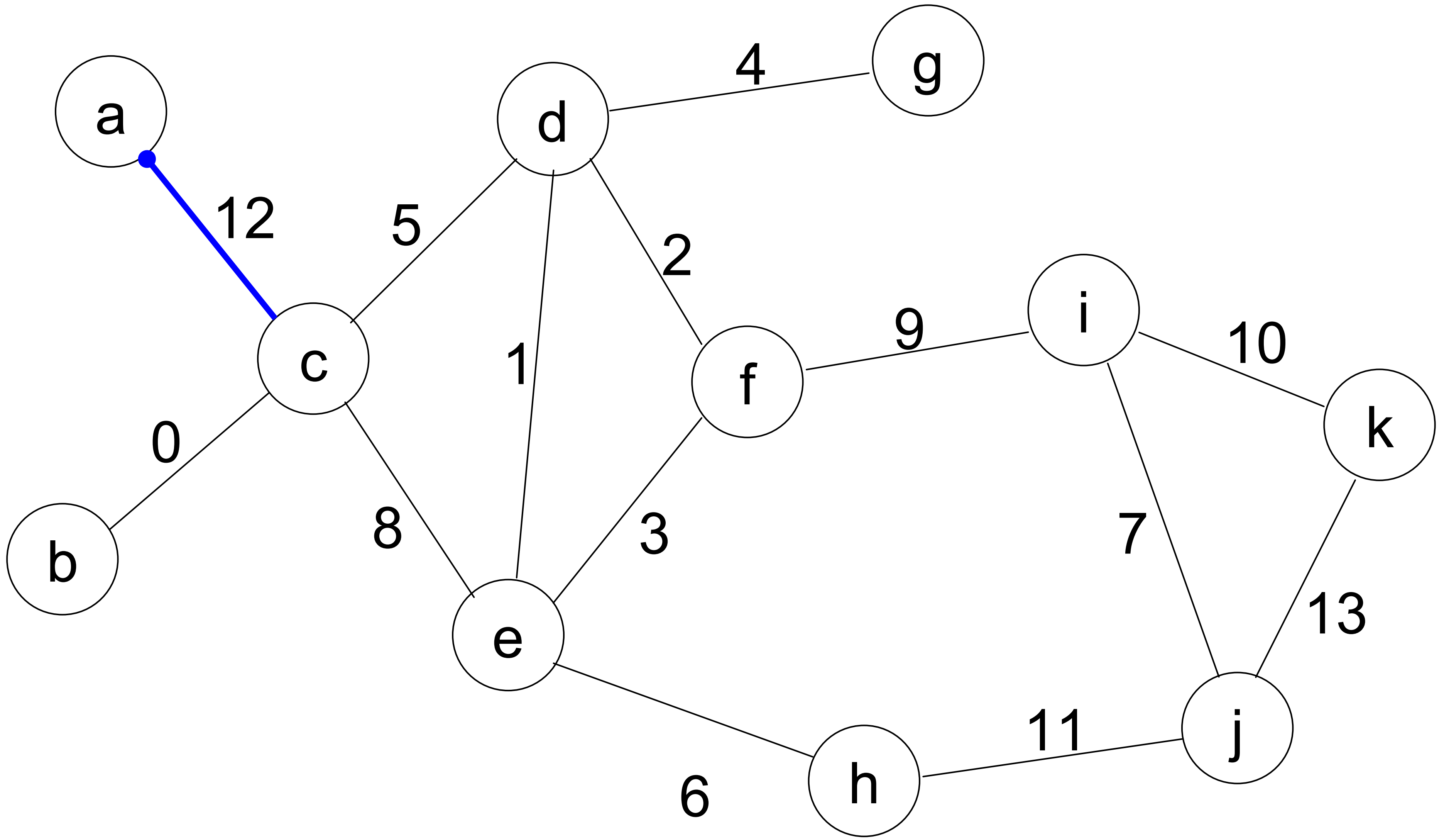
Minimum spanning tree



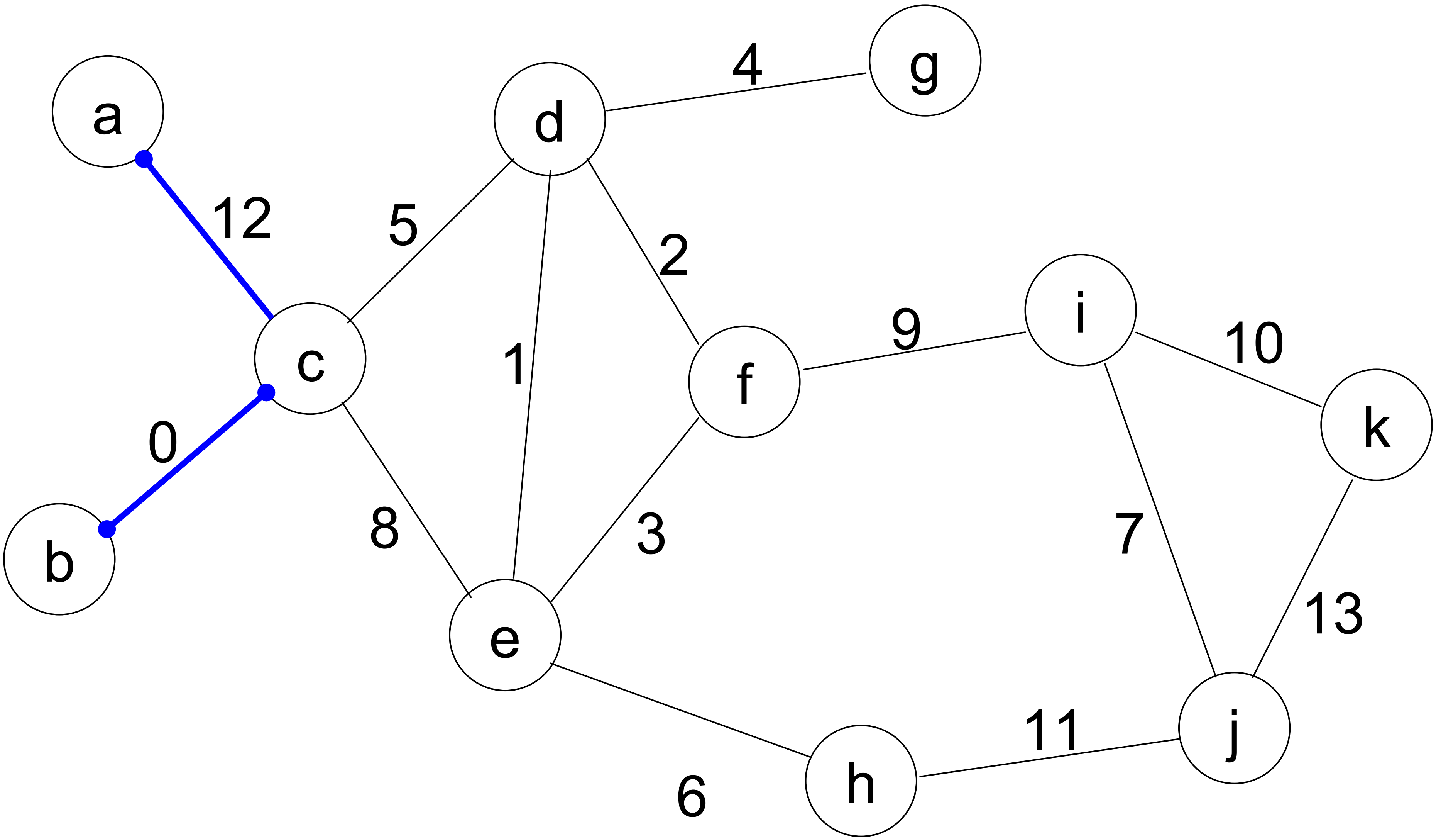
Minimum spanning tree



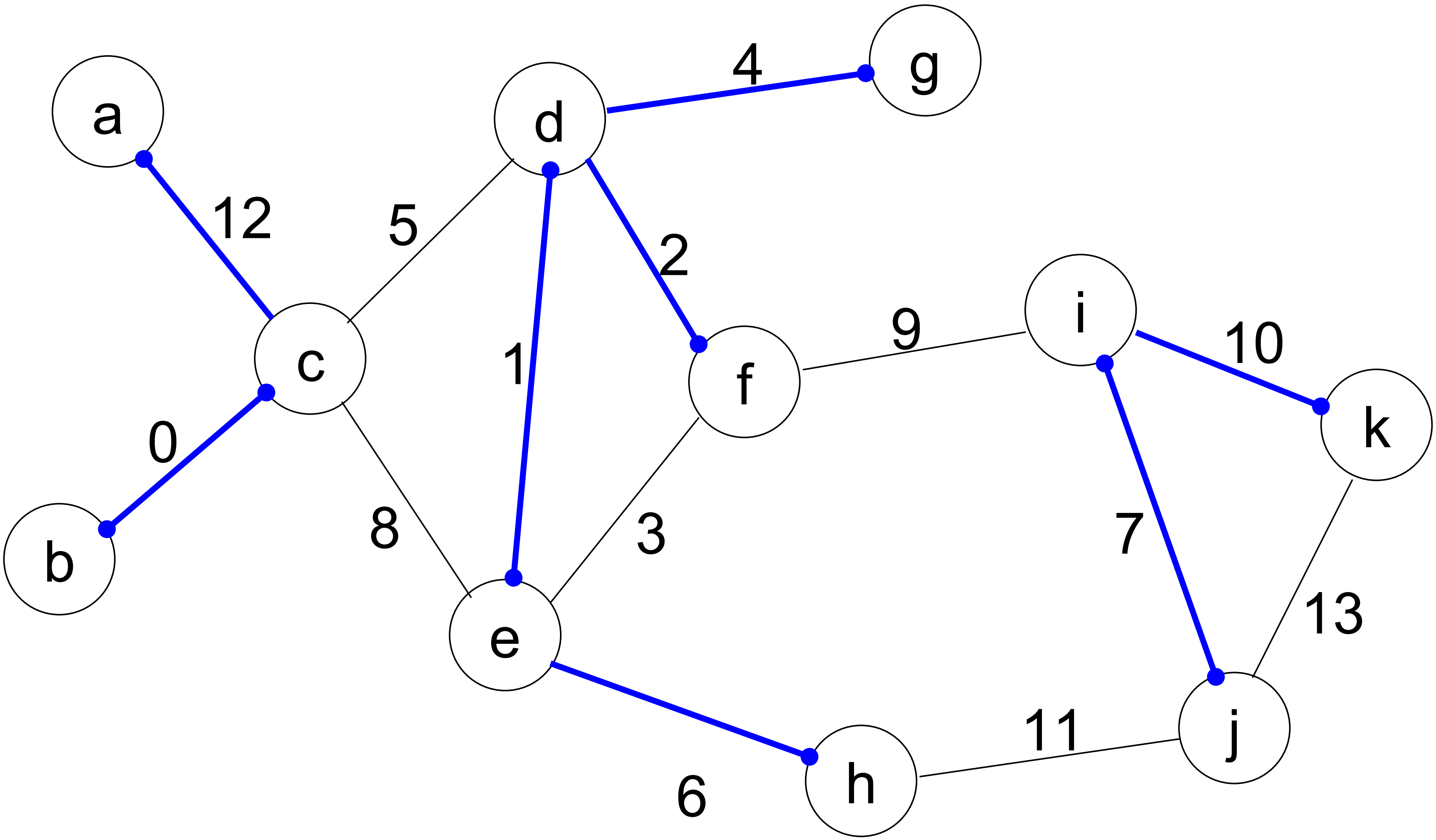
Minimum spanning tree



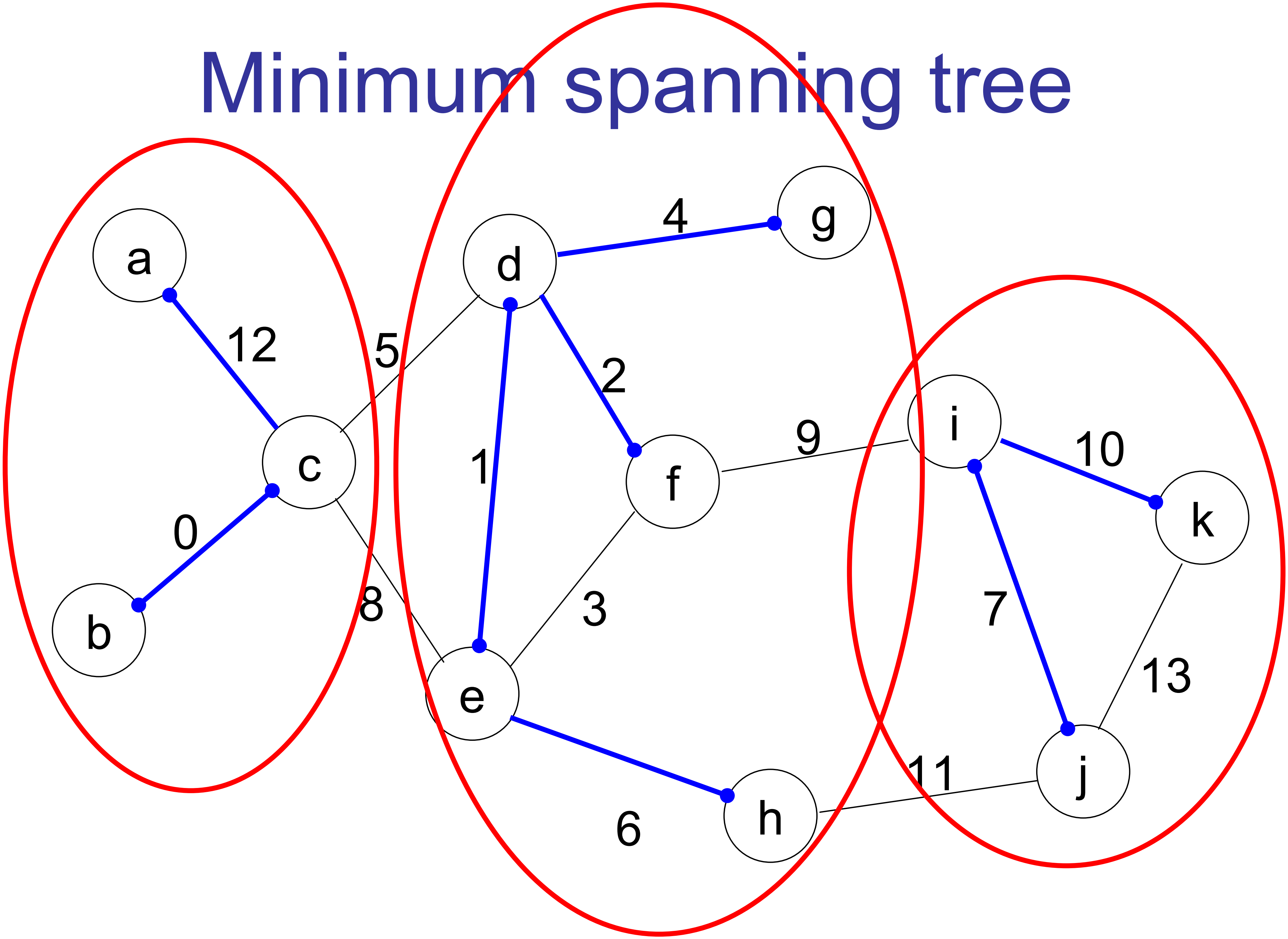
Minimum spanning tree



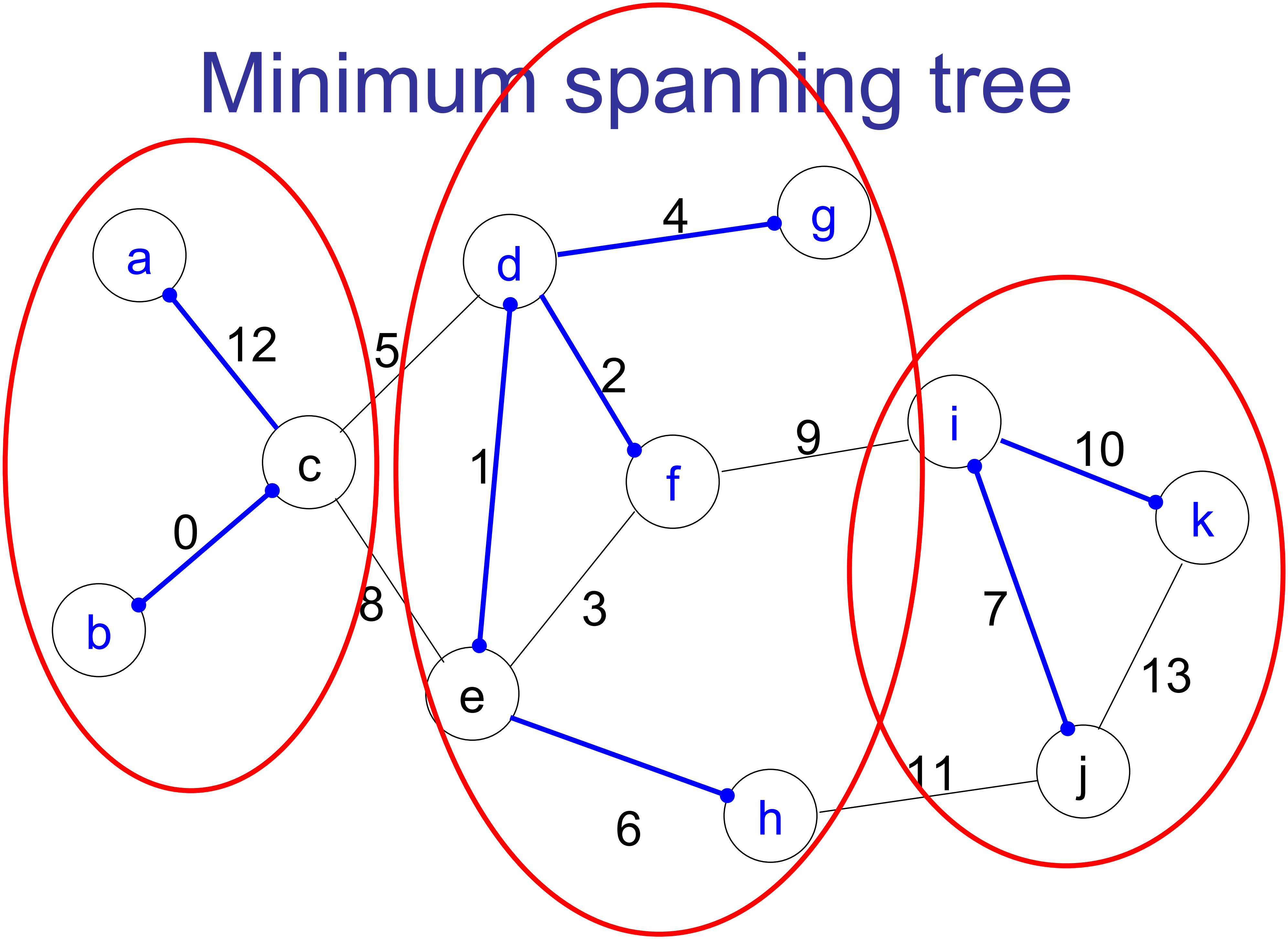
Minimum spanning tree



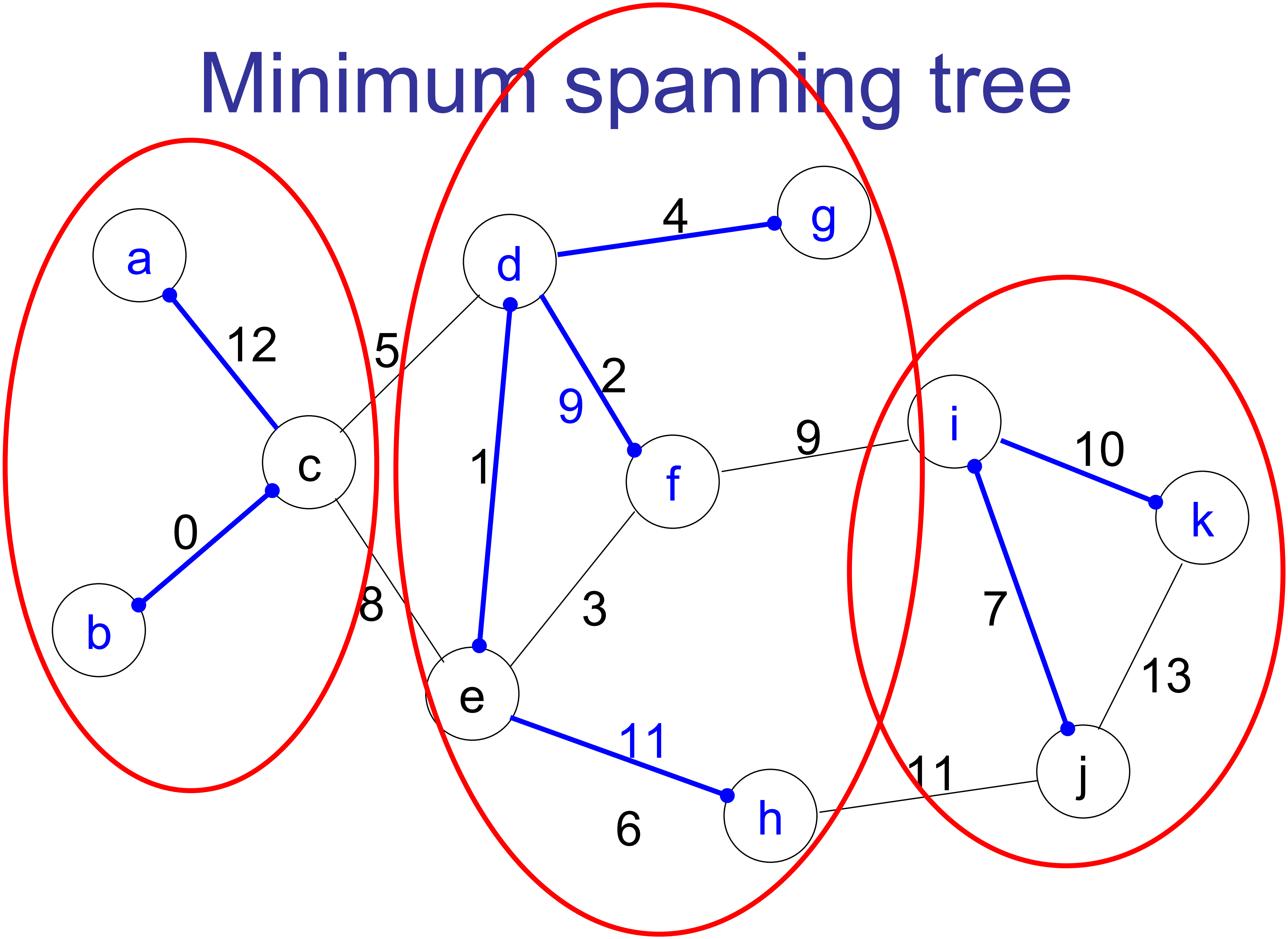
Minimum spanning tree



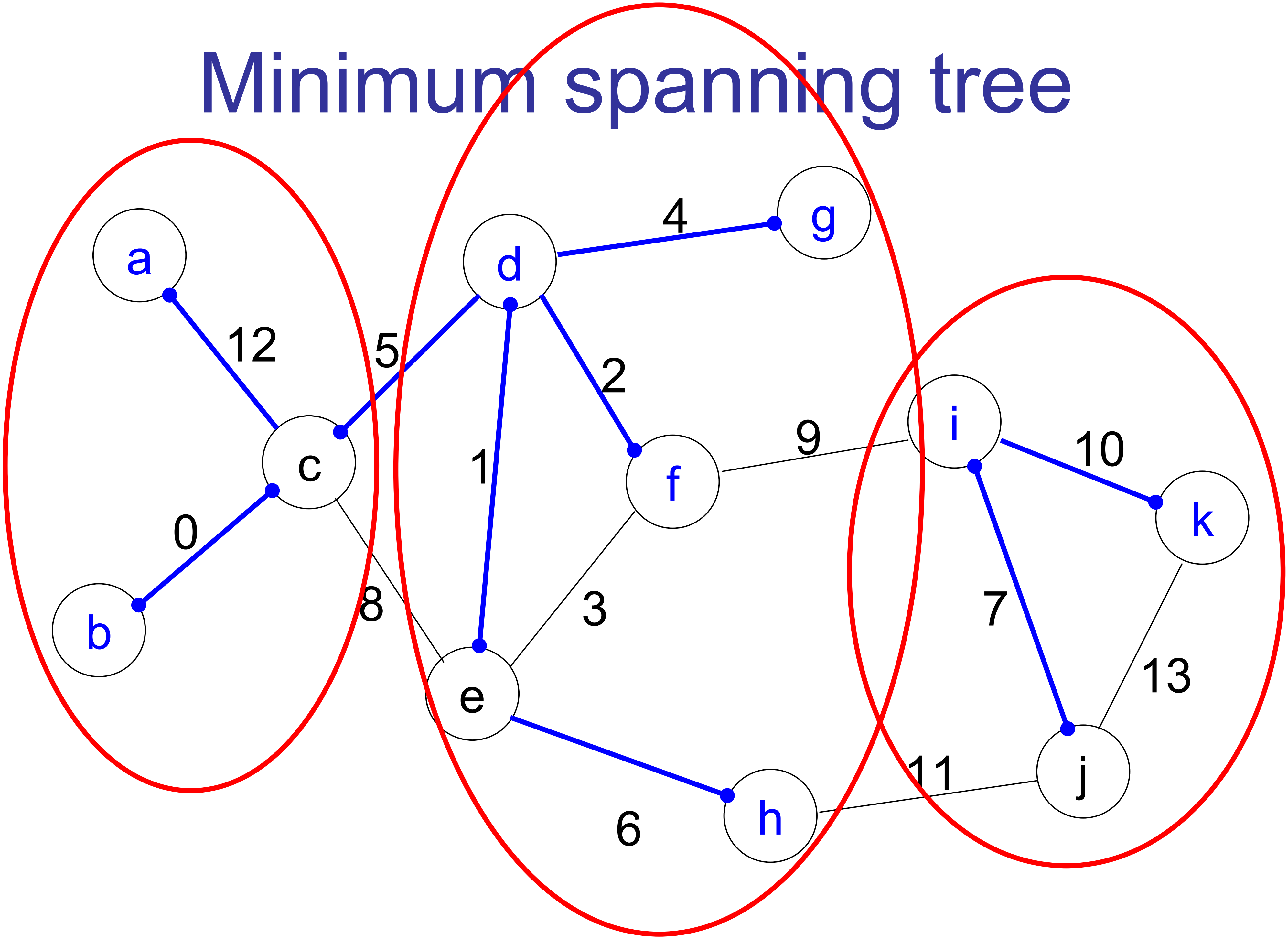
Minimum spanning tree



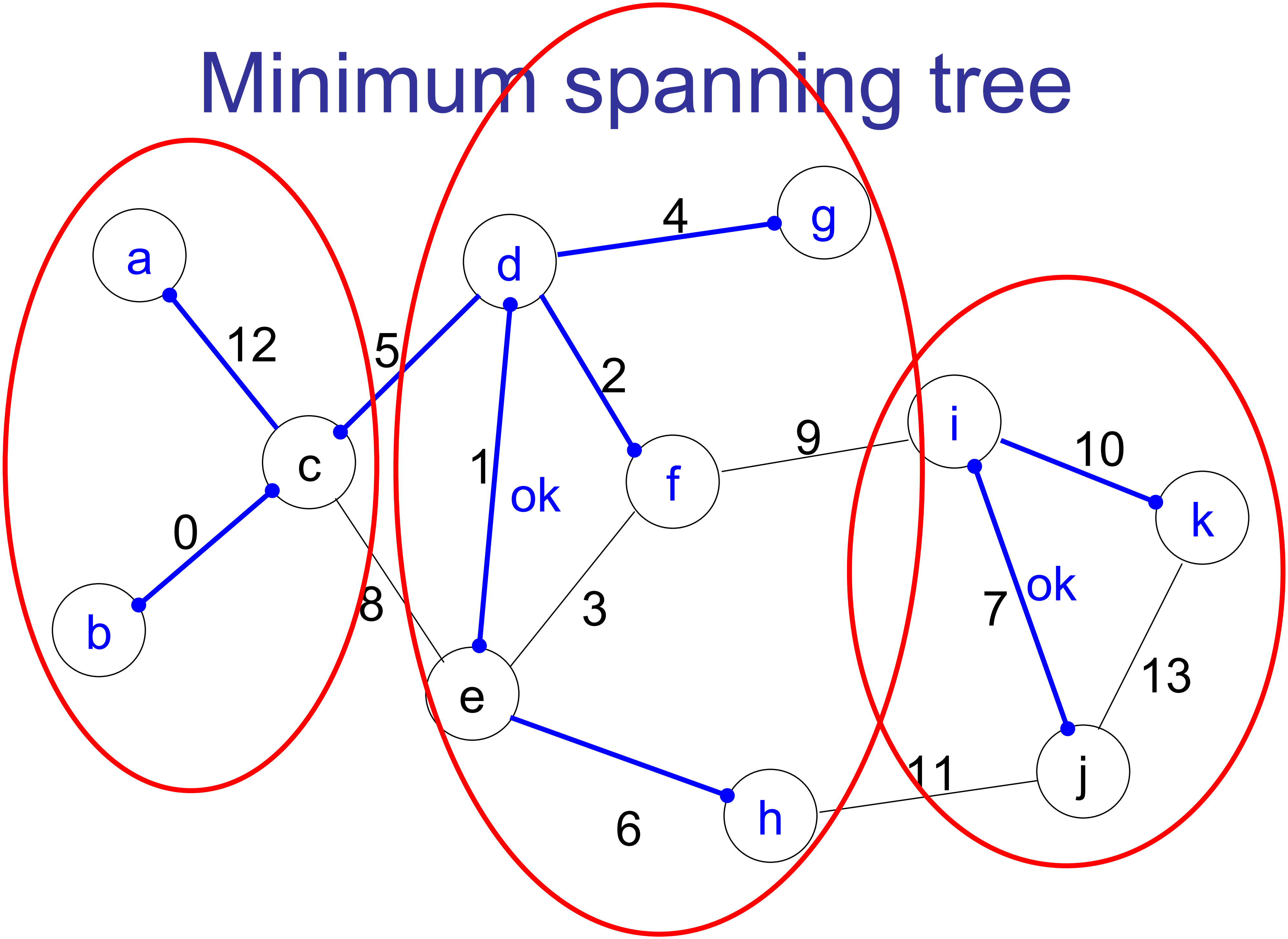
Minimum spanning tree



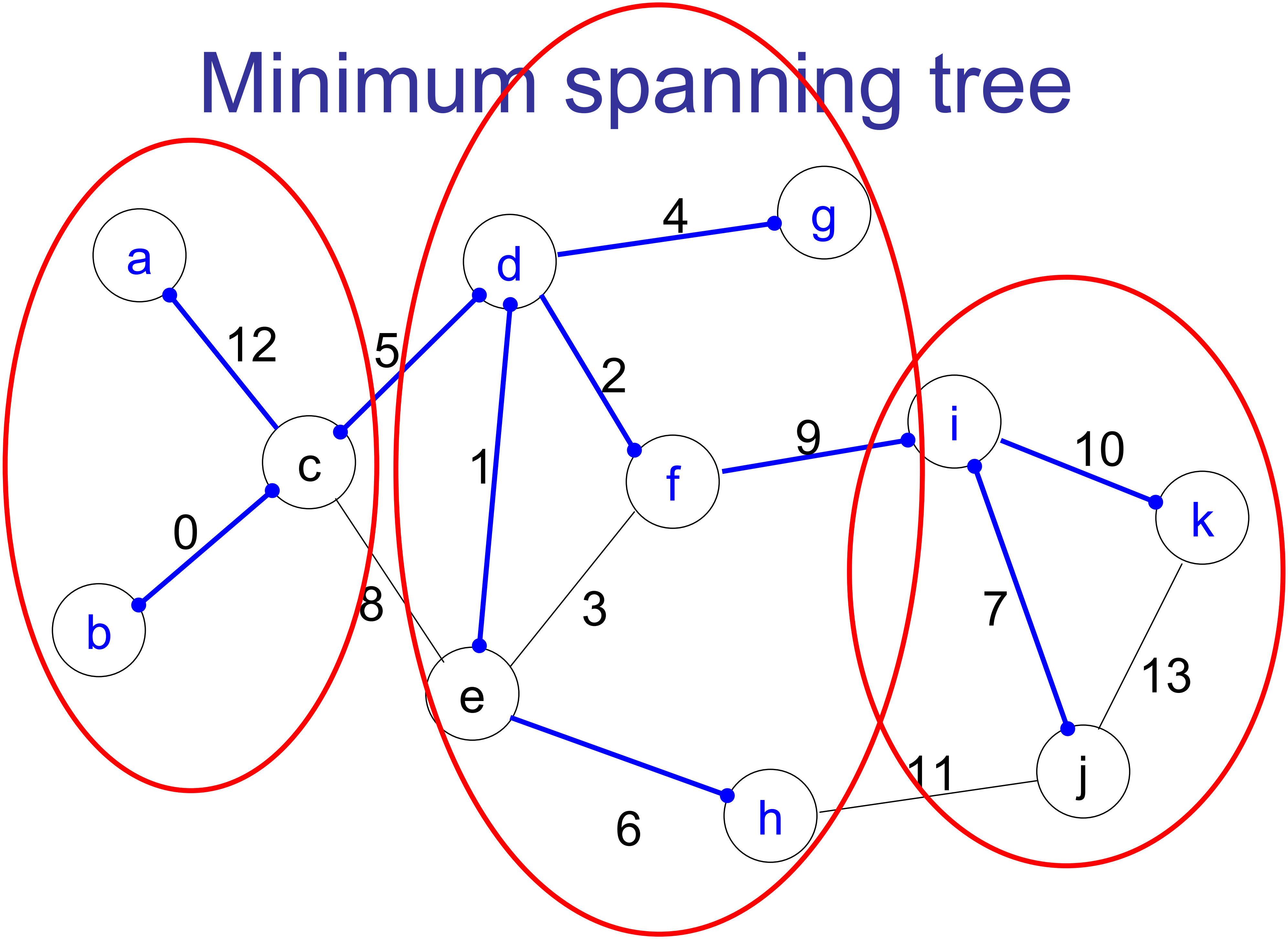
Minimum spanning tree



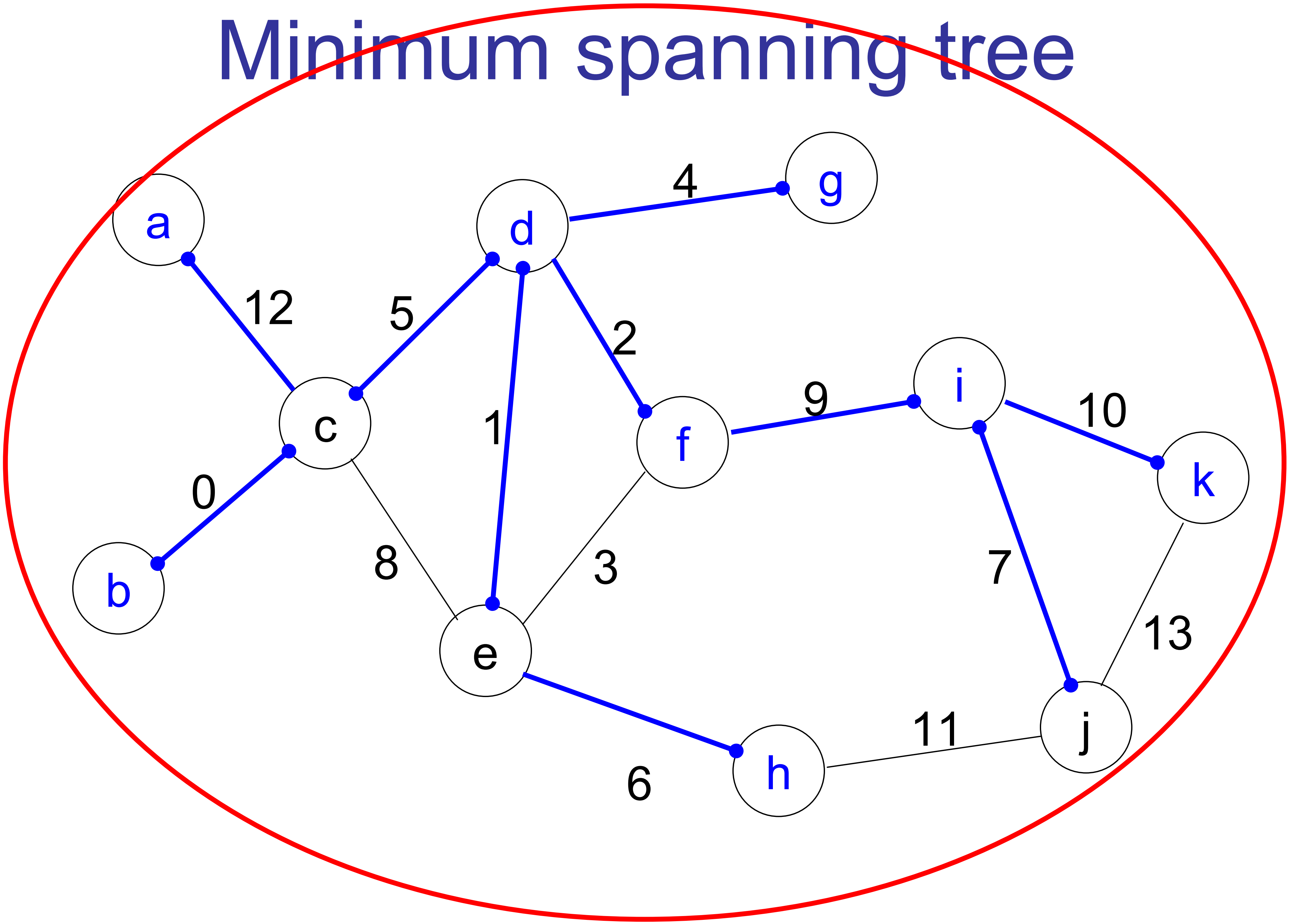
Minimum spanning tree



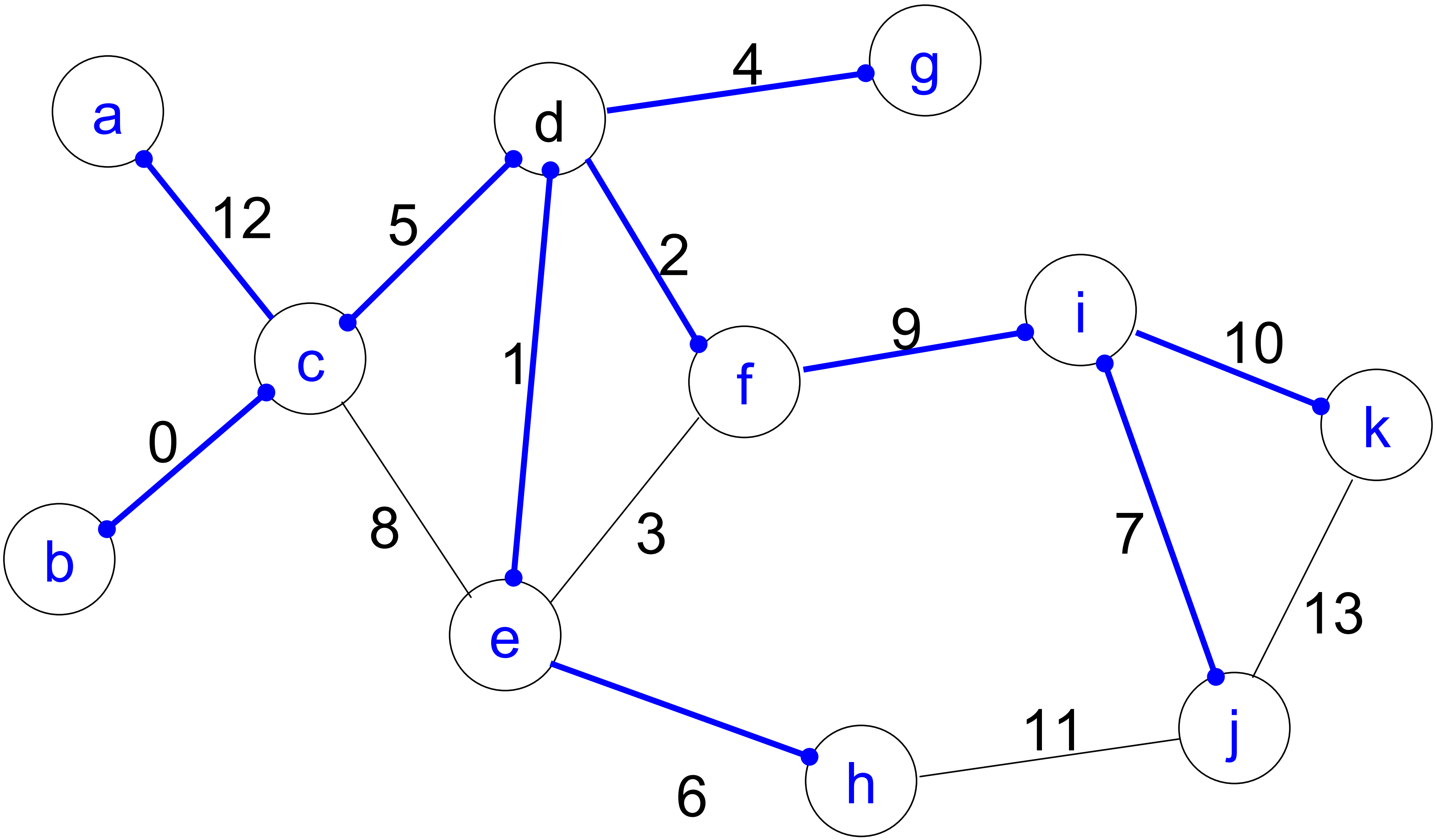
Minimum spanning tree



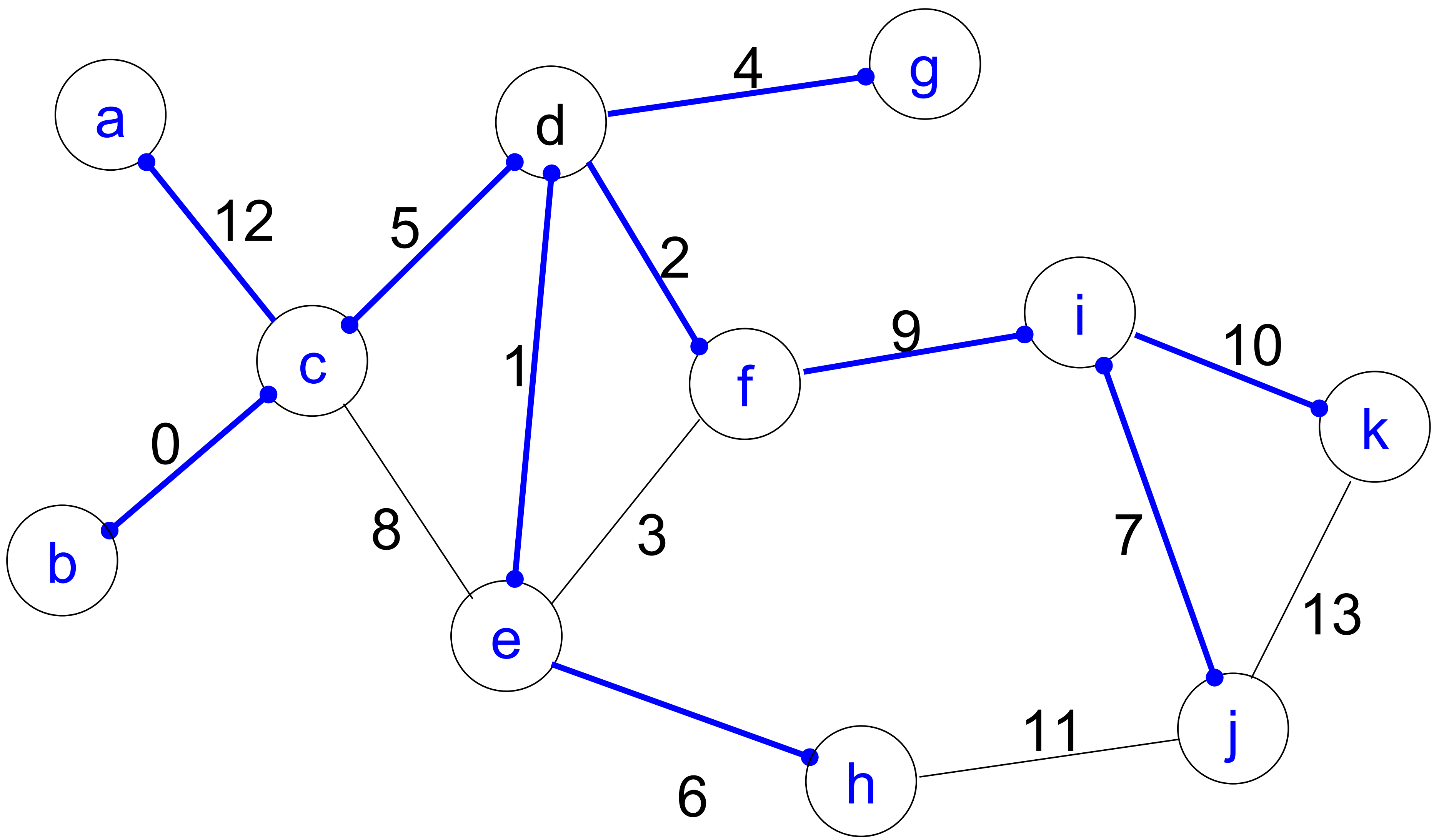
Minimum spanning tree



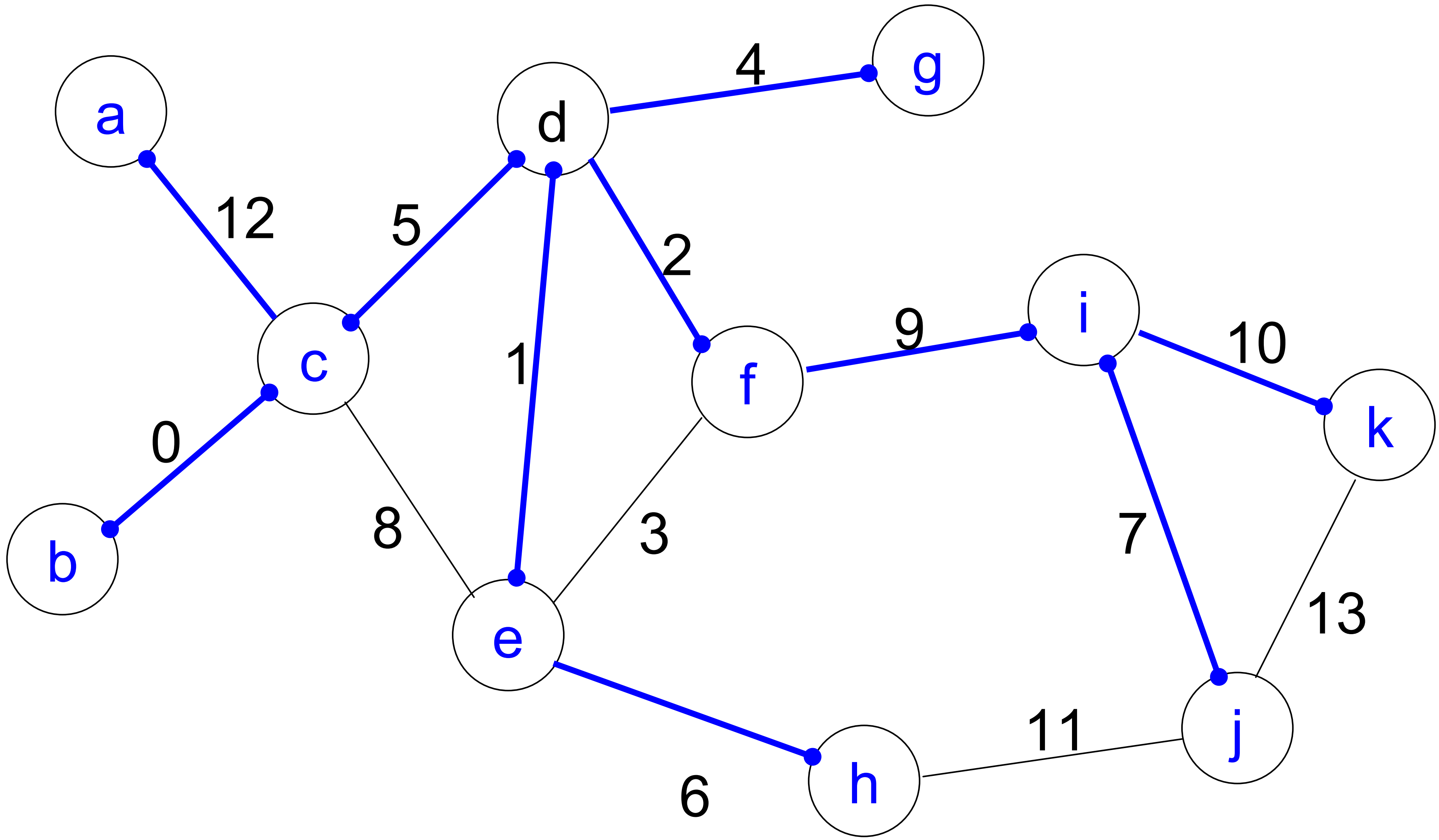
Minimum spanning tree



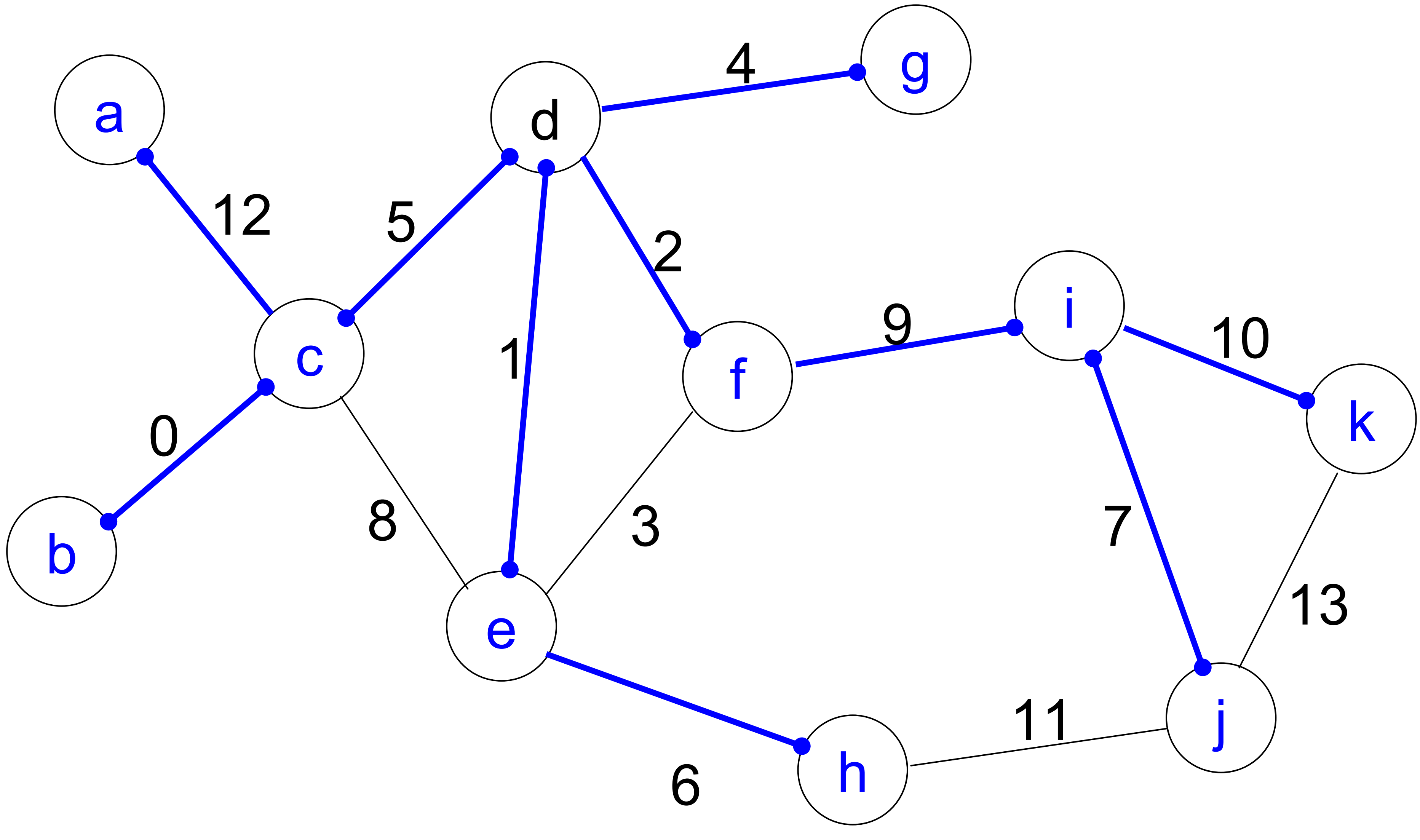
Minimum spanning tree



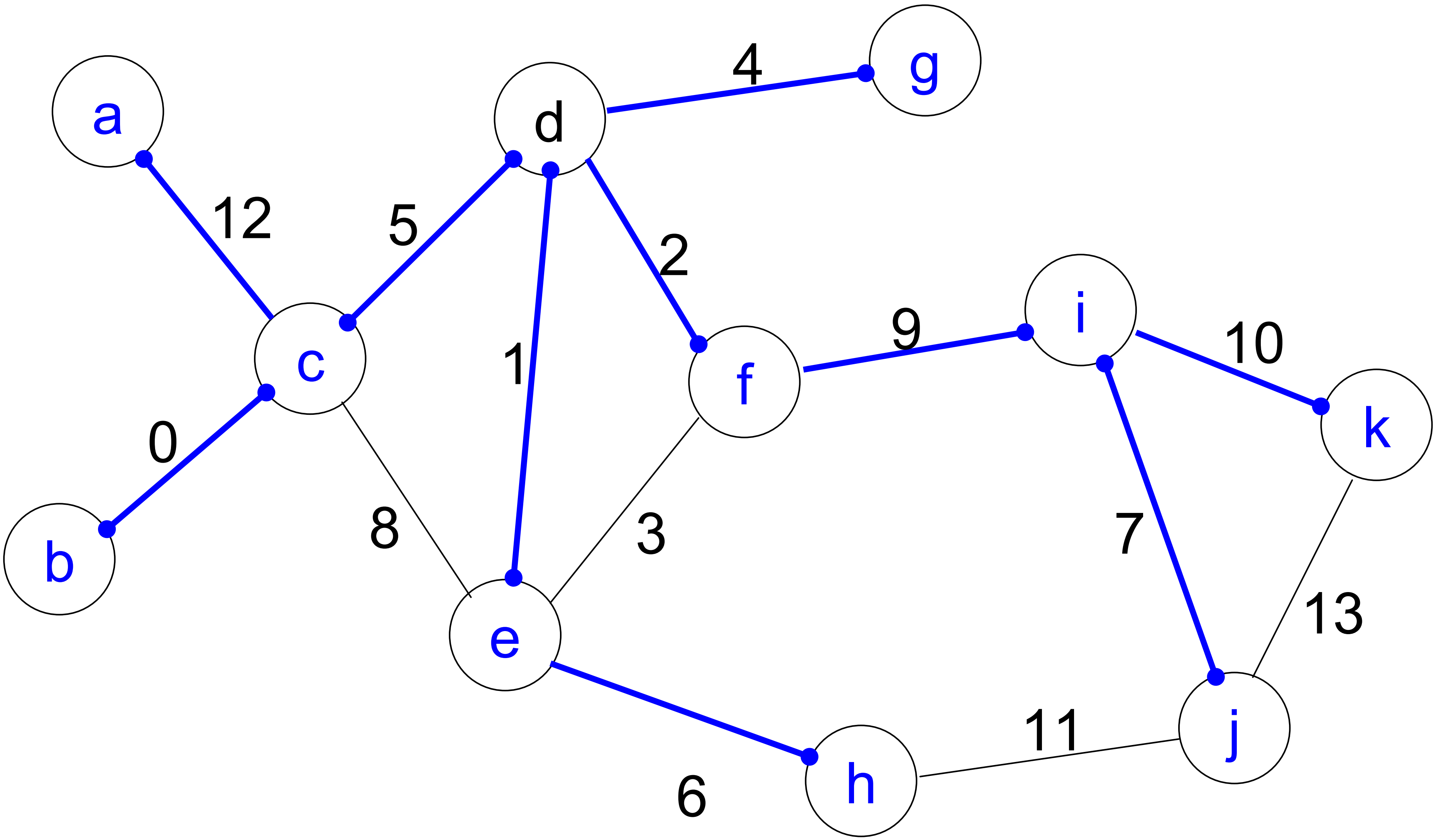
Minimum spanning tree



Minimum spanning tree



Minimum spanning tree



Simplified GHS MST Algorithm

- Proof?
- Use invariants; but this is complicated because the algorithm is complicated.
- Complexity:
  - Time: O(n log n)
    - n rounds for each level
    - log n levels, because there are $\geq 2^k$ nodes in each level k component.
  - Messages: O((n + |E|) log n)
    - Naïve analysis.
    - At each level, O(n) messages sent on tree edges, O(|E|) messages overall for all the test messages and their responses.
  - Messages: O(n log n + |E|)
    - A surprising, significant reduction.
    - Trick also works in asynchronous setting.
    - Has implications for other problems, such as leader election.

O(n log n + |E|) message complexity

- Each process marks its incident edges as rejected when they are discovered to lead to the same component; no need to retest them.
- At each level, tests candidate edges one a a time, in order of increasing weight, until the first one is found that leads outside (or exhaust candidates)
- Rejects all edges that are found to lead to same component.
- At next level, resumes where it left off.

- O(n log n + |E|) bound:
  - O(n) for messages on tree edges at each phase, O(n log n) total.
  - Test, accept (different component), reject (same component):
    - Amortized analysis.
    - Test-reject: Each (directed) edge has at most one test-reject, for O(|E|) total.
    - Test-accept: Can accept the same directed edge several times; but at most one test-accept per node per level, O(n log n) total.

Where/how did we use synchrony?

- Leader election
- Breadth-first search
- Shortest paths
- Minimum spanning tree

We will see these algorithms again in the asynchronous setting.

Spanning tree $\rightarrow$ Leader

- Given any spanning tree of an undirected graph, elect a leader:
  - Convergecast from the leaves, until messages meet at a node (which can become the leader) or cross on an edge (choose endpoint with the larger UID).
  - Complexity: Time O(n); Messages O(n)

- Given any weighted connected undirected graph, with known n, but no leader, elect a leader:
  - First use GHS MST to get a spanning tree, then use the spanning tree to elect a leader.
  - Complexity: Time O(n log n); Messages O(n log n + |E|).
  - Example: In a ring, O(n log n) time and messages.

Other graph problems…

- We can define a distributed version of practically any graph problem: maximal independent set (MIS), dominating set, graph coloring,…
- Most of these have been well studied.
- For example…

Maximal Independent Set

- Subset I of vertices V of undirected graph G = (V,E) is independent if no two G-neighbors are in V.
- Independent set I is maximal if no strict superset of I is independent.
- Distributed MIS problem:
  - Assume: No UIDs, nodes know (good upper bound on) n.
  - Required:
    - Compute an MIS I of the network graph.
    - Each process in I should output winner, others output loser.
- Application: Wireless network transmission
  - A transmitted message reaches neighbors in the graph; they receive the message if they are in "receive mode".
  - Let nodes in the MIS transmit messages simultaneously, others receive.
  - Independence guarantees that all transmitted messages are received by all neighbors (since neighbors don't transmit at the same time).
  - Neglecting collisions here---some strategy (backoff and retransmission, or coding) is needed for this.
- Unsolvable by deterministic algorithm, in some graphs.
- Randomized algorithm [Luby]:

Luby's MIS Algorithm (sketch)

- Each process chooses a random val in $\{1,2,\ldots,n^4\}$.
  - Large enough set so it's very likely that all numbers are distinct.
- Neighbors exchange vals.
- If node i's val > all neighbors' vals, then process i declares itself a winner and notifies its neighbors.
- Any neighbor of a winner declares itself a loser, notifies its neighbors.
- Processes reconstruct the remaining graph, eliminating winners, losers, and edges incident on winners and lowers.
- Repeat on the remaining graph, until no nodes are left.

- Theorem: If LubyMIS ever terminates, it produces an MIS.
- Theorem: With probability 1, it eventually terminates; the expected number of rounds until termination is O(log n).
- Proof: LTTR.

Termination theorem for Luby MIS

- Theorem: With probability 1, Luby MIS eventually terminates; the expected number of rounds until termination is O(log n).

- Proof: Key ideas
  - Define sum(i) = $\Sigma_{j \in nbrs(i)}$ 1/degree(j).
    - Sum of the inverses of the neighbors' degrees.
  - Lemma 1: In one stage of Luby MIS, for each in the graph, the probability that i is a loser (neighbor of a winner) is $\geq$ 1/8 sum(i).
  - Lemma 2: The expected number of edges removed from G in one stage is $\geq$ |E| / 8.
  - Lemma 3: With probability at least 1/16, the number of edges removed from G at a single stage is $\geq$ |E| / 16.

Next time

- Distributed consensus
- Reading: Sections 5.1, 6.1-6.3

MIT OpenCourseWare
http://ocw.mit.edu

6.852J / 18.437J Distributed Algorithms
Fall 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.