# Data Structure - Spring 2022

# 17. Graph – Part 2

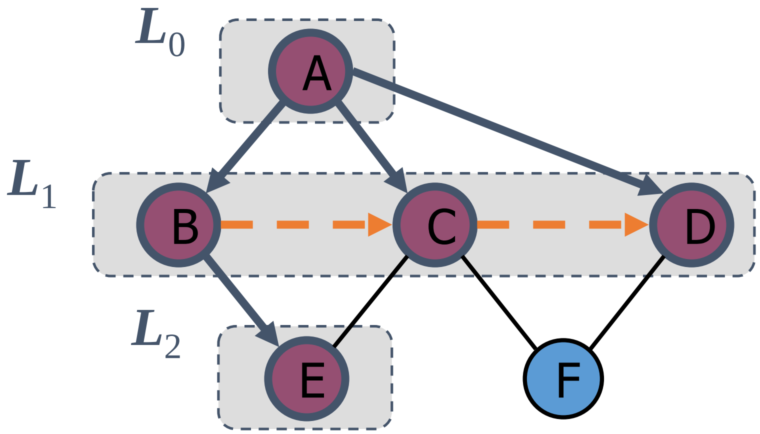
**Walid Abdullah Al**
Computer and Electronic Systems Engineering
Hankuk University of Foreign Studies

TA: **Seong Joo Kim**

**Based on:**
Goodrich, Chapter 9,11
Karumanchi, Chapter 6-7
Slides by Prof. Yung Yi, KAIST
Slides by Prof. Chansu Shin, HUFS

Computer Vision Lab
Hankuk University of Foreign Studies

# Breadth-First Search

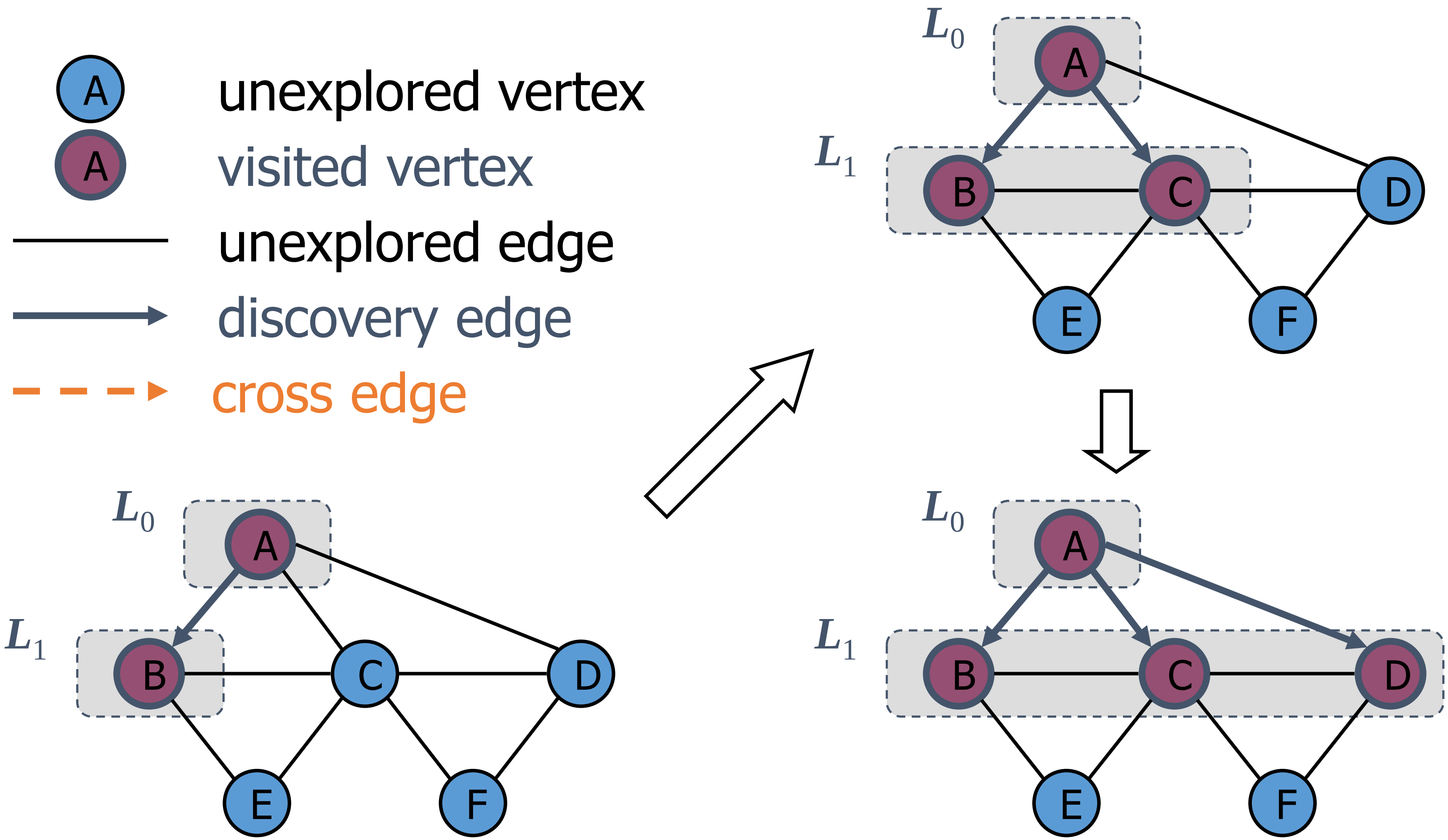


# Breadth-First Search

- Breadth-first search (BFS) is another general technique for traversing a graph

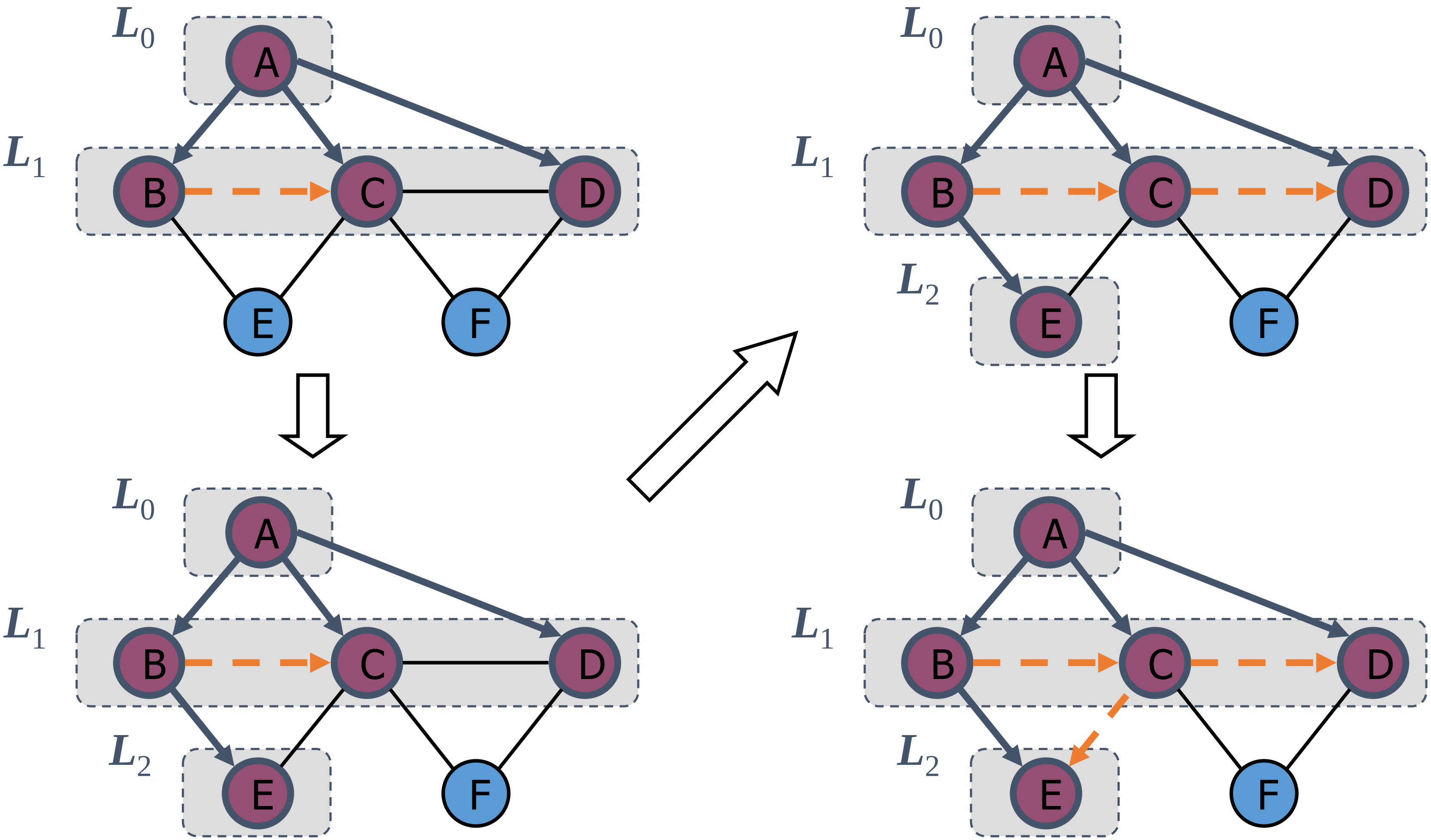
- Let’s look at the example

# Example

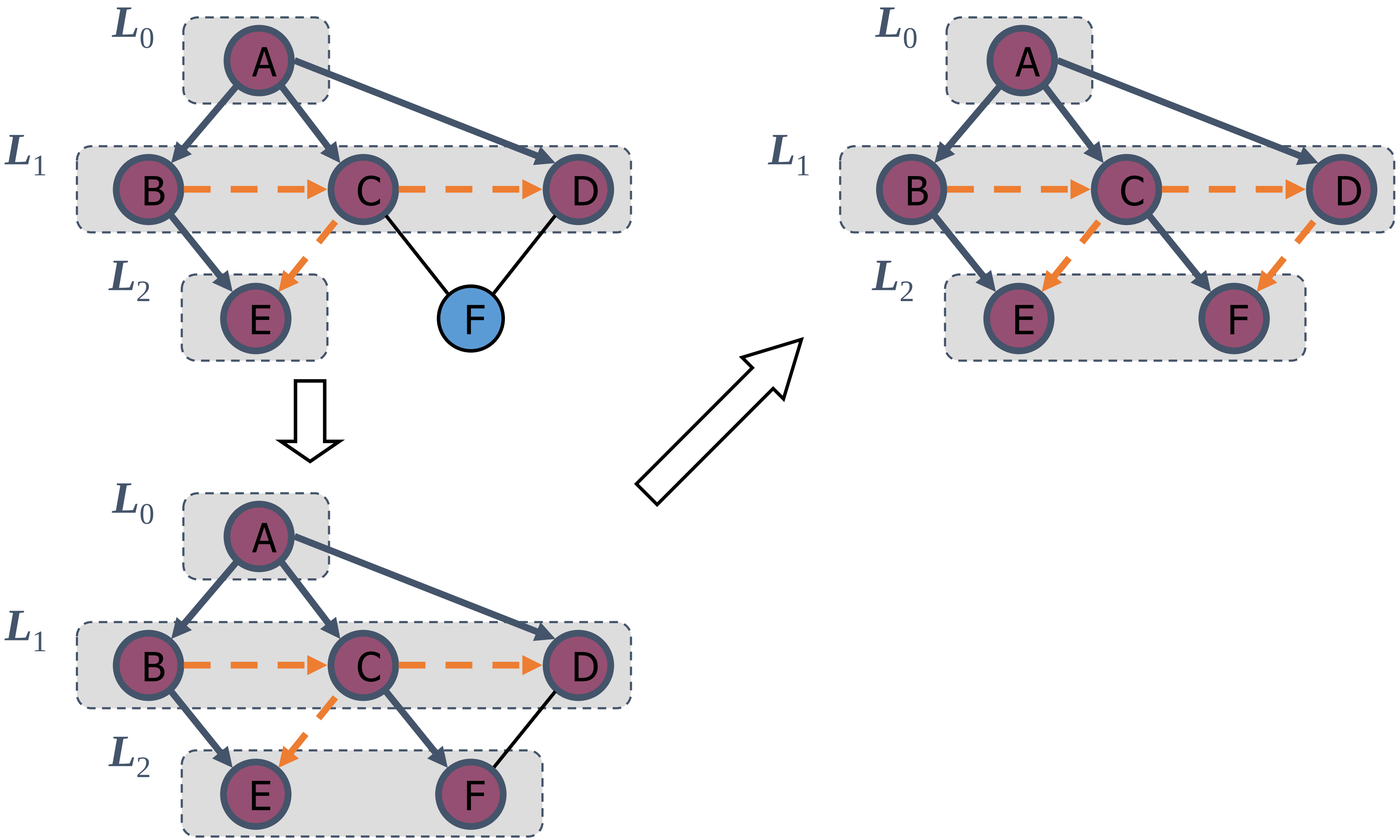


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
$L_0$
$L_1$
A
B
C
D
E
F

# Example (cont.)



# Example (cont.)



# Breadth-First Search

- **A BFS traversal of a graph G**
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

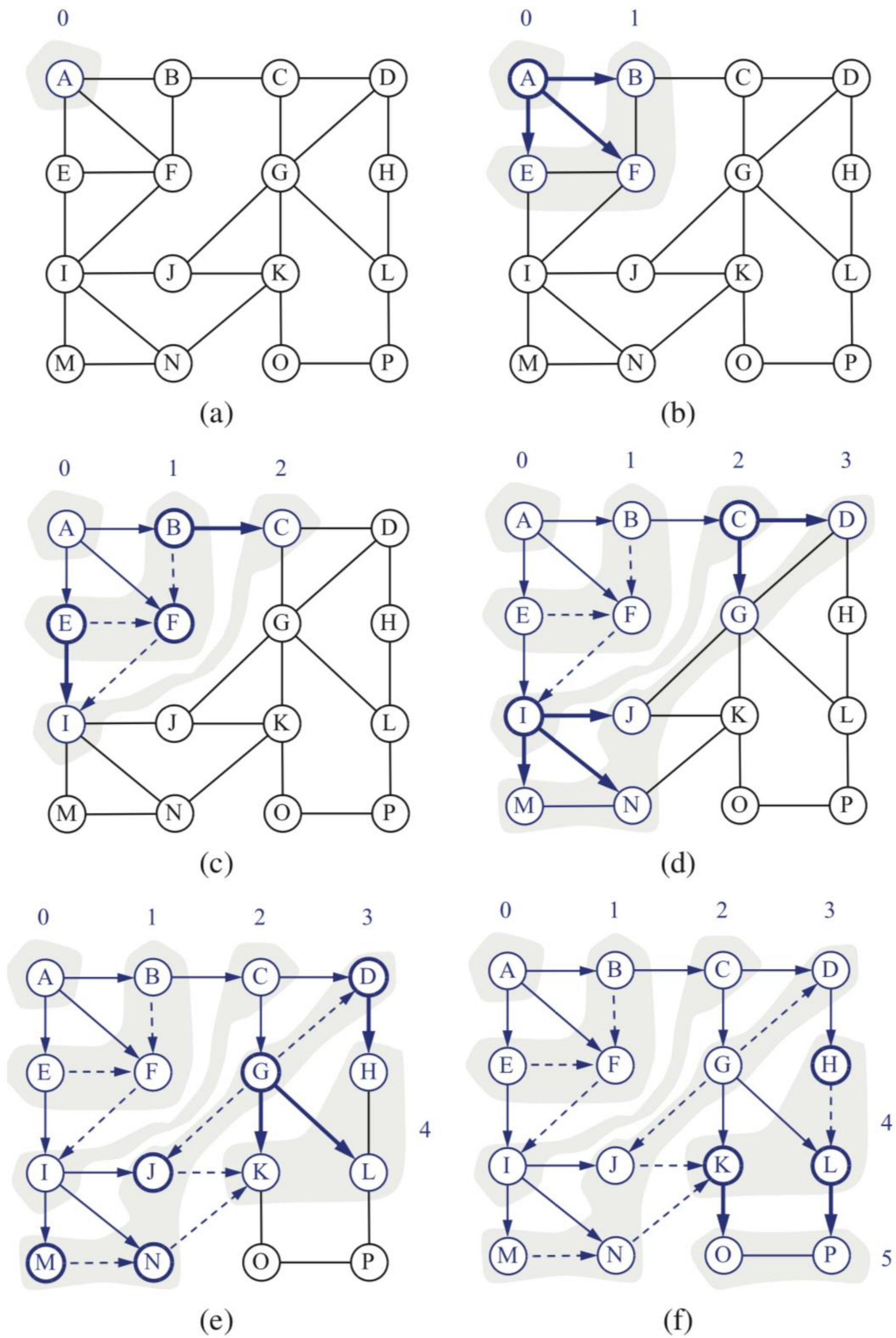
- **BFS on a graph with $n$ vertices and $m$ edges takes $O(n+m)$ time**
- **BFS can be further extended to solve other graph problems**
  - Find and report a path between two given vertices
  - Can label each vertex by the length of a shortest path (in terms of # of edges) from the start vertex s
  - Find a simple cycle, if there is one

# BFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

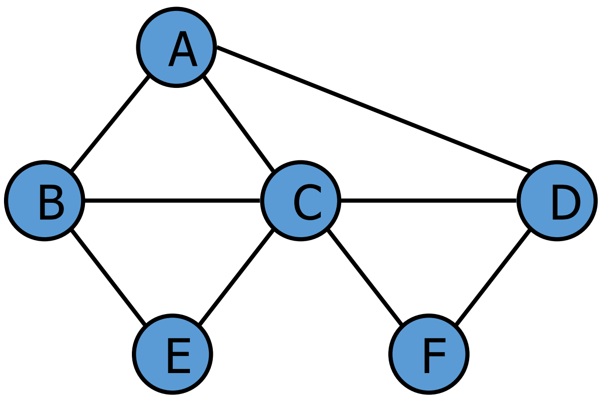
```
Algorithm BFS(G)
   Input graph G
   Output labeling of the edges
      and partition of the
      vertices of G
  for all u ∈ G.vertices()
   u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
   e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
   if v.getLabel() = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertBack(s)
  s.setLabel(VISITED)
  i ← 0
  while ¬Li.empty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ v.incidentEdges()
        if e.getLabel() = UNEXPLORED
          w ← e.opposite(v)
          if w.getLabel() = UNEXPLORED
            e.setLabel(DISCOVERY)
            w.setLabel(VISITED)
            Li+1.insertBack(w)
          else
            e.setLabel(CROSS)
    i ← i + 1
```



0
A B C D
E F G H
I J K L
M N O P
(a)
0 1
(b)
0 1 2
(c)
0 1 2 3
(d)
0 1 2 3 4
(e)
0 1 2 3 4 5
(f)

# Properties




**Notation**

$G_s$: connected component of $s$

**Property 1**

**$BFS(G, s)$ visits all the vertices and edges of $G_s$**

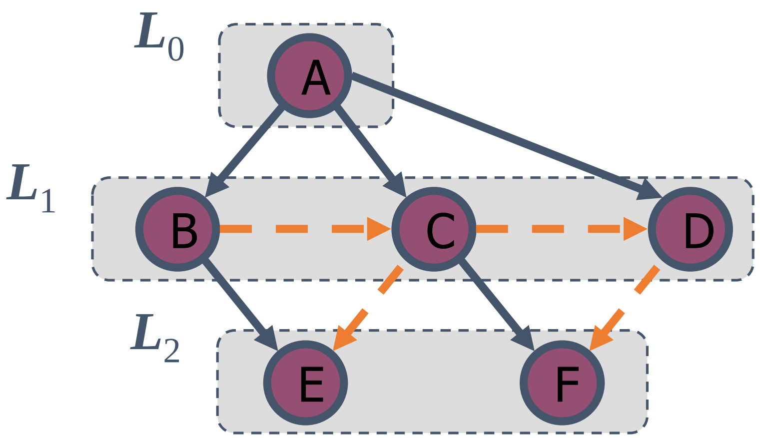
**Property 2**

**The discovery edges labeled by $BFS(G, s)$ form a spanning tree $T_s$ of $G_s$**

**Property 3**

**For each vertex $v$ in $L_i$**

- The path of $T_s$ from $s$ to $v$ has $i$ edges
- Every path from $s$ to $v$ in $G_s$ has at least $i$ edges (i.e., find a shortest path)



# Analysis

- **Setting/getting a vertex/edge label takes $O(1)$ time**
- **Each vertex is labeled twice**
  - once as UNEXPLORED
  - once as VISITED
- **Each edge is labeled twice**
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- **Each vertex is inserted once into a sequence $L_i$**
- **Method incidentEdges is called once for each vertex**
- **BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure**
  - Recall that $\sum_v \deg(v) = 2m$

# Applications

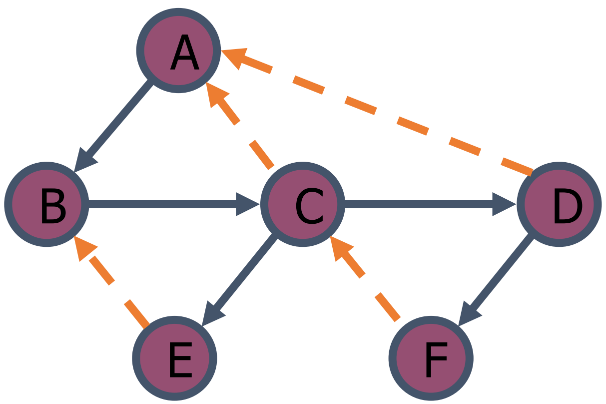
- **Using the template method pattern, we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n+m)$ time**
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists

# DFS vs. BFS

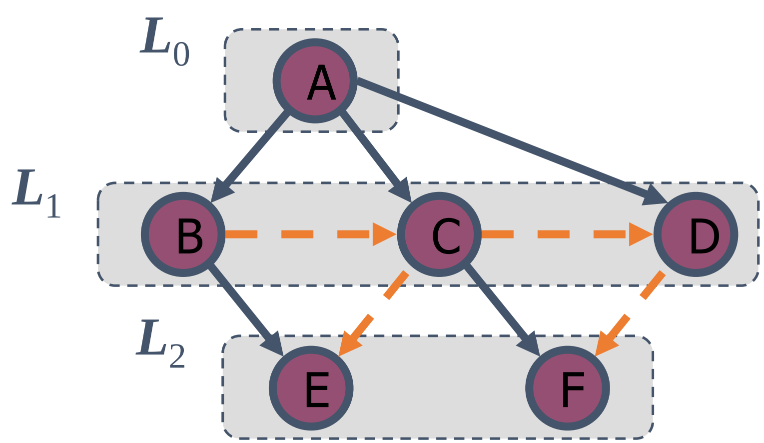
| Applications | DFS | BFS |
|---|---|---|
| Spanning forest, connected components, paths, cycles | √ | √ |
| Shortest paths | | √ |
| Biconnected components (how?) | √ | |

Biconnected components:

- Connected
- Even after removing any vertex the graph remains connected




DFS




BFS

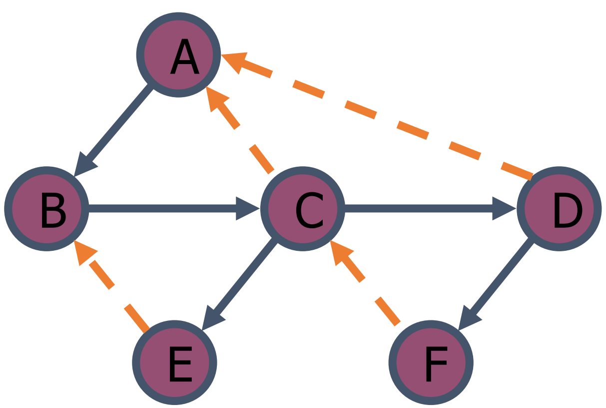
# DFS vs. BFS (cont.)

**Back edge** $(v,w)$

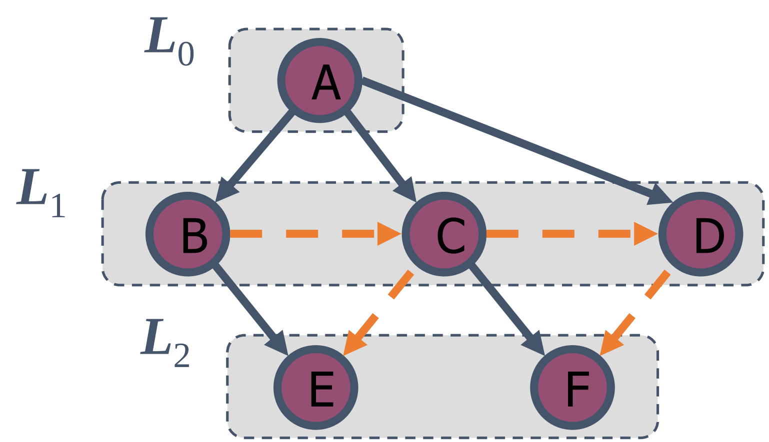
- $w$ is an ancestor of $v$ in the tree of discovery edges

**Cross edge** $(v,w)$

- $w$ is in the same level as $v$ or in the next level

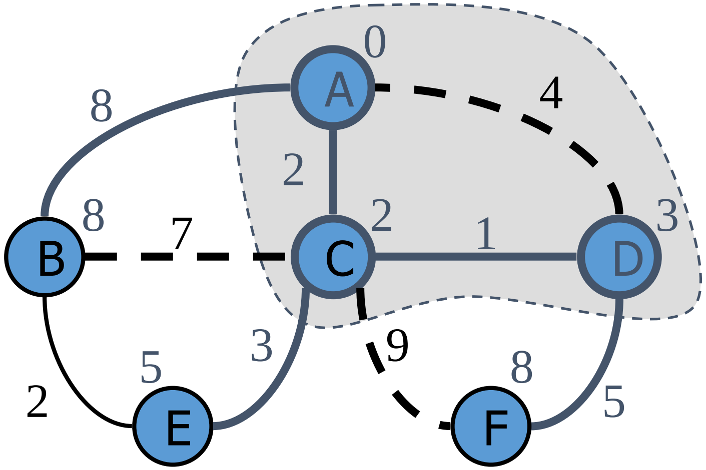



DFS



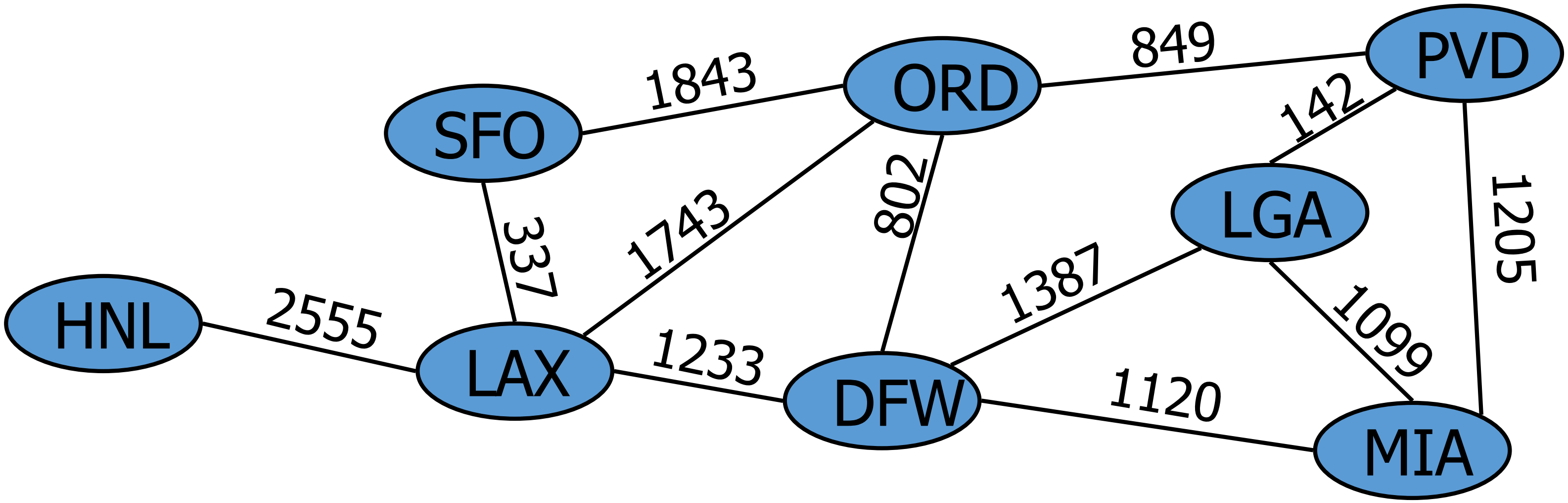

BFS

# Shortest Paths



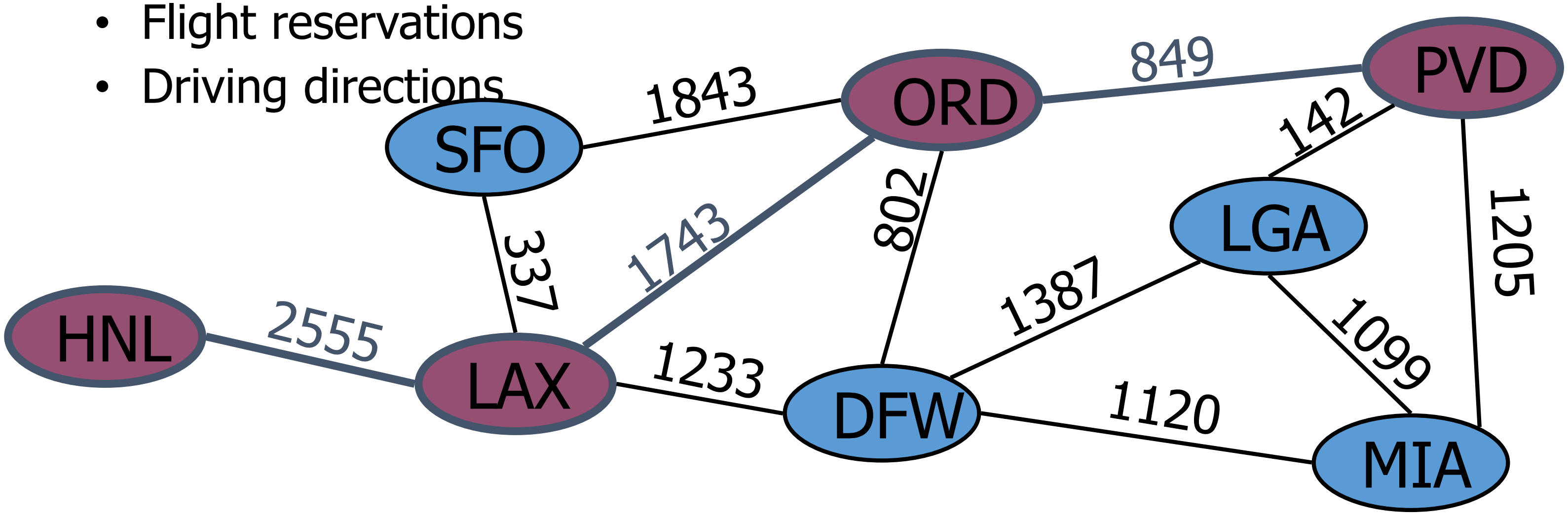
# Weighted Graphs

- **In a weighted graph, each edge has an associated numerical value, called the weight of the edge**
- **Edge weights may represent, distances, costs, etc.**
- **Example:**
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



# Shortest Paths

- **Given a weighted graph and two vertices $u$ and $v$, we want to find a path of minimum total weight between $u$ and $v$.**
  - Length of a path is the sum of the weights of its edges.
- **Example:**
  - Shortest path between Providence and Honolulu
- **Applications**
  - Internet packet routing
  - Flight reservations
  - Driving directions



# Shortest Path Properties

## Property 1:

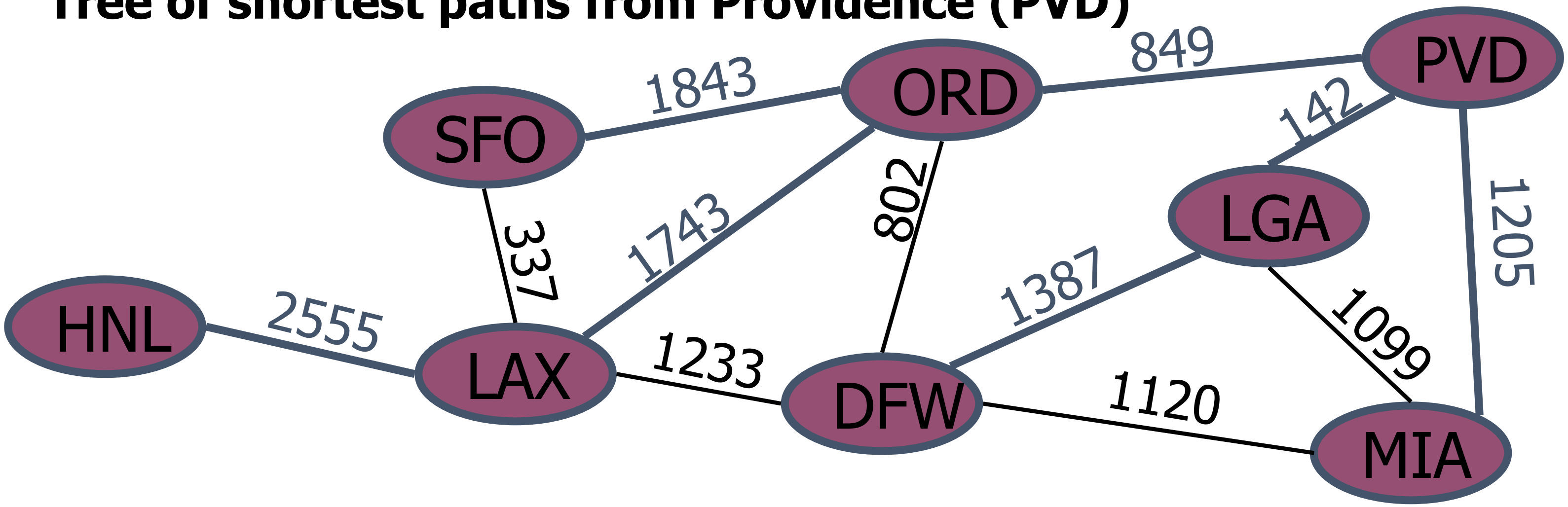
**A subpath of a shortest path is itself a shortest path**

## Property 2:

**There is a tree of shortest paths from a start vertex to all the other vertices**

## Example:

**Tree of shortest paths from Providence (PVD)**



# Our goal and Initial Ideas

- **Goal**
  - Given a source vertex s, compute the shortest paths to all other vertices

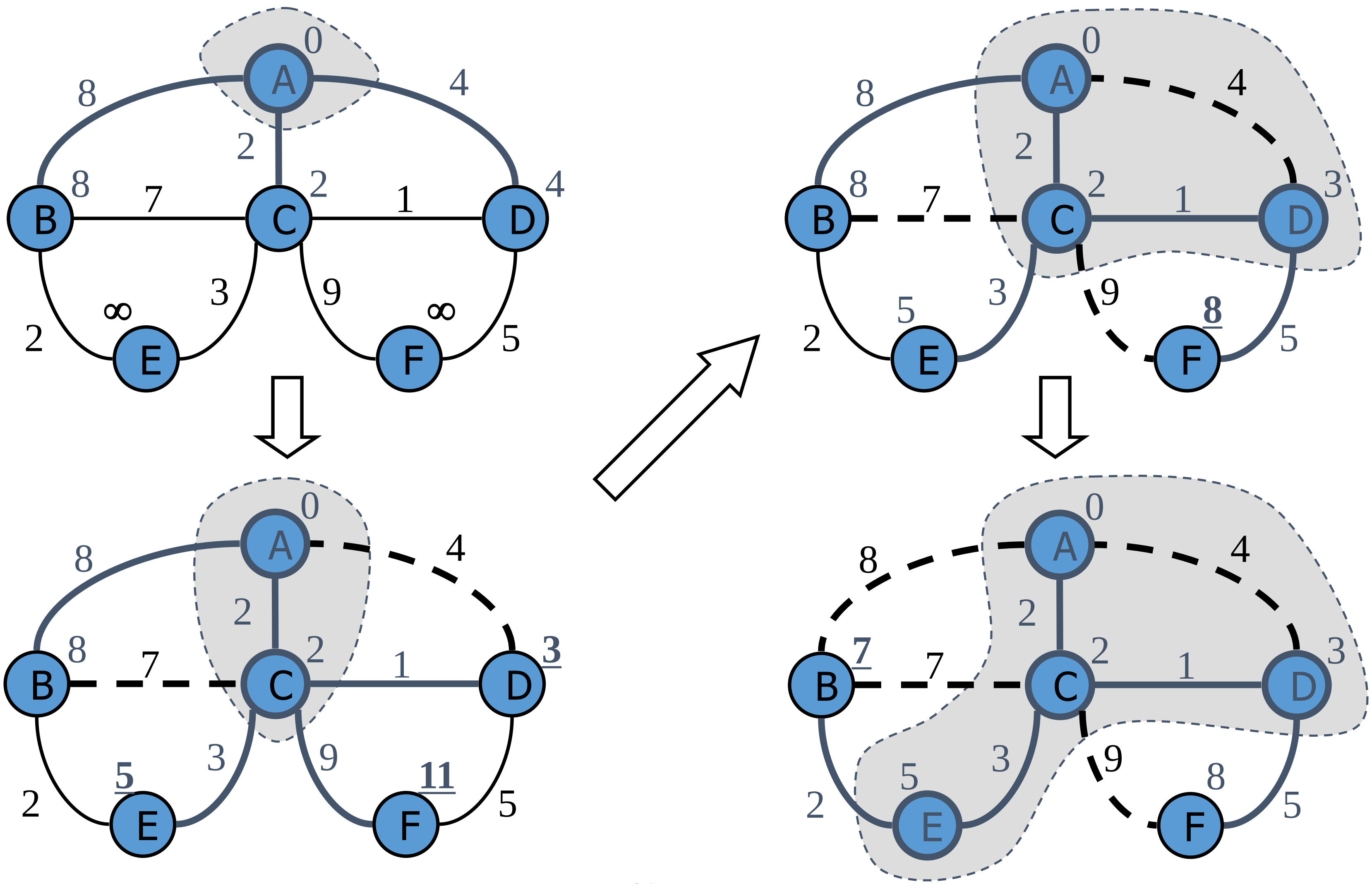
- **Initial Ideas**
  - Compute all the paths from the source ***s*** to other vertices
  - Take the minimums
  - How much complexity?
    - Exponential (not a polynomial time algorithm)
  - Why is this algorithm stupid?
    - Ignore the wisdom from computing the minimum path for computing other minimum paths

# Dijkstra’s Algorithm (1)

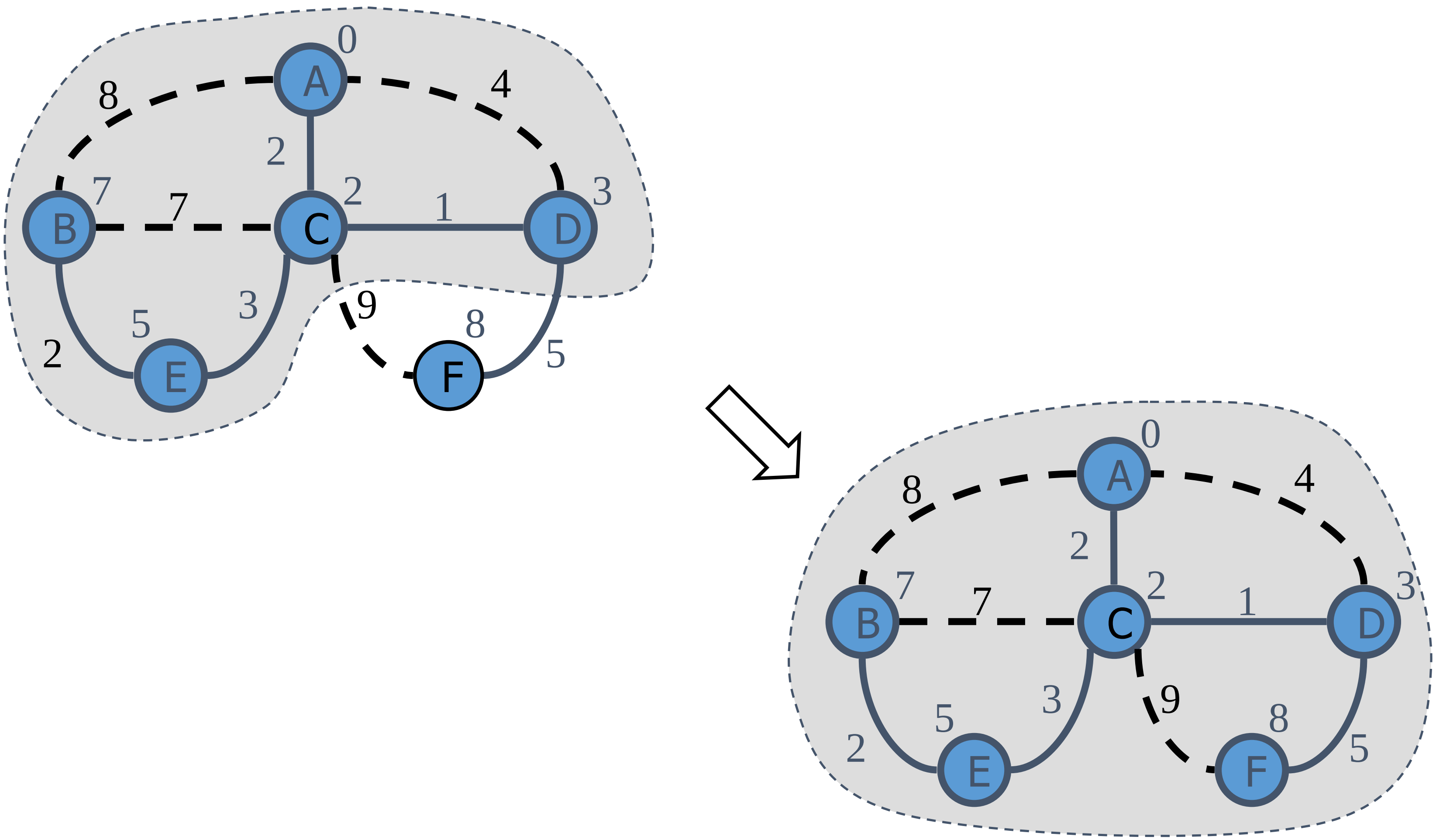
- **The distance of a vertex $v$ from a vertex $s$ is the length of a shortest path between $s$ and $v$**
- **Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex $s$**
- **Assumptions:**
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

- **We grow a “cloud” of vertices, beginning with $s$ and eventually covering all the vertices**
  - Remember the “wisdom”
- **Example**
  - What is your distance to ”Obama” in facebook? 50
  - Suppose that MoonJaein becomes your friend
  - What is your distance to “Obama” then?
    - Probably much shorter than 50. Maybe 2? ☺

# Example first



# Example (cont.)



# Dijkstra's Algorithm (2)

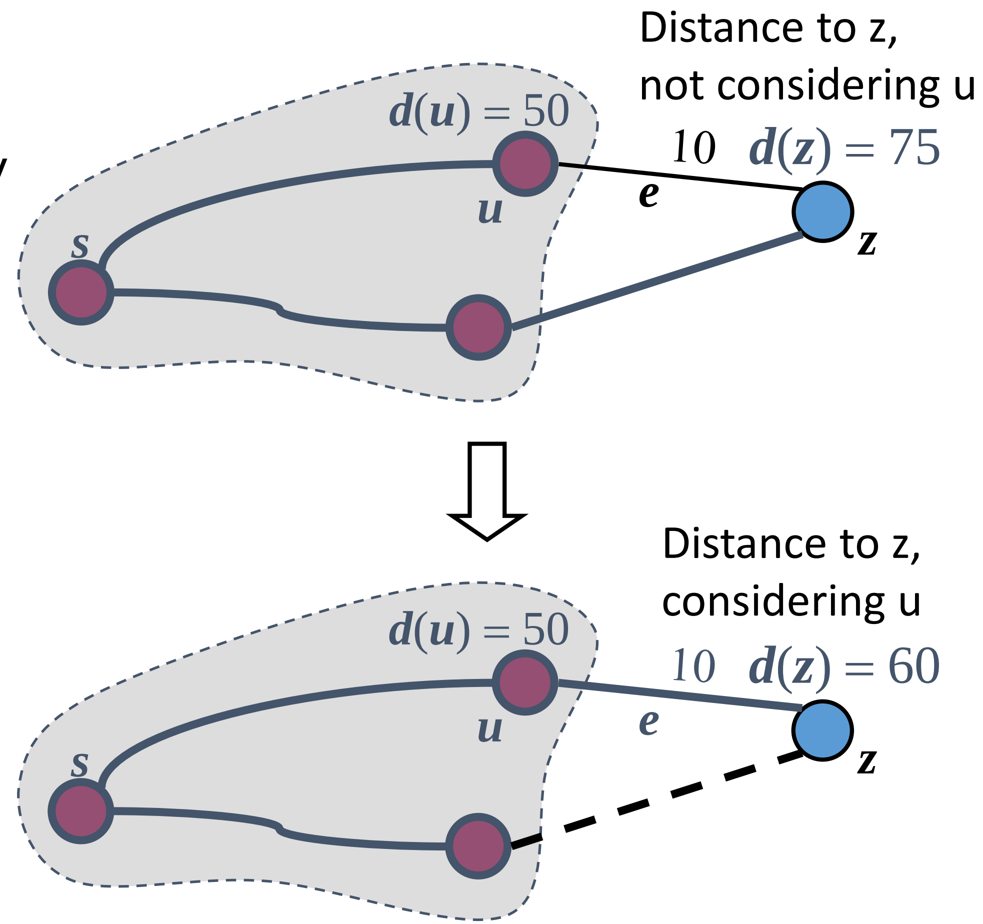
- **We store with each vertex $v$ a label $d(v)$ representing the distance of $v$ from $s$ in the subgraph consisting of the cloud and its adjacent vertices**
- **At each step**
  - We add to the cloud the vertex $u$ outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to $u$
  - Greedy method: we solve the problem at hand by repeatedly selecting the best choice from among those available in each iteration

> Raise your quality standards as high as you can live with, avoid wasting your time on routine problems, and always try to work as closely as possible at the boundary of your abilities. Do this, because it is the only way of discovering how that boundary should be moved forward.
>
> Edsger Dijkstra

# Edge Relaxation

- **Consider an edge $e = (u,z)$ such that**
  - $u$ is the vertex most recently added to the cloud
  - $z$ is not in the cloud
- **The relaxation of edge $e$ updates distance $d(z)$ as follows:**

$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



# Recall: Priority Queue ADT

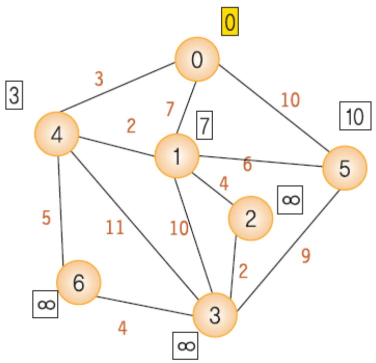
- **A priority queue stores a collection of entries**
- **Typically, an entry is a pair (key, value), where the key indicates the priority**
- **Main methods of the Priority Queue ADT**
  - insert(e) inserts an entry e
  - removeMin()
    removes the entry with smallest key
- **Additional methods**
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()

# Dijkstra's Algorithm

- **A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud**
    - Key: distance
    - Value: vertex
    - Recall that method ***replaceKey***(***l***,***k***) changes the key of entry ***l*** with ***k***
- **We store two labels with each vertex:**
    - Distance
    - Entry in priority queue
- **We take out the vertex with the minimum distance so far**

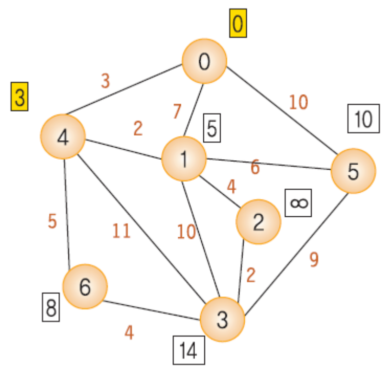
```
Algorithm DijkstraDistances(G, s)
    Q ← new heap-based priority queue
    for all v ∈ G.vertices()
        if v = s
            v.setDistance(0)
        else
            v.setDistance(∞)
        l ← Q.insert(v.getDistance(), v)
        v.setEntry(l)
    while ¬Q.empty()
        l ← Q.removeMin()
        u ← l.getValue() // take out the closest node
        for all e ∈ u.incidentEdges() { relax e }
            z ← e.opposite(u)
            r ← u.getDistance() + e.weight()
            if r < z.getDistance()
                z.setDistance(r)
                Q.replaceKey(z.getEntry(), r)
```

# Dijkstra’s alg. using distance array



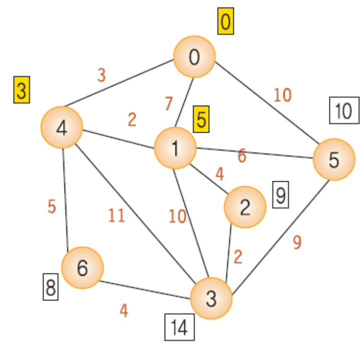

S={0}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 7 | ∞ | ∞ | 3 | 10 | ∞ |



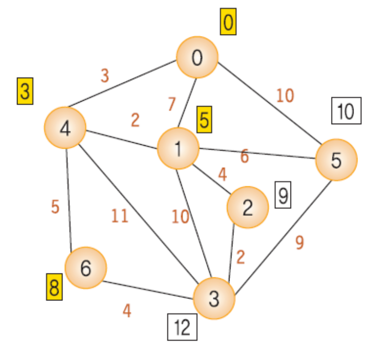

S={0, 4}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 5 | ∞ | 14 | 3 | 10 | 8 |



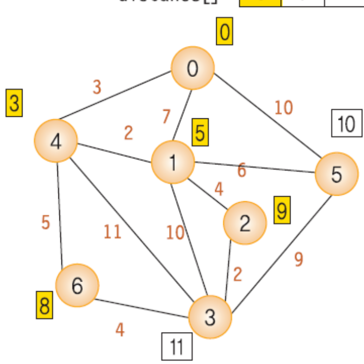

S={0, 4, 1}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 5 | 9 | 14 | 3 | 10 | 8 |



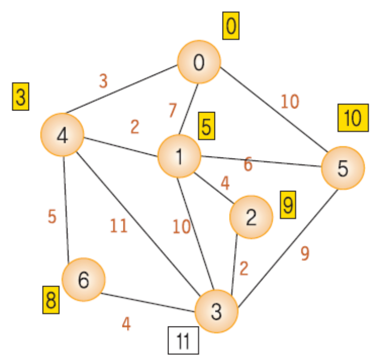

S={0, 4, 1, 6}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 5 | 9 | 12 | 3 | 10 | 8 |



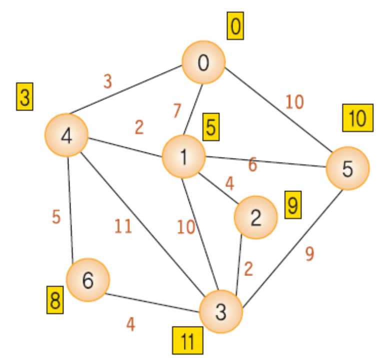

S={0, 4, 1, 6, 2}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 5 | 9 | 11 | 3 | 10 | 8 |




S={0, 4, 1, 6, 2, 5}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 5 | 9 | 11 | 3 | 10 | 8 |

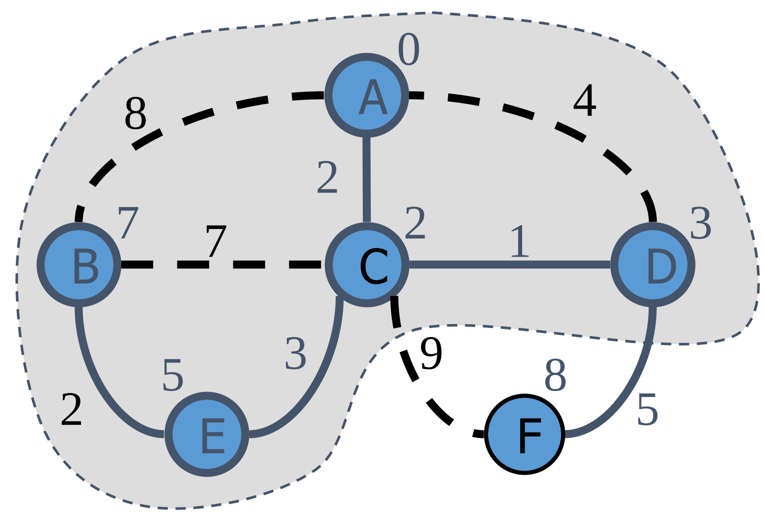



S={0, 4, 1, 6, 2, 5, 3}

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| distance[]= | 0 | 5 | 9 | 11 | 3 | 10 | 8 |

# Why Dijkstra's Algorithm Works

- **Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.**
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex




- (Question) Why not working for non-negative weight?

# Analysis of Dijkstra's Algorithm

- **Graph operations**
  - incidentEdges is called once for each $\boldsymbol{v}$
- **Label operations**
  - We set/get the distance and locator labels of vertex $z$, $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- **Priority queue operations**
  - Each $\boldsymbol{v}$ is inserted once into and removed once from the PQ, where each insertion or removal takes $O(\log n)$ time → total $nO(\log n)$
  - The key of a vertex in the PQ is modified at most $\deg(v)$ times, where each key change takes $O(\log n)$ time
- **Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure**
  - Recall that $\Sigma_v \deg(v) = 2m$
- **The running time can also be expressed as $O(m \log n)$ since the graph is connected**

**Algorithm** *DijkstraDistances*($G$, $s$)
  $Q \leftarrow$ new heap-based priority queue
  **for all** $v \in G.vertices()$
    **if** $v = s$
      $v.setDistance(0)$
    **else**
      $v.setDistance(\infty)$
    $l \leftarrow Q.insert(v.getDistance(), v)$
    $v.setEntry(l)$
  **while** $\neg Q.empty()$
    $l \leftarrow Q.removeMin()$
    $u \leftarrow l.getValue()$ // take out the closest node
    **for all** $e \in u.incidentEdges()$ { relax $e$ }
      $z \leftarrow e.opposite(u)$
      $r \leftarrow u.getDistance() + e.weight()$
      **if** $r < z.getDistance()$
        $z.setDistance(r)$
        $Q.replaceKey(z.getEntry(), r)$