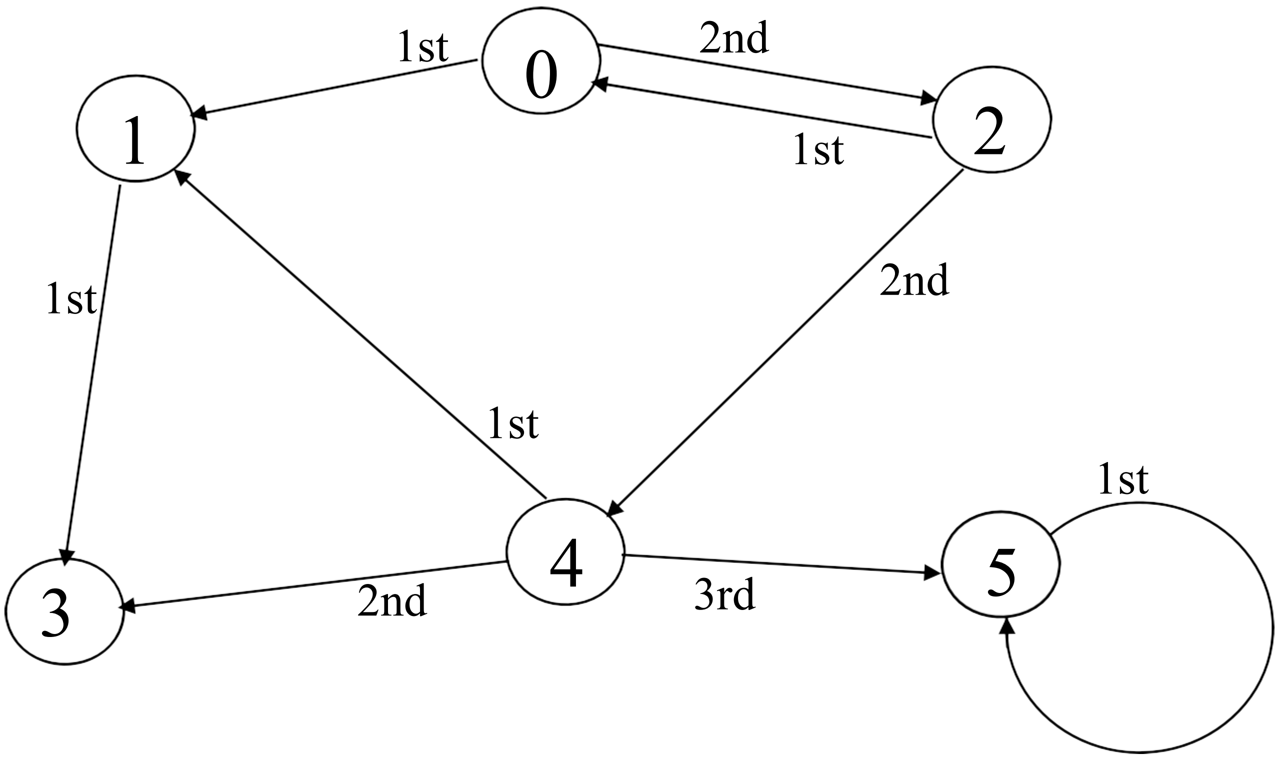
SCC120 Fundamentals of Computer Science
Unit 5: Graphs (Traversals)

Jidong Yuan
yuanjd@bjtu.edu.cn

Overview

- Depth-First Traversal
- Breadth-First Traversal
- Testing for “Strongly Connected”
- Dijkstra’s algorithm: Finding the Shortest Path in a Graph
- Finding the Minimal Spanning Tree

Depth-First Traversal: An Example



Depth-First Traversal: An E

"Consider" = push
"Back-up" = pop
"Visit" = record in order
"Mark" = record as visited

• start at node 0	OK: visit 0, mark 0	try 1st arc from 0
• consider node 1	OK: visit 1, mark 1	try 1st arc from 1
• consider node 3	OK: visit 3, mark 3	no arc from 3
•	back-up to 1	no other arc from 1
•	back-up to 0	try 2nd arc from 0
• consider node 2	OK: visit 2, mark 2	try 1st arc from 2
• consider node 0	0 already marked	try 2nd arc from 2
• consider node 4	OK: visit 4, mark 4	try 1st arc from 4
• consider node 1	1 already marked	try 2nd arc from 4
• consider node 3	3 already marked	try 3rd arc from 4
• consider node 5	OK: visit 5, mark 5	try 1st arc from 5
• consider node 5	5 already marked	no other arc from 5
•	back-up to 4	no other arc from 4
•	back-up to 2	no other arc from 2
•	back-up to 0	no other arc from 0
• Exit		

Depth-First Traversal: An Example

- The order of visitation is:
  - 0, 1, 3, ↑ ↑ 2, 4, 5, ↑ ↑ ↑

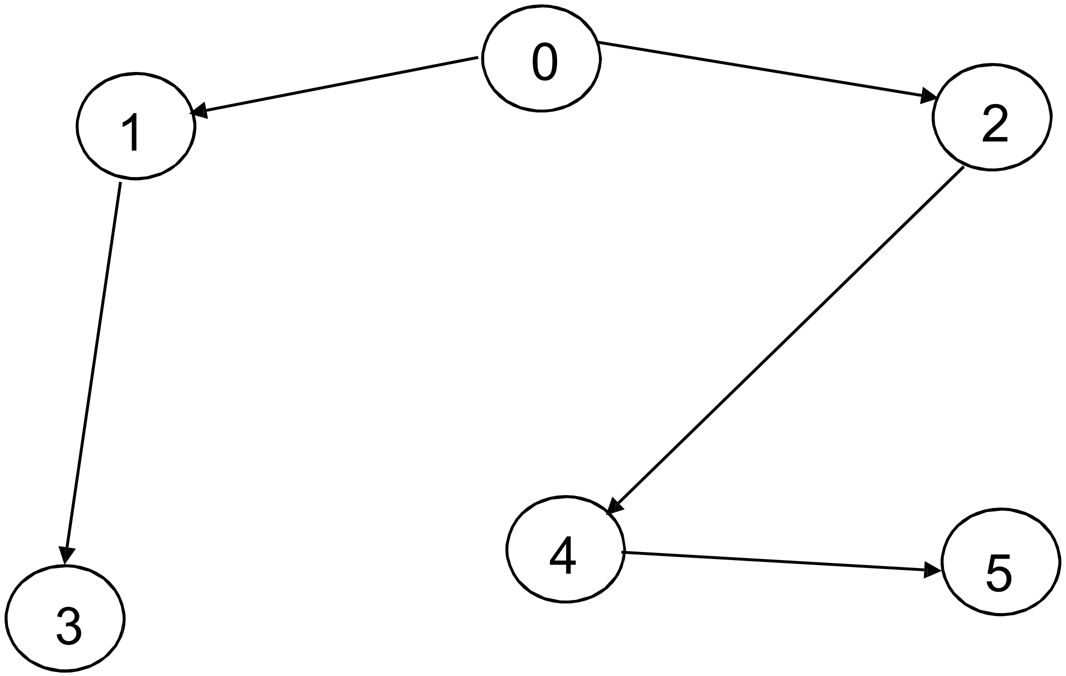
- ↑ represents “back-up” i.e. retracing your steps

Depth-First Traversal: Comments

- In defining a traversal, the edges out of any given node must be taken in some specific (though arbitrary) order
- In our graph, each edge out of a particular node is labelled 1st, 2nd, or 3rd
- If these labels are changed, the order of visitation may be changed, but the same set of nodes will ultimately be visited
- Need to use a **stack** to remember the nodes

Spanning Tree

- If during the traversal process, we “highlight” each movement along an edge to an *unvisited* node, we obtain a reduced graph which contains no loops



Spanning Tree

- This is known as a *spanning tree* for the graph, rooted at the start node of the traversal (here 0)
- “Spanning” here means “visiting every node of the graph”

Depth-First Traversal: Algorithm

```
void depthFirstTraversal(Graph G, Node N)
{
    visitNode(N);
    record visit to node N;
    for each node X attached to N
        if X has not been visited
            depthFirstTraversal(G, X);
} // end of method depthFirstTraversal
```

Depth-First Traversal: Comments

- This algorithm is *recursive*
- If there is something we need to do at each node, it can be done in the method *visitNode*
  - e.g. to test, count, update or output the value of the node
- The algorithm doesn't specify how the graph is represented, nor how the visits are recorded

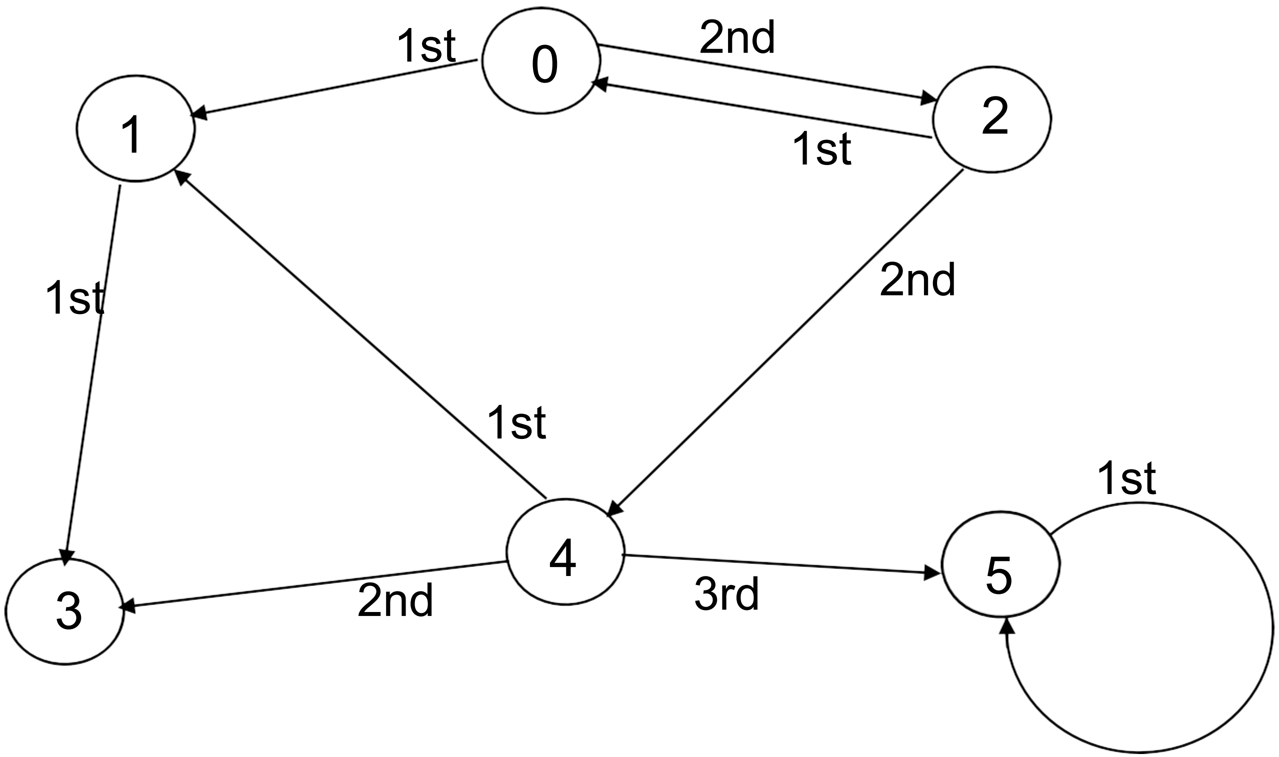
Depth-First Traversal: Comments

- This involves the hidden use of a **stack**, the runtime stack, to record
  - where we have got to in the depth-first traversal
  - how we got there
  - and where else we have to visit
- The algorithm could be rewritten in an iterative form, but would then have to include an explicit stack to record this information

Overview

- Depth-First Traversal
- Breadth-First Traversal
- Testing for “Strongly Connected”
- Dijkstra’s algorithm: Finding the Shortest Path in a Graph
- Finding the Minimal Spanning Tree

Breadth-First Traversal: An Example



Breadth-First Traversal: Example

"Add", "remove" from queue
"Visit" = record in order
"Mark" = record as visited

start at node 0	visit 0, mark 0, add 0	
	remove 0	try 1st arc from 0
consider node 1	visit 1, mark 1, add 1	try 2nd arc from 0
consider node 2	visit 2, mark 2, add 2	no other arc from 0
	remove 1	try 1st arc from 1
consider node 3	visit 3, mark 3, add 3	no other arc from 1
	remove 2	try 1st arc from 2
consider node 0	0 already visited	try 2nd arc from 2
consider node 4	visit 4, mark 4, add 4	no other arc from 2
	remove 3	no arc from 3
	remove 4	try 1st arc from 4
consider node 1	1 already visited	try 2nd arc from 4
consider node 3	3 already visited	try 3rd arc from 4
consider node 5	visit 5, mark 5, add 5	no other arc from 4
	remove 5	try 1st arc from 5
consider node 5	5 already visited	no other arc from 5
	queue empty	exit

Breadth-First Traversal: An Example

Order of visitation is

0, 1, 2, 3, 4, 5

Breadth-First Traversal

- We visit nodes of increasing distance from the start node (i.e. nodes of distance 1, nodes of distance 2, …, and so on)
- Use a **queue** to remember recently visited nodes (we use a queue to allow the algorithm to return to a node after visiting the other same-distance nodes)

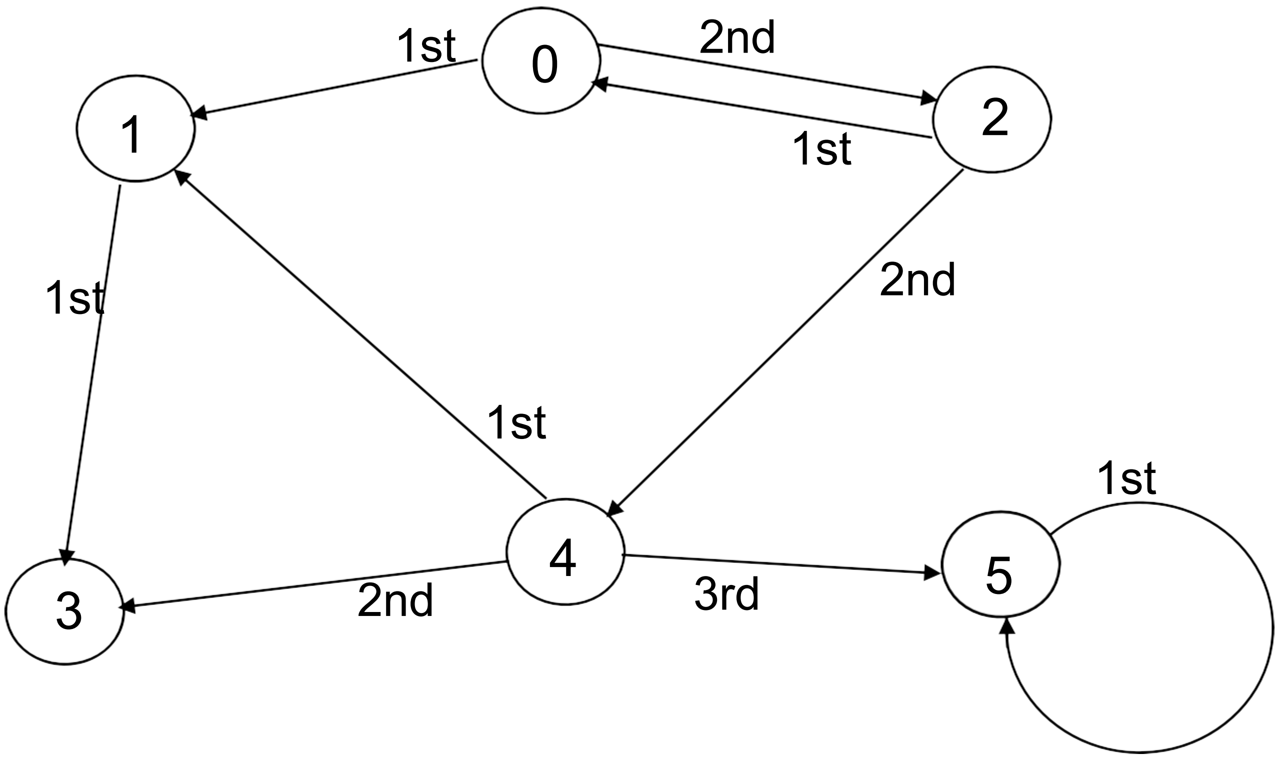
Breadth-First Traversal: Algorithm

```
void breadthFirstTraversal(Graph G, node N)
{
    Queue Q = new Queue();
    visitNode(N);     record visit to N;    Q.add(N);
    while (!Q.isEmpty()) {
        node X = Q.remove();
        for each unvisited node W attached to X {
            visitNode(W);  record visit to W;  Q.add(W);
        }
    }
} // end of method breadthFirstTraversal
```

Overview

- Depth-First Traversal
- Breadth-First Traversal
- Testing for “Strongly Connected”
- Dijkstra’s algorithm: Finding the Shortest Path in a Graph
- Finding the Minimal Spanning Tree

Is it Strongly Connected?



Is it Strongly Connected?

- A directed graph is *strongly connected* if we can travel from each node to any other node
- In the graph G:
  - depthFirstTraversal(G, 0) visits the other five nodes
  - depthFirstTraversal(G, 2) visits the other five nodes
  - depthFirstTraversal(G, 1) visits only node 3
  - and for nodes 3, 4, and 5, depthFirstTraversal doesn’t visit all nodes
  - so G is not strongly connected

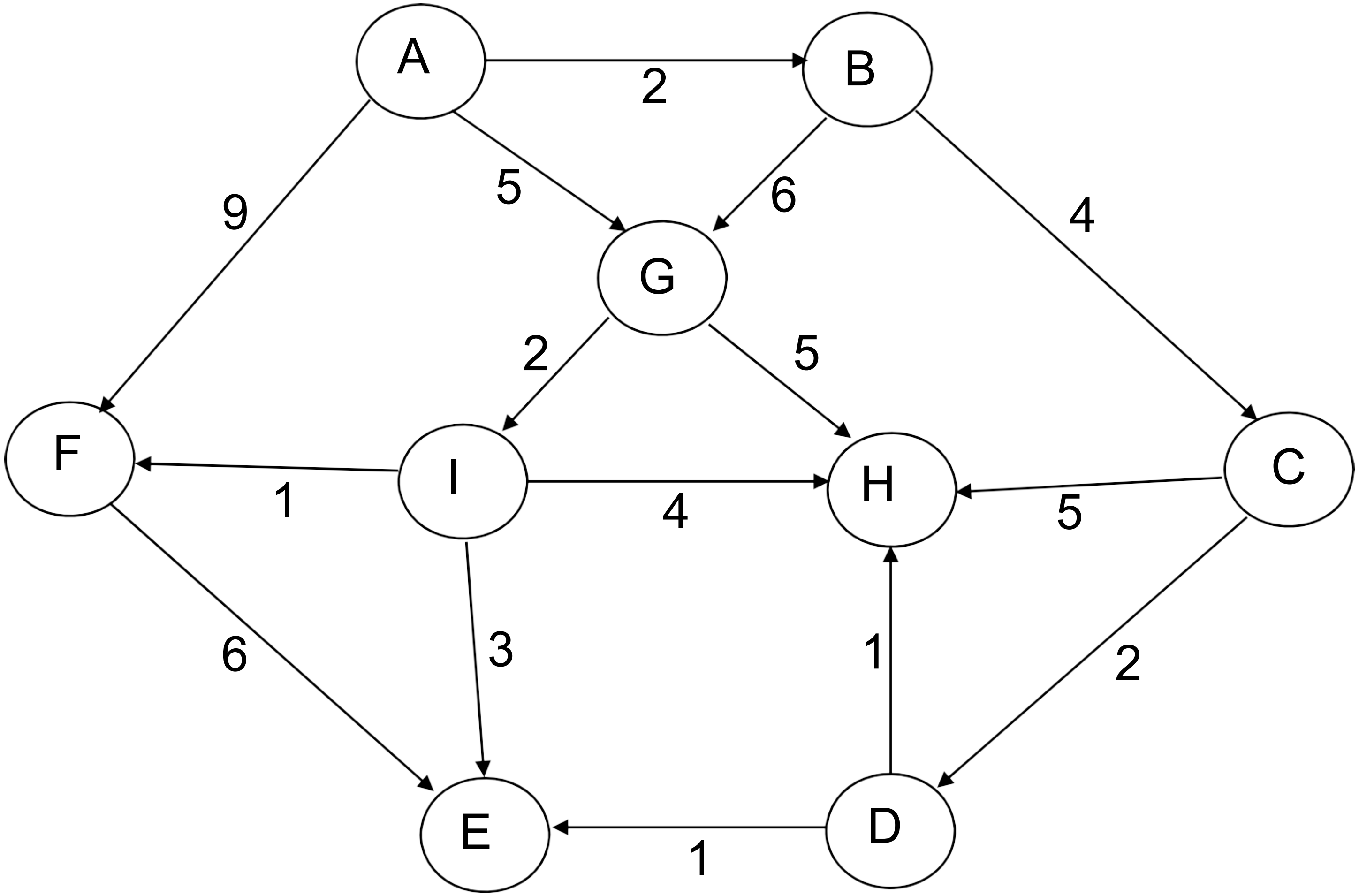
An Algorithm for Testing for "Strong Connection"

- This gives us an algorithm for checking whether a graph is strongly connected
  - We do a traversal from each node in the graph in turn
  - We check that each traversal is complete; that is, visits all the other nodes

Overview

- Depth-First Traversal
- Breadth-First Traversal
- Testing for “Strongly Connected”
- Dijkstra’s algorithm: Finding the Shortest Path in a Graph
- Finding the Minimal Spanning Tree

Finding the Shortest Path in a Graph



Finding the Shortest Path in a Graph

- We have a directed graph with values (distances, weights, costs) associated with each edge
- The distance (cost) of a path between two nodes is the sum of the values of the edges on the path
- *Dijkstra's* algorithm: finds the shortest path between two nodes

Dijkstra’s Algorithm

- We divide the nodes of the graph into three groups

  A: those which have been added to the tree of nodes whose shortest path has already been found

  - the “tree”

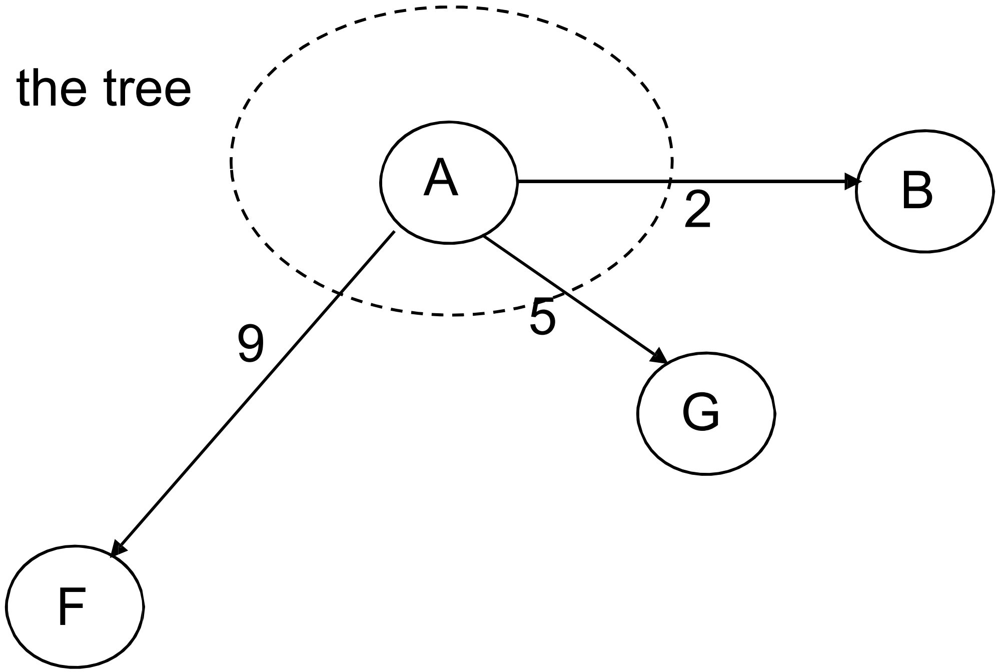
  B: those not in group A, but connected by an edge to some node in group A

  - the “fringe”

  C: all the other nodes

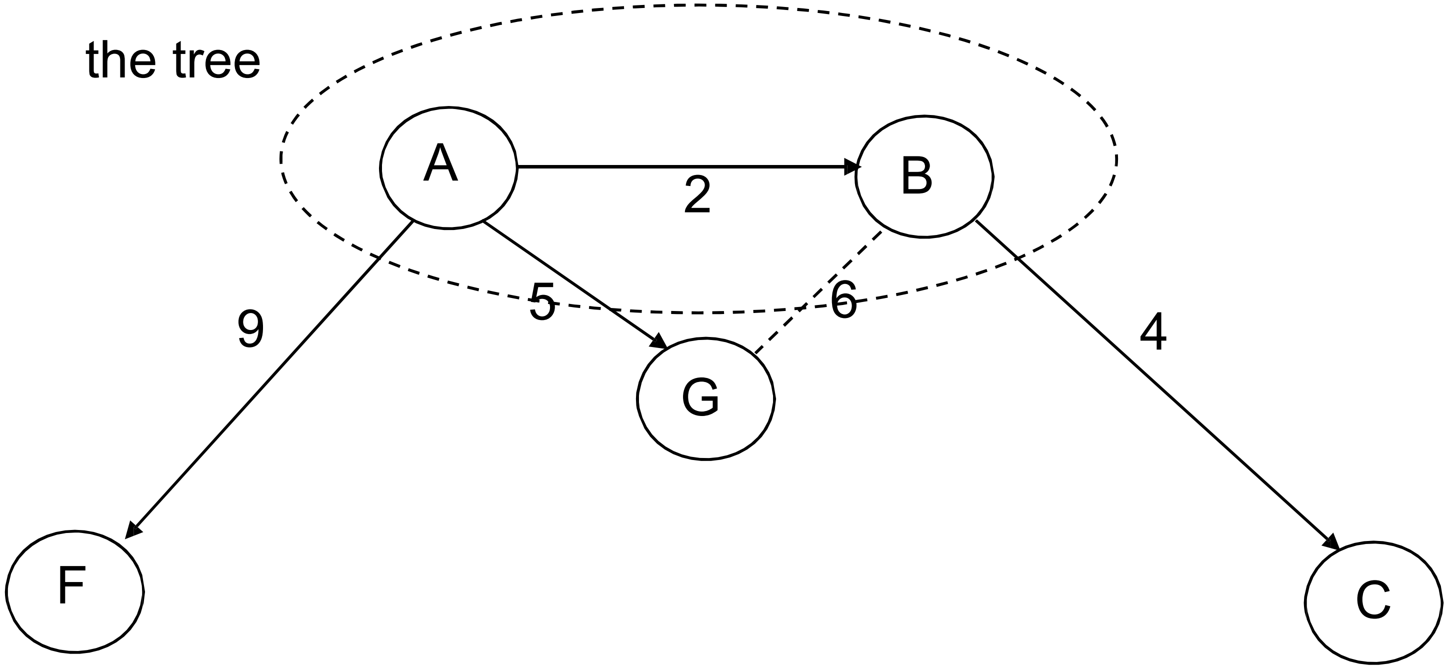
  - the “unseen” nodes

Example (shortest path from A to H)



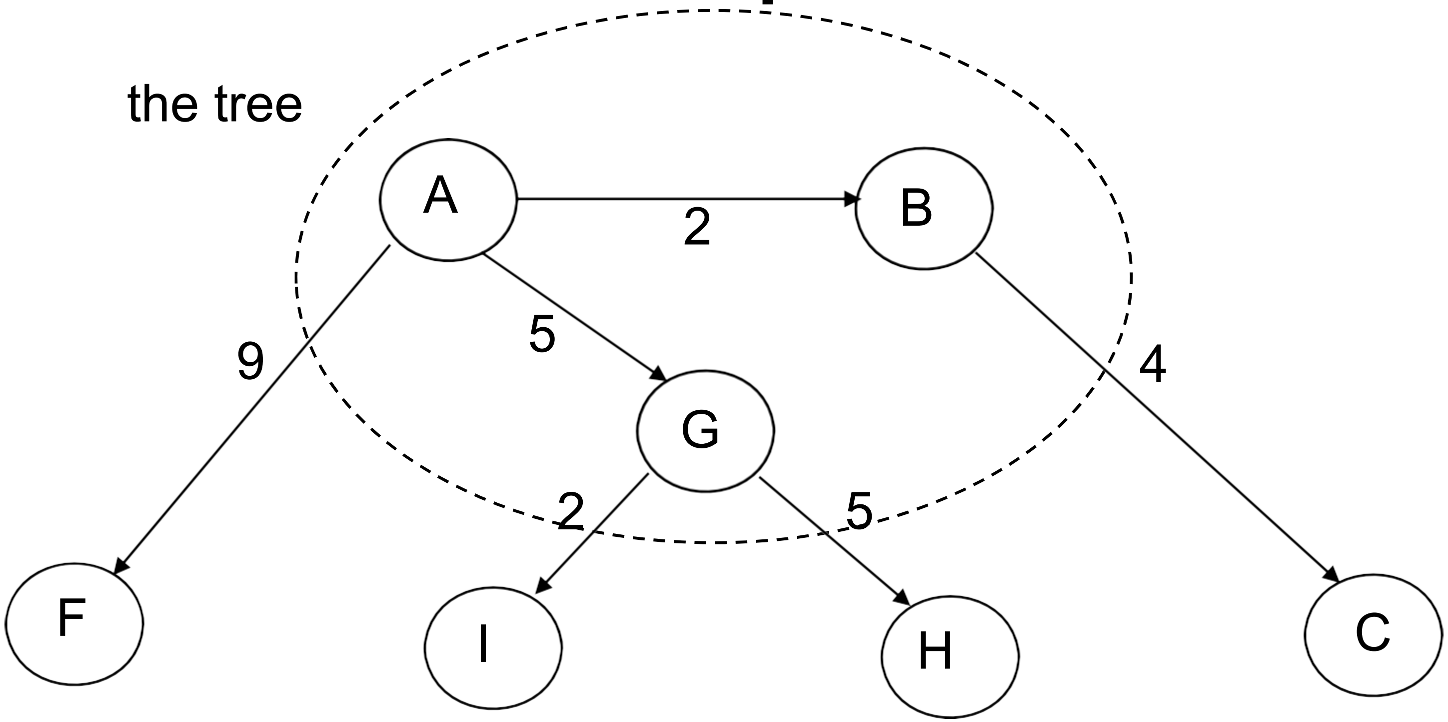
- tree {A}; fringe {B (2), F (9), G (5)}; choose (A, B)

Example



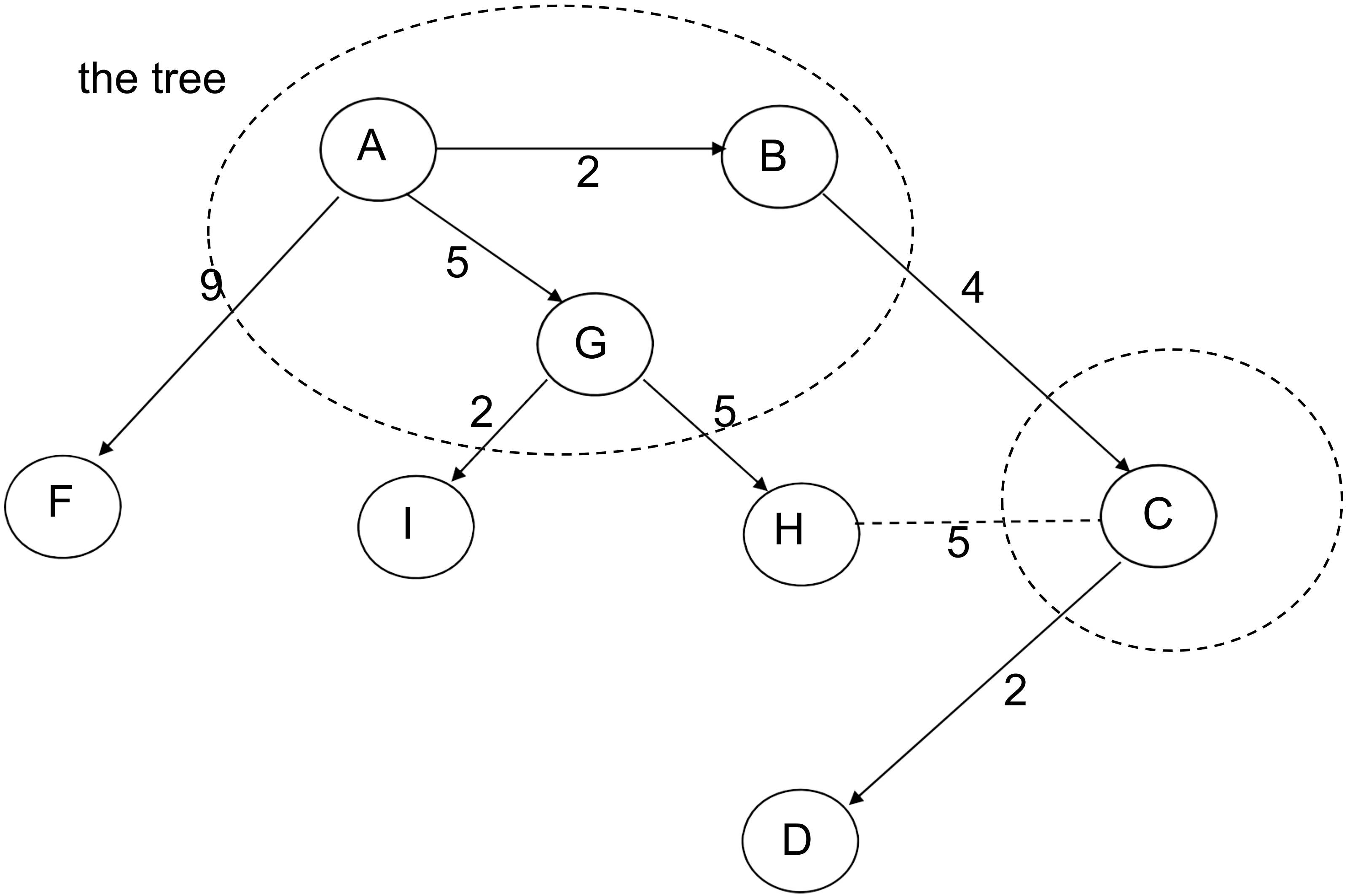
- tree {A, B}; fringe {C (6), F (9), G (still 5)}; choose (A, G)

Example



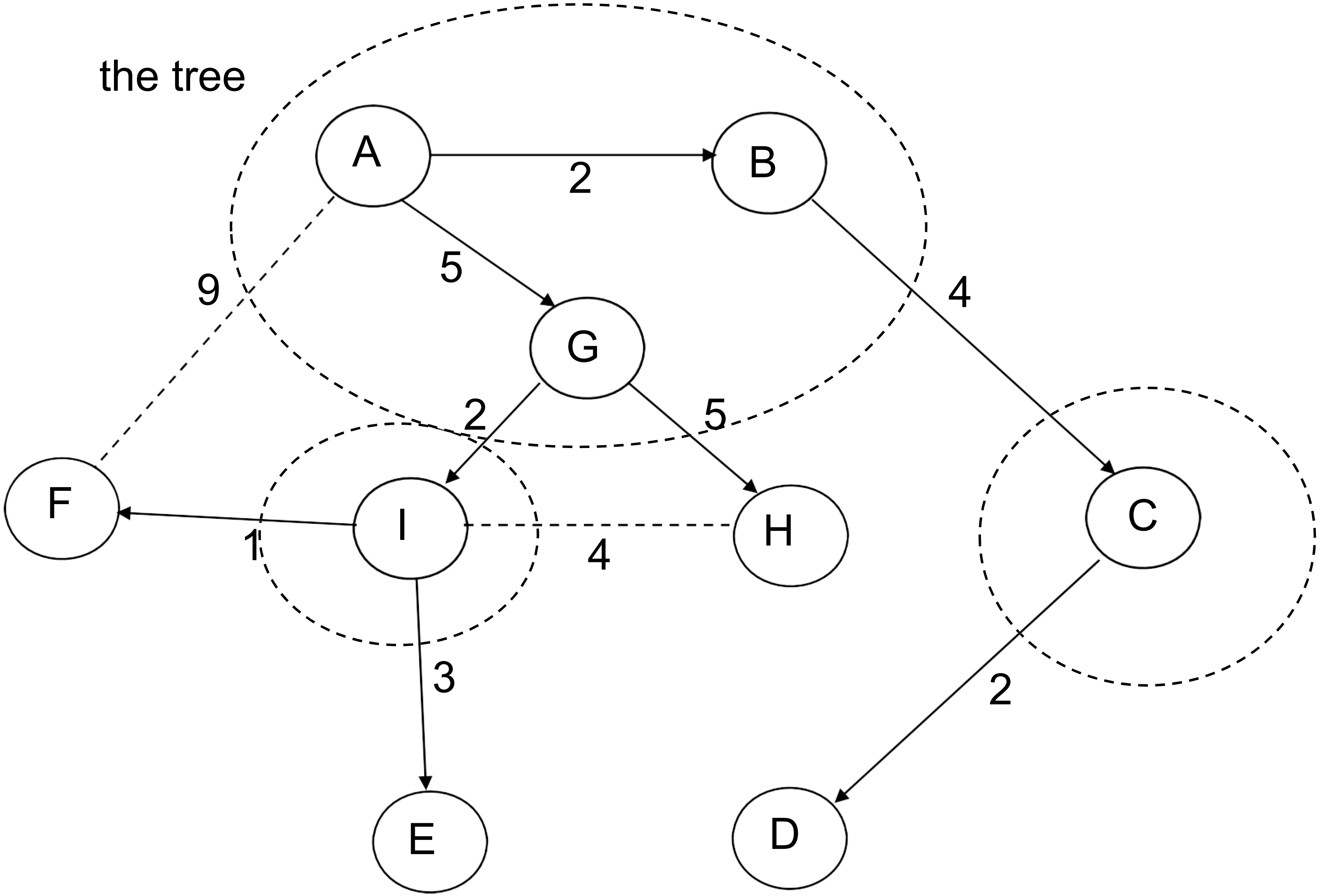
- tree {A, B, G}; fringe {C (6), F (9), H (10), I (7)}; choose (B, C)

Example



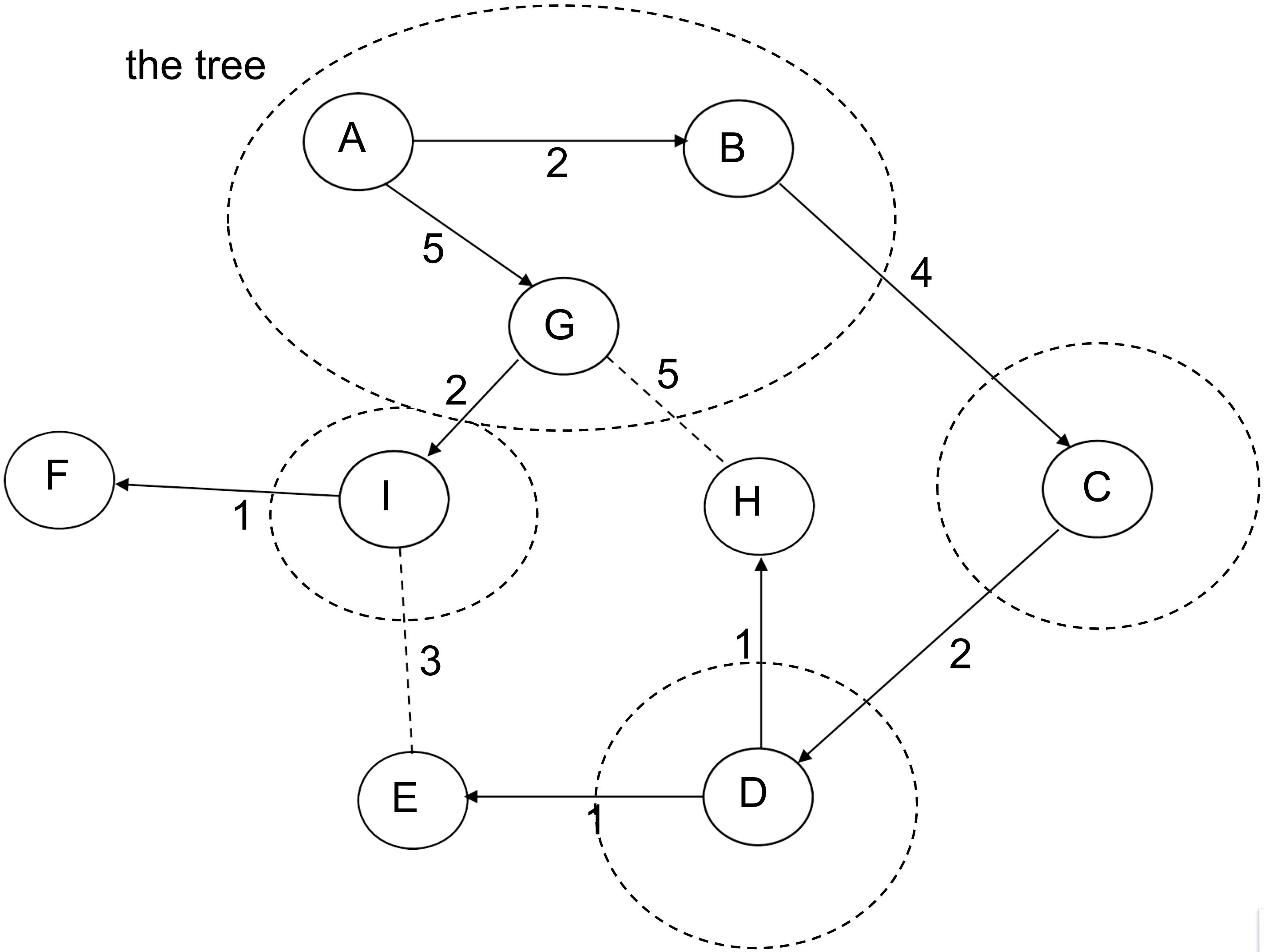
- tree {A, B, C, G}; fringe {D (8), F (9), H (still 10), I (7)}; choose (G, I)

Example



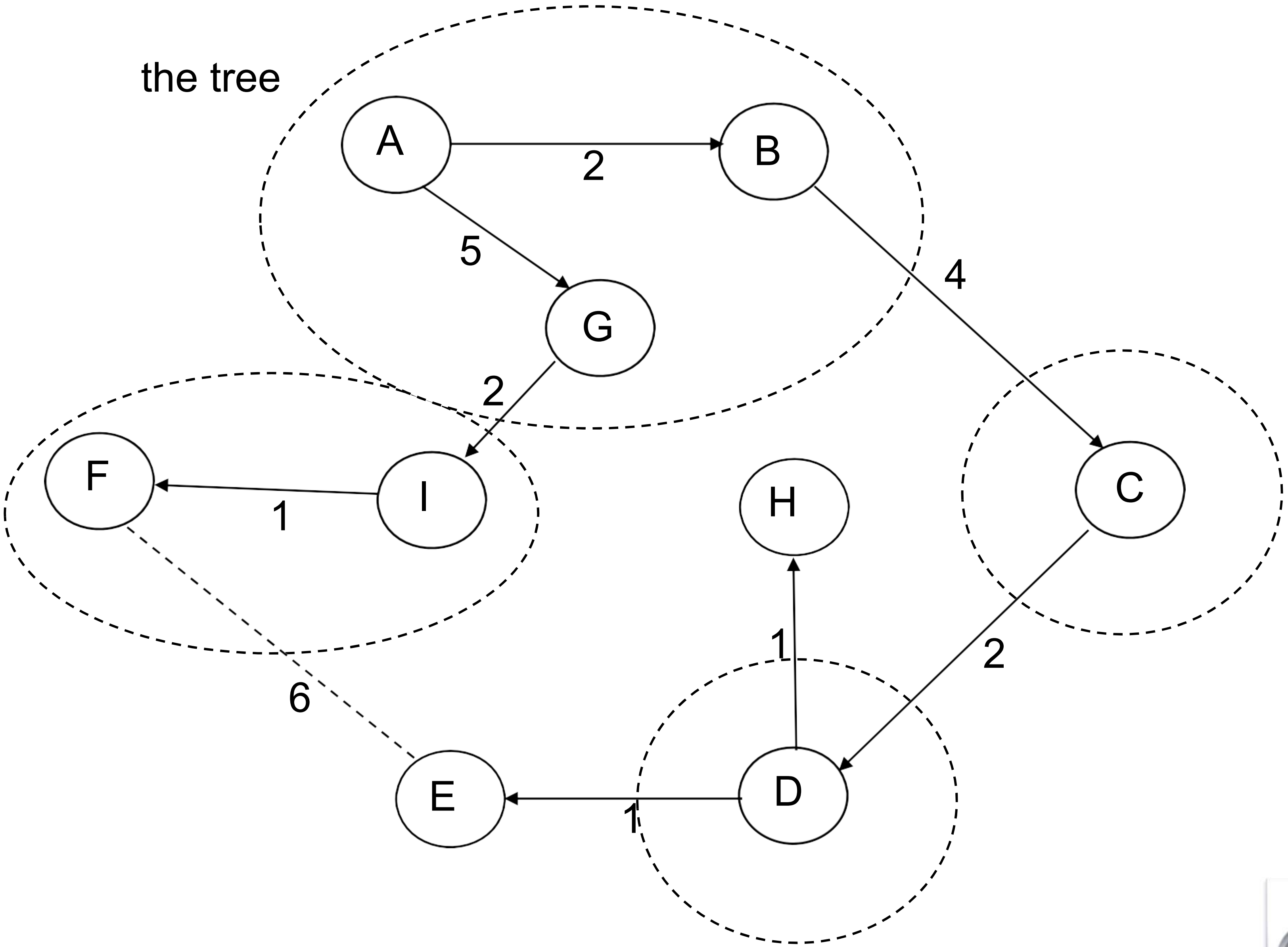
- tree {A, B, C, G, I}; fringe {D (8), E (10), F (now 8), H (still 10)}; choose (C, D) or (I, F) - we'll choose (C, D)

Example



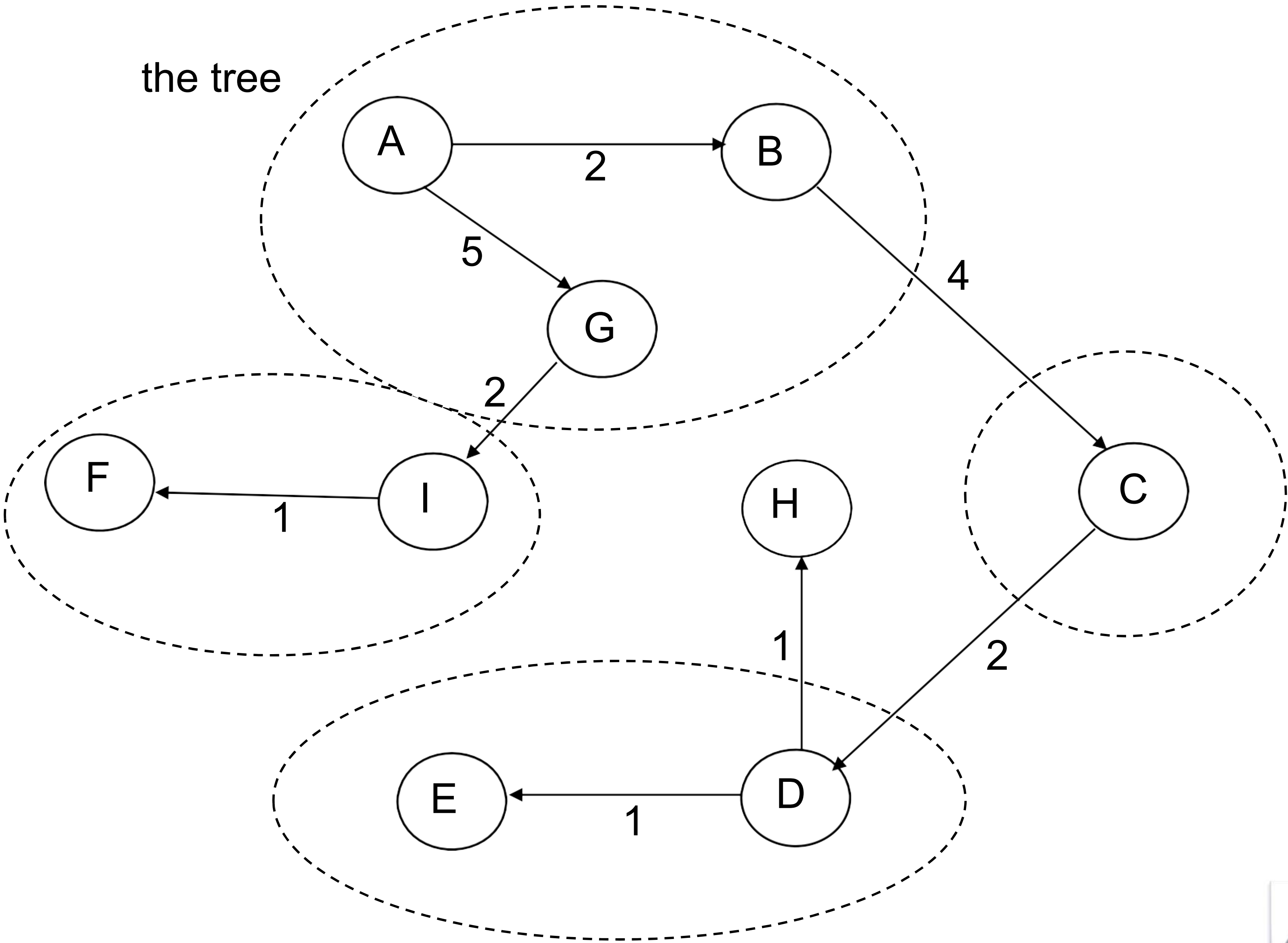
- tree {A, B, C, D, G, I}; fringe {E (now 9), F (8), H (now 9)}; choose (I, F)

Example



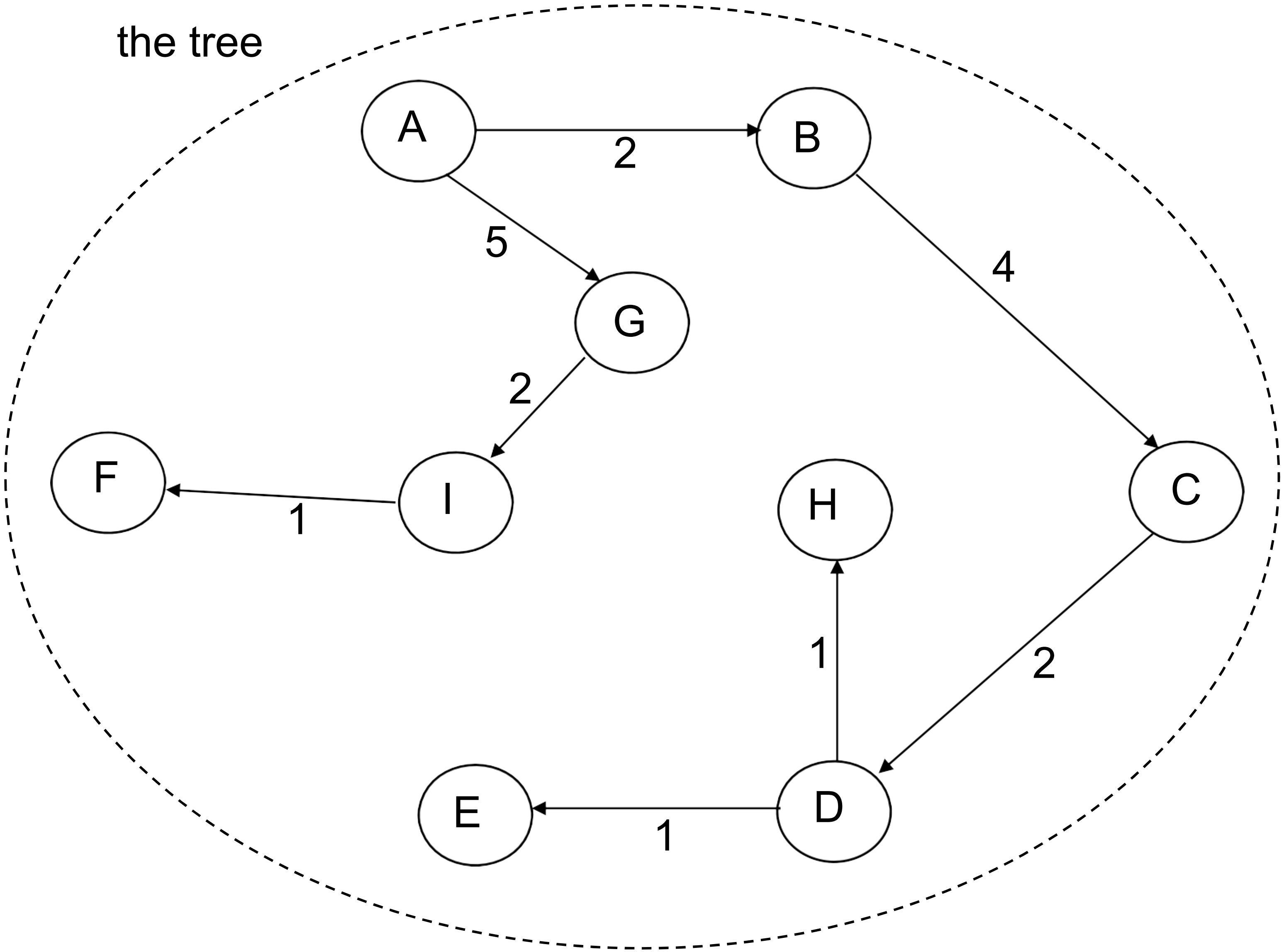
- tree {A, B, C, D, F, G, I}; fringe {E (still 9), H (9)}; choose (D, E) or (D, H) - we'll choose (D, E)

Example



- tree {A, B, C, D, E, F, G, I}; fringe {H (9)}; choose (D, H)

Example



- tree {A, B, C, D, E, F, G, H, I}; fringe { }

Solution Found

- So we stop, because the specified end node H is now part of the tree
- We could carry on and find shortest paths to all the rest of the nodes
  - in this case there are no more
- We now know that there is at least one path from A to H, and the shortest one has length 9, but we don't know what the path is
- When we do the algorithm, whenever we add an edge to a node, we must remember where it came from

Remember where we came from

- When we added (A, B), we remember parent(B) = A
- When we added (A, G), we remember parent(G) = A
- When we added (B, C), we remember parent(C) = B
- When we added (G, I), we remember parent(I) = G
- When we added (C, D), we remember parent(D) = C
- When we added (I, F), we remember parent(F) = I
- When we added (D, E), we remember parent(E) = D
- When we added (D, H), we remember parent(H) = D

Solution

So the shortest path from A to H is (in reverse):

H

parent(H) = D

parent(D) = C

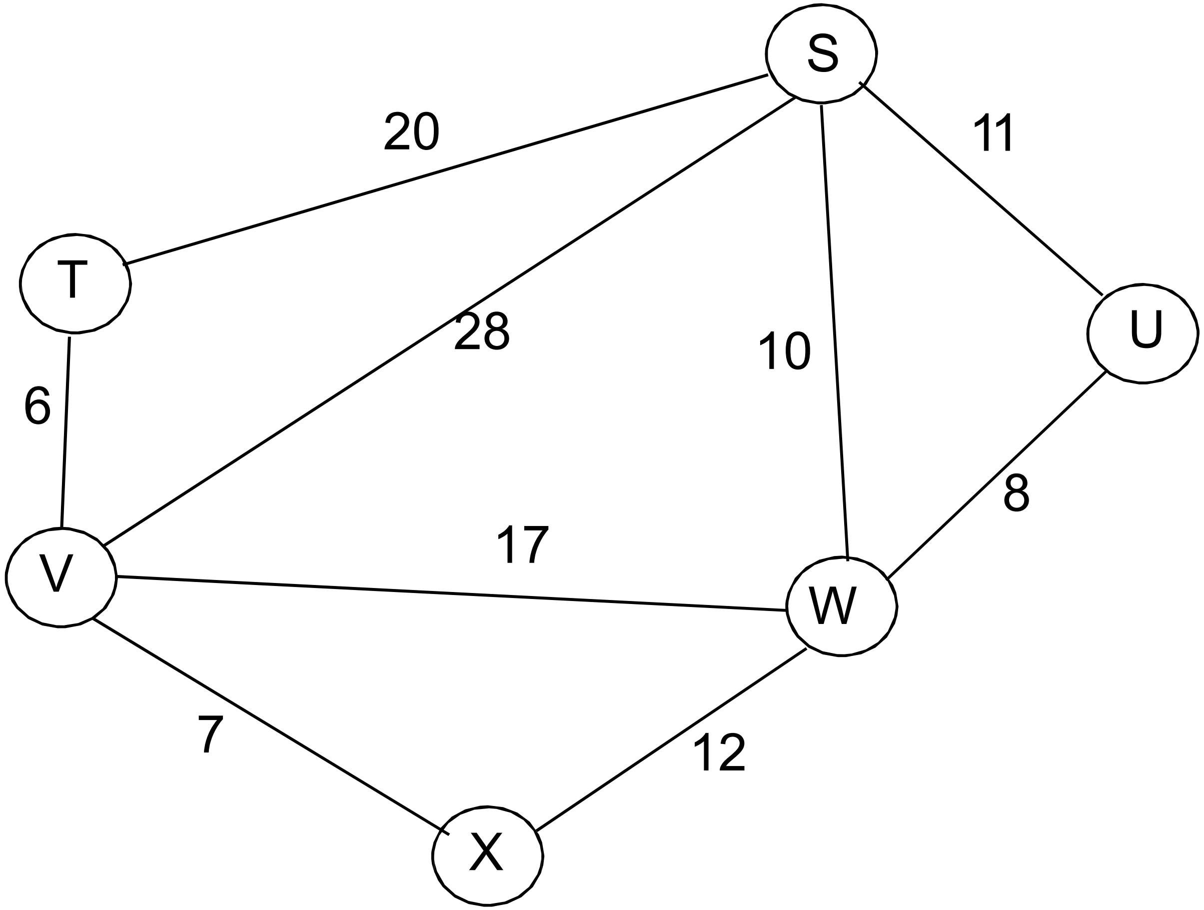
parent(C) = B

parent(B) = A

Overview

- Depth-First Traversal
- Breadth-First Traversal
- Testing for “Strongly Connected”
- Dijkstra’s algorithm: Finding the Shortest Path in a Graph
- Finding the Minimal Spanning Tree

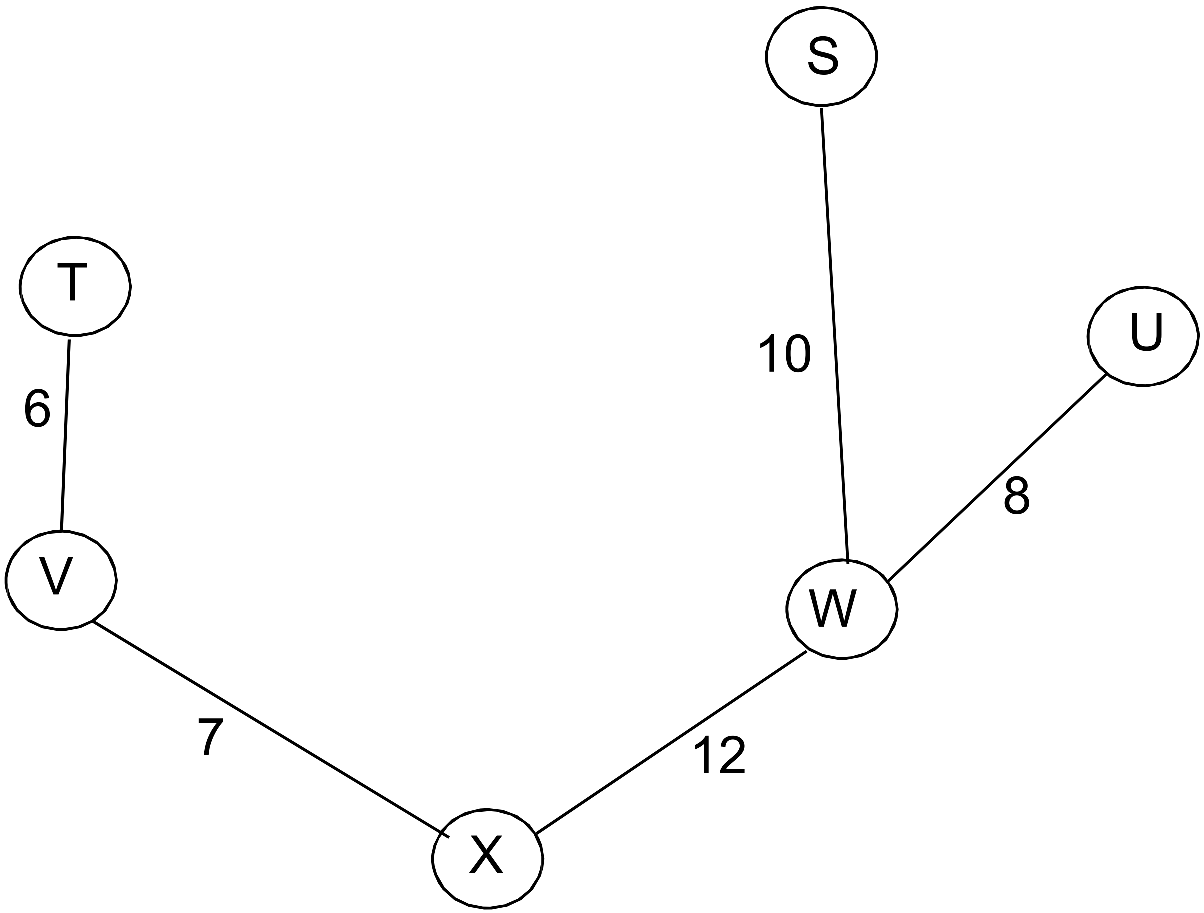
Finding the Minimal Spanning Tree



Steps for finding Minimal Spanning Tree

- select (T, V), cost 6 add to Tree
- select (V, X), cost 7 add to Tree
- select (U, W), cost 8 add to Tree
- select (S, W), cost 10 add to Tree
- select (S, U), cost 11 reject: loop (S, U, W)
- select (W, X), cost 12 add to Tree
- terminate as we have added 5 = (6 - 1) edges

Minimal Spanning Tree



A Minimal Spanning Tree Algorithm

- We start by defining an “empty Tree”
  - This is going to end up as the minimal spanning tree
  - This contains all the nodes of G
  - But (initially) none of the edges of G

A Minimal Spanning Tree Algorithm

- We then (repeatedly) remove the cheapest (lowest-valued) edge from G and add it to Tree (provided that it does not create a loop in Tree)
- We do this until we have added N-1 edges to the Tree, after which we stop (as any further edges are unnecessary)

Testing for the Loop

- How do we test if the edge to be added will not create a loop in the tree?

- One way would be to start at one of the ends of the edge and see it you can reach the other end of the edge by travelling only through edges already added to the tree
  - but this could be expensive

Testing for the Loop

• initially	S | T | U | V | W | X	6 components
• add (T, V)	S | T, V | U | W | X	5 components
• add (V, X)	S | T, V, X | U | W	4 components
• add (U, W)	S | T, V, X | U, W	3 components
• add (S, W)	S, U, W | T, V, X	2 components
• reject (S, U)	in the same component	
• add (W, X)	S, T, U, V, W, X	1 component

SCC120 ADT (weeks 5-10)

- Week 5 Abstractions; Set
  Stack
- Week 6 Queues
  Priority Queues
- Week 7 Graphs (Terminology)
  Graphs (Traversals)
- Week 8 …
- Week 9
- Week 10