

INFDEV026A - Algoritmiek

Unit 5

G. Costantini, F. Di Giacomo, G. Maggiore

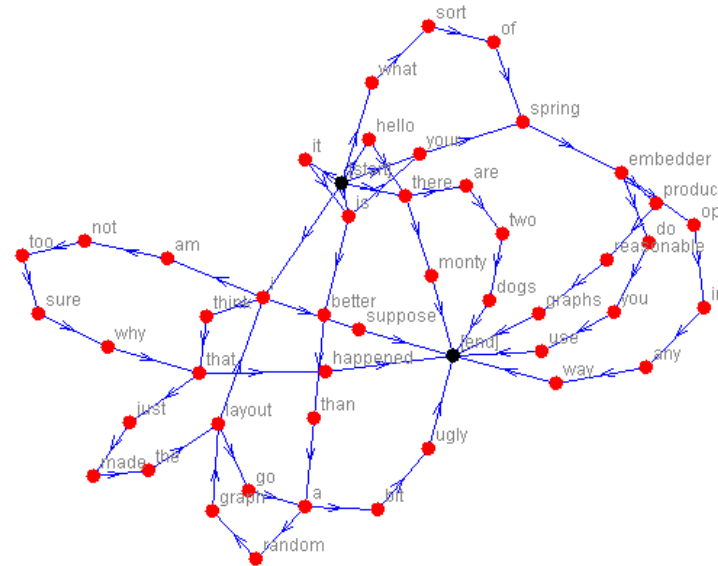
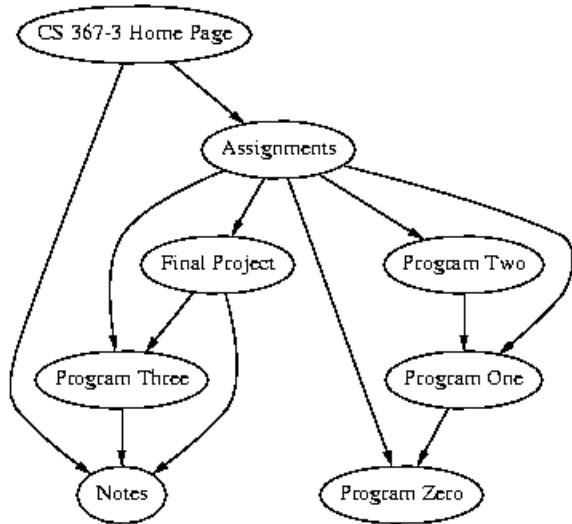
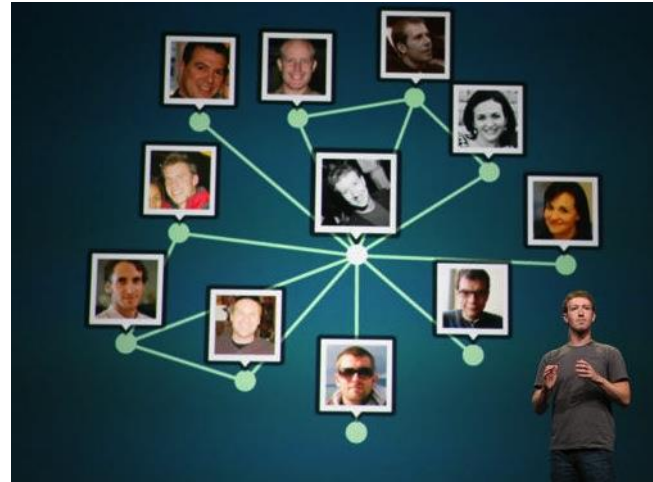
costg@hr.nl, giacf@hr.nl, maggg@hr.nl - Office H4.204

Today

- ▶ ~~Why is my code slow?~~
 - ▶ ~~Empirical and complexity analysis~~
- ▶ ~~How do I order my data?~~
 - ▶ ~~Sorting algorithms~~
- ▶ ~~How do I structure my data?~~
 - ▶ ~~Linear, tabular, recursive data structures~~
- ▶ How do I represent relationship networks?
 - ▶ Graphs

More detailed agenda

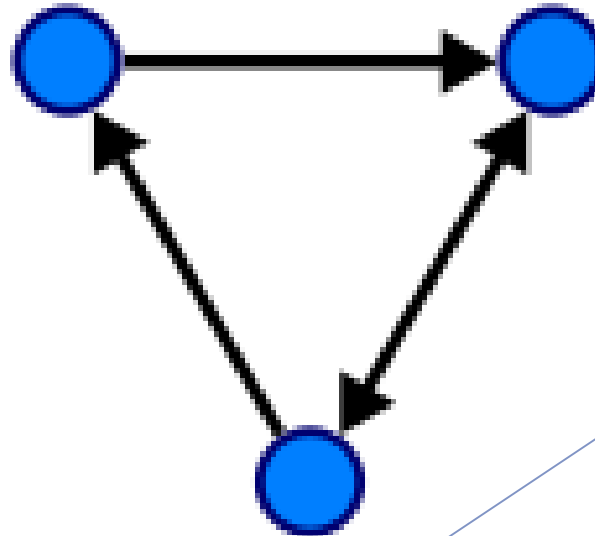
- ▶ What are (di)graphs?
- ▶ How do we represent a (di)graph?
 - ▶ Adjacency list, adjacency matrix, incidence matrix
- ▶ How can we traverse/visit a graph?
 - ▶ BFS, DFS
- ▶ How can we find the shortest path between two nodes of a graph?
 - ▶ Dijkstra's algorithm



Graphs

Graphs - Definition

- ▶ Nonlinear structure made by
 - ▶ finite (and possibly mutable) set of *nodes* or *vertices*
 - ▶ set of ordered/unordered pairs of these nodes, known as *edges* or *arcs*
 - ▶ edge (x, y) is said to **point** or **go from** x to y
 - ▶ may also associate to each edge some edge *value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.)



Graphs - Definition

► Simple graph \rightarrow pair $G = (V, E)$ where

- V and E are finite sets
- V = vertices (nodes); E = edges (arcs)
- every element of E is a two-element subset of V

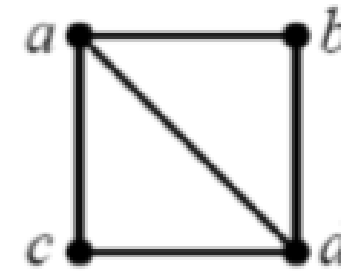
► Graph size \rightarrow # elements of $V \rightarrow |V|$

► Given an edge $e = \{a, b\} = ab = ba$

- e connects/is incident with the two vertices a and b
- a and b are adjacent/incident upon e /the terminal points of e

► Path from a to $b \rightarrow$ sequence of edges which form a chain of connected vertices from a to b , with all distinct vertices

► length of a path = number of edges forming the path



$$V = \{a, b, c, d\}$$

$$E = \{ab, ac, ad, bd, cd\}$$

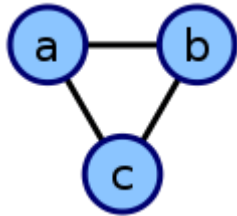
Graphs - Representations

- ▶ Possible data structures for the representation of graphs
 - ▶ **Adjacency list**
 - ▶ Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices
 - ▶ **Adjacency matrix**
 - ▶ A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices
 - ▶ **Incidence matrix**
 - ▶ A two-dimensional matrix, in which the rows represent the vertices and columns represent the edges

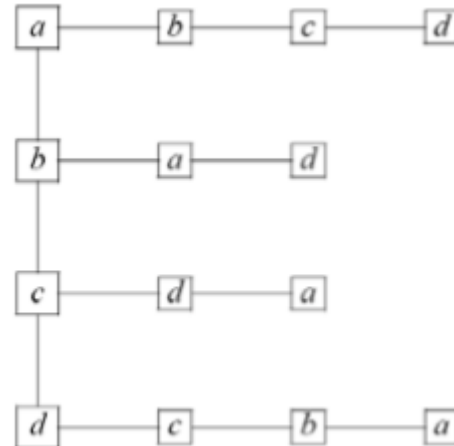
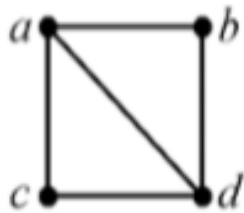
Graphs - Representations

► Adjacency list

- collection of unordered lists, one for each vertex in the graph
- each list describes the set of neighbors of its vertex



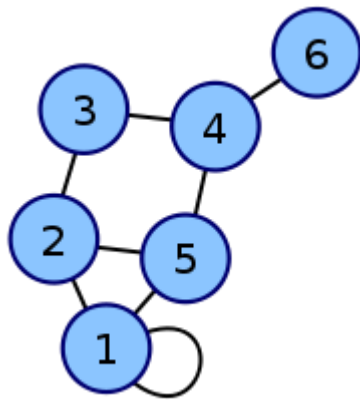
a	adjacent to	b,c
b	adjacent to	a,c
c	adjacent to	a,b



Graphs - Representations

► Adjacency matrix

- represents which vertices of a graph are adjacent to which other vertices
- rows and columns represent both the vertices
- given a cell at row $i \rightarrow$ column j is *True(1)* if there is an edge connecting i to j



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Graphs - Representations

► Adjacency matrix properties

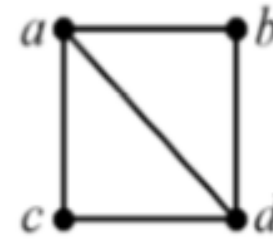
- the matrix is symmetric ($a[i][j] == a[j][i]$ will be true $\forall i, j$)
- the number of *True*(1) entries is twice the number of edges
- different orderings of the vertex set V will result in different adjacency matrices for the same graph
- preferred representation when the graph is *dense* (= many edges)
 - When the graph is sparse (= few edges), adjacency lists are more efficient



	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	F	T	T	T
<i>b</i>	T	F	F	T
<i>c</i>	T	F	F	T
<i>d</i>	T	T	T	F

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	0	1	1	1
<i>b</i>	1	0	0	1
<i>c</i>	1	0	0	1
<i>d</i>	1	1	1	0

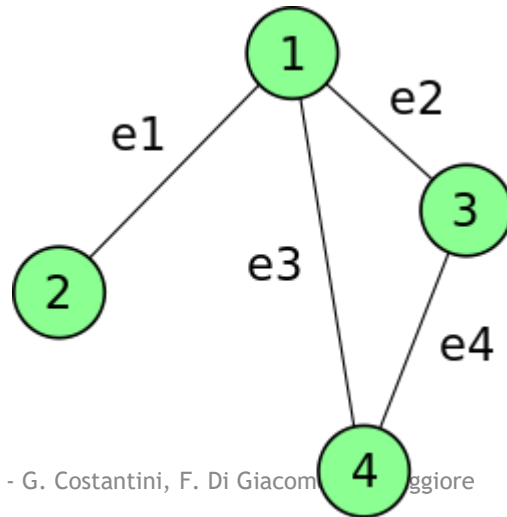
Graphs - Representations



a	1	1	1	0	0
b	1	0	0	1	0
c	0	1	0	0	1
d	0	0	1	1	1

► Incidence matrix

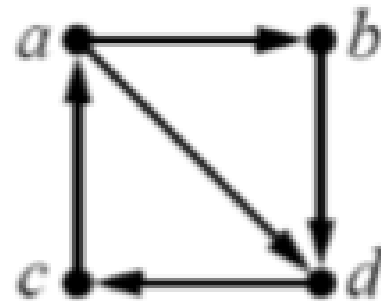
- shows the relationship between two classes of objects: vertices (rows) and edges (columns)
- given a cell at row $i \rightarrow$ column j is *True(1)* if vertex i is incident upon edge j



$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Graphs - Definition of digraph

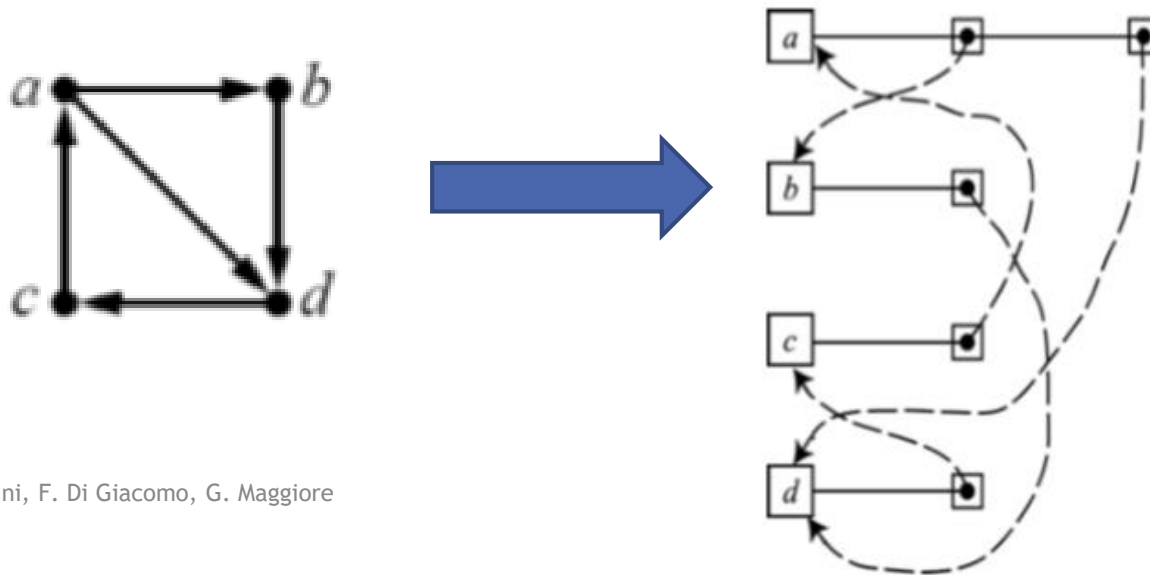
- ▶ A *digraph* (or **directed graph**) is a pair $G = (V, E)$ where V is a finite set and E is a set of ordered pairs of elements of V
 - ▶ Difference with simple graphs: edges have a DIRECTION
 - ▶ If $e = (a, b)$...
 - ▶ the edge e *emanates/is incident from* vertex a
 - ▶ the edge e *terminates/is incident to* vertex b



Graphs - Representation of digraphs

► Adjacency list of a digraph

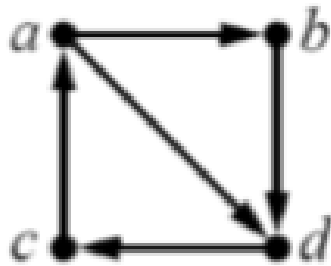
- for each vertex in the graph, store a list containing the edges that emanate from that vertex
- same as the adjacency list for a graph, except that the links are *not duplicated* unless there are edges going both ways between a pair of vertices



Graphs - Representation of digraphs

► Adjacency matrix of a digraph

- cell at row i , column j is *True* if there is an edge emanating from vertex i and terminating at vertex j

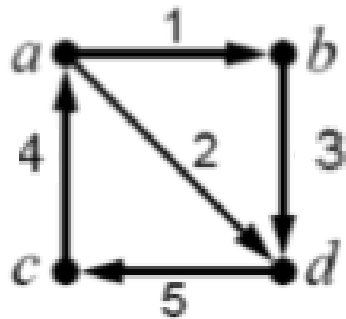


	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	F	T	F	T
<i>b</i>	F	F	F	T
<i>c</i>	T	F	F	F
<i>d</i>	F	F	T	F

Graphs - Representation of digraphs

► Incidence matrix of a digraph

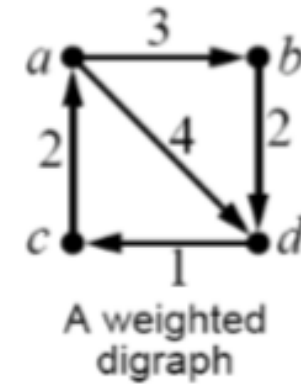
- cell at row i , column j has value
 - 1 if edge j emanates from vertex i
 - -1 if edge j terminates at vertex i



a	1	1	0	-1	0
b	-1	0	1	0	0
c	0	0	0	1	-1
d	0	-1	-1	0	1

Graphs - Some more terminology

- ▶ Path from a to b in a digraph
 - ▶ Same concept as in undirected graphs
- ▶ **Weighted digraph** \rightarrow pair (V, w) where
 - ▶ V is a finite set of vertices
 - ▶ w is a function that assigns to each pair (x, y) of vertices either a positive integer or ∞ (infinity)
 - ▶ called *weight function*
 - ▶ cost/time/distance for moving directly from x to y ; ∞ means no edge from x to y
 - ▶ Weighted graph $\rightarrow w$ is symmetric ($w(x, y) = w(y, x)$)



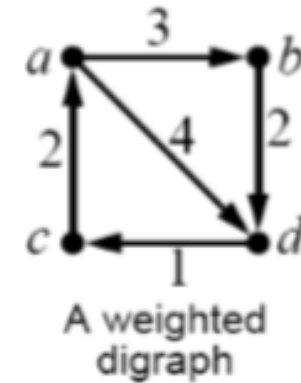
Graphs - Some more terminology

- ▶ **Weighted path length**

- ▶ sum of the weights of the edges along the path

- ▶ **Shortest distance from x to y**

- ▶ minimum weighted path length among all the paths from x to y
 - ▶ *Dijkstra's Shortest Path Algorithm* → finding the shortest path from one vertex to each other vertex in a (di)graph

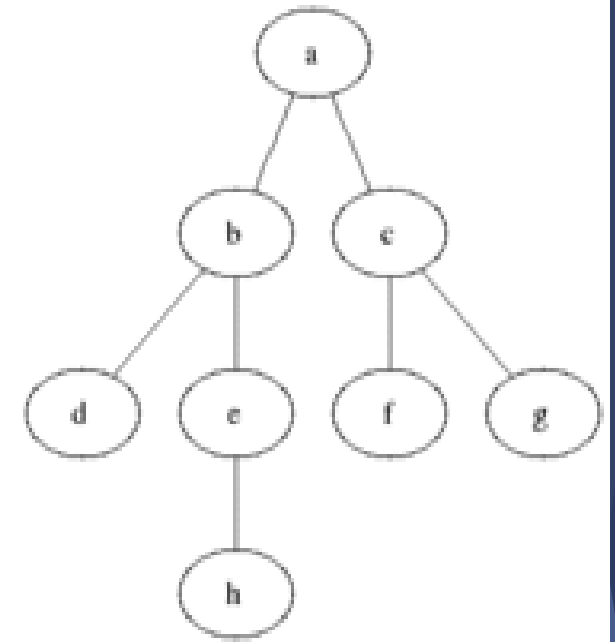


Graphs - Traversal algorithms

- ▶ *Graph traversal* → visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way
- ▶ Possible algorithms
 - ▶ **BFS** (Breadth First Search)
 - ▶ Inspect all neighbors of a node; then for each neighbor inspect all its unvisited neighbors, etc...
 - ▶ **DFS** (Depth First Search)
 - ▶ Start from one neighbor and go as far as possible in that direction before continuing with exploring the other neighbors

Graph - BFS traversal algorithm

- ▶ Search is limited to essentially two operations
 - ▶ visit and inspect a node of a graph
 - ▶ gain access to visit the nodes that neighbor the currently visited node
- ▶ Algorithm
 - ▶ begins at a root node and inspects all the neighboring nodes
 - ▶ for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on
- ▶ Complexity $\rightarrow O(|V| + |E|)$

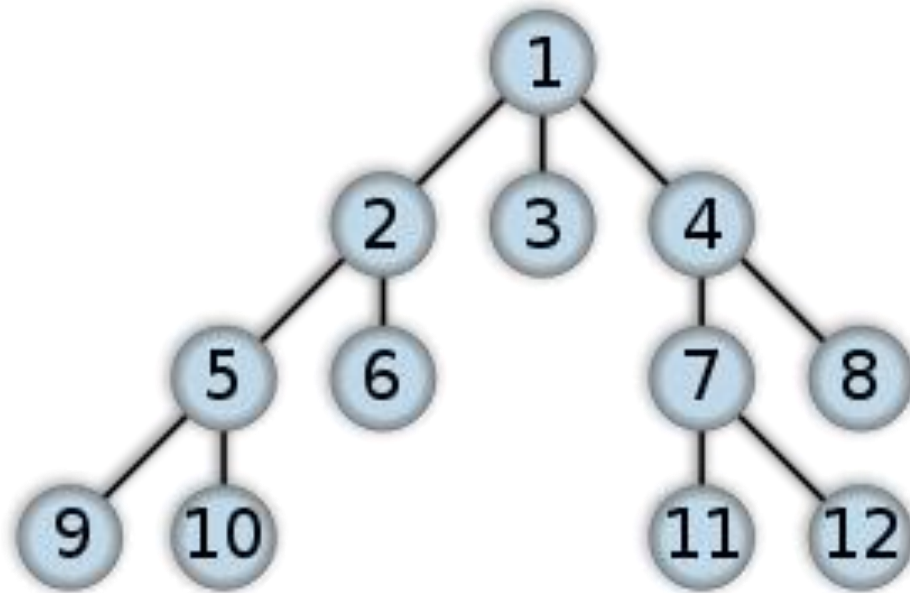


Graph - BFS traversal algorithm

- ▶ **Queue** data structure used to store intermediate results as it traverses the graph
 1. Enqueue the root node
 2. Dequeue a node and examine it
 - ▶ [If the element sought is found in this node, quit the search and return a result]
 - ▶ Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered
 3. If the queue is empty, every node on the graph has been examined [quit the search and return "not found"]
 4. If the queue is not empty, repeat from Step 2

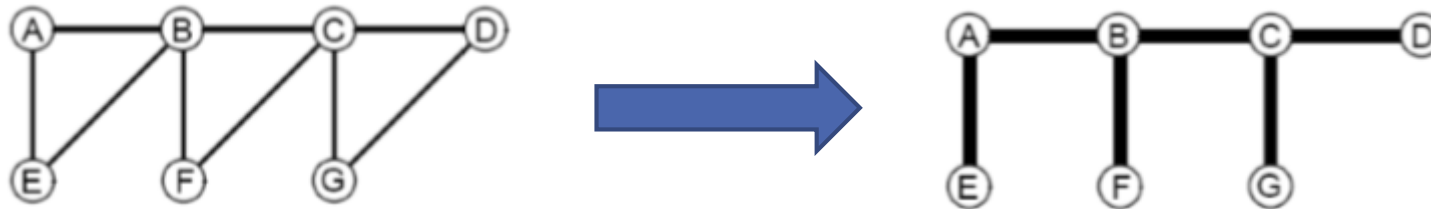
Graph - BFS traversal algorithm

- Result of a BFS traversal



Graph - BFS traversal algorithm

- ▶ BFS traversal
 - ▶ Returned list of visited vertices: A, B, E, C, F, D, G



Graph - BFS traversal algorithm

Algorithm pseudocode:

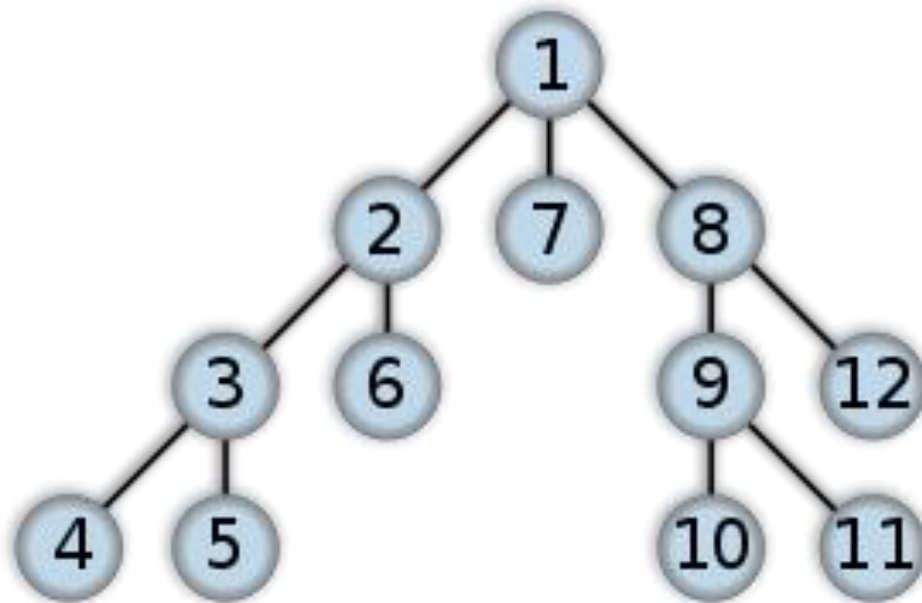
```
Breadth-First-Search(Graph, root)
  for each node n in Graph:
    n.distance = INFINITY
    n.parent = NIL
  create empty queue Q
  root.distance = 0
  Q.enqueue(root)
  while Q is not empty:
    current = Q.dequeue()
    for each node n that is adjacent to current:
      if n.distance == INFINITY:
        n.distance = current.distance + 1
        n.parent = current
        Q.enqueue(n)
```

Graph - DFS traversal algorithm

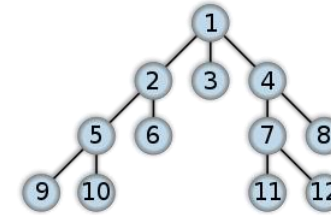
- ▶ Algorithm
 - ▶ Starts at a root node
 - ▶ Explores as far as possible along each branch before backtracking
- ▶ Complexity $\rightarrow O(|V| + |E|)$
- ▶ Difference with BFS
 - ▶ DSF uses a stack instead of a queue
 - ▶ *Push* only the first unvisited neighbour of the top element of the stack
 - ▶ *Pop* from the stack if there are no other unvisited neighbours
 - ▶ A recursive implementation is possible

Graph - DFS traversal algorithm

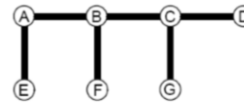
- Result of a DFS traversal



BFS was...

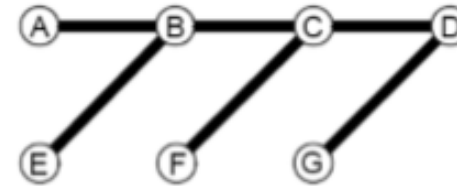
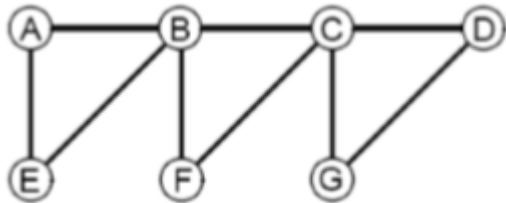


Graph - DFS traversal algorithm

BFS was... 

- ▶ DFS traversal

- ▶ Returned list of visited vertices: A, B, C, D, G, F, E



Graph - DFS traversal algorithm

Algorithm pseudocode (recursive)

```
procedure DFS(G,v):  
    label v as discovered  
    for all edges from v to w in G.adjacentEdges(v) do  
        if vertex w is not labeled as discovered then  
            recursively call DFS(G,w)
```

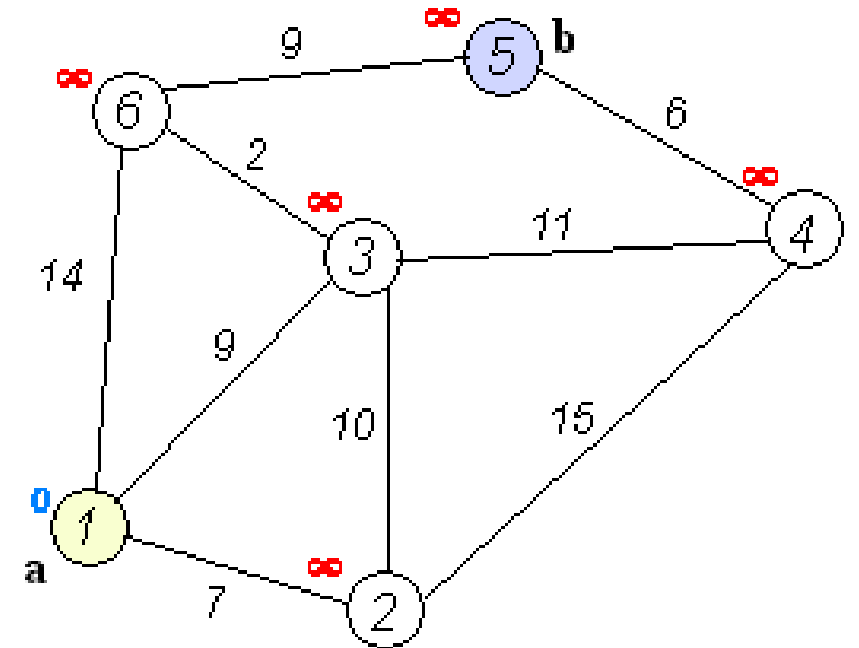
Graph - DFS traversal algorithm

Algorithm pseudocode (iterative)

```
procedure DFS-iterative(G,v):  
  let S be a stack  
  S.push(v)  
  while S is not empty  
    v = S.pop()  
    if v is not labeled as discovered:  
      label v as discovered  
      for all edges from v to w in G.adjacentEdges(v) do  
        S.push(w)
```

Graphs - Dijkstra's algorithm

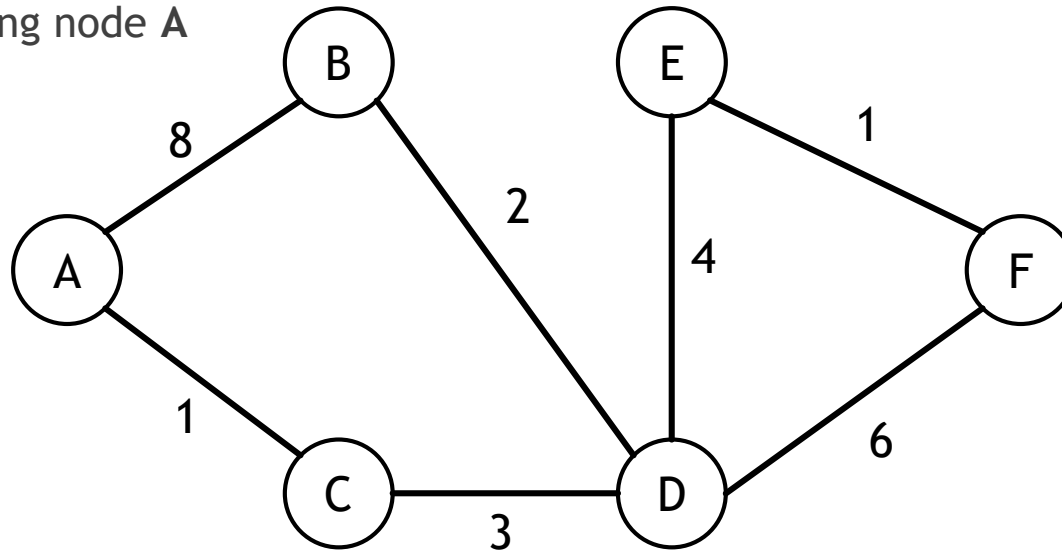
- ▶ Single-source shortest path problem
 - ▶ for a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e., the shortest path) between that vertex and every other vertex
- ▶ Informal steps of the algorithm
 - ▶ Pick the unvisited vertex with the lowest-distance
 - ▶ Calculate the distance through it to each unvisited neighbor
 - ▶ Update the neighbor's distance if smaller
 - ▶ Mark as visited when done with neighbors



Graphs - Dijkstra's algorithm

► Example

► Starting node A

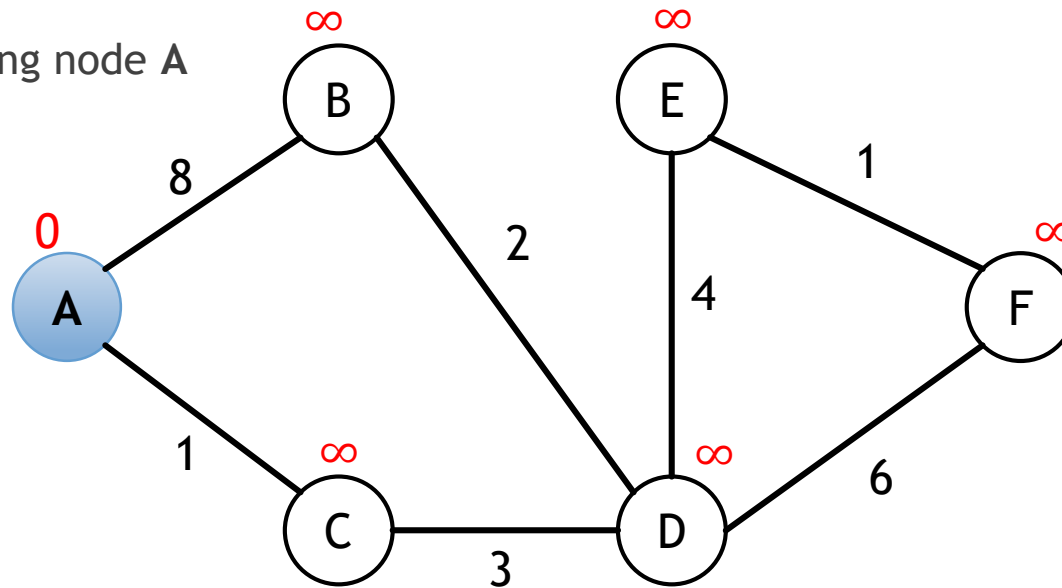


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

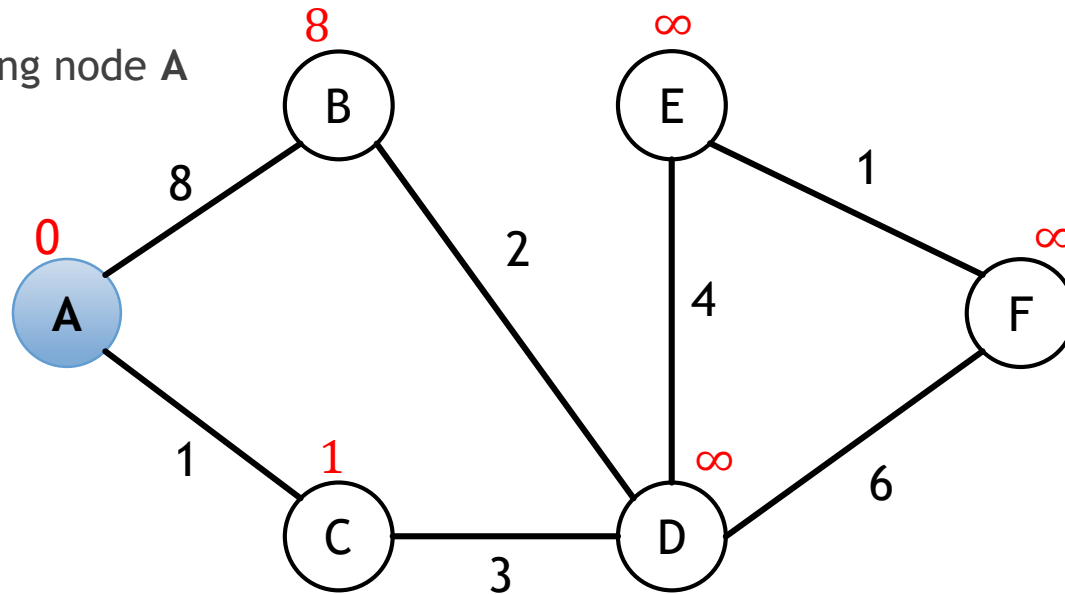


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

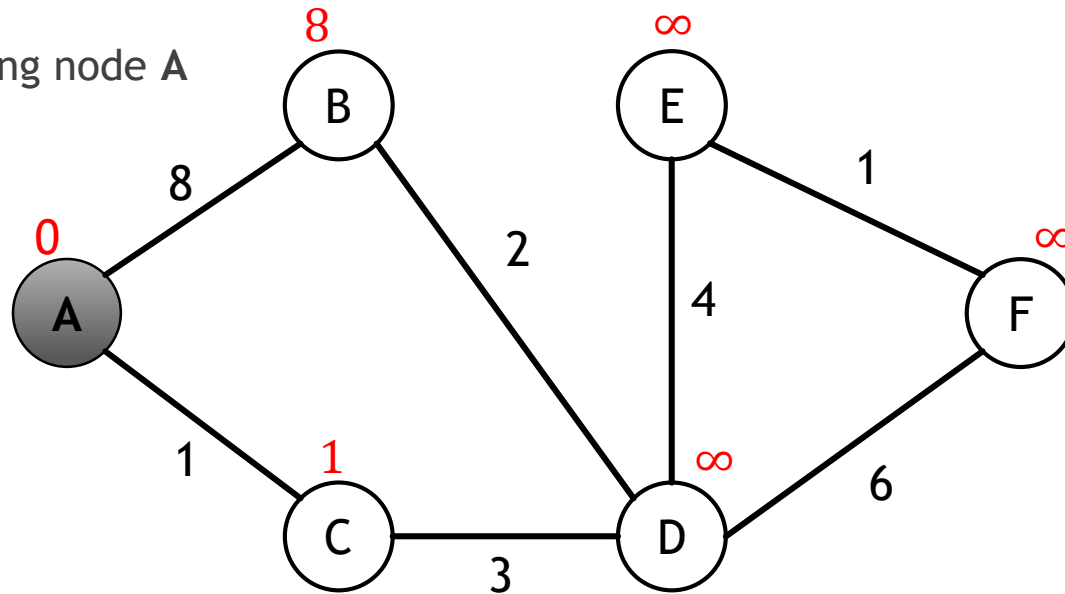


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

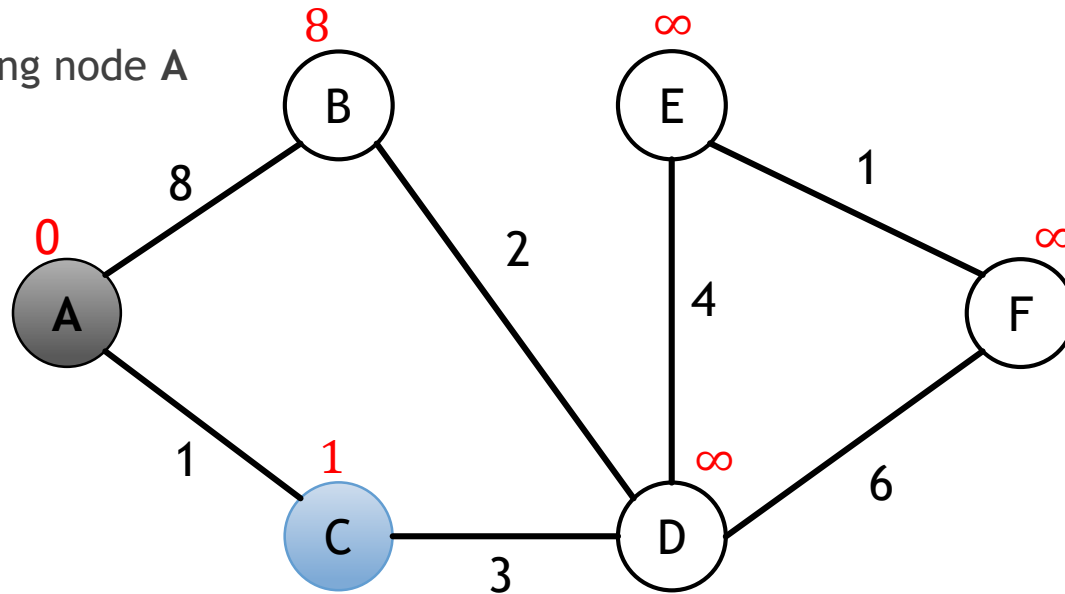


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

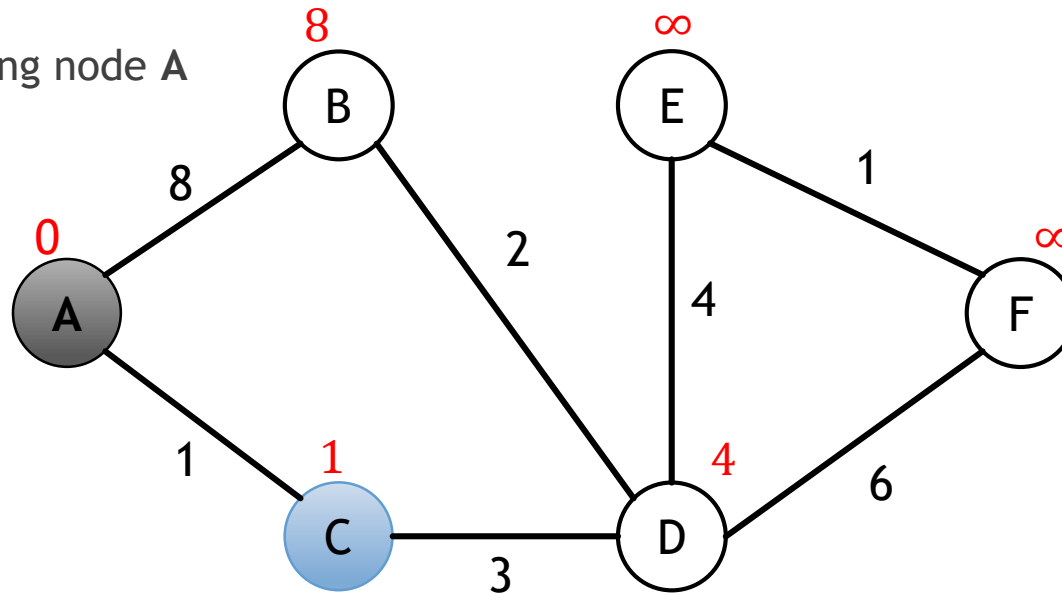


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

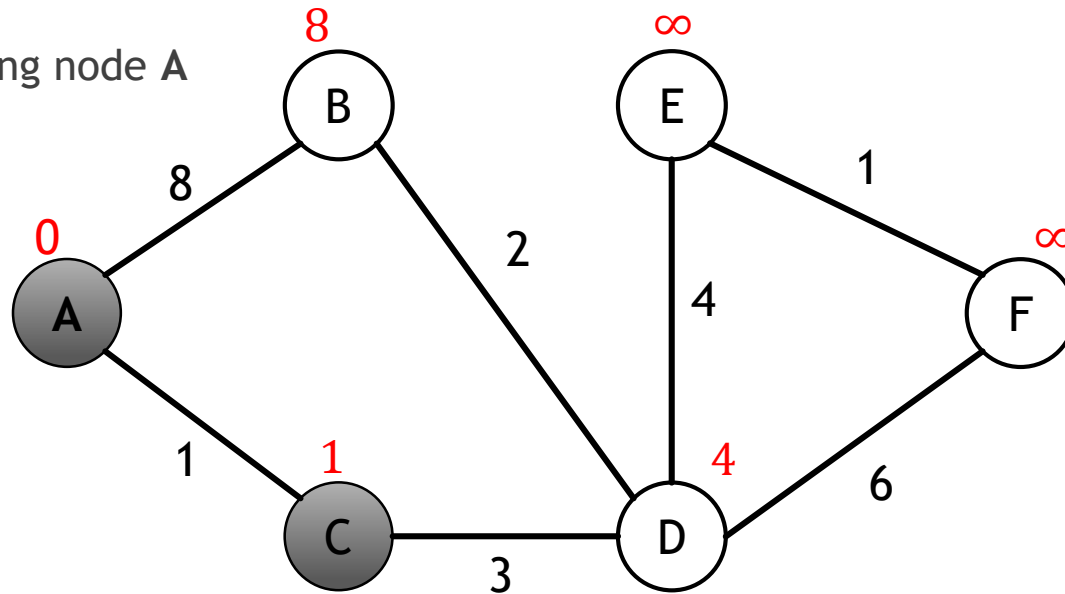


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

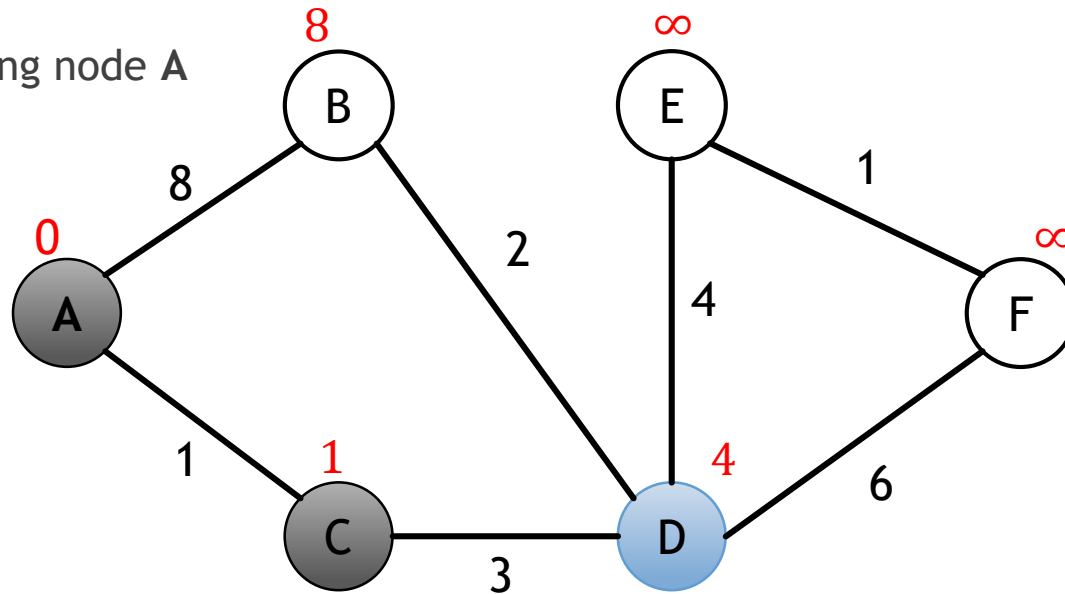


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

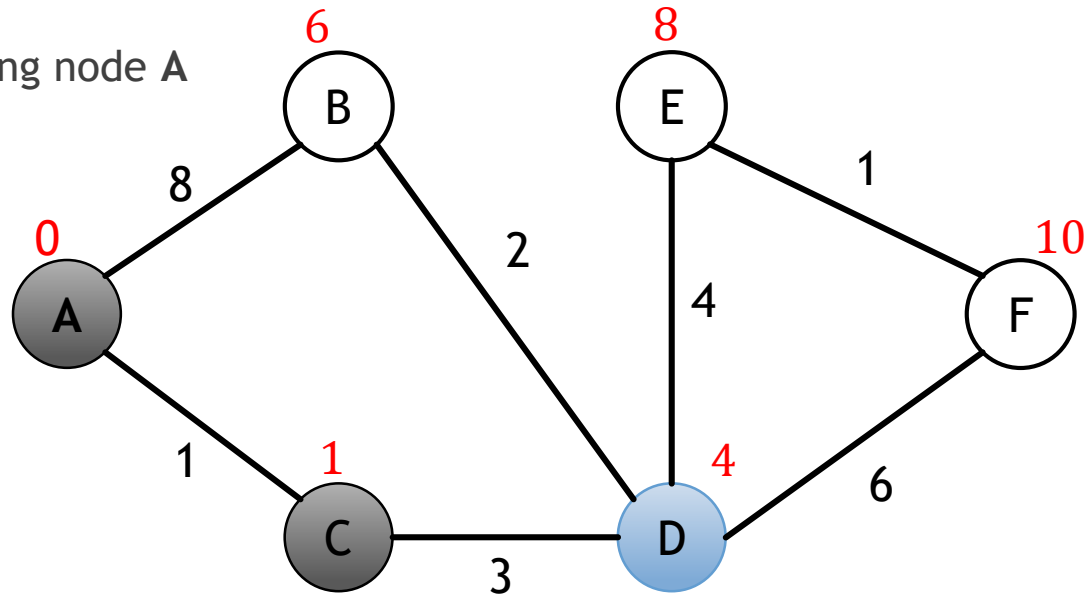


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

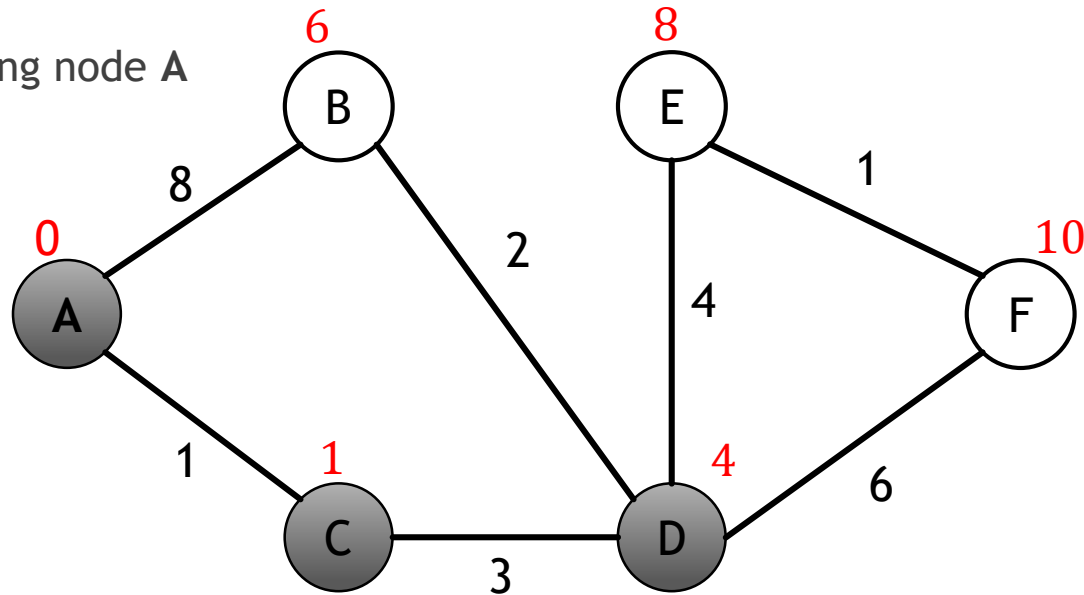


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

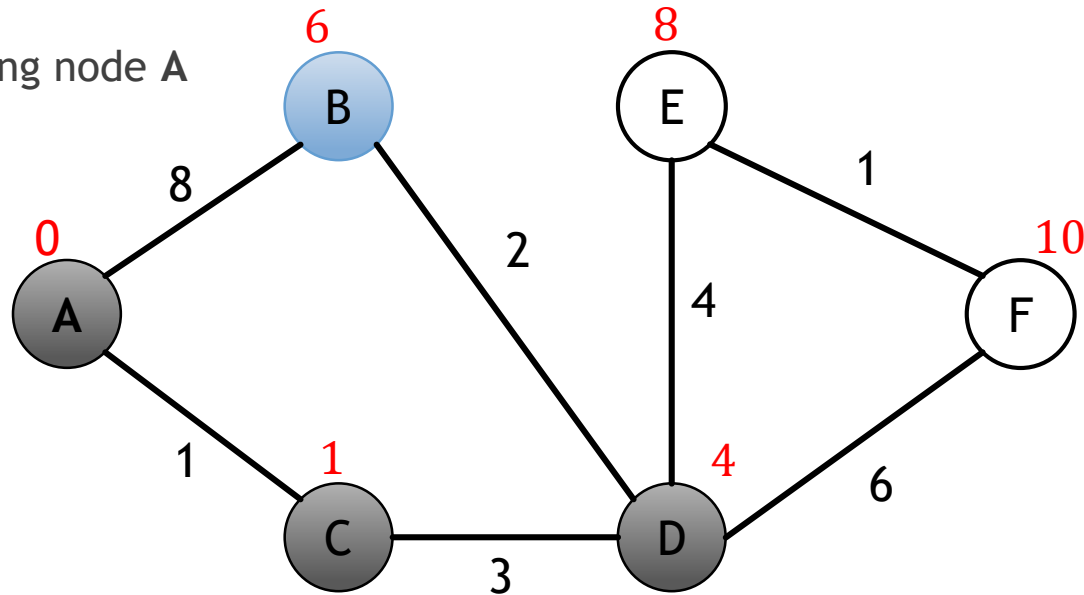


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

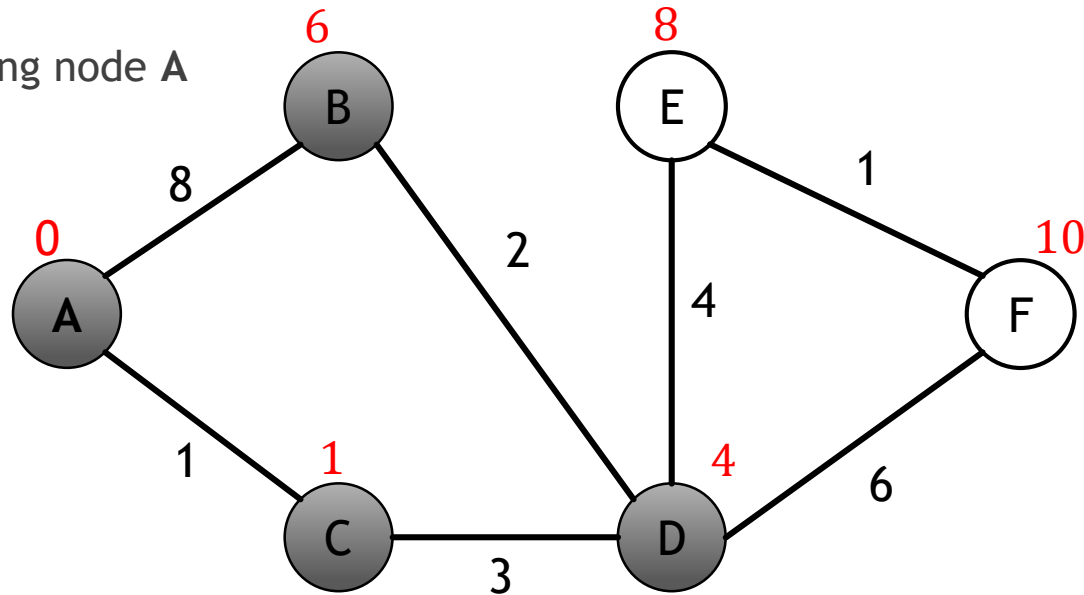


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

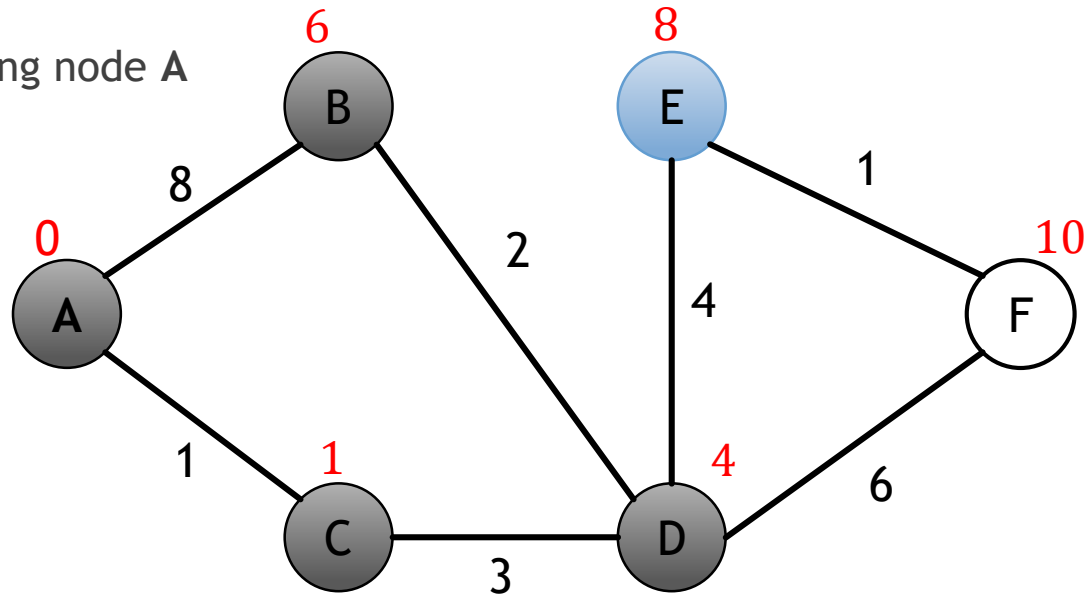


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

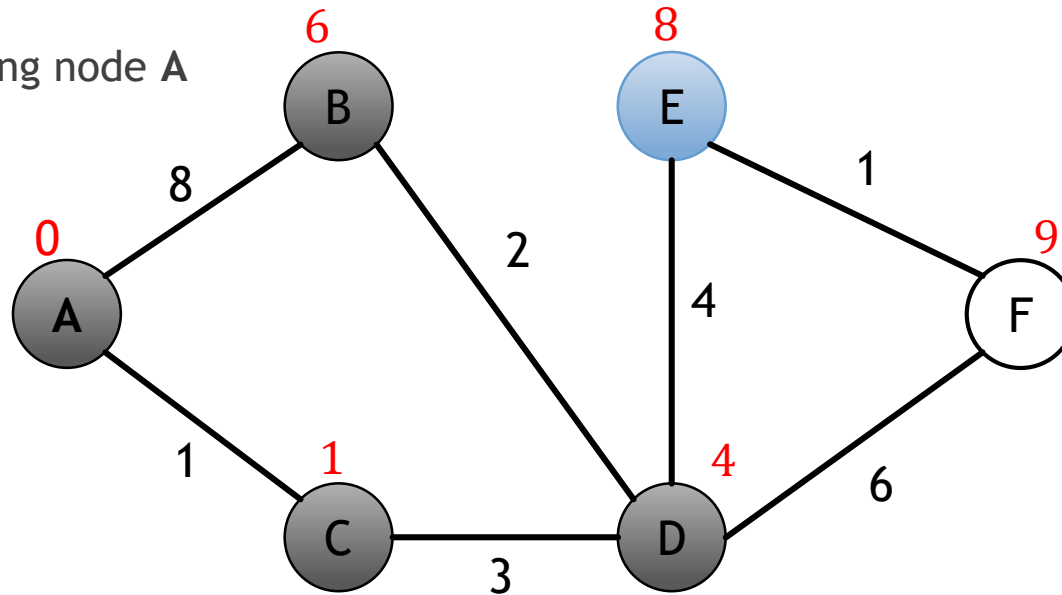


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

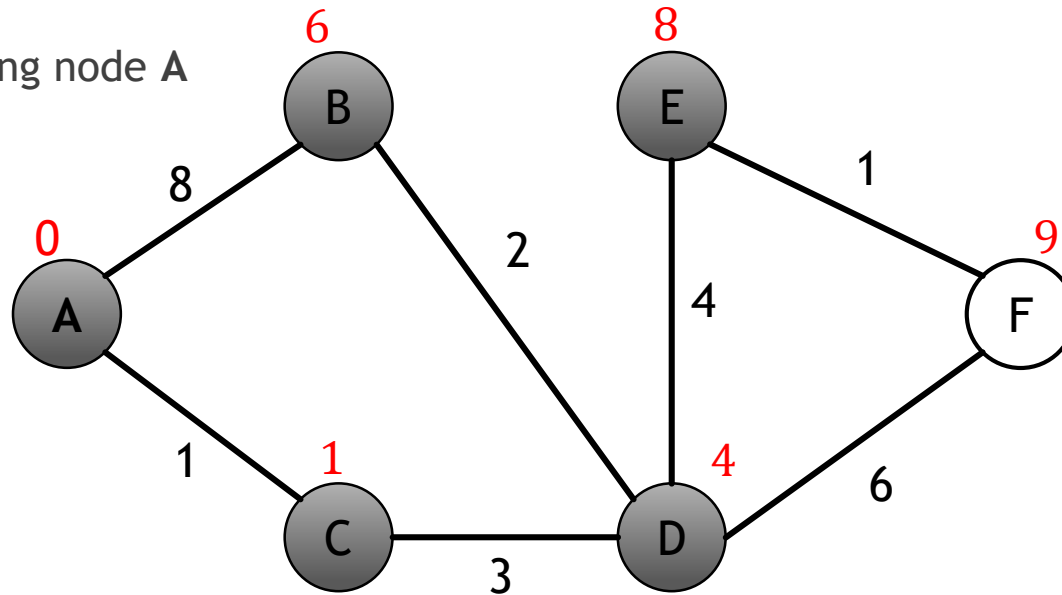


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

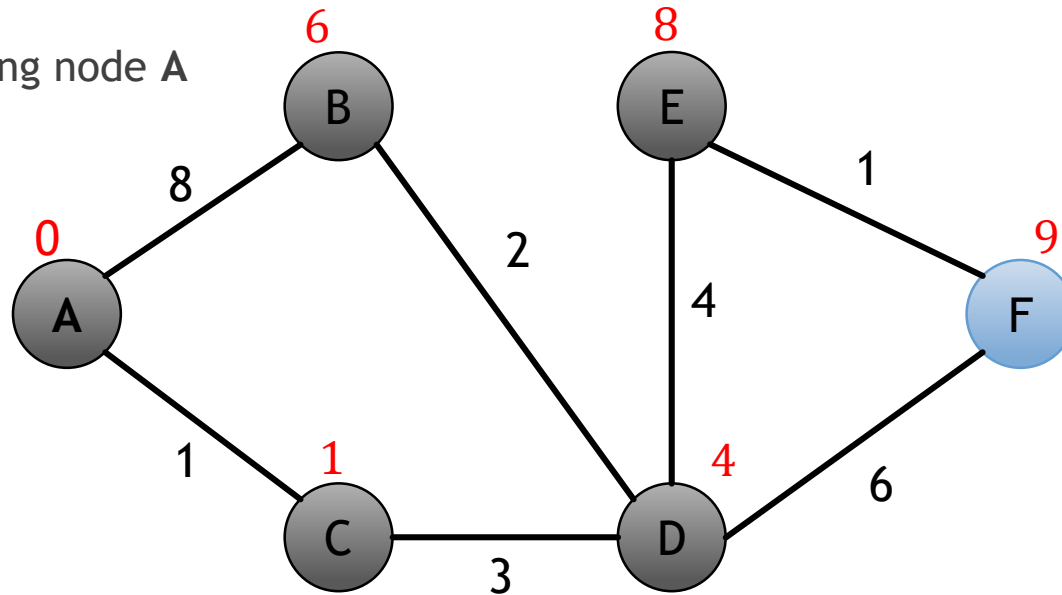


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A

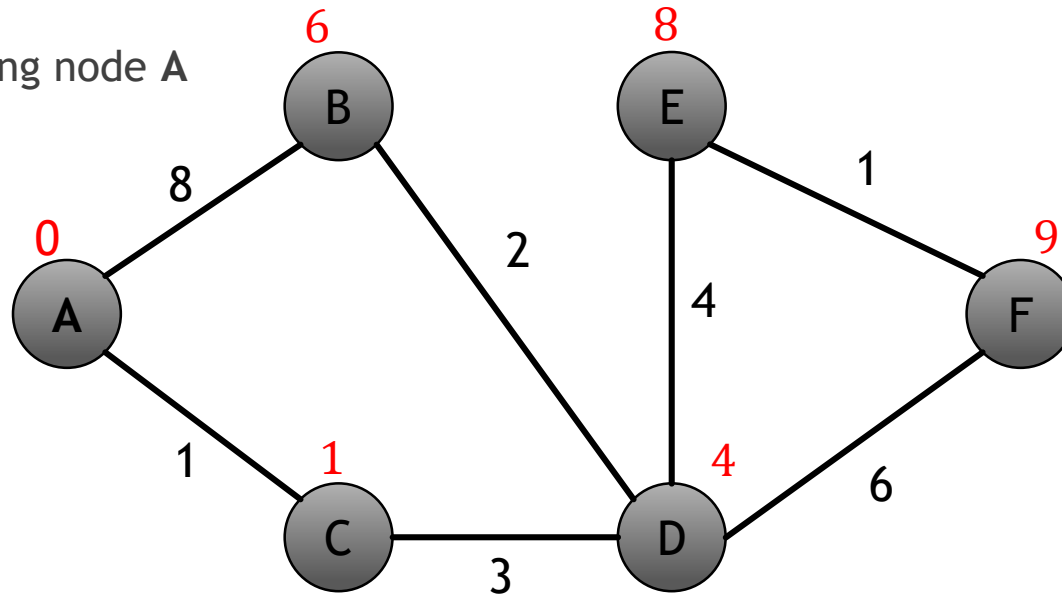


- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm

► Example

► Starting node A



- Pick the unvisited vertex with the lowest-distance
- Calculate the distance through it to each unvisited neighbor
- Update the neighbor's distance if smaller
- Mark as visited when done with neighbors

Graphs - Dijkstra's algorithm (pseudocode)

```
function Dijkstra(Graph, source):  
    create vertex set Q  
    for each vertex v in Graph:           // Initialization  
        dist[v] ← INFINITY                // Unknown distance from source to v  
        prev[v] ← UNDEFINED              // Previous node in optimal path from source  
        add v to Q                        // All nodes initially in Q (unvisited nodes)  
    dist[source] ← 0                      // Distance from source to source  
    while Q is not empty:  
        u ← vertex in Q with min dist[u] // Source node will be selected first  
        remove u from Q  
  
        for each neighbor v of u:         // where v is still in Q.  
            alt ← dist[u] + length(u, v)  
            if alt < dist[v]:              // A shorter path to v has been found  
                dist[v] ← alt  
                prev[v] ← u  
    return dist[], prev[]
```

Graphs - Dijkstra's algorithm

- ▶ Main steps of the algorithm
 - ▶ Pick the unvisited vertex with the lowest-distance
 - ▶ Calculate the distance through it to each unvisited neighbor
 - ▶ Update the neighbor's distance if smaller
 - ▶ Mark visited when done with neighbors

Graphs - Dijkstra's algorithm

1. Assign to every node a tentative distance value: set it to zero for the initial node and to infinity (∞) for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and **calculate their tentative distances**. Compare the newly calculated *tentative* distance to the current assigned value and **assign the smaller one**.
 - ▶ For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be $6 + 2 = 8$. If *B* was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, **mark the current node as visited** and remove it from the *unvisited set*. A visited node will never be checked again.
5. **Select the unvisited node that is marked with the smallest tentative distance**, and set it as the new "current node" then go back to step 3.

Graphs - Dijkstra's algorithm

- ▶ Complexity: $O(|V|^2)$
 - ▶ where $|V|$ is the amount of nodes
 - ▶ Improvable with a Fibonacci heap into: $O(|E| + |V| \log |V|)$

Homework

- ▶ *Implement the BFS algorithm*
- ▶ *Implement the DFS algorithm*
- ▶ *Implement Dijkstra algorithm*