ENG 346
Data Structures and Algorithms for Artificial Intelligence
Graphs

Dr. Mehmet PEKMEZCİ

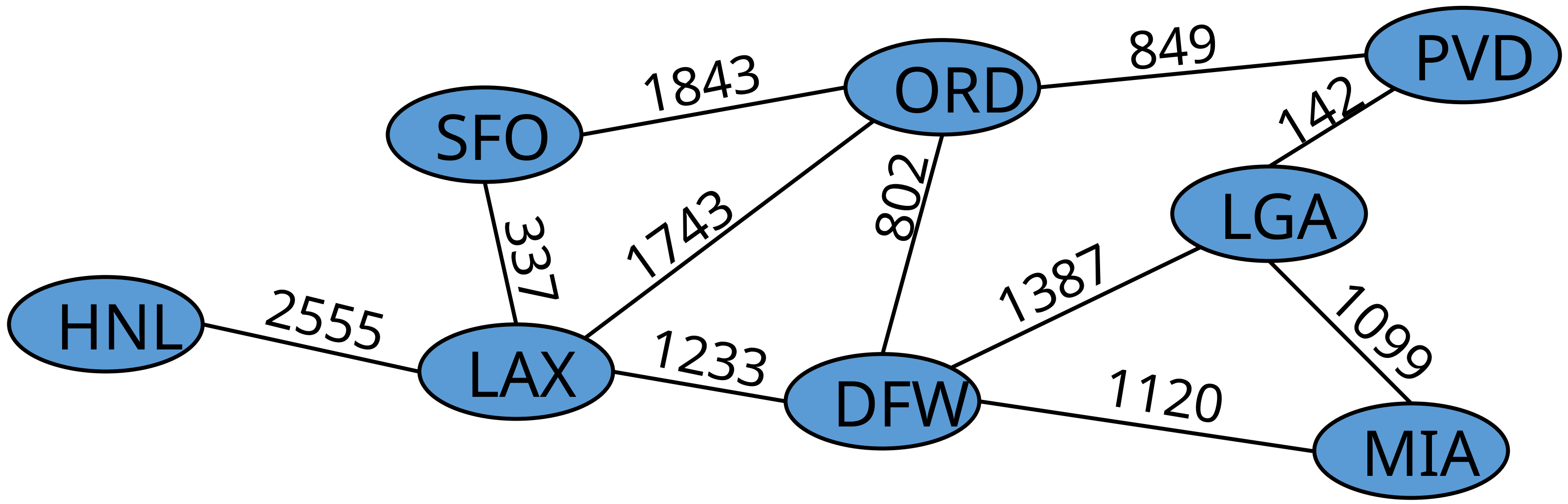
mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

ENG-346-FALL-2025 Teams code is **4b108kr**

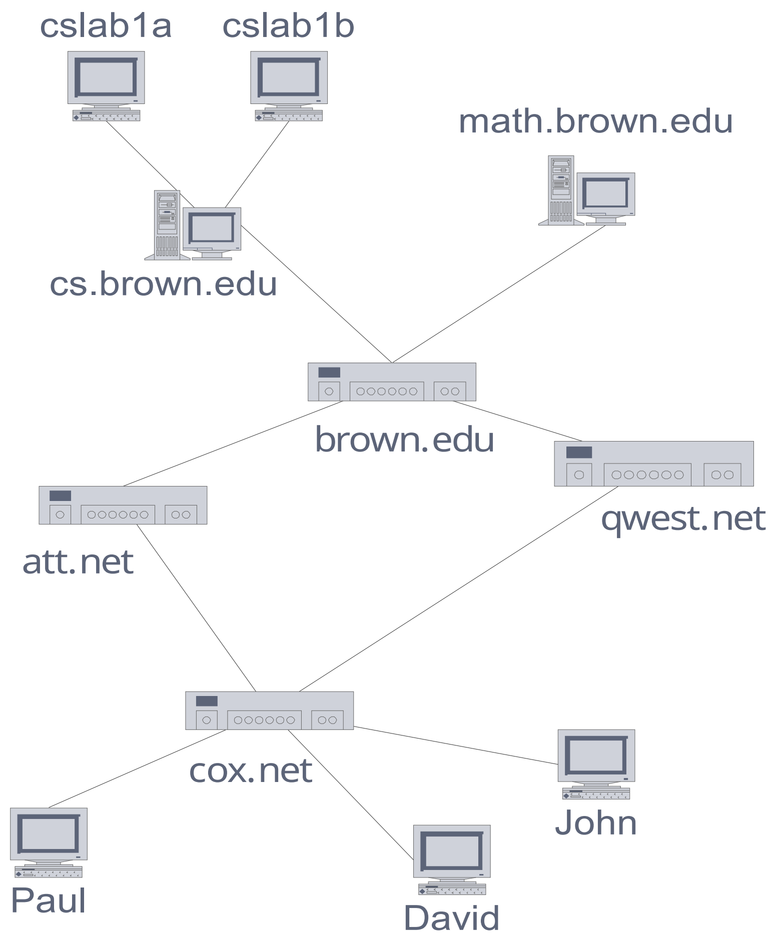
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



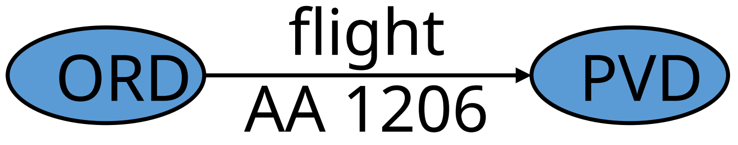
Applications

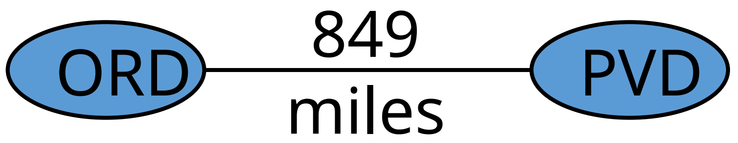
- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



Edge Types

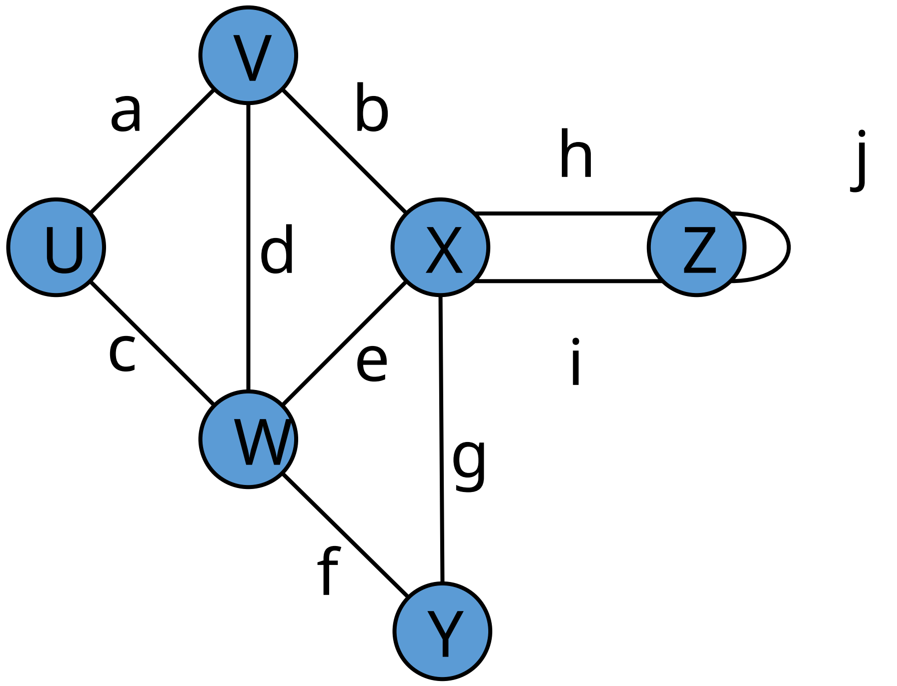
- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network





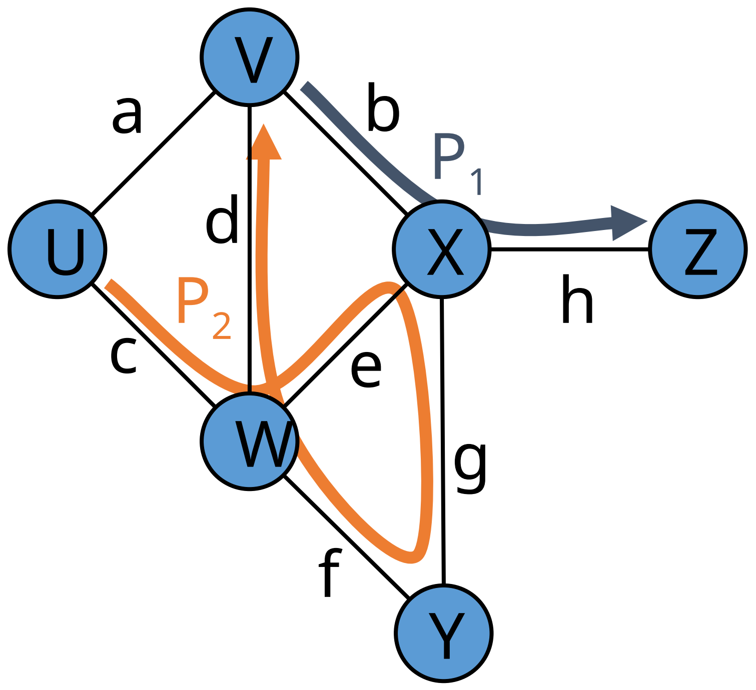
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent/Neighbor vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



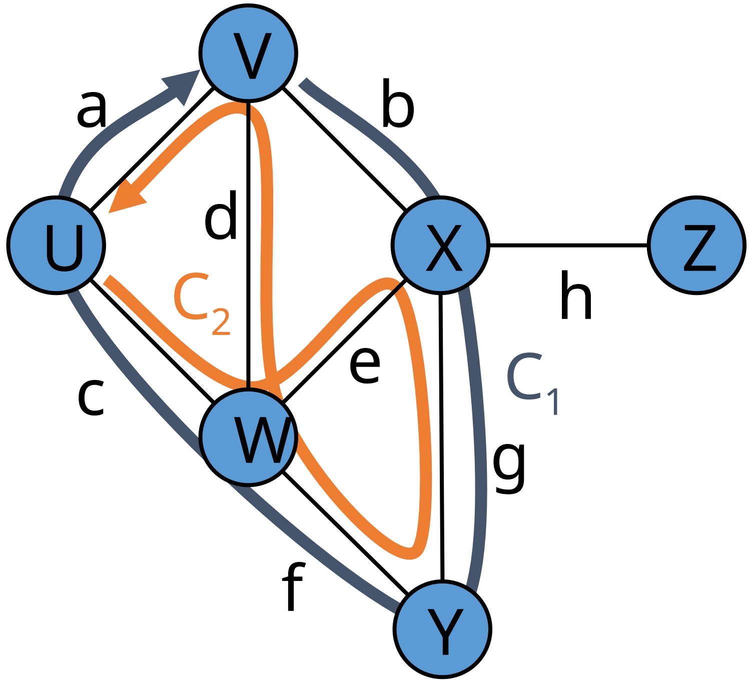
Terminology – continued

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology – continued

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a, ᆵ) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a, ᆵ) is a cycle that is not simple



Properties

Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

Property 2

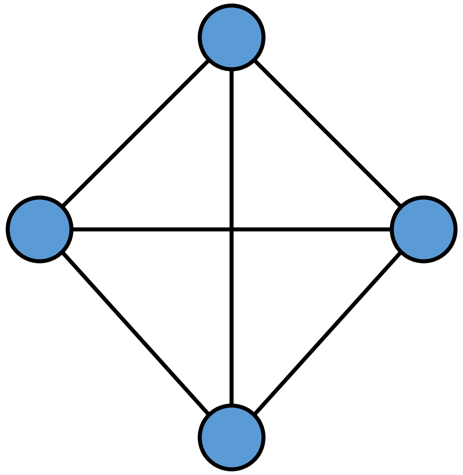
In an undirected graph with no self-loops and no multiple edges

$$m \leq n(n-1)/2$$

Proof: each vertex has degree at most $(n-1)$

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

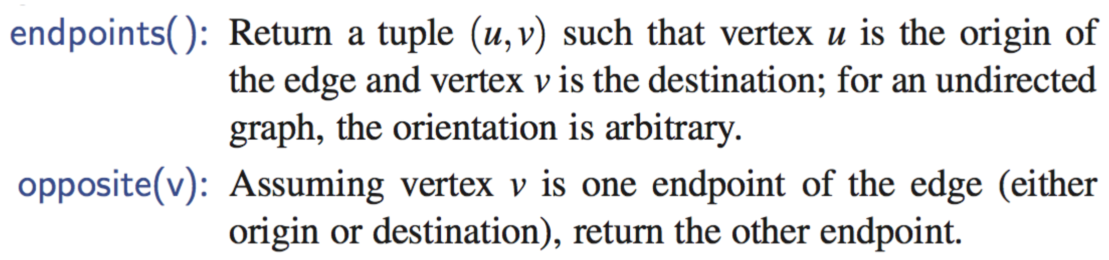


Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.
- In addition, we assume that an Edge supports the following methods:



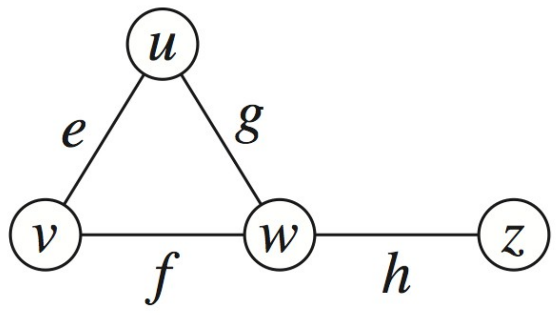
endpoints( ):	Return a tuple (u, v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.
opposite(v):	Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

Graph ADT

vertex_count( ):	Return the number of vertices of the graph.
vertices( ):	Return an iteration of all the vertices of the graph.
edge_count( ):	Return the number of edges of the graph.
edges( ):	Return an iteration of all the edges of the graph.
get_edge(u,v):	Return the edge from vertex *u* to vertex *v*, if one exists; otherwise return None. For an undirected graph, there is no difference between get_edge(u,v) and get_edge(v,u).
degree(v, out=True):	For an undirected graph, return the number of edges incident to vertex *v*. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex *v*, as designated by the optional parameter.
incident_edges(v, out=True):	Return an iteration of all edges incident to vertex *v*. In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to False.
insert_vertex(x=None):	Create and return a new Vertex storing element *x*.
insert_edge(u, v, x=None):	Create and return a new Edge from vertex *u* to vertex *v*, storing element *x* (None by default).
remove_vertex(v):	Remove vertex *v* and all its incident edges from the graph.
remove_edge(e):	Remove edge *e* from the graph.

Edge List Structure

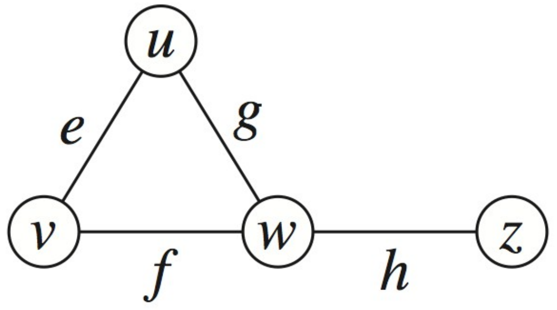
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



edges = [(u,v), (u,w), (v,w), (w,z)]

Adjacency List Structure

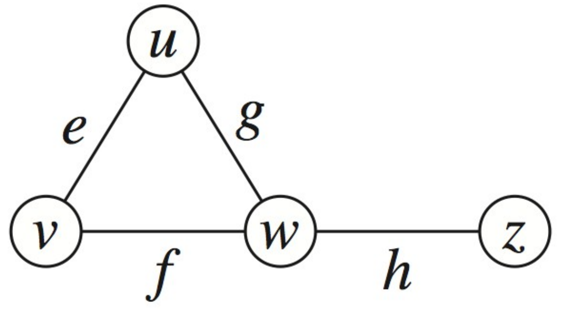
- Lists neighbors for each vertex

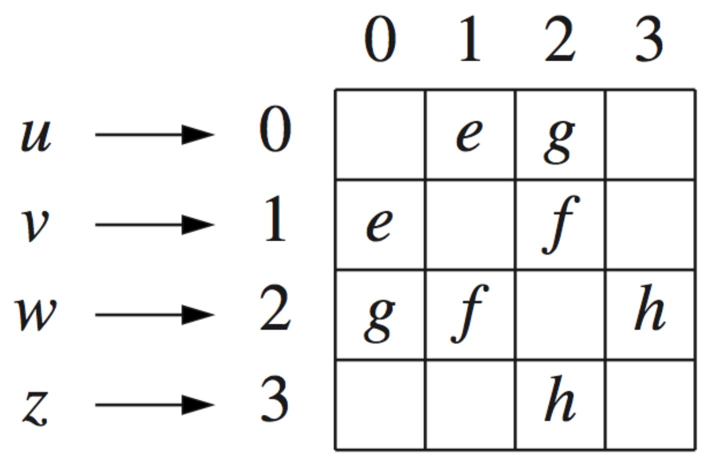


```
{
        u: [v, w],
        v: [u, w],
        w: [u, v, z],
        z: [w]
}
```

Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge





$$\begin{array}{ccc|c|c|c|c|}
 & & & 0 & 1 & 2 & 3 \\ \hline
u & \longrightarrow & 0 & & e & g & \\ \hline
v & \longrightarrow & 1 & e & & f & \\ \hline
w & \longrightarrow & 2 & g & f & & h \\ \hline
z & \longrightarrow & 3 & & & h & \\ \hline
\end{array}$$

Performance

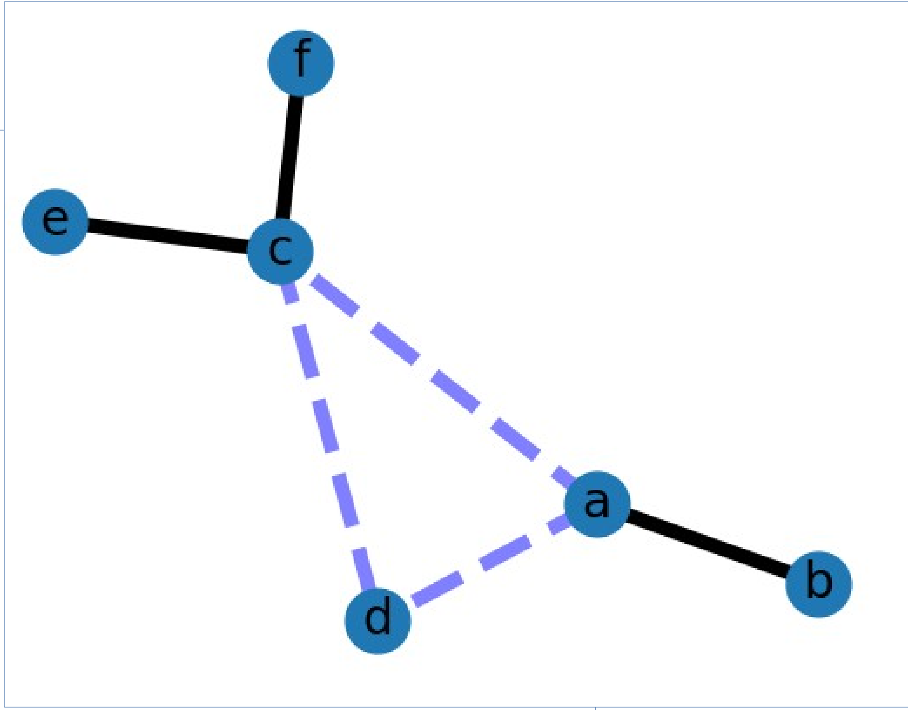
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

Python Graph Libs

- NetworkX : General graph implementation.
- Pytorch Geometric : Generally used in Graph Neural Network implementations.

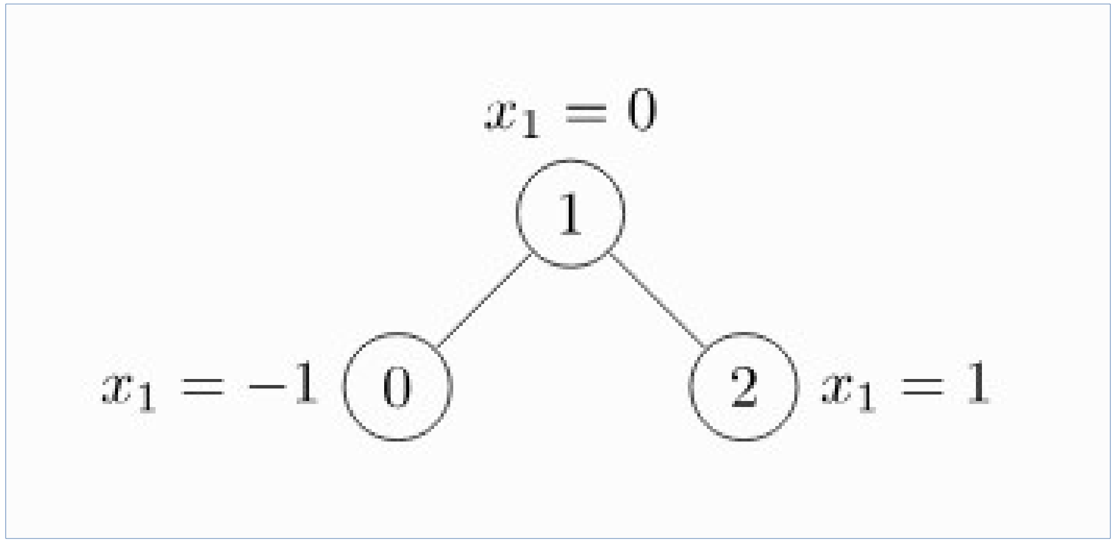
NetworkX Example

```
import matplotlib.pyplot as plt
import networkx as nx
G = nx.Graph()
G.add_edge('a', 'b', weight=0.6)
G.add_edge('a', 'c', weight=0.2)
G.add_edge('c', 'd', weight=0.1)
G.add_edge('c', 'e', weight=0.7)
G.add_edge('c', 'f', weight=0.9)
G.add_edge('a', 'd', weight=0.3)
elarge = [(u, v) for (u, v, d) in G.edges(data=True) if d['weight'] > 0.5]
esmall = [(u, v) for (u, v, d) in G.edges(data=True) if d['weight'] <= 0.5]
pos = nx.spring_layout(G)  # positions for all nodes
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_edges(G, pos, edgelist=elarge,width=6)
nx.draw_networkx_edges(G, pos, edgelist=esmall,width=6, alpha=0.5, edge_color='b', style='dashed')
nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')
plt.axis('off')
plt.show()
```



Pytorch Geometric Example

```
import torch
from torch_geometric.data import Data
edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)
data = Data(x=x, edge_index=edge_index)
>>> Data(edge_index=[2, 4], x=[3, 1])
```



Python Graph Implementation

- We use a variant of the ***adjacency map*** representation.
- For each vertex *v*, we use a Python dictionary to represent the secondary incidence map *I*(*v*).
- The list *V* is replaced by a top-level dictionary *D* that maps each vertex *v* to its incidence map *I*(*v*).
    - Note that we can iterate through all vertices by generating the set of keys for dictionary *D*.
- A vertex does not need to explicitly maintain a reference to its position in *D*, because it can be determined in *O*(1) expected time.
- Running time bounds for the adjacency-list graph ADT operations, given above, become ***expected*** bounds.

Vertex Class

```
#------------------------ nested Vertex class ------------------------
class Vertex:
  """Lightweight vertex structure for a graph."""
  __slots__ = '_element'

  def __init__(self, x):
    """Do not call constructor directly. Use Graph's insert_vertex(x)."""
    self._element = x

  def element(self):
    """Return element associated with this vertex."""
    return self._element

  def __hash__(self):            # will allow vertex to be a map/set key
    return hash(id(self))
```

Edge Class

```
#------------------------- nested Edge class -------------------------
class Edge:
  """Lightweight edge structure for a graph."""
  __slots__ = '_origin', '_destination', '_element'

  def __init__(self, u, v, x):
    """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    """Return (u,v) tuple for vertices u and v."""
    return (self._origin, self._destination)

  def opposite(self, v):
    """Return the vertex that is opposite v on this edge."""
    return self._destination if v is self._origin else self._origin

  def element(self):
    """Return element associated with this edge."""
    return self._element

  def __hash__(self):              # will allow edge to be a map/set key
    return hash( (self._origin, self._destination) )
```

Graph, Part 1

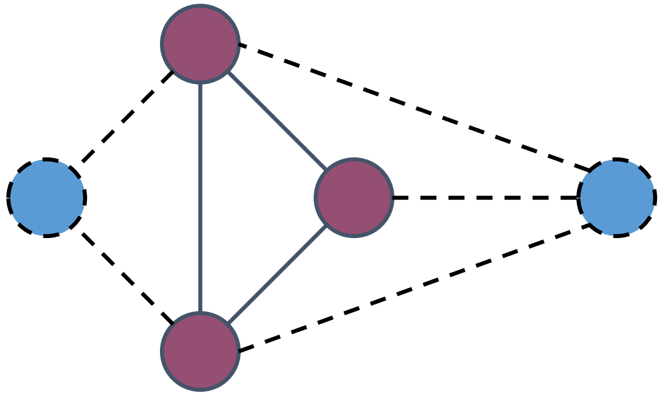
```
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for undirected
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected.

    Property is based on the original declaration of the graph, not its contents.
    """
    return self._incoming is not self._outgoing # directed if maps are distinct

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)

  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys( )

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count edges
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )          # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values( ):
      result.update(secondary_map.values( ))       # add edges to resulting set
    return result
```

Graph, Part

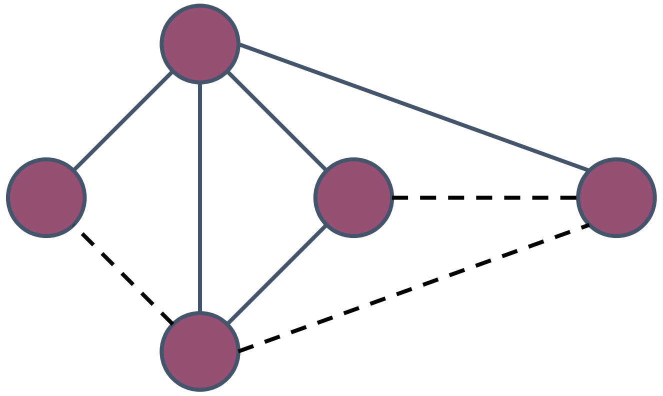
```
def get_edge(self, u, v):
  """Return the edge from u to v, or None if not adjacent."""
  return self._outgoing[u].get(v)          # returns None if v not adjacent

def degree(self, v, outgoing=True):
  """Return number of (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to count incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  return len(adj[v])

def incident_edges(self, v, outgoing=True):
  """Return all (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to request incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  for edge in adj[v].values():
    yield edge

def insert_vertex(self, x=None):
  """Insert and return a new Vertex with element x."""
  v = self.Vertex(x)
  self._outgoing[v] = { }
  if self.is_directed():
    self._incoming[v] = { }          # need distinct map for incoming edges
  return v

def insert_edge(self, u, v, x=None):
  """Insert and return a new Edge from u to v with auxiliary element x."""
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



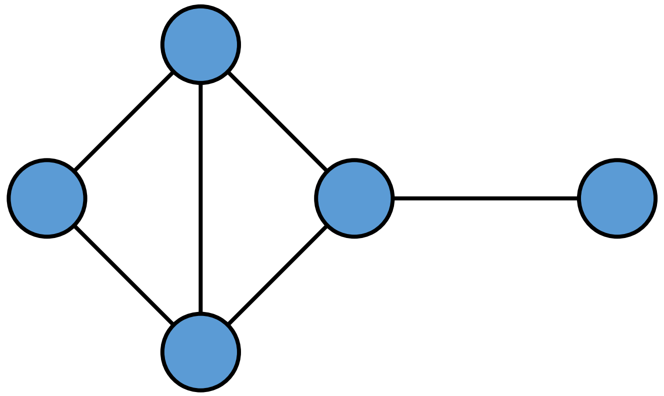
Subgraph

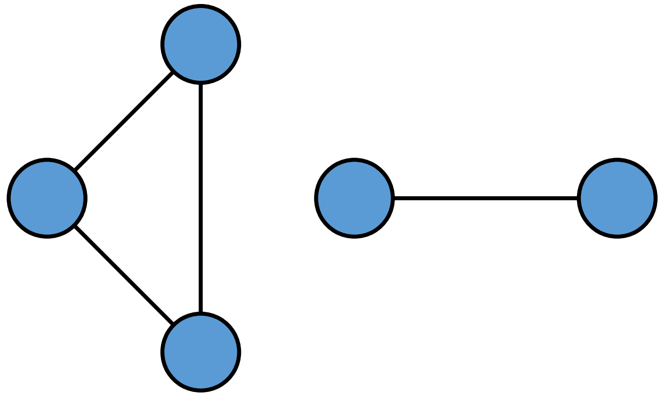


Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

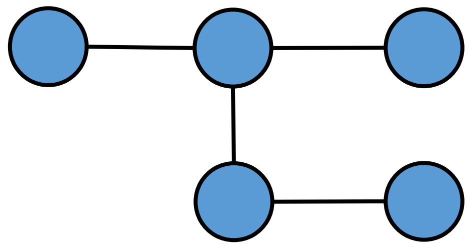

Connected graph

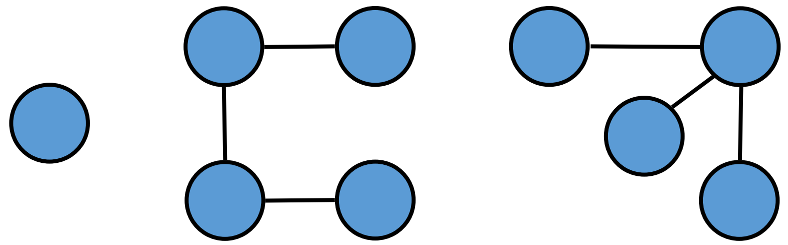

Non connected graph with two connected components

Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

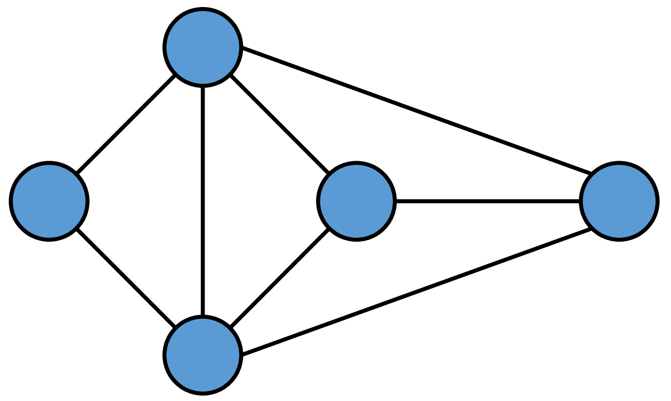
  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

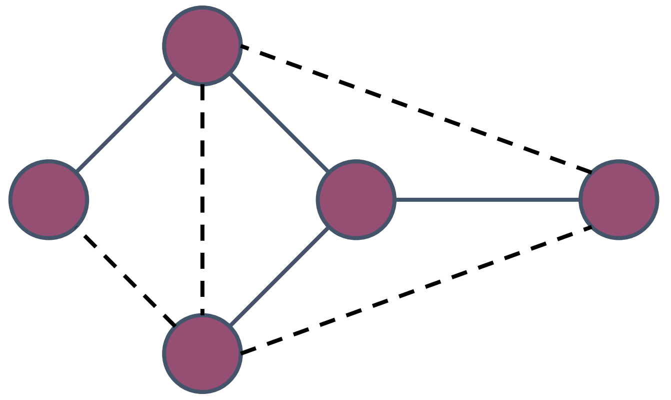

Tree


Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest


Graph


Spanning tree

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS 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

- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm

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

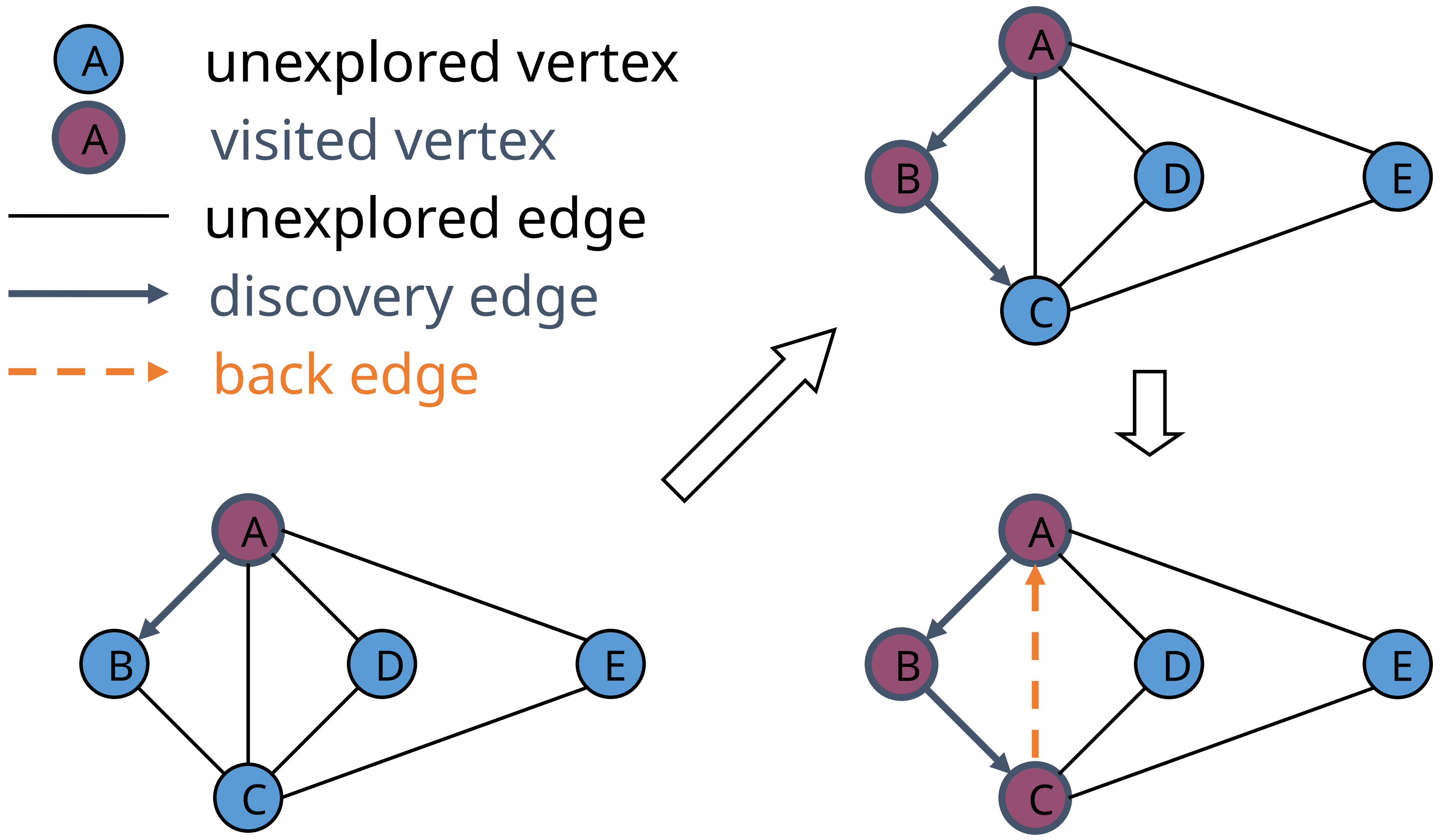
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u $\in$ G.vertices()
      setLabel(u, UNEXPLORED)
  for all e $\in$ G.edges()
      setLabel(e, UNEXPLORED)
  for all v $\in$ G.vertices()
      if getLabel(v) = UNEXPLORED
          DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
  setLabel(v, VISITED)
  for all e $\in$ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
          w $\leftarrow$ opposite(v,e)
          if getLabel(w) = UNEXPLORED
              setLabel(e, DISCOVERY)
              DFS(G, w)
          else
              setLabel(e, BACK)
```

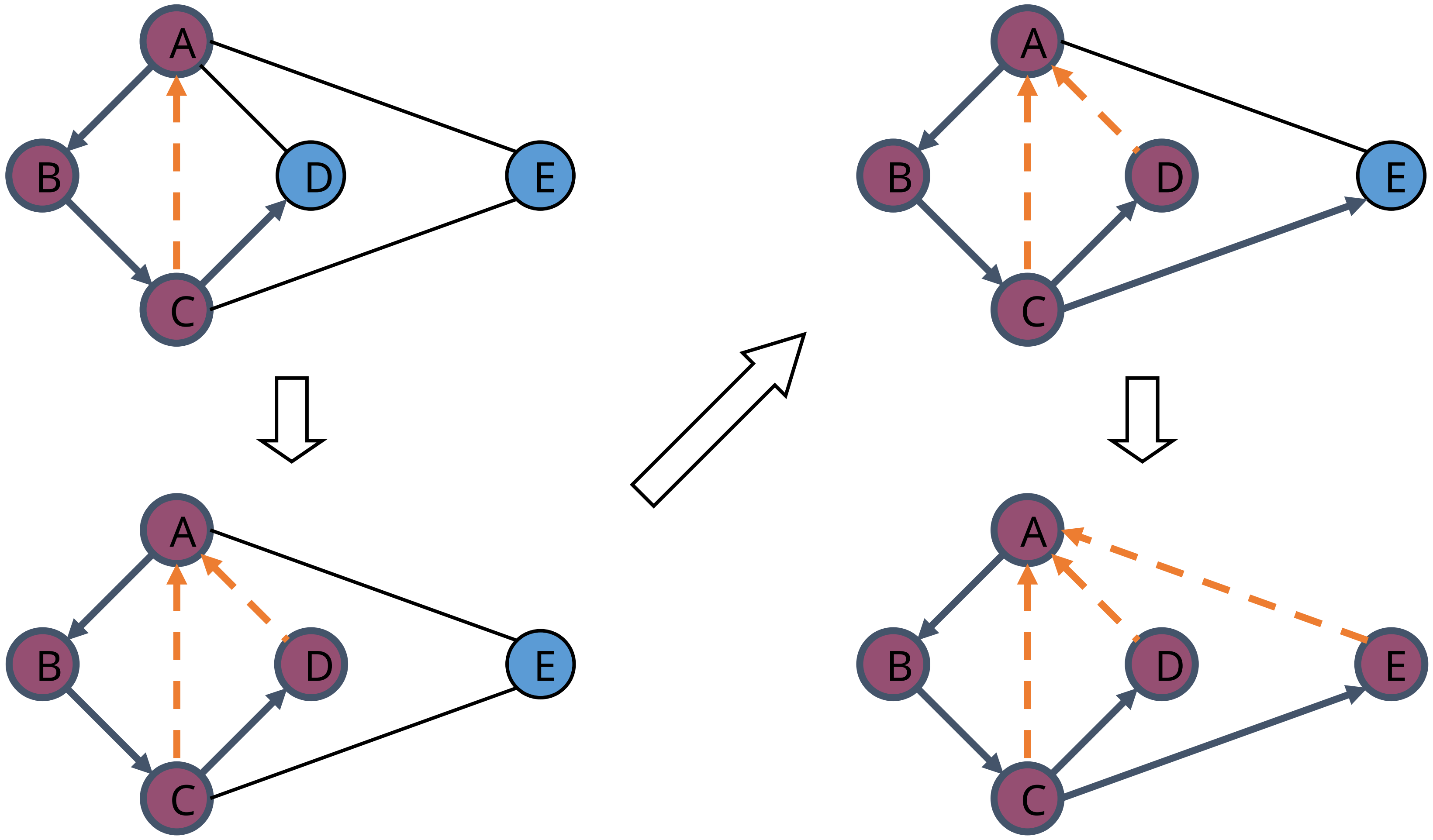
Python Implementation

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):           # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:               # v is an unvisited vertex
      discovered[v] = e                   # e is the tree edge that discovered v
      DFS(g, v, discovered)               # recursively explore from v
```

Example

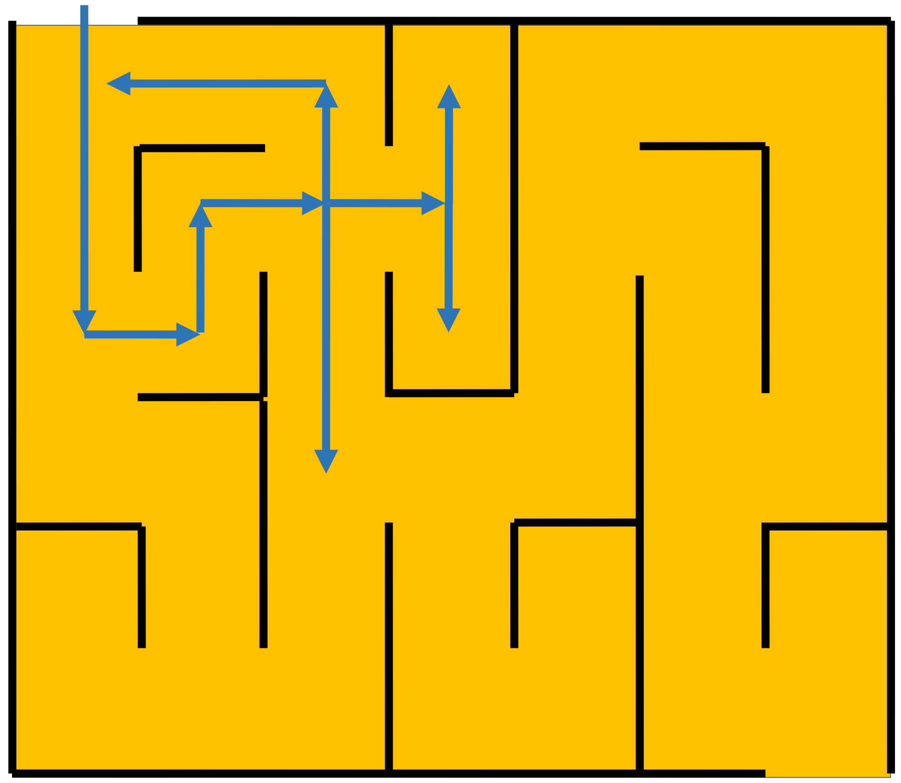

A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge
A
B
C
D
E
A
B
C
D
E
A
B
C
D
E

Example (cont.)



DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



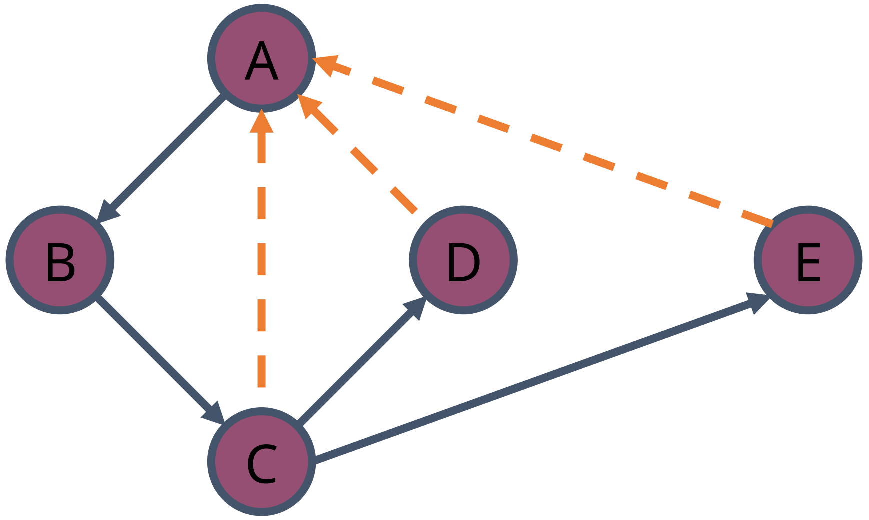
Properties of DFS

Property 1

DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v

Property 2

The discovery edges labeled by ***DFS*(*G*, *v*)** form a spanning tree of the connected component of ***v***



Analysis of DFS

- 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 BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call ***DFS***(G, u) with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
    S.pop(v)
```

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all  e ∈  G.incidentEdges(v)
        if  getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                pathDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- 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 $\boldsymbol{n}$ vertices and $\boldsymbol{m}$ edges takes $\boldsymbol{O}(\boldsymbol{n}+\boldsymbol{m})$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

BFS Algorithm

GEBZE TECHNICAL UNIVERSITY

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

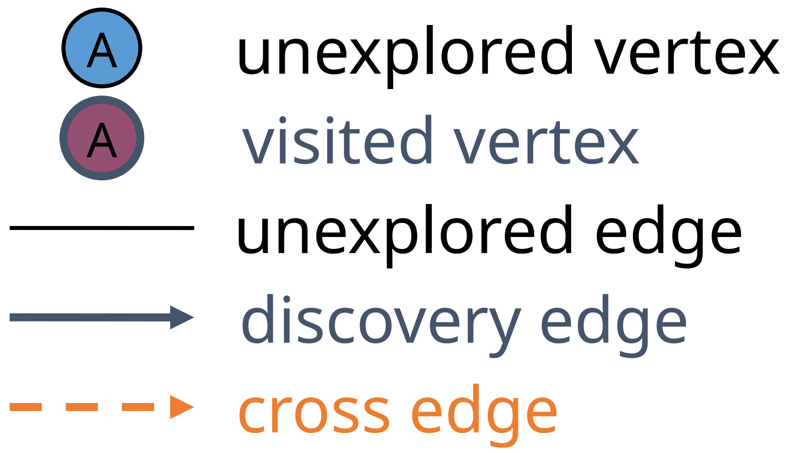
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices  of G
  for all $u \in G.vertices()$
      setLabel(u, UNEXPLORED)
  for all $e \in G.edges()$
      setLabel(e, UNEXPLORED)
  for all $v \in G.vertices()$
      if getLabel(v) =
  UNEXPLORED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

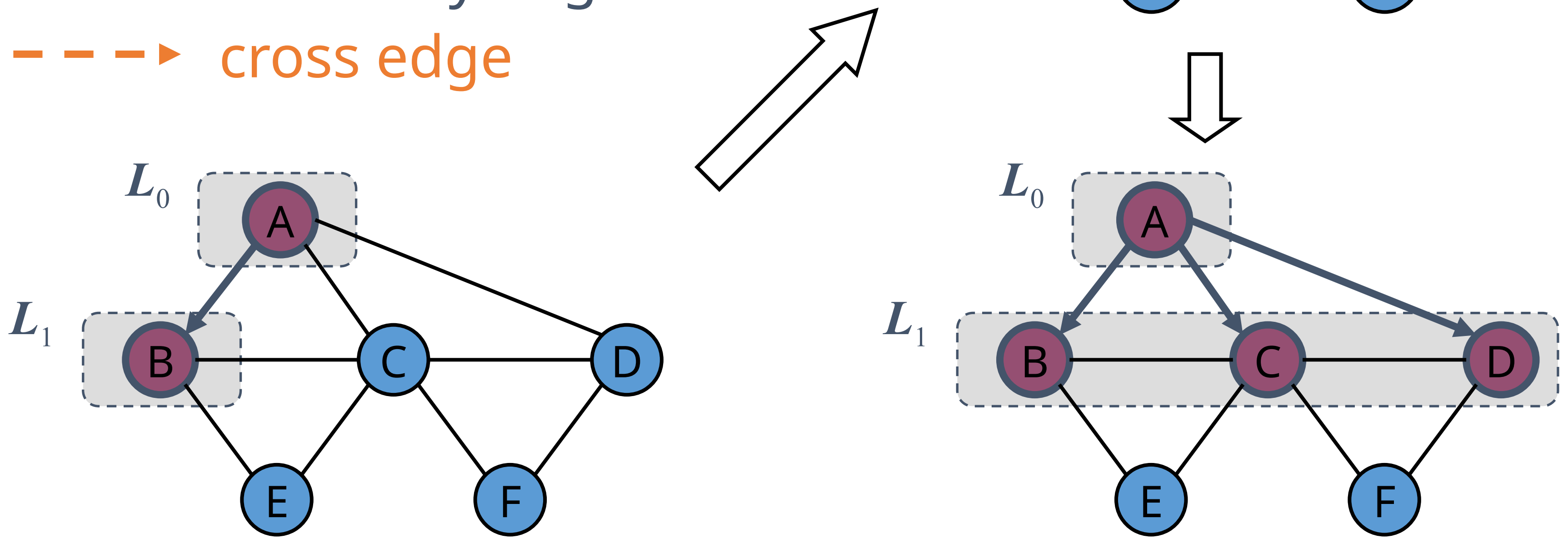
```
Algorithm BFS(G, s)
  $L_0 \leftarrow$ new empty sequence
  $L_0.addLast(s)$
  setLabel(s, VISITED)
  $i \leftarrow 0$
  while $\neg L_i.isEmpty()$
    $L_{i+1} \leftarrow$ new empty sequence
    for all $v \in L_i.elements()$
      for all $e \in G.incidentEdges(v)$
        if getLabel(e) = UNEXPLORED
          $w \leftarrow opposite(v,e)$
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            $L_{i+1}.addLast(w)$
          else
            setLabel(e, CROSS)
    $i \leftarrow i + 1$
```

Python Implementation

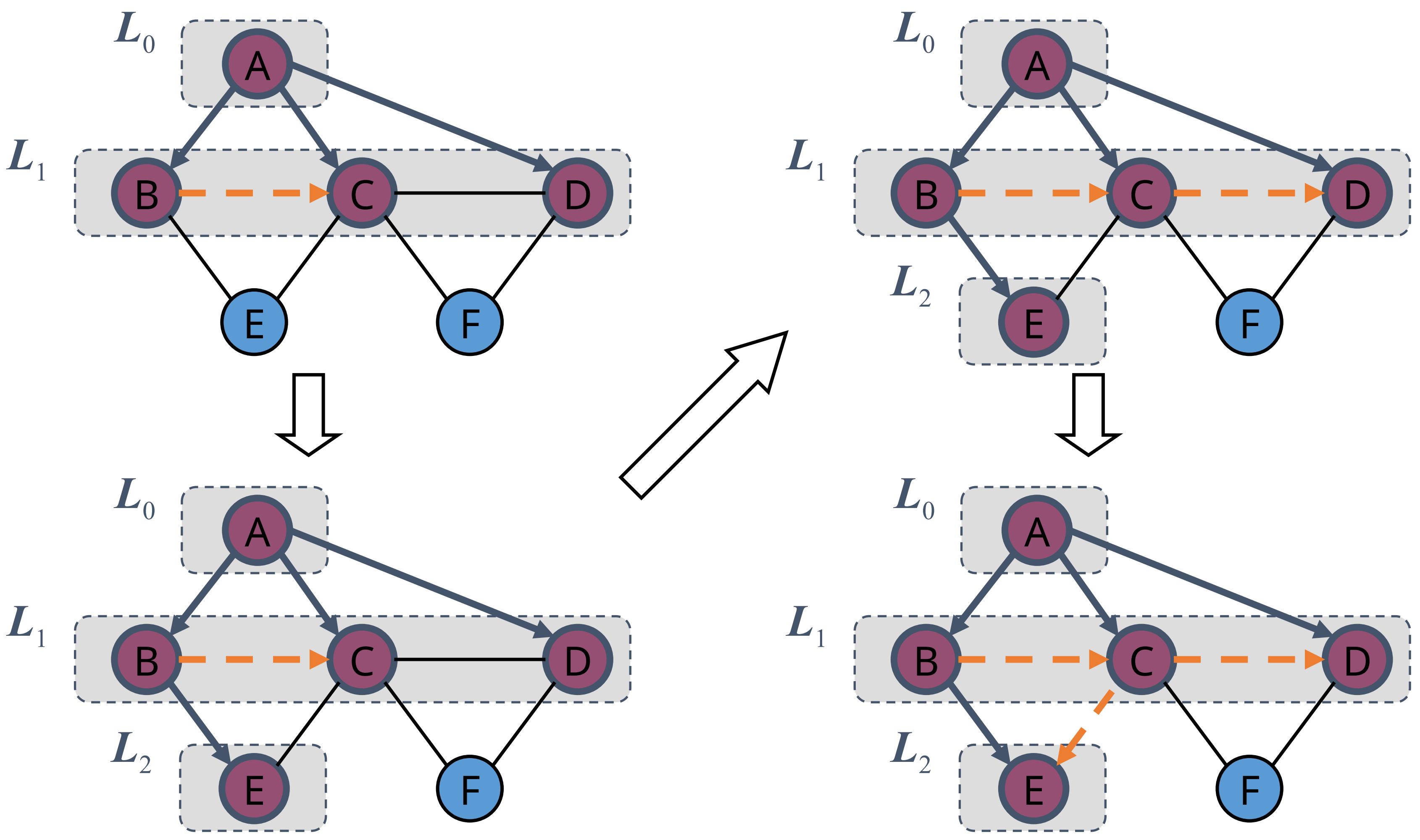
```
def BFS(g, s, discovered):
  """Perform BFS of the undiscovered portion of Graph g starting at Vertex s.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the BFS (s should be mapped to None prior to the call).
  Newly discovered vertices will be added to the dictionary as a result.
  """
  level = [s]                          # first level includes only s
  while len(level) > 0:
    next_level = [ ]                   # prepare to gather newly found vertices
    for u in level:
      for e in g.incident_edges(u):    # for every outgoing edge from u
        v = e.opposite(u)
        if v not in discovered:        # v is an unvisited vertex
          discovered[v] = e            # e is the tree edge that discovered v
          next_level.append(v)         # v will be further considered in next pass
    level = next_level                 # relabel 'next' level to become current
```

Example

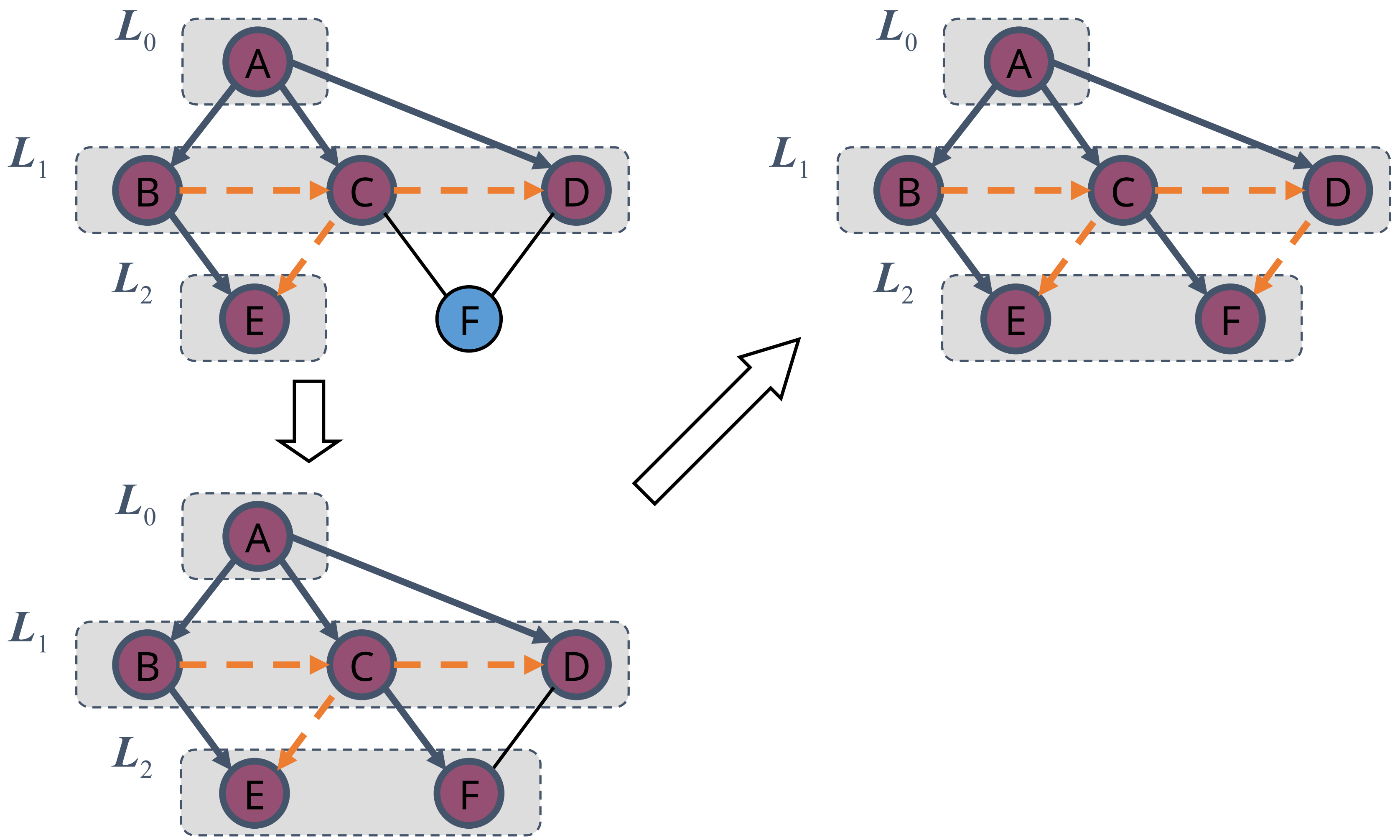

A
A
unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge


L_0
L_1
A
B
C
D
E
F

Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

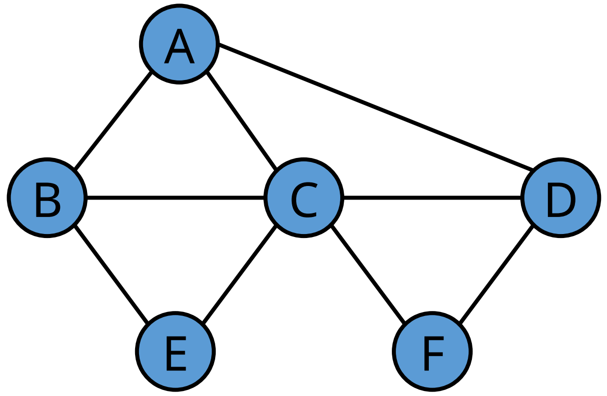
Property 2

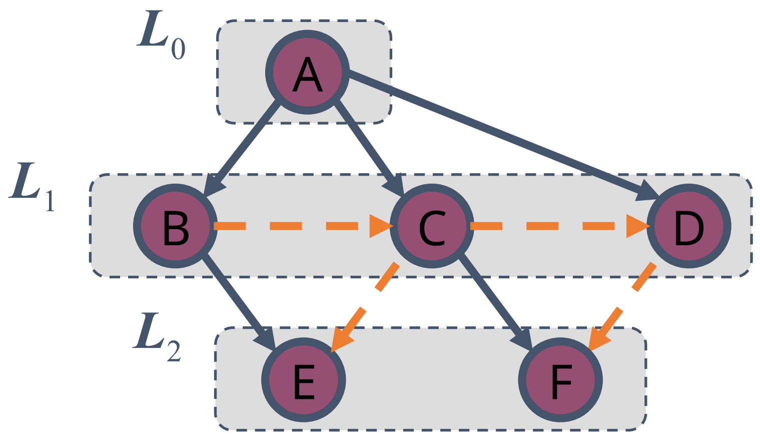
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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





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

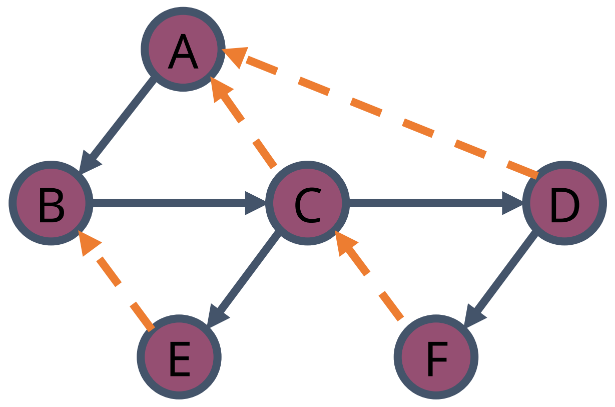
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n+m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find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

DFS vs BFS

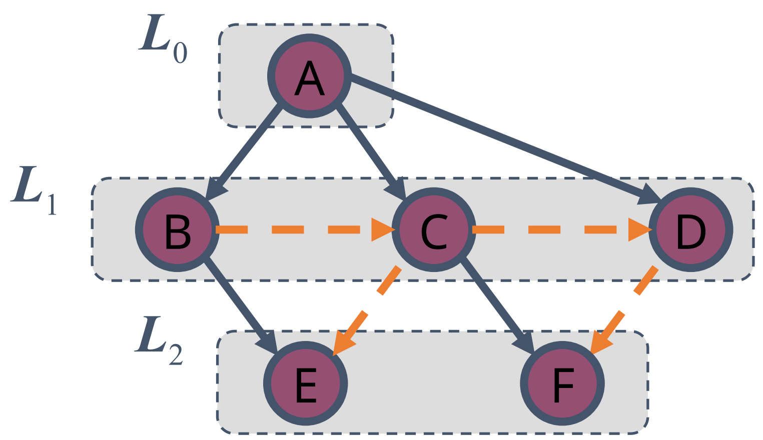
	Depth First Search (DFS)	Breadth First Search (BFS)
Data Structure	DFS uses Stack data structure.	BFS uses Queue data structure
Definition	DFS traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.
Conceptual Difference	DFS builds the tree sub-tree by sub-tree.	BFS builds the tree level by level.
Approach used	It works on the concept of LIFO (Last In First Out).	It works on the concept of FIFO (First In First Out).
Suitable for	DFS is more suitable when there are solutions away from source.	BFS is more suitable for searching vertices closer to the given source.
Applications	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc.	BFS is used in various applications such as bipartite graphs, shortest paths, etc.

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	



DFS



BFS

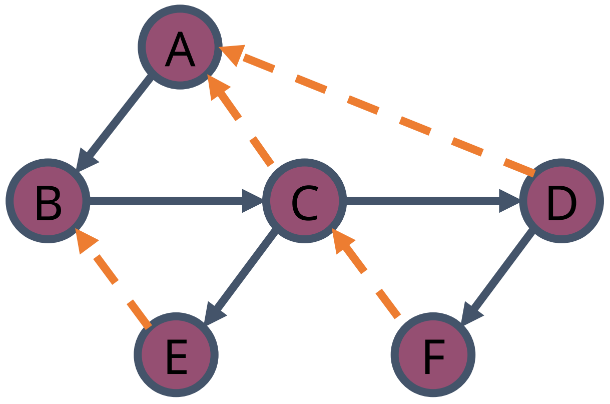
DFS vs. BFS (cont.)

Back edge (v,w)

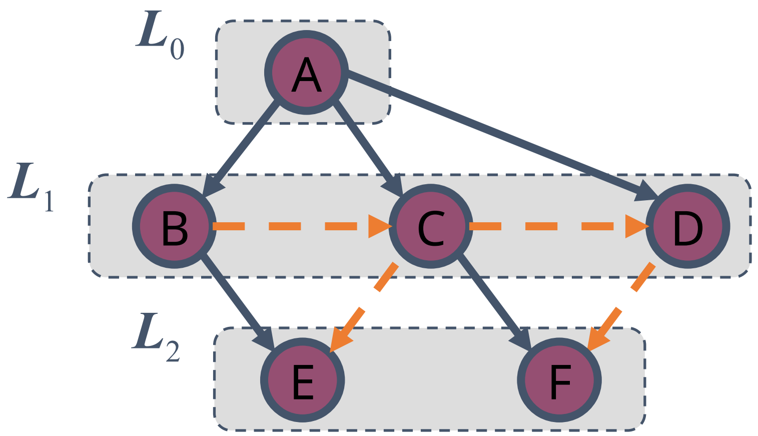
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

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



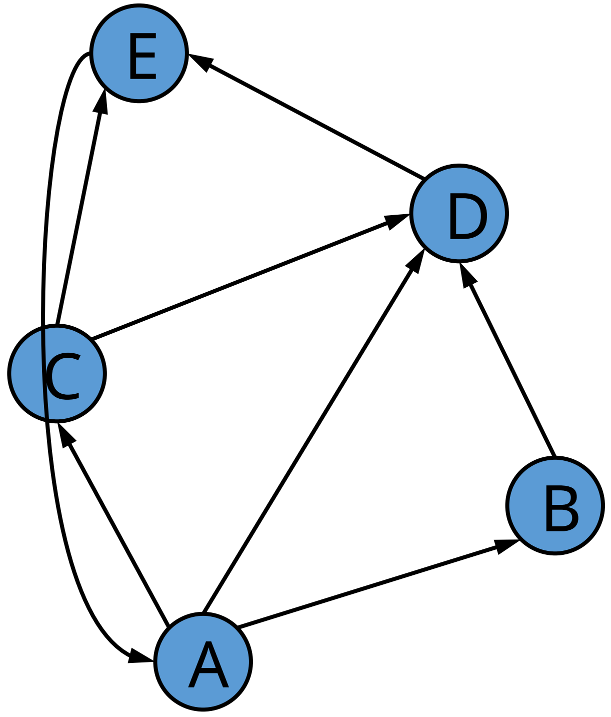
DFS



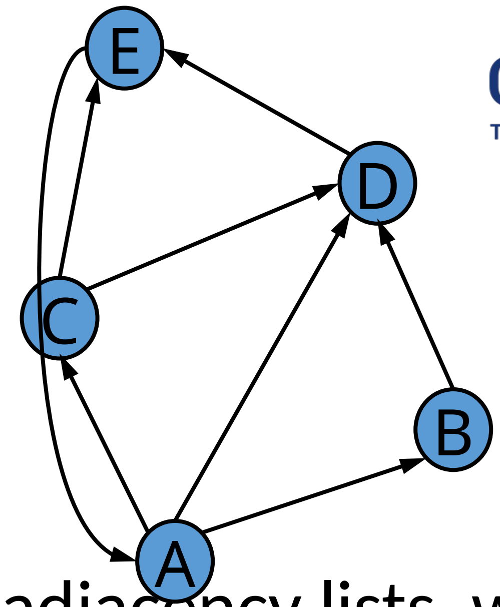
BFS

Directed Graphs

- A directed graph is a graph whose edges are all directed
- Applications
  - one-way streets
  - flights
  - task scheduling



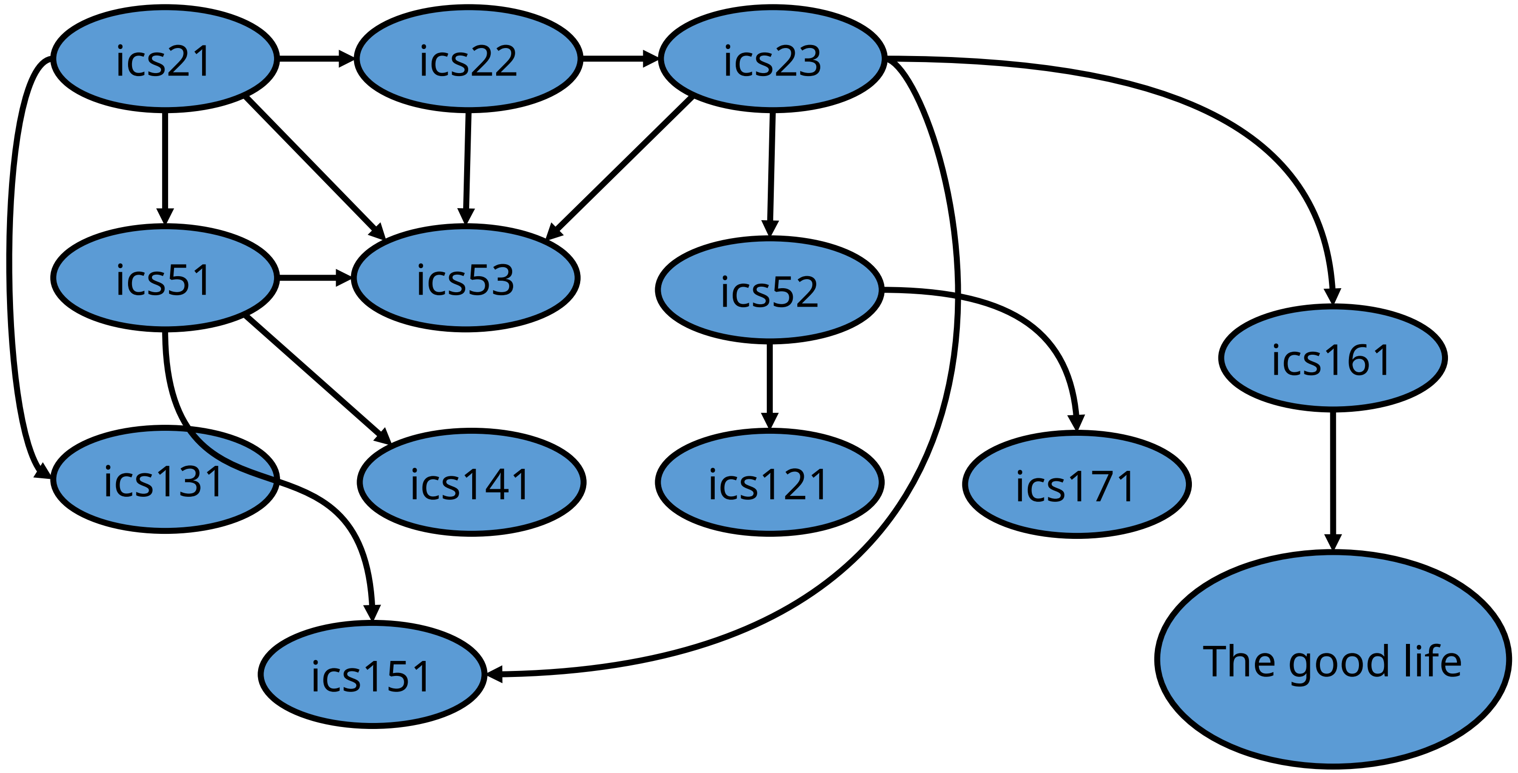
Directed Graph Properties



- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a,b) goes from a to b, but not b to a
- If G is simple, $m \leq n(n-1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

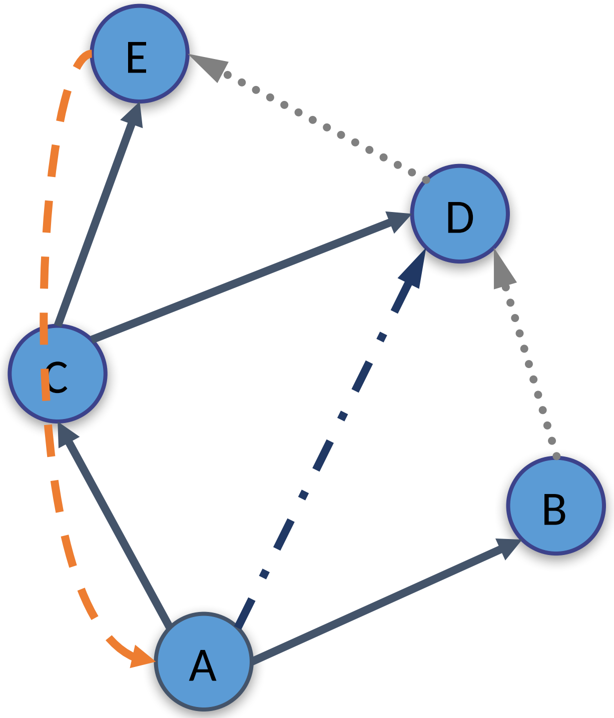
Directed Graph Application

- Scheduling: edge (a,b) means task a must be completed before b can be started



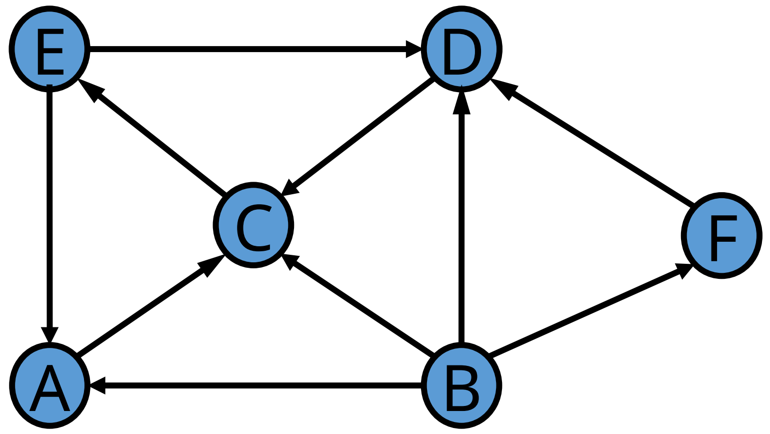
Directed DFS

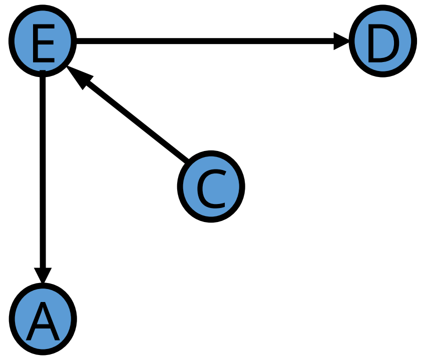
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s

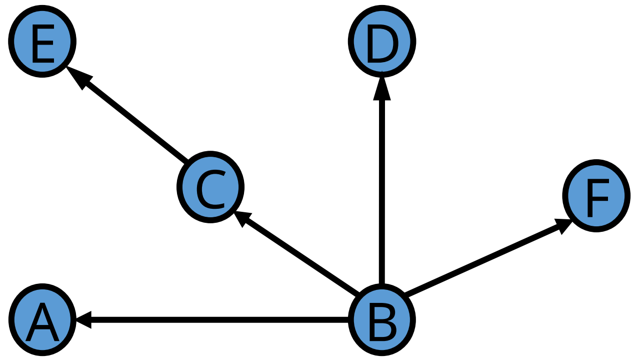


Reachability

- DFS tree rooted at v: vertices reachable from v via directed paths

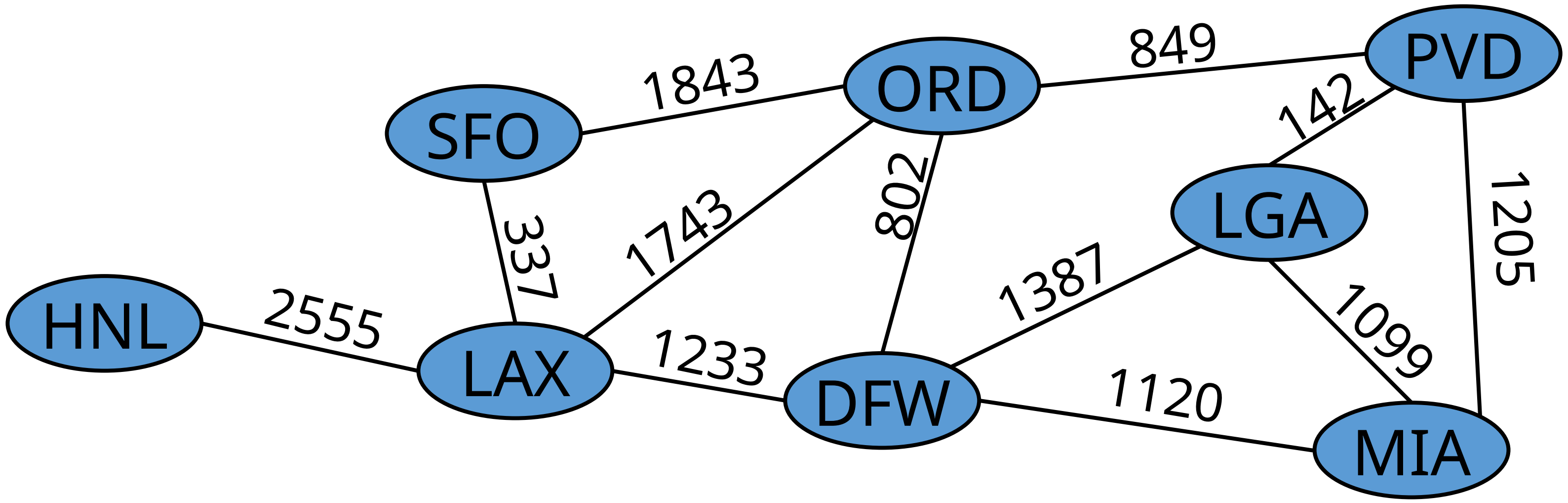






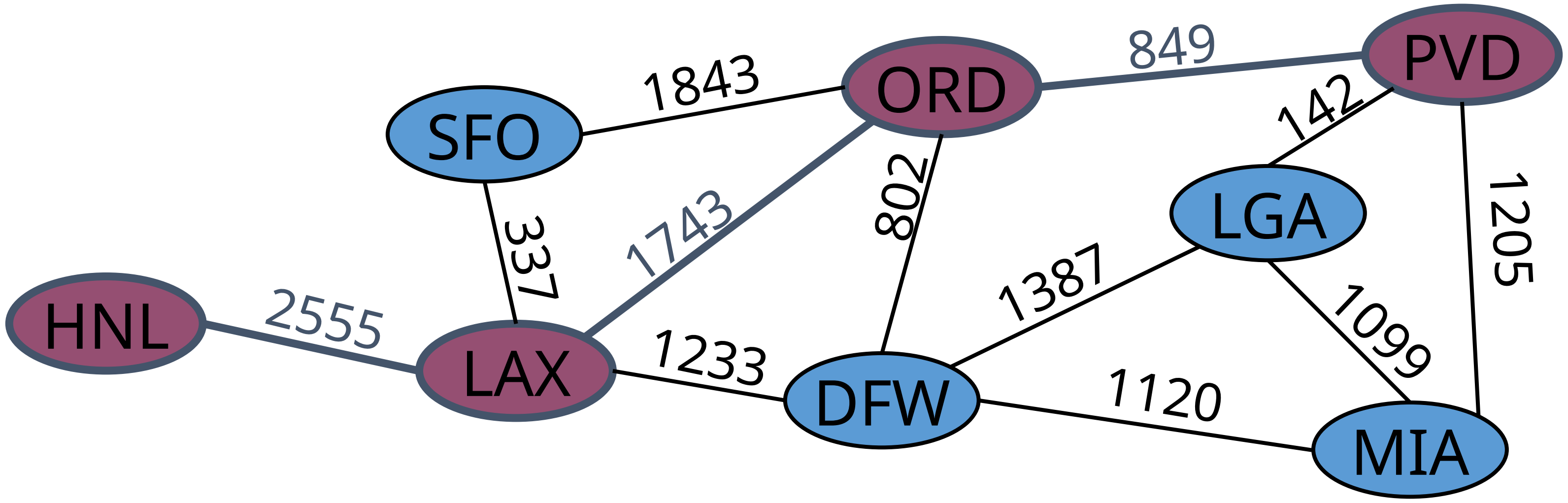
Weighted Graphs

- 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:

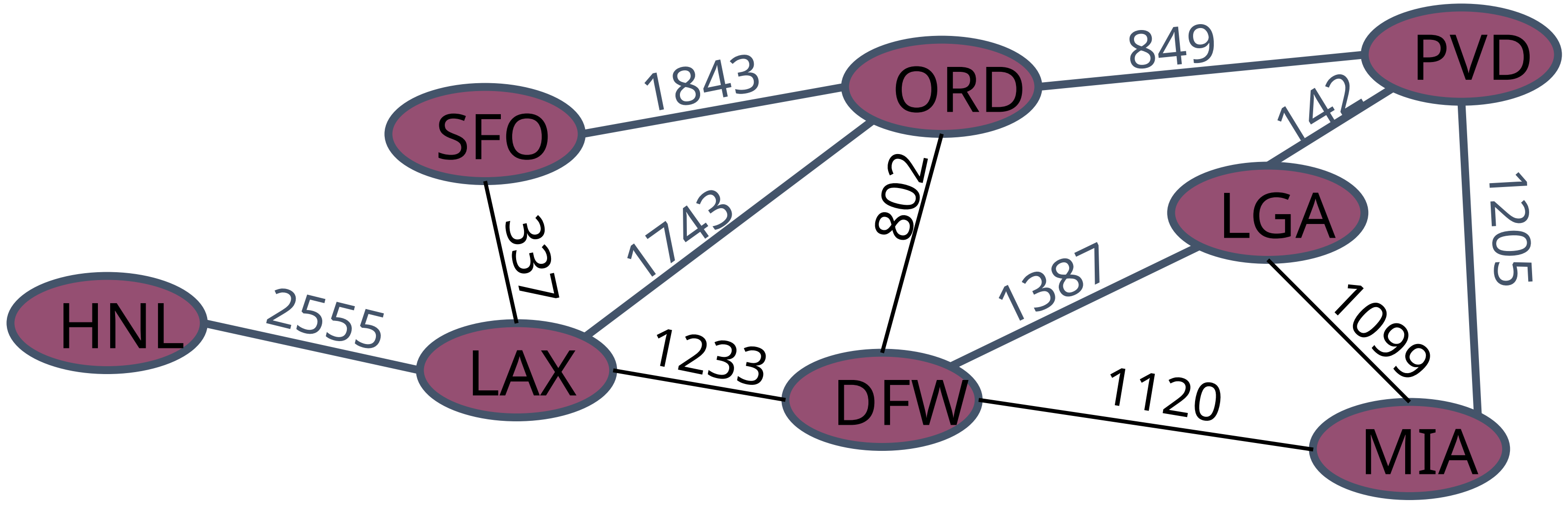
A subpath of a shortest path is itself a shortest path

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



Dijkstra’s Algorithm

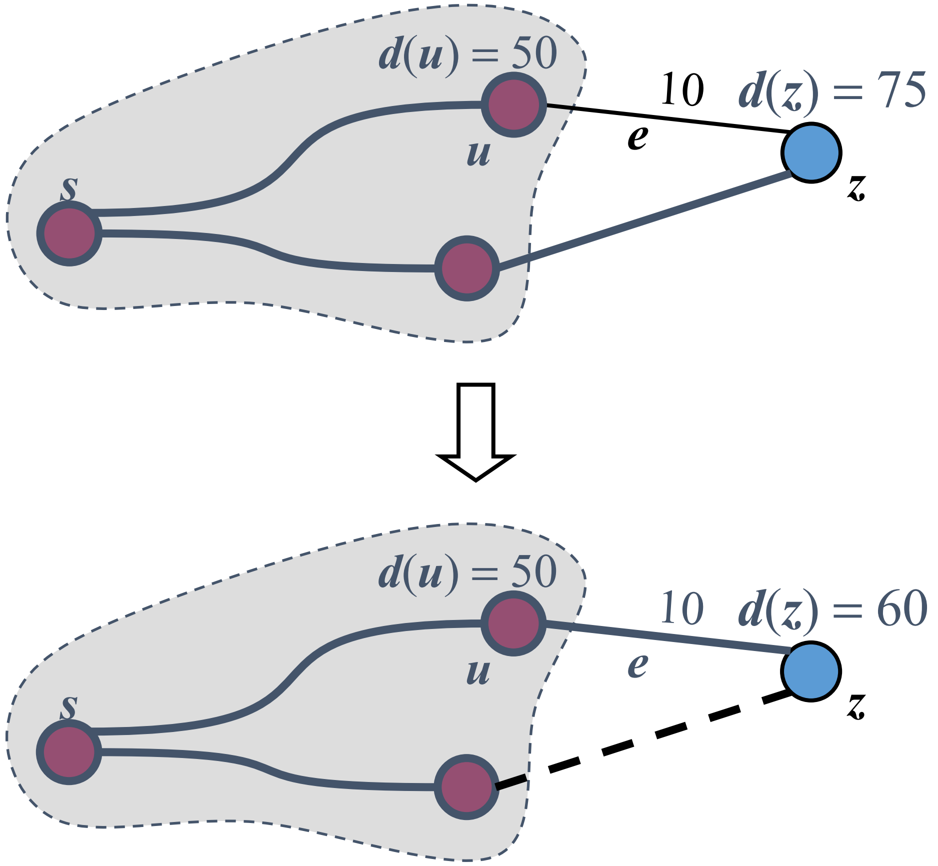
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a “cloud” of vertices, beginning with s and eventually covering all the vertices
- 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

Edge Relaxation

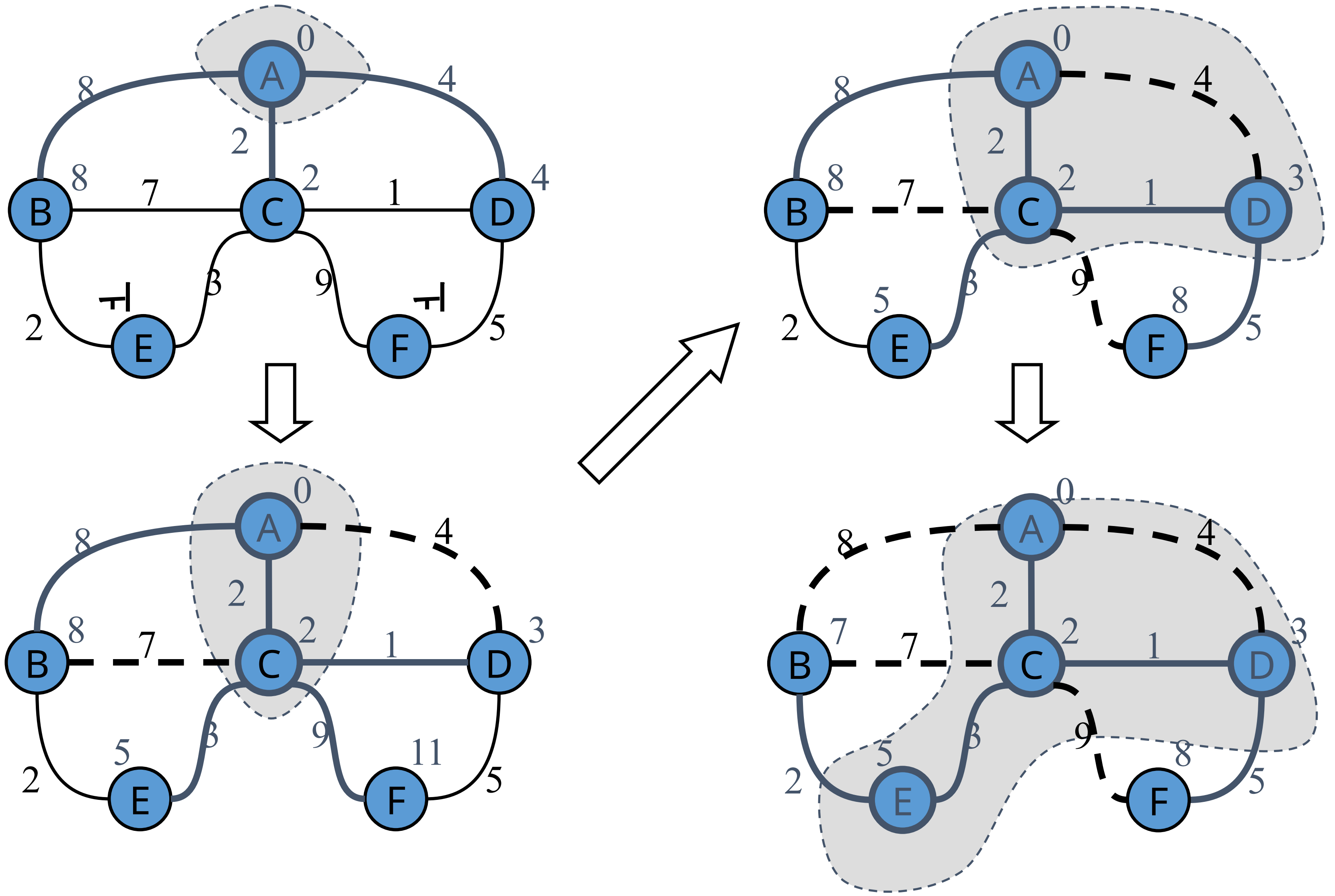
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud

- The relaxation of edge e updates distance $d(z)$ as follows:

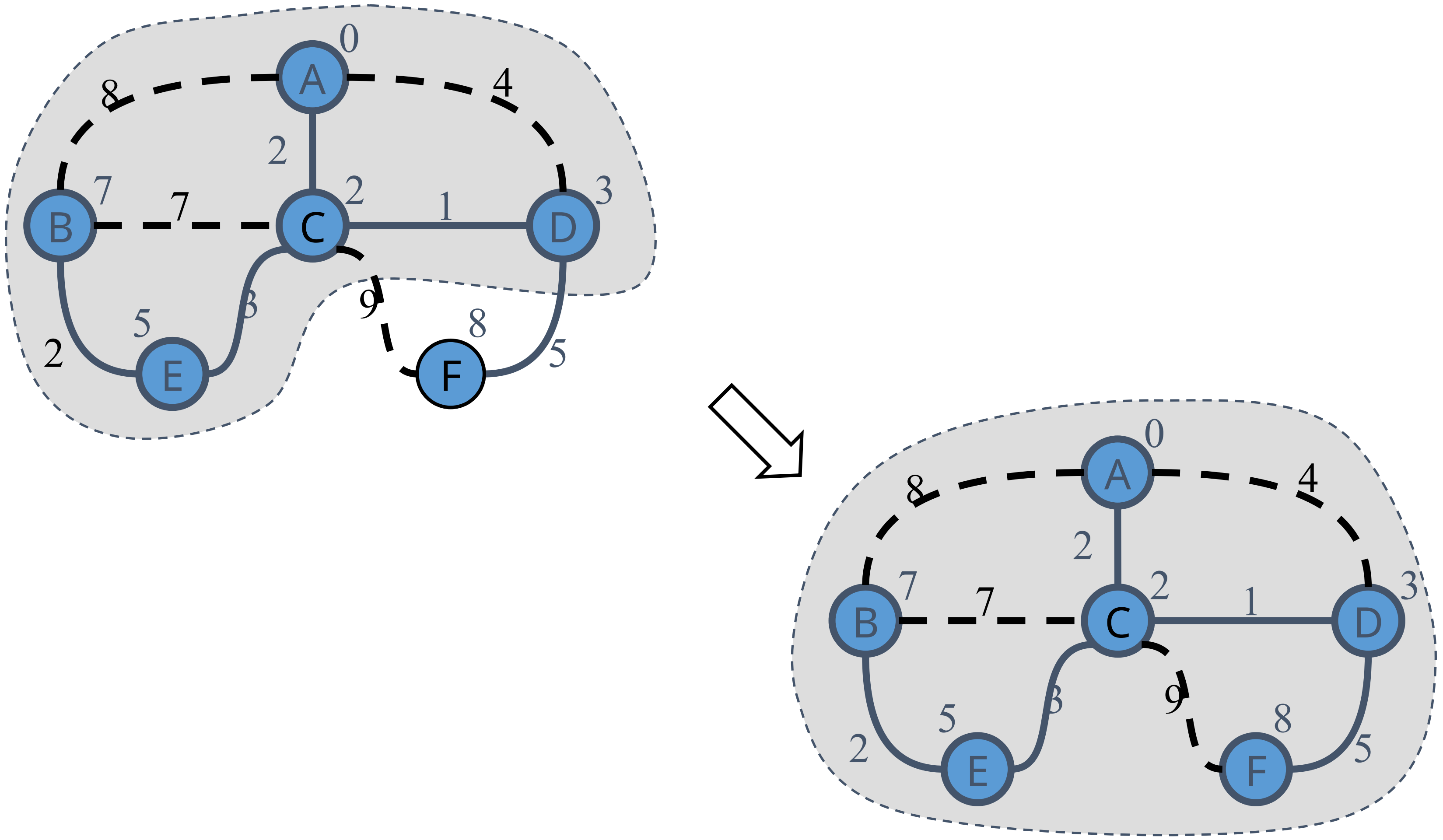
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Example



Example (cont.)



Dijkstra’s Algorithm

```
Algorithm ShortestPath($G, s$):
    Input: A weighted graph $G$ with nonnegative edge weights, and a distinguished
        vertex $s$ of $G$.
    Output: The length of a shortest path from $s$ to $v$ for each vertex $v$ of $G$.
    Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
    Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
    while $Q$ is not empty do
        {pull a new vertex $u$ into the cloud}
        $u =$ value returned by $Q$.remove_min()
        for each vertex $v$ adjacent to $u$ such that $v$ is in $Q$ do
            {perform the relaxation procedure on edge $(u,v)$}
            if $D[u] + w(u,v) < D[v]$ then
                $D[v] = D[u] + w(u,v)$
                Change to $D[v]$ the key of vertex $v$ in $Q$.
    return the label $D[v]$ of each vertex $v$
```

Analysis of Dijkstra's Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- 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 vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ 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/map 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

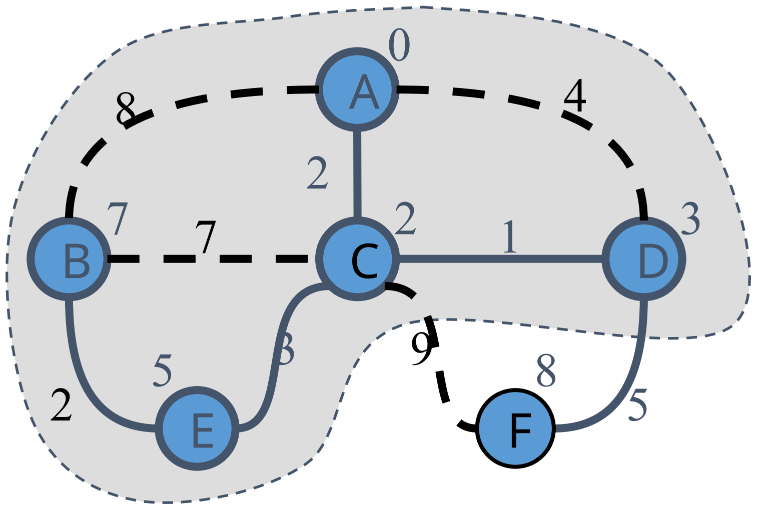
Python Implementation

```
def shortest_path_lengths(g, src):
  """Compute shortest-path distances from src to reachable vertices of g.

  Graph g can be undirected or directed, but must be weighted such that
  e.element() returns a numeric weight for each edge e.

  Return dictionary mapping each reachable vertex to its distance from src.
  """
  d = { }                                  # d[v] is upper bound from s to v
  cloud = { }                              # map reachable v to its d[v] value
  pq = AdaptableHeapPriorityQueue( )       # vertex v will have key d[v]
  pqlocator = { }                          # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if v is src:
      d[v] = 0
    else:
      d[v] = float('inf')                  # syntax for positive infinity
    pqlocator[v] = pq.add(d[v], v)         # save locator for future updates

  while not pq.is_empty( ):
    key, u = pq.remove_min( )
    cloud[u] = key                         # its correct d[u] value
    del pqlocator[u]                       # u is no longer in pq
    for e in g.incident_edges(u):          # outgoing edges (u,v)
      v = e.opposite(u)
      if v not in cloud:
        # perform relaxation step on edge (u,v)
        wgt = e.element( )
        if d[u] + wgt < d[v]:                      # better path to v?
          d[v] = d[u] + wgt                        # update the distance
          pq.update(pqlocator[v], d[v], v)         # update the pq entry

  return cloud                             # only includes reachable vertices
```

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)$\geq$d(D), F's distance cannot be wrong. That is, there is no wrong vertex



0
A
8
4
2
7
7
2
1
3
B
C
D
9
3
8
5
5
2
E
F