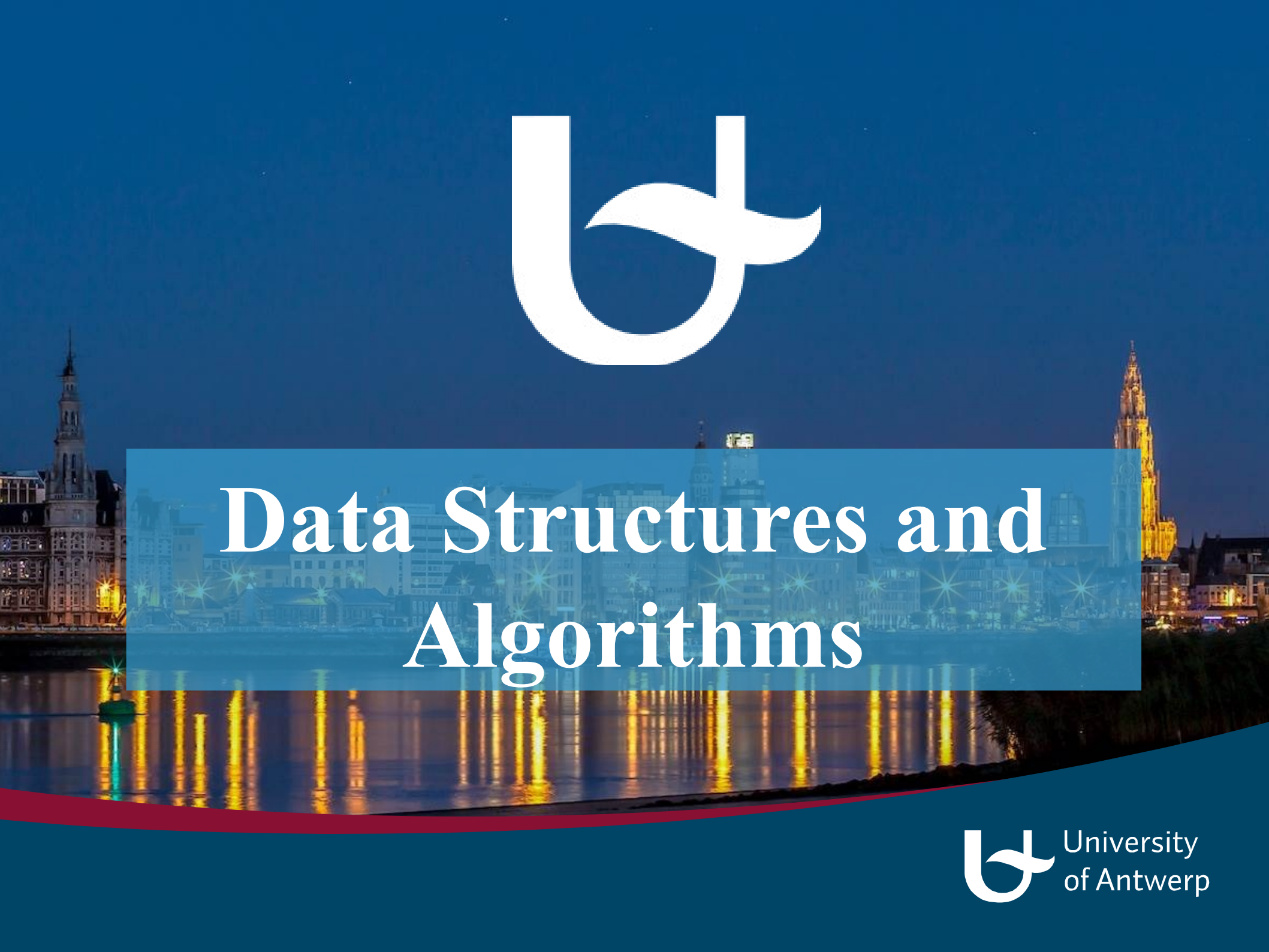


Data Structures and Algorithms

University of Antwerp

Graphs

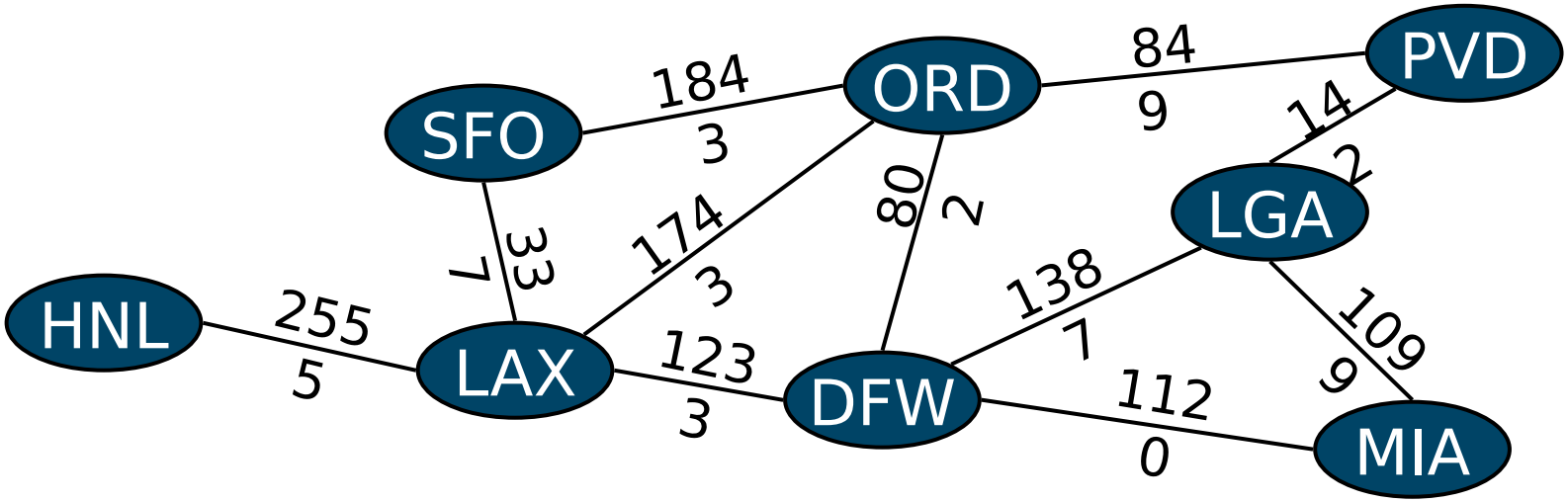
Lecturer: Dr. Ali Anwar

Objectives

- Introduction of
  - Graphs
    - Applications, Terminology, Properties, Methods
  - Subgraphs
  - DFS (Depth-First Search), BFS (Breadth-First Search)
  - Directed Graphs
    - Applications, Properties
  - Shortest Paths
    - Properties, Algorithms, Shortest Path Trees

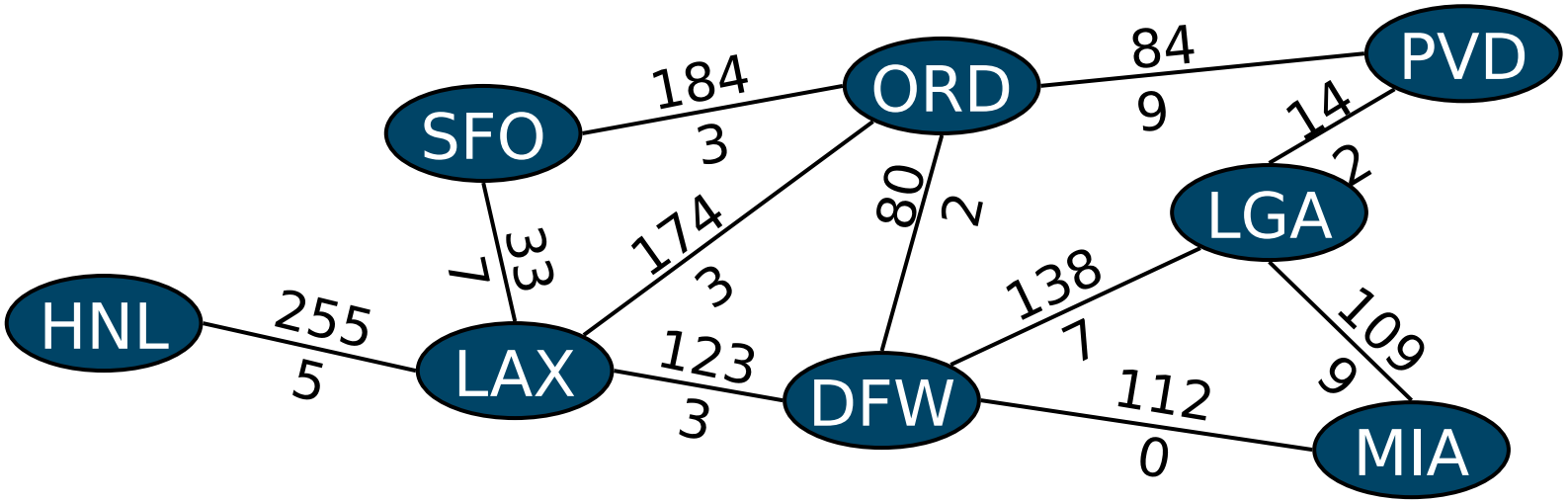
Part 1

Graphs



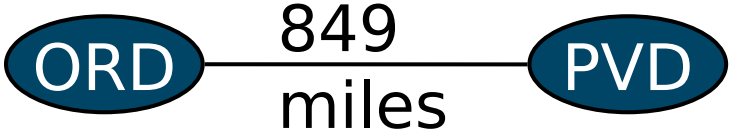
Graphs

- 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



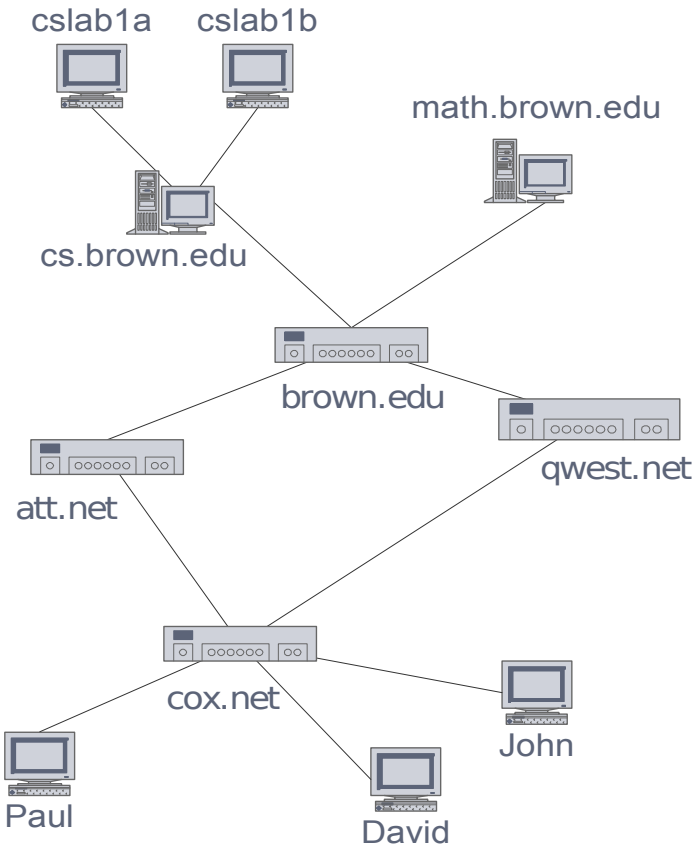
Edge Type

- 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



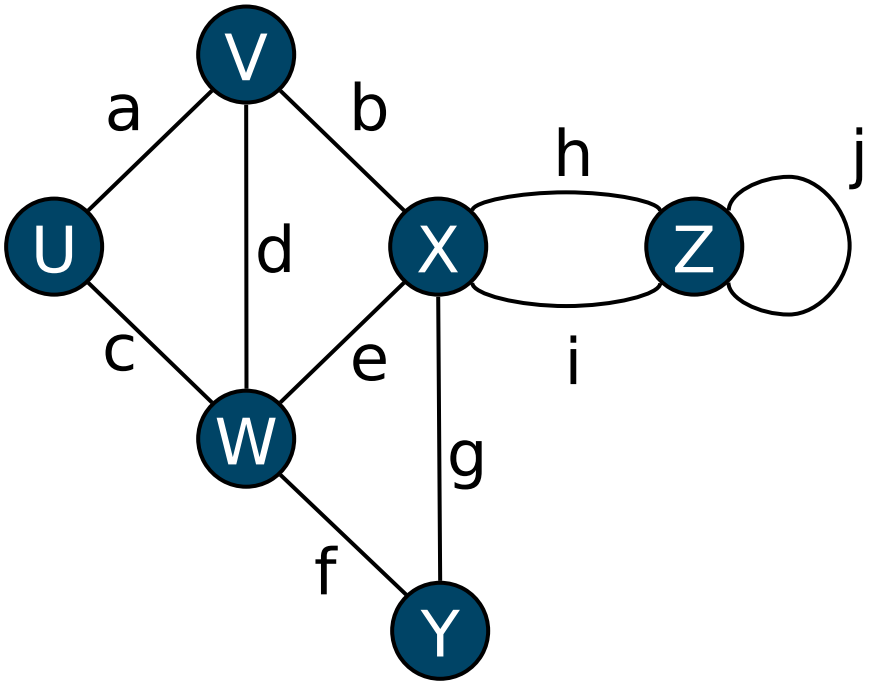
Applications

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



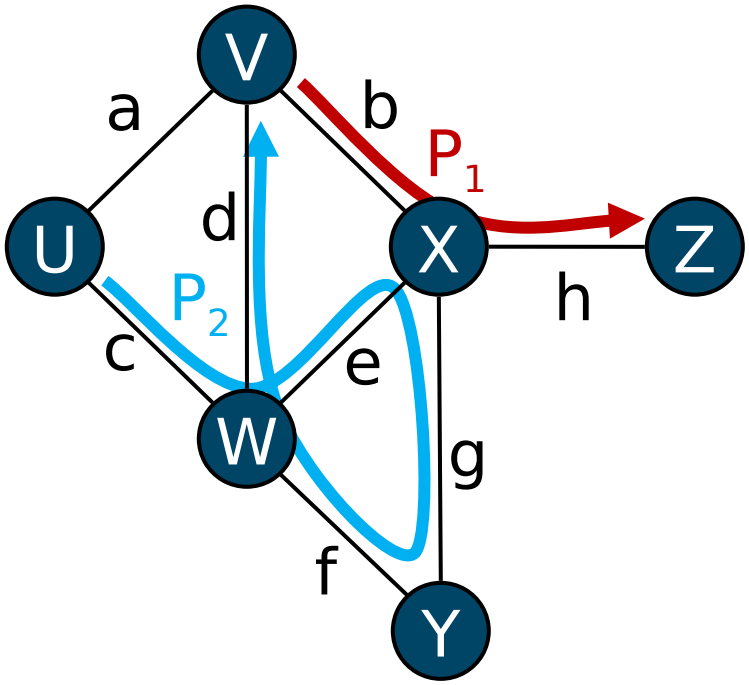
Terminology

- 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 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



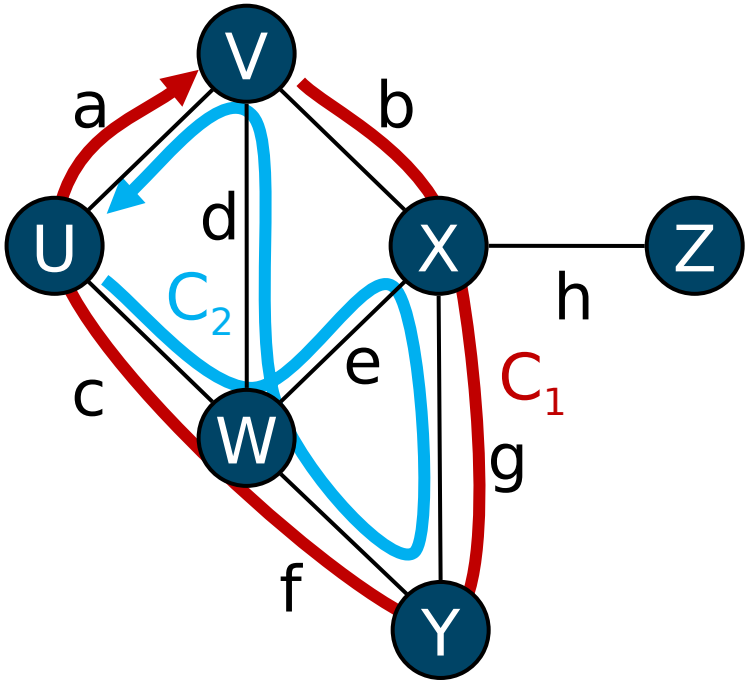
Terminology (cont.)

- 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 (cont.)

- 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,↩) is a simple cycle
  - C_2 =(U,c,W,e,X,g,Y,f,W,d,V,a,↩) 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

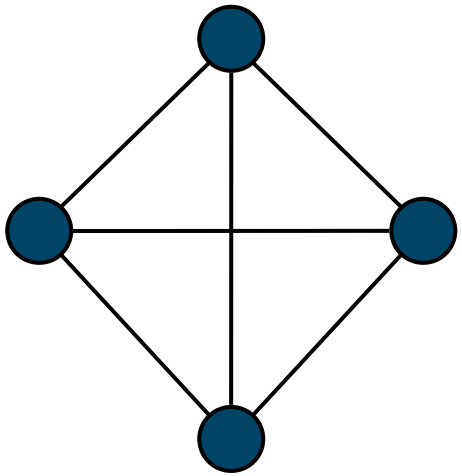
$$m \leq n\,(n - 1)/2$$

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

What is the bound for a directed graph?

Notation

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



Example

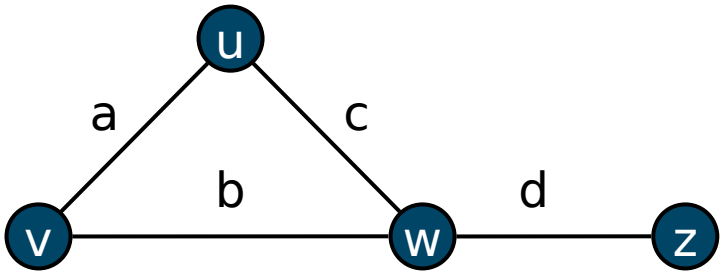
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

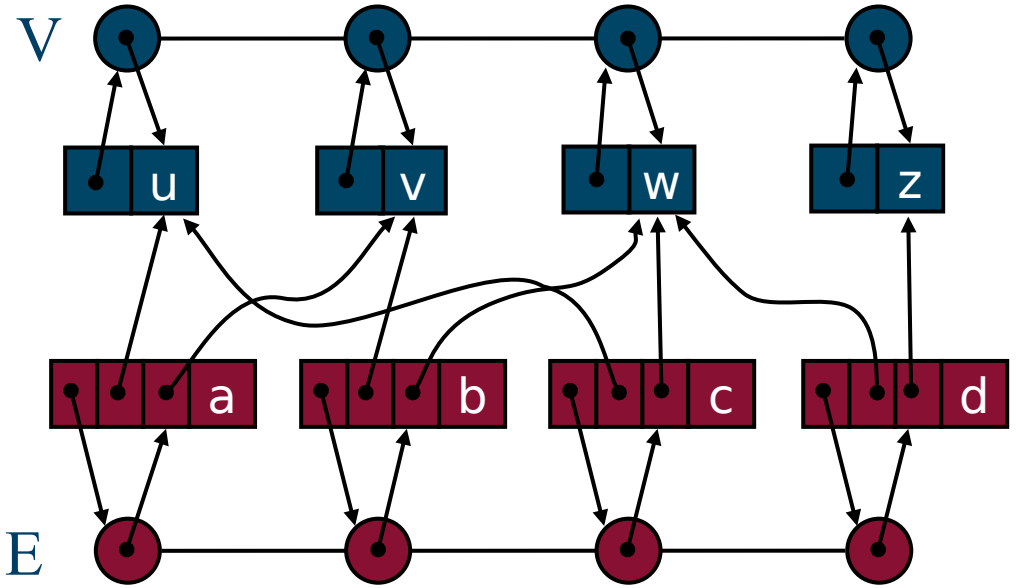
Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - *e.endVertices*(): a list of the two end vertices of *e*
  - *e.opposite*(*v*): the vertex opposite of v on e
  - *u.isAdjacentTo*(*v*): true if *u* and *v* are adjacent
  - **v*: reference to element associated with vertex *v*
  - **e*: reference to element associated with edge *e*
- Update methods
  - *insertVertex*(*o*): insert a vertex storing element *o*
  - *insertEdge*(*v*, *w*, *o*): insert an edge (*v*,*w*) storing element *o*
  - *eraseVertex*(*v*): remove vertex *v* (and its incident edges)
  - *eraseEdge*(*e*): remove edge *e*
- Iterable collection methods
  - *incidentEdges*(*v*): list of edges incident to *v*
  - *vertices*(): list of all vertices in the graph
  - *edges*(): list of all edges in the graph

Edge List Structure

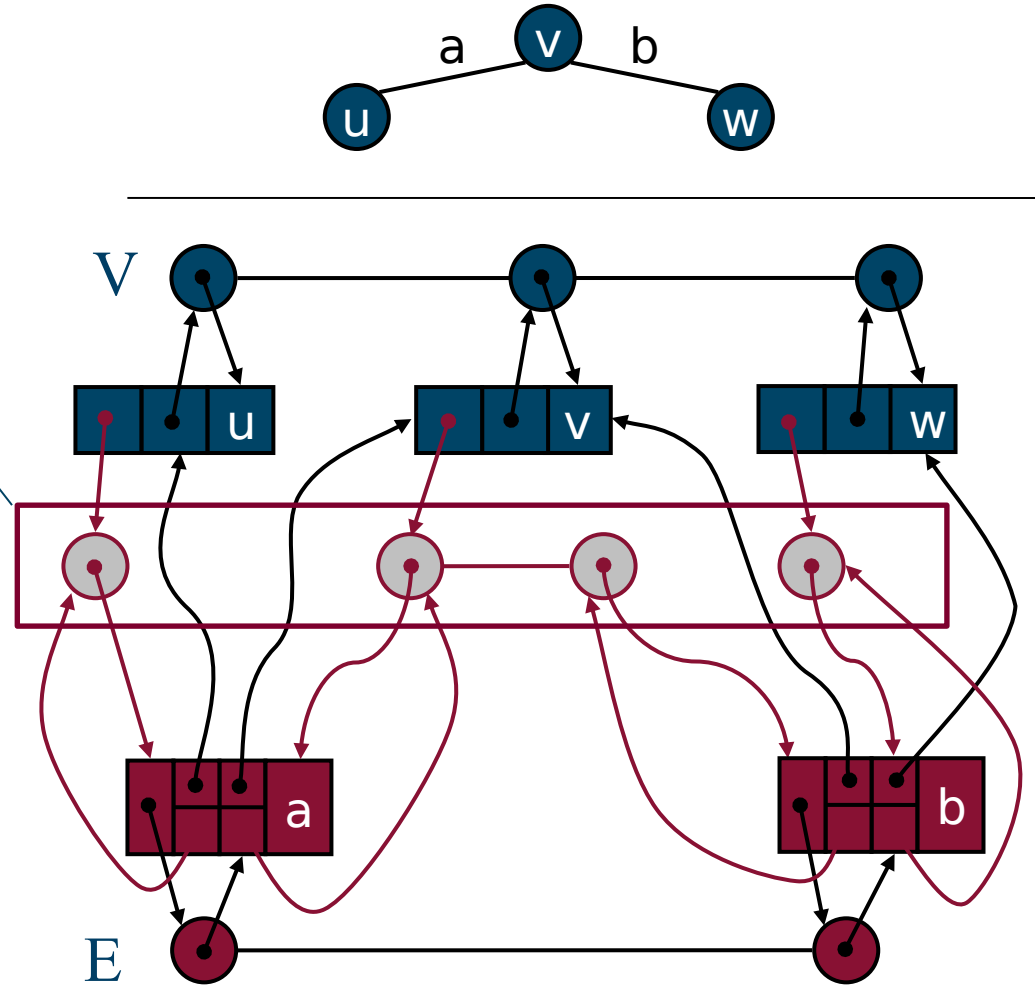
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects





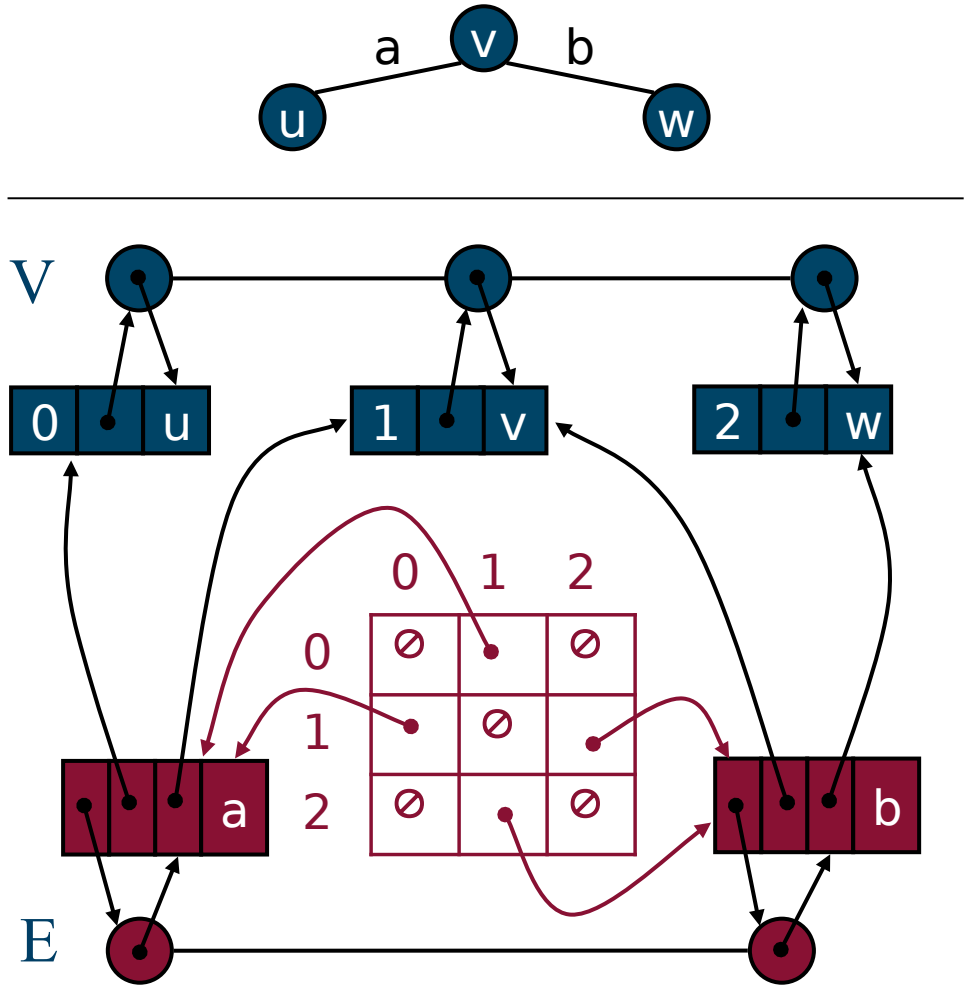
Adjacency List Structure

- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



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 – A[i,j] holds reference to (v,w)
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



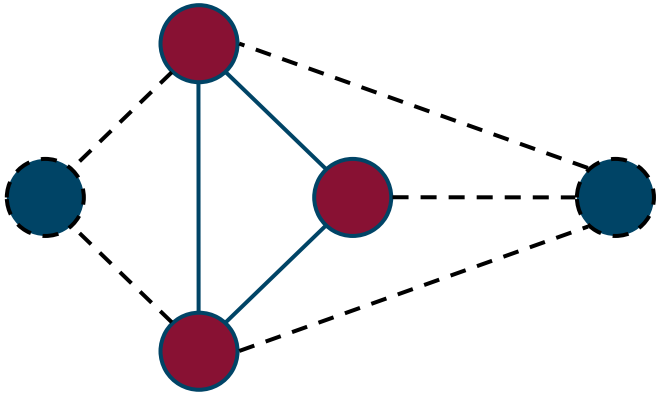
Performance

• n vertices, m edges • no parallel edges • no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	n + m	n + m	n^2
v.incidentEdges()	m	deg(v)	n
u.isAdjacentTo(v)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
eraseVertex(v)	m	deg(v)	n^2
eraseEdge(e)	1	1	1

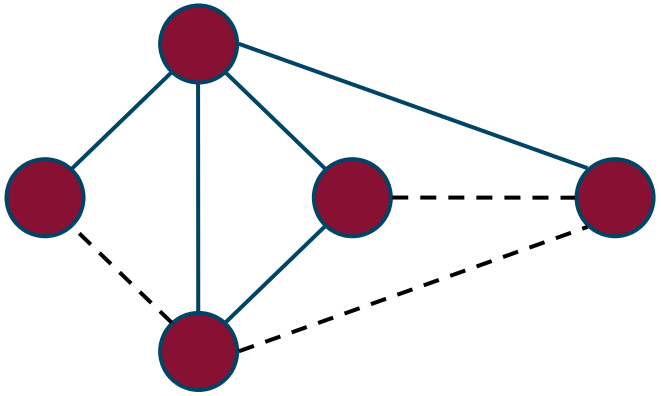
HW: Read Section 13.2 in Goodrich

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*



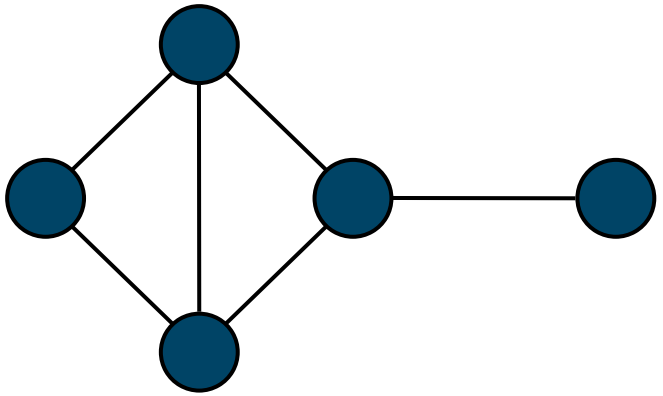
Subgraph



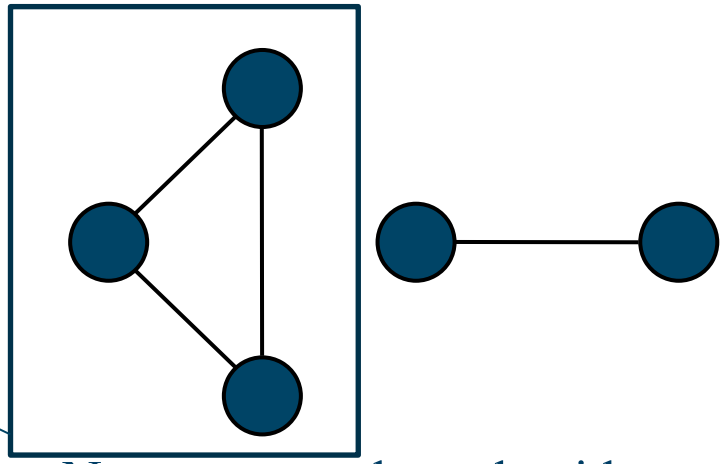
Spanning subgraph

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



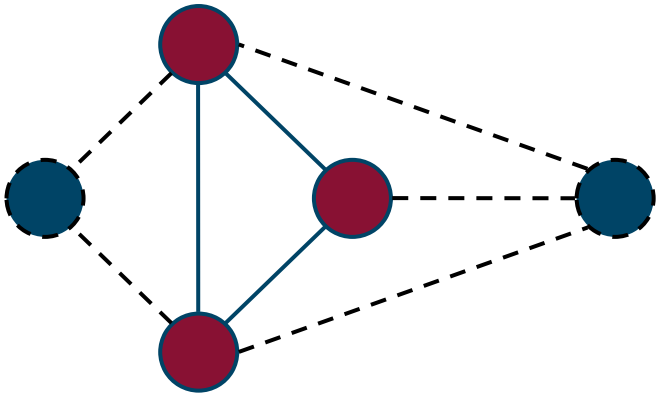
Connected graph



Non connected graph with two connected components

Part 2

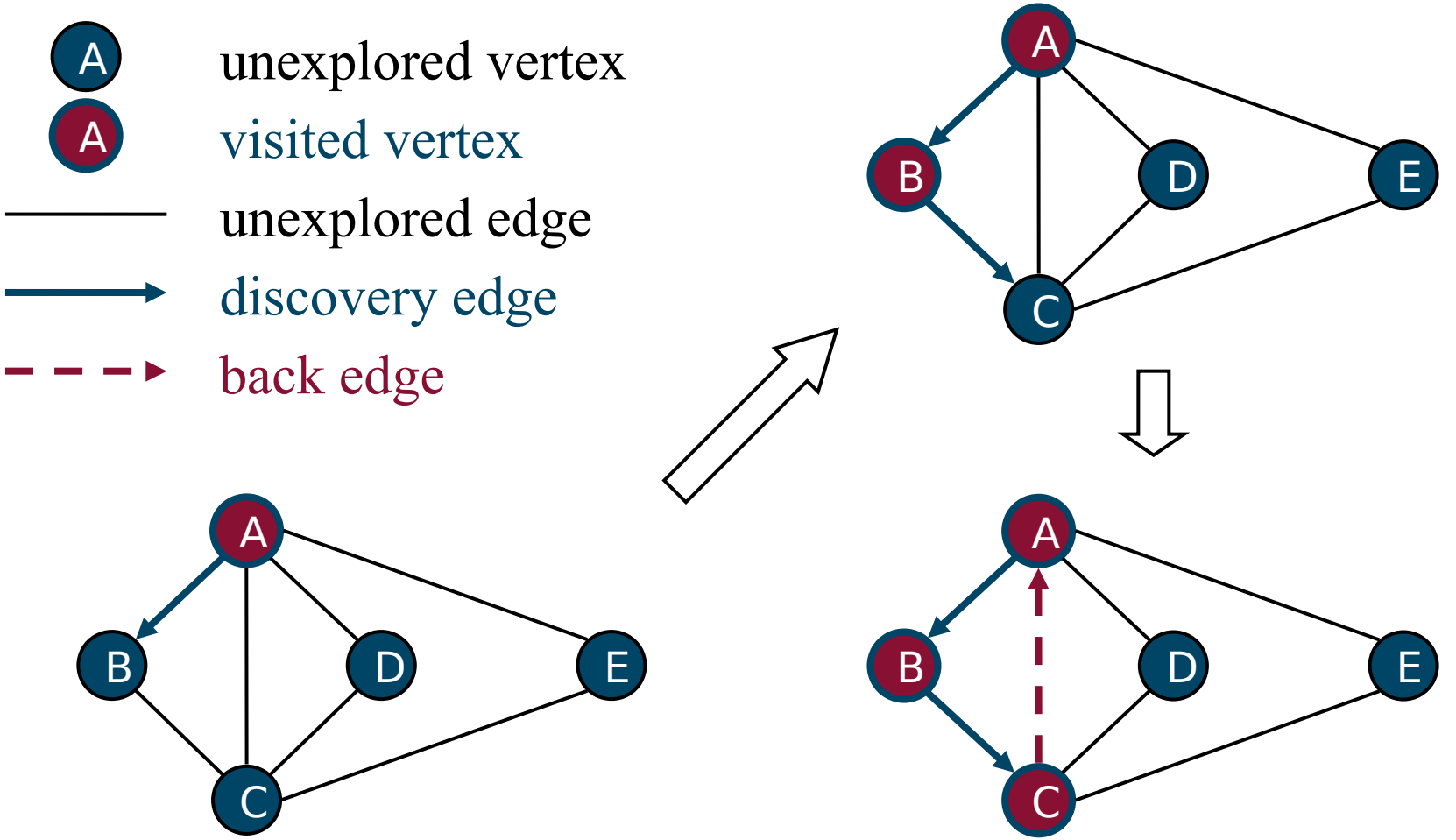
DFS, BFS



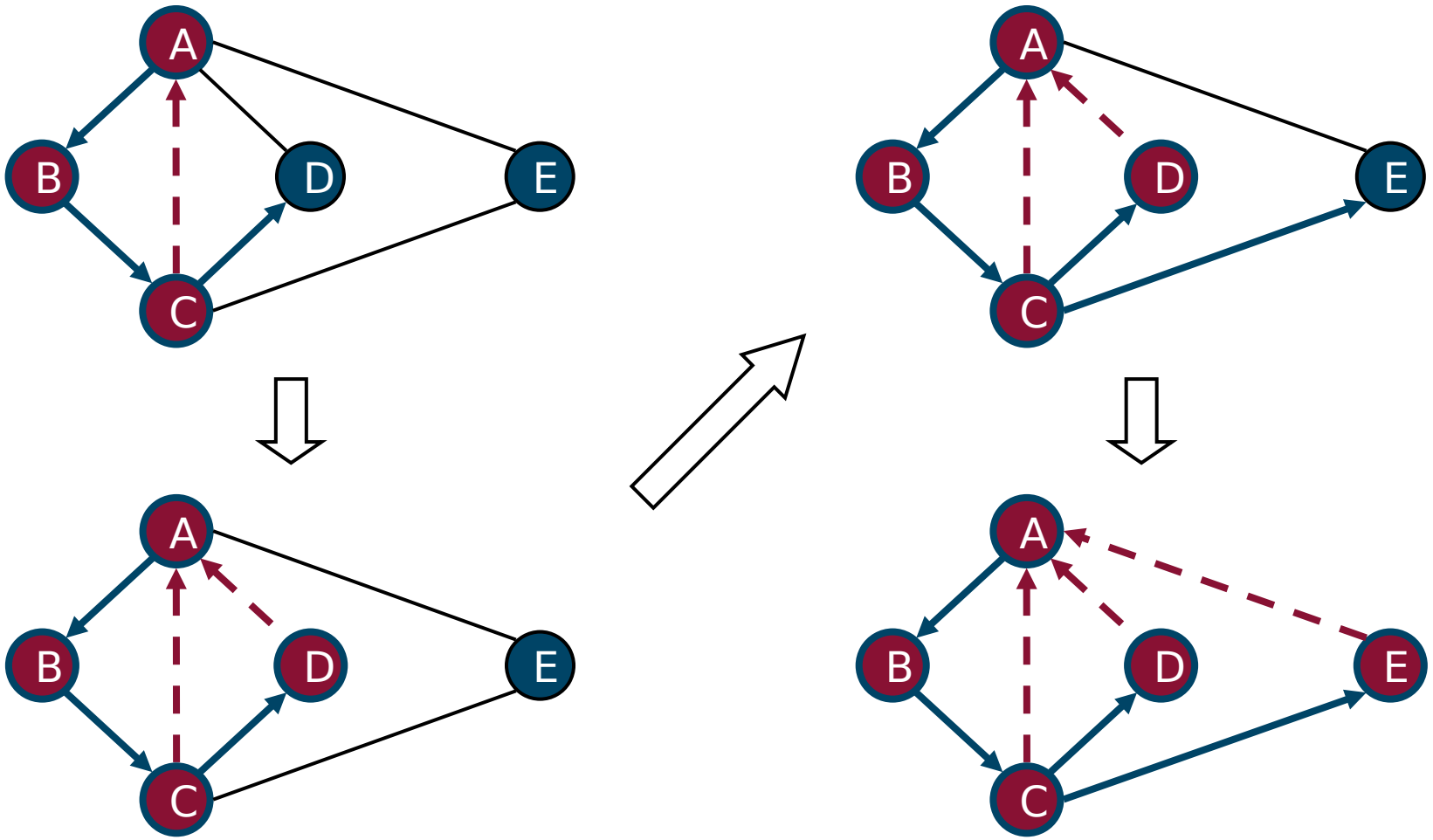
Depth-First Search

- 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

Example


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

Example (cont.)



DFS Algorithm

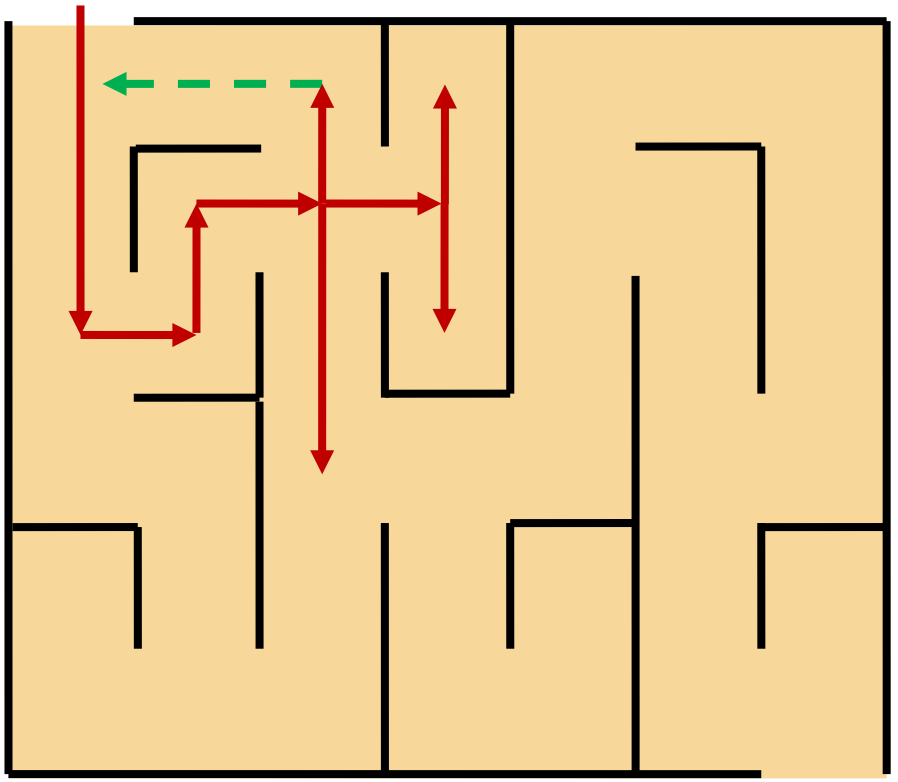
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm DFS(G)
        Input graph G
        Output labeling of the edges
of G
                as discovery edges
and
                back edges
  for all u ∈ G.vertices()

  u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()

  e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
        if v.getLabel() =
  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
        v.setLabel(VISITED)
  for all e ∈ G.incidentEdges(v)
        if e.getLabel() =
  UNEXPLORED
                w ← e.opposite(v)
                if w.getLabel() =
  UNEXPLORED

  e.setLabel(DISCOVERY)
                DFS(G, w)
        else
```

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)



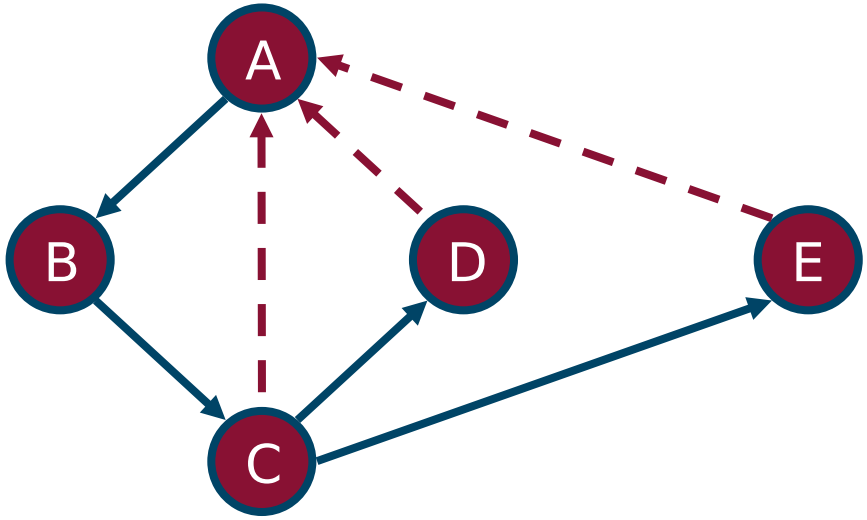
Properties of DFS

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***



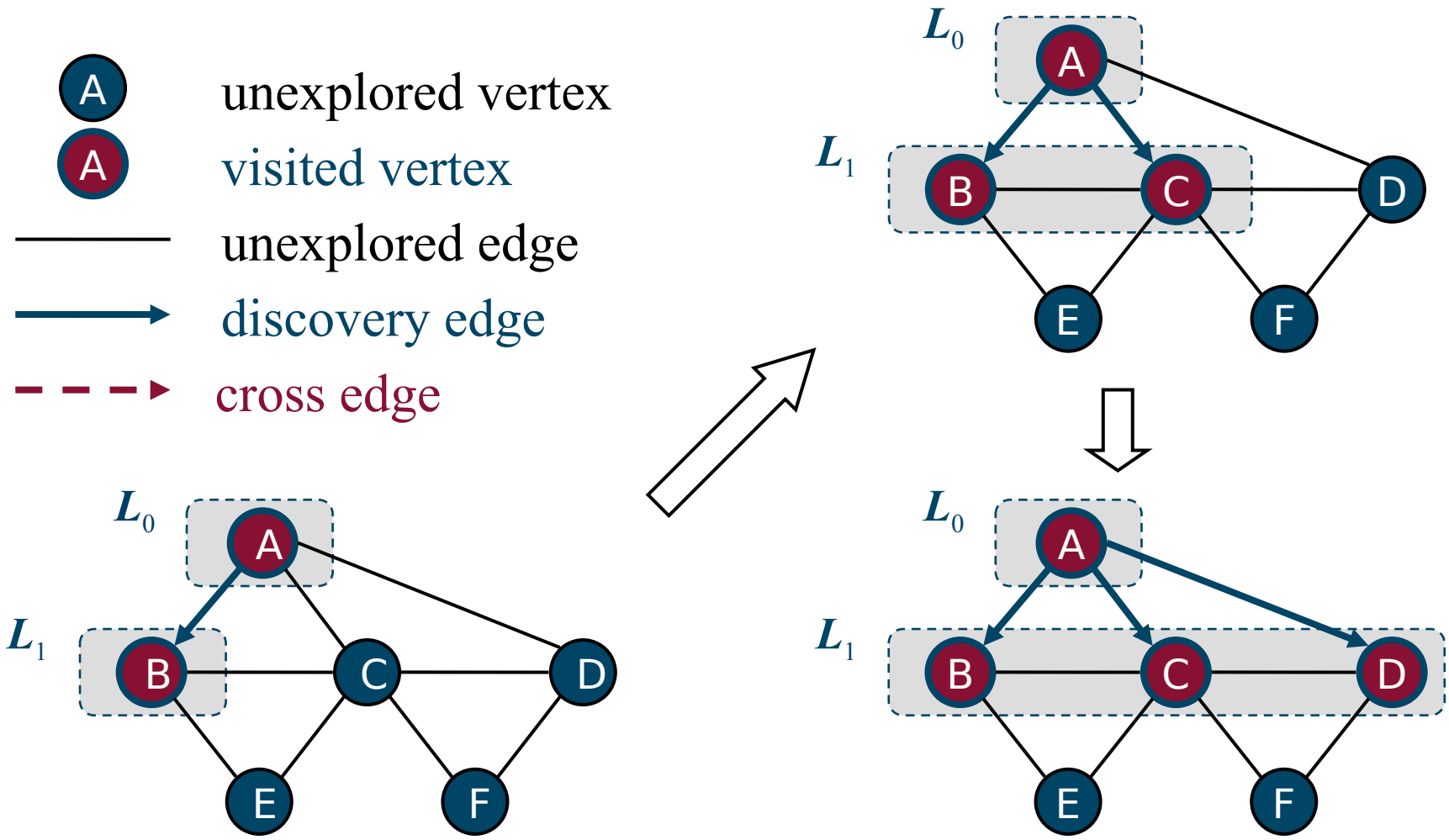
Analysis of DFS

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

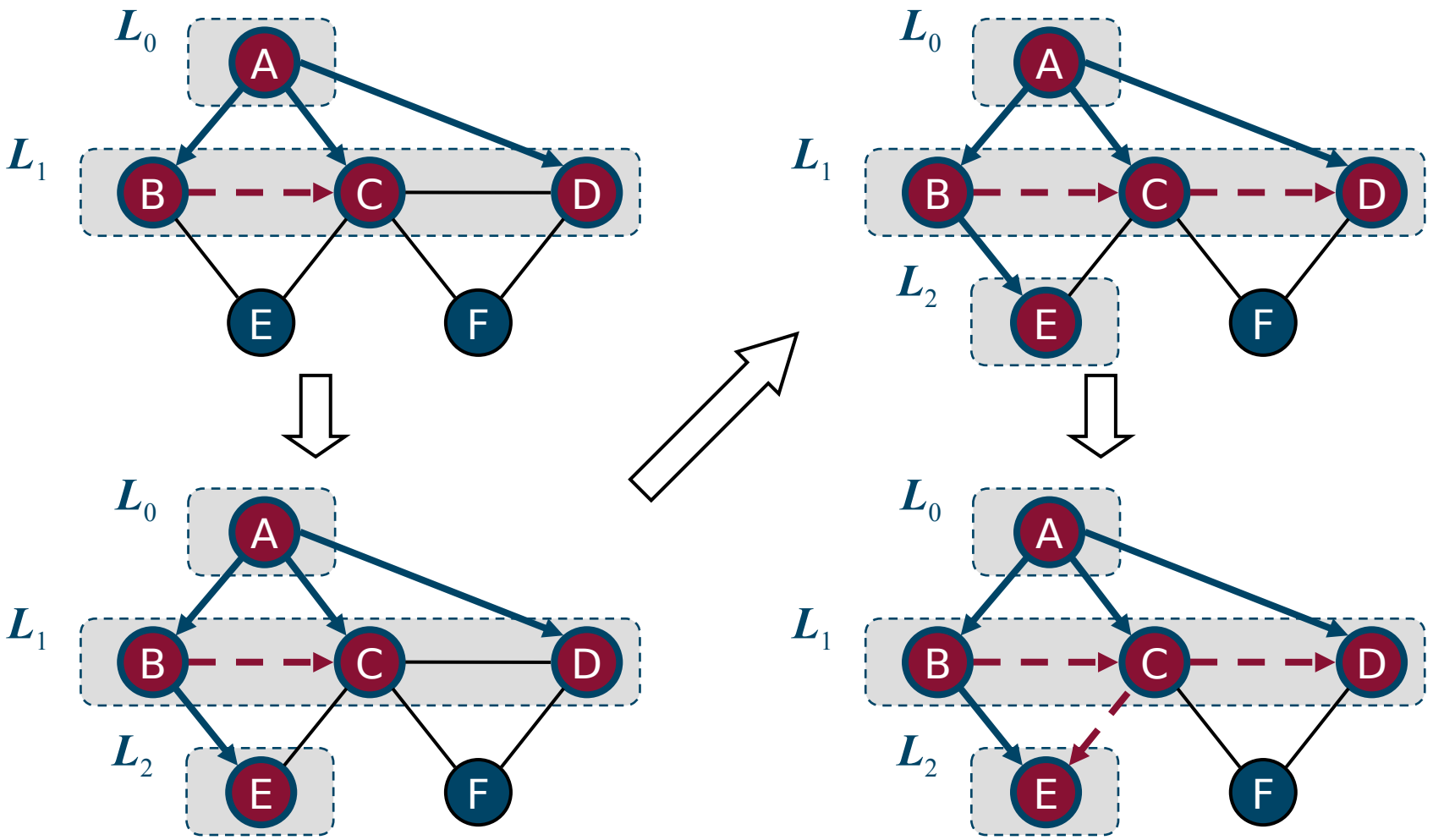
Breadth-First Search

- 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 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 with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

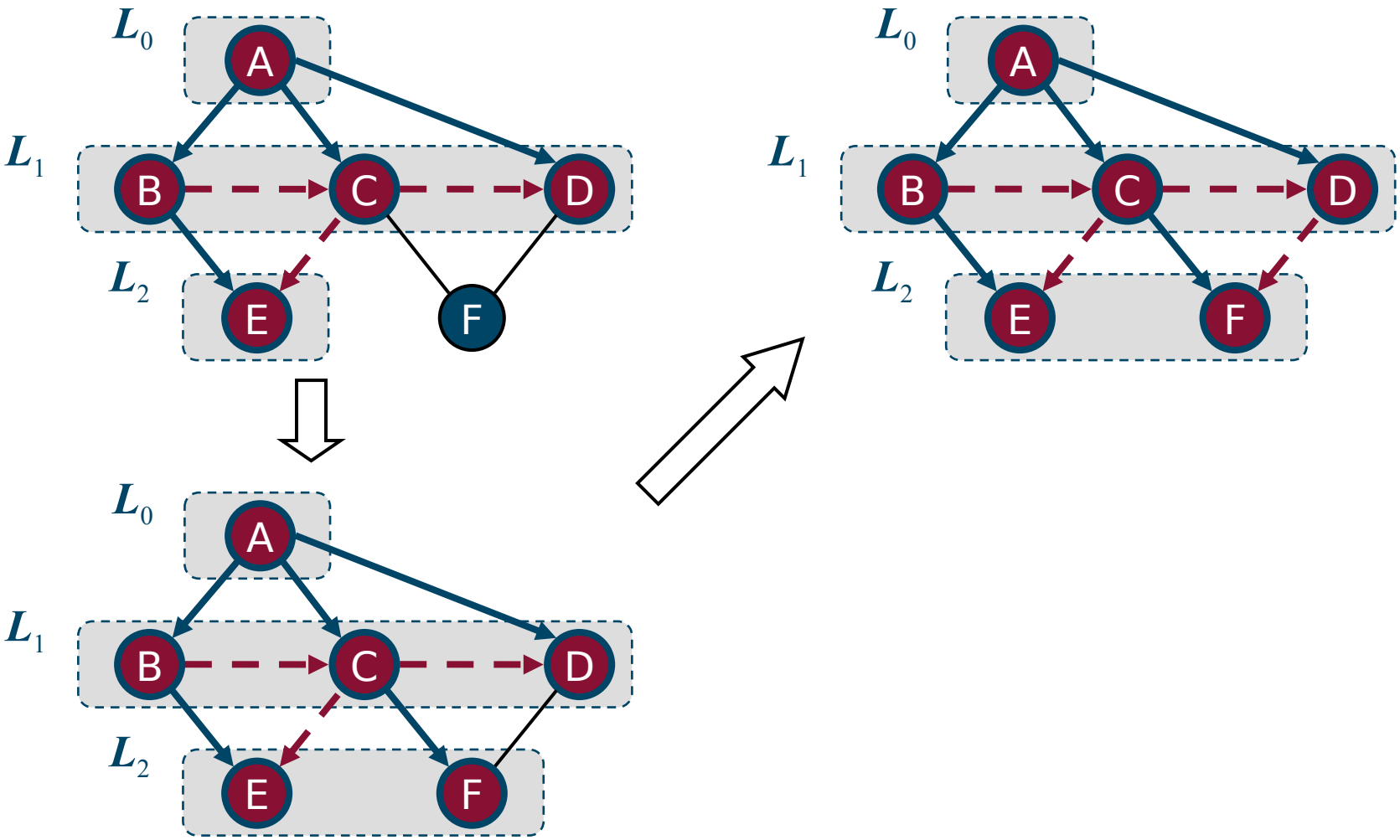
Example


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.)



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)
```

Properties

Notation

G_s: connected component of s

Property 1

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

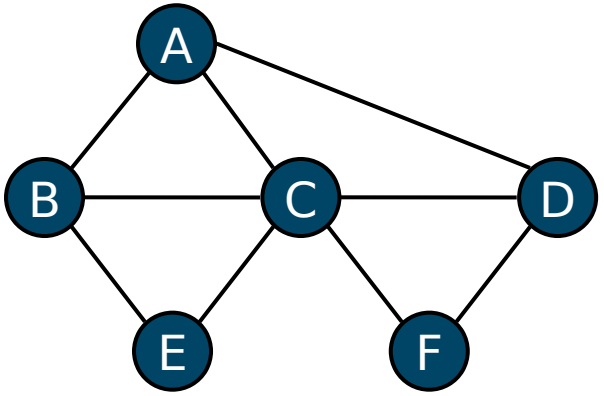
Property 2

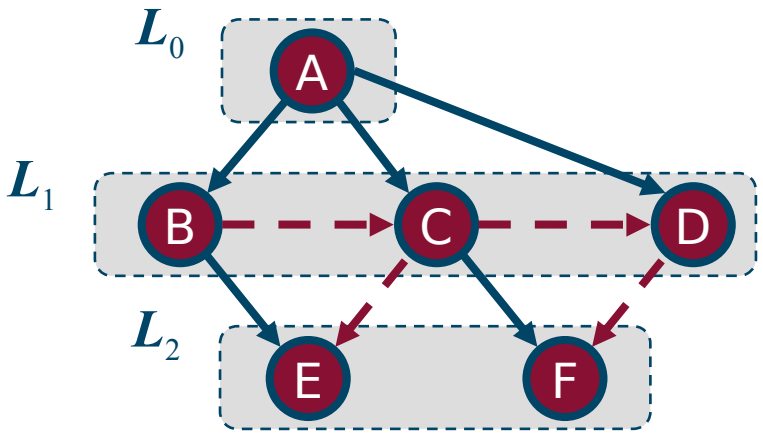
The discovery edges labeled by ***BFS*(*G*, *s*)** form a spanning tree T_s of G_s

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



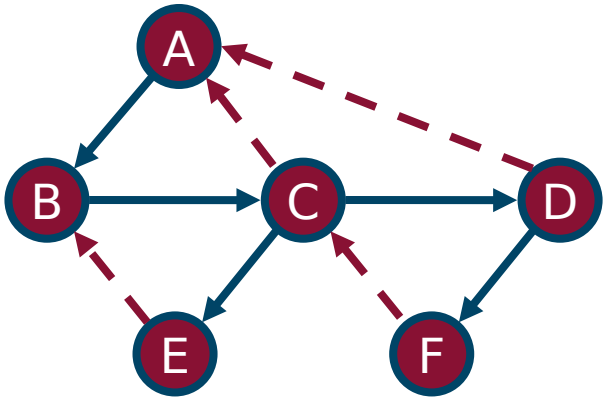


Analysis

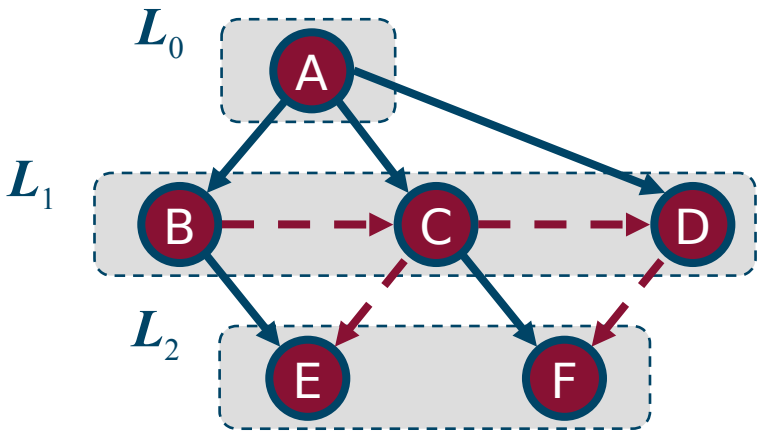
- Setting/getting a vertex/edge label takes O(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method *incidentEdges* is called once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

DFS vs. BFS

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



DFS



BFS

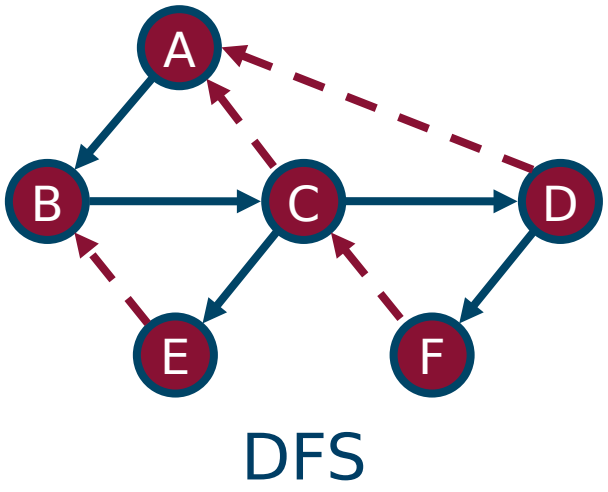
DFS vs. BFS (cont.)

Back edge (*v*,*w*)

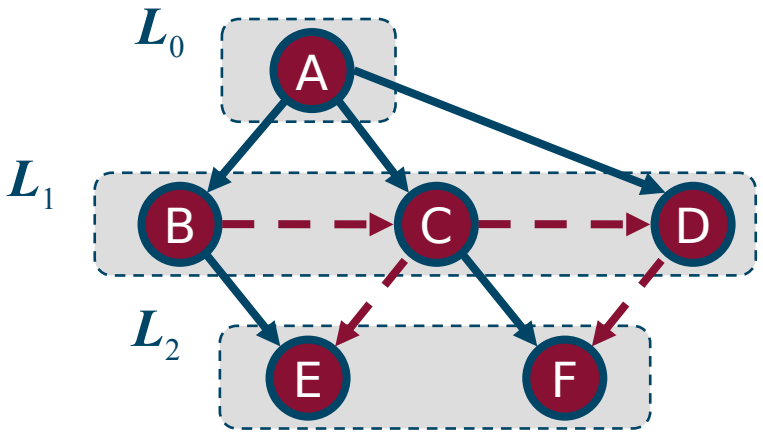
- *w* is an ancestor of *v* in the tree of discovery edges

Cross edge (*v*,*w*)

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



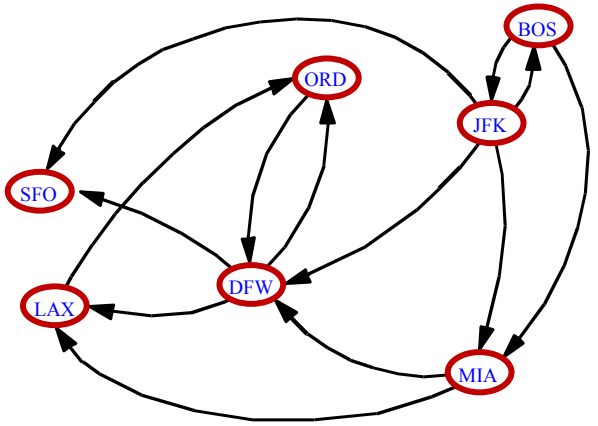
DFS



BFS

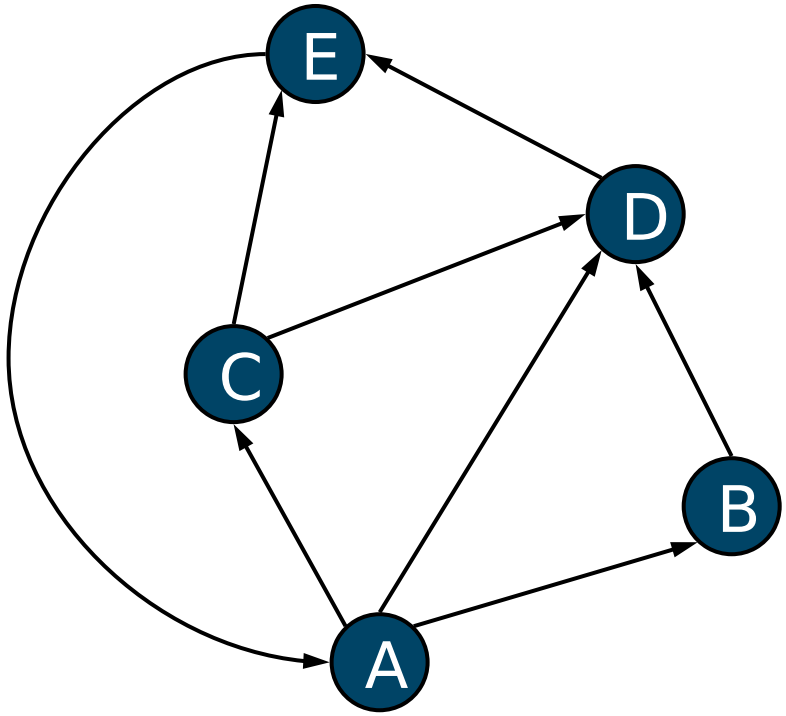
Part 3

Directed Graphs



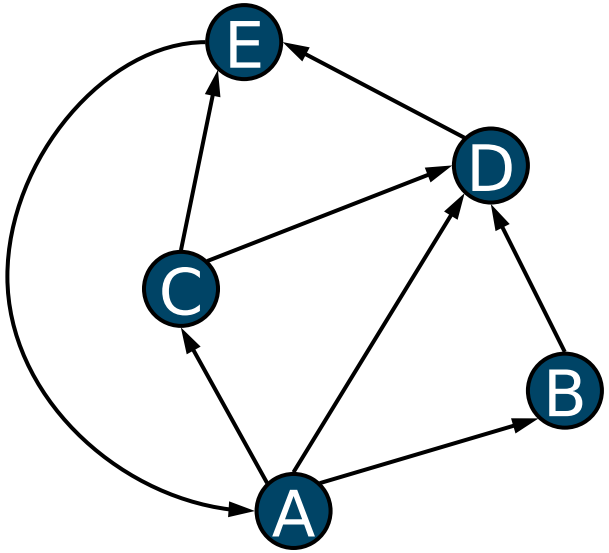
Digraphs

- A digraph is a graph whose edges are all directed
  - Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling



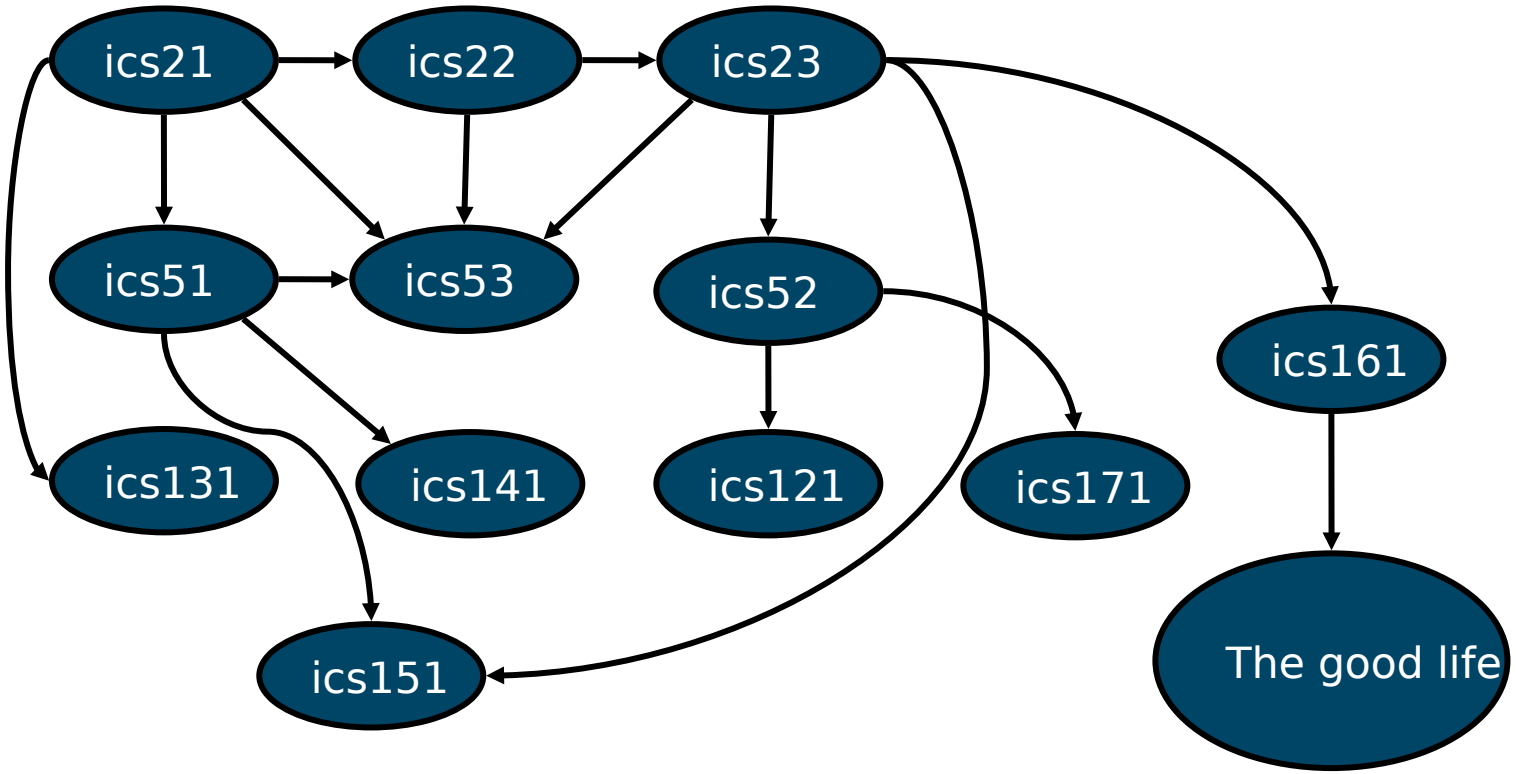
Digraph Properties

- 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, $m < n \cdot (n - 1)$
- 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



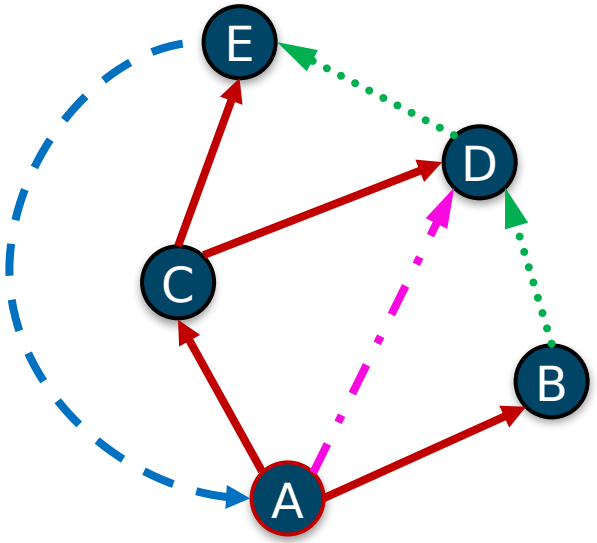
Digraph Application

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



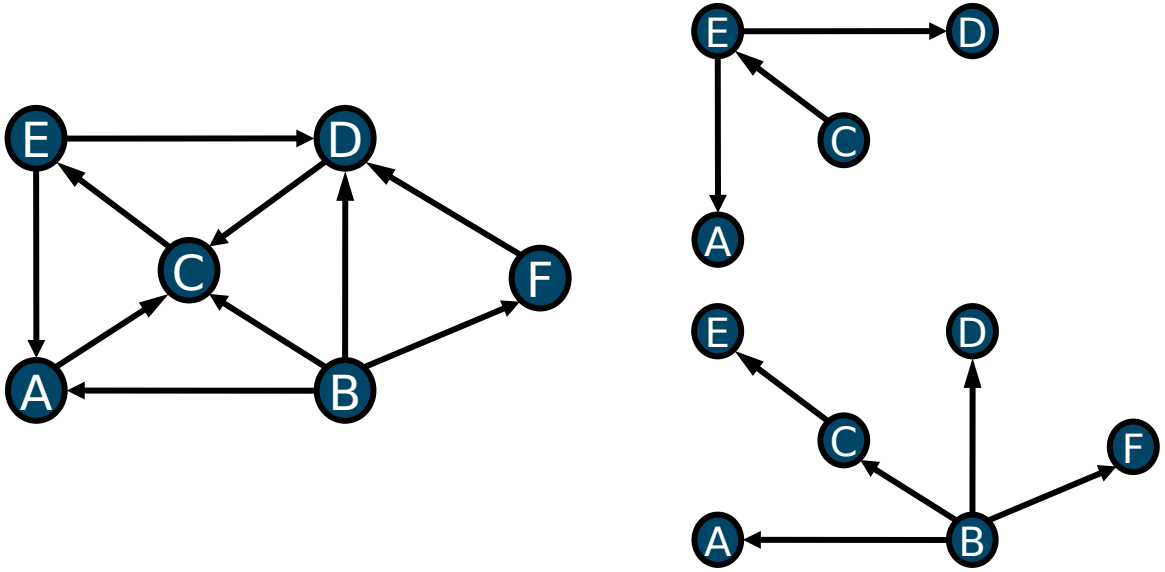
Directed DFS

- 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*



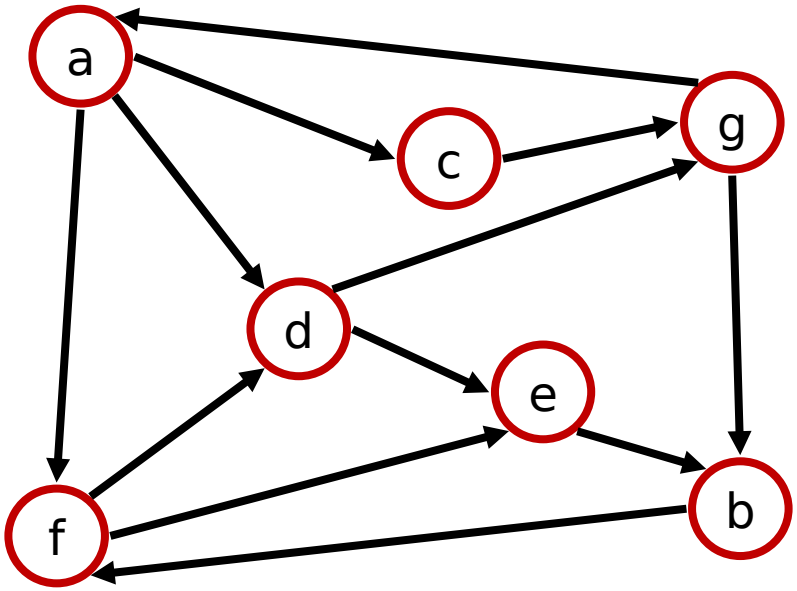
Reachability

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



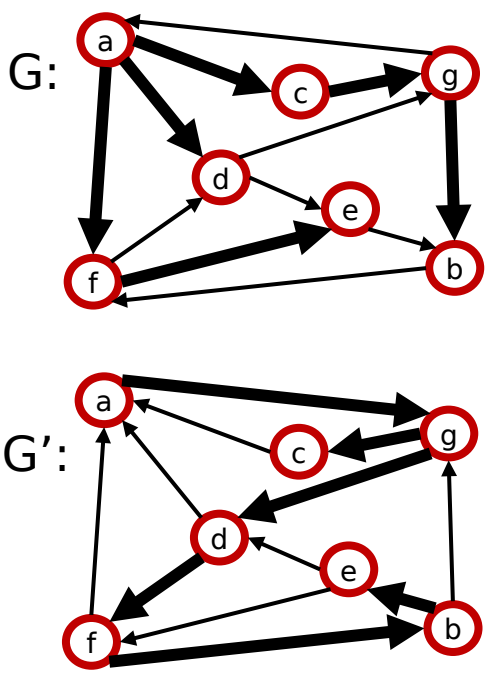
Strong Connectivity

- Each vertex can reach all other vertices



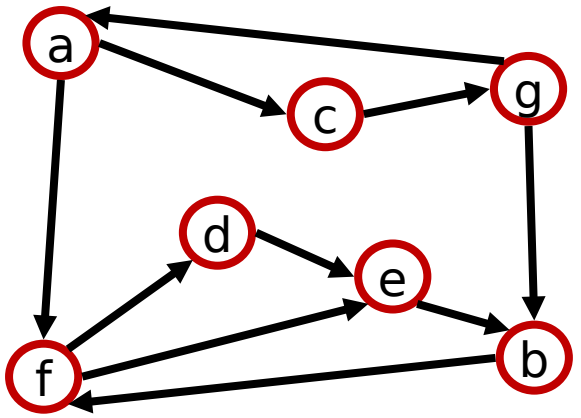
Strong Connectivity Algorithm

- Pick a vertex *v* in *G*
- Perform a DFS from *v* in *G*
  - If there's a *w* not visited, print "no"
- Let *G'* be *G* with edges reversed
- Perform a DFS from *v* in *G'*
  - If there's a *w* not visited, print "no"
  - Else, print "yes"
- Running time: O(*n*+*m*)



Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).

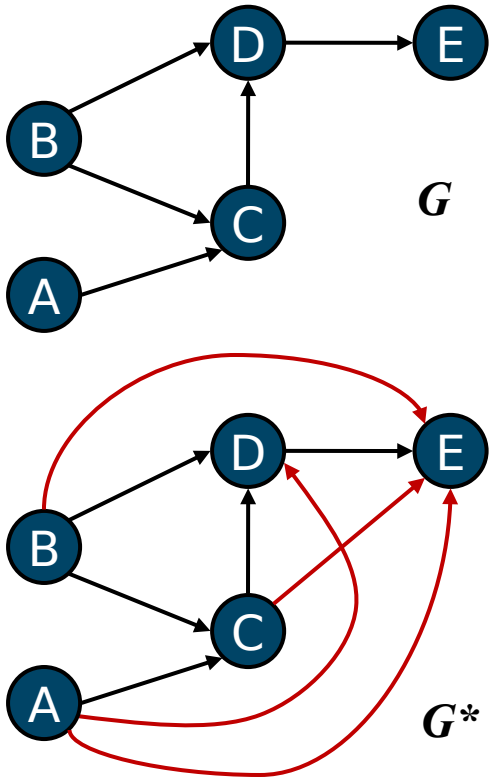


{ a , c , g }

{ f , d , e , b }

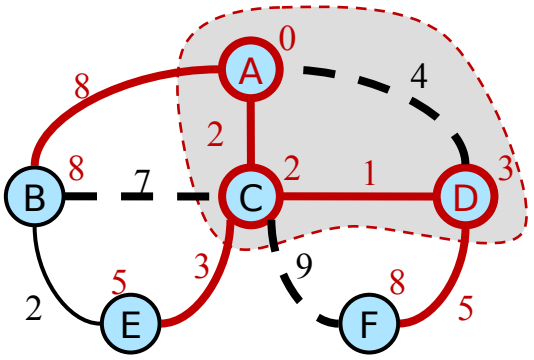
Transitive Closure

- Given a digraph G, the transitive closure of G is the digraph G^* such that
  - G^* has the same vertices as G
  - if G has a directed path from u to v ($u \neq v$), G^* has a directed edge from u to v
- The transitive closure provides reachability information about a digraph



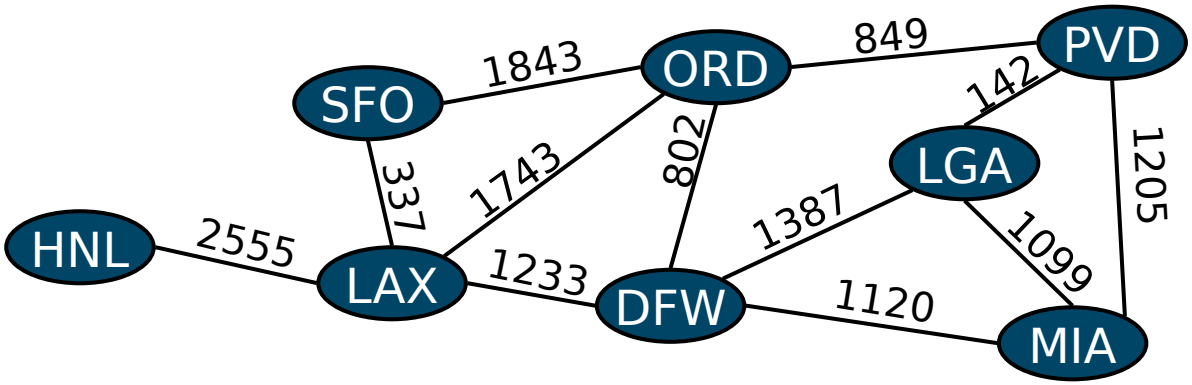
Part 4

Shortest Paths



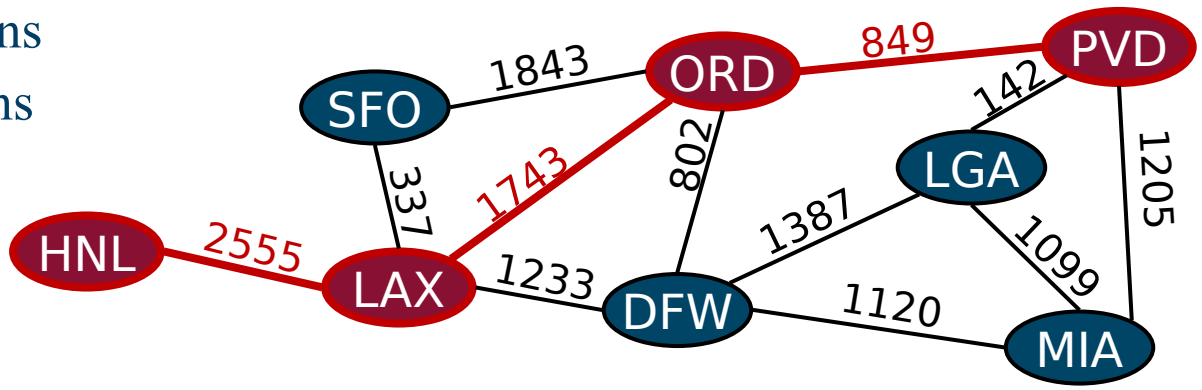
Weighted Graphs

- 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:

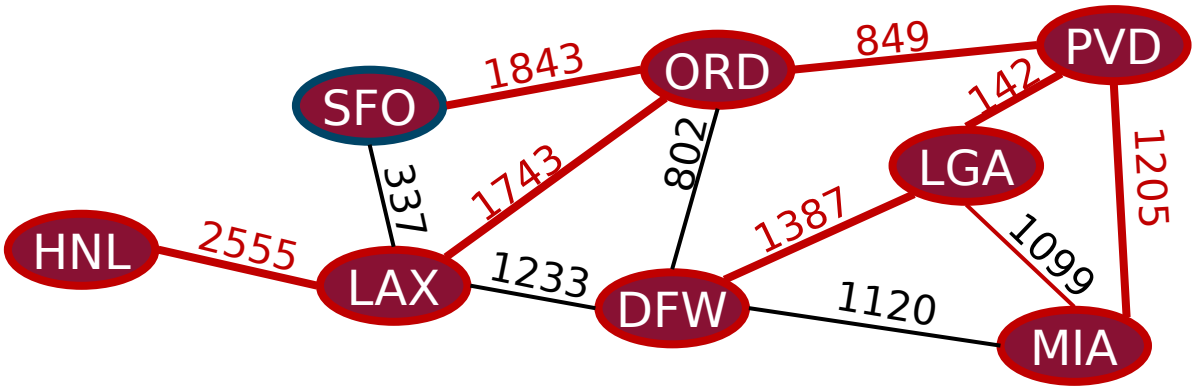
  A subpath of a shortest path is itself a shortest path

- 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

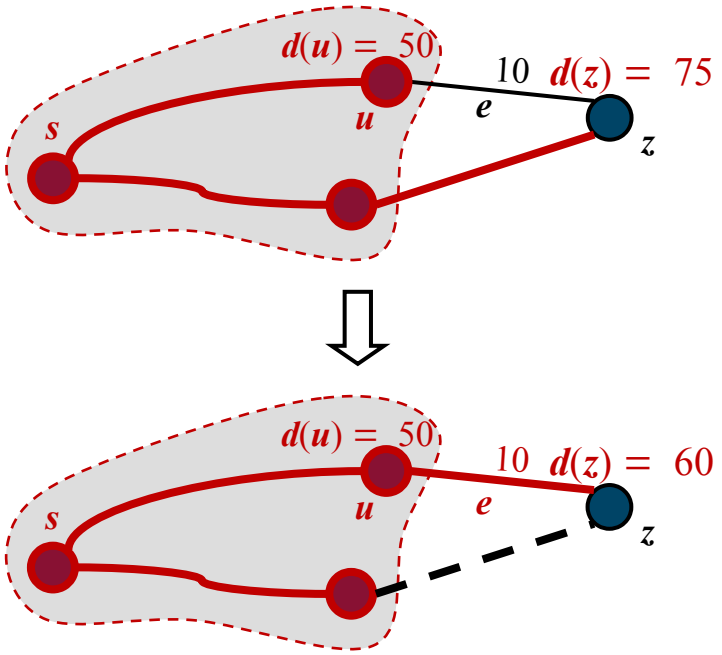


Dijkstra's Algorithm

- 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

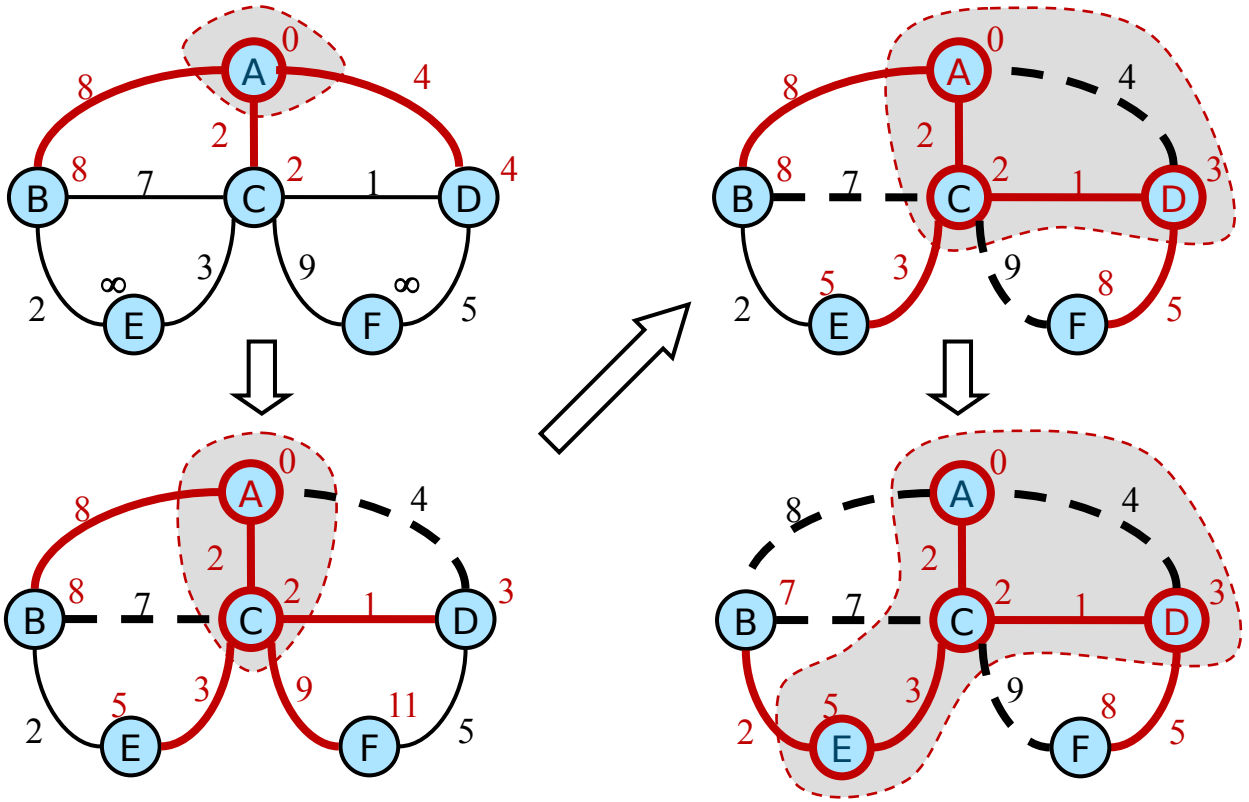
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud



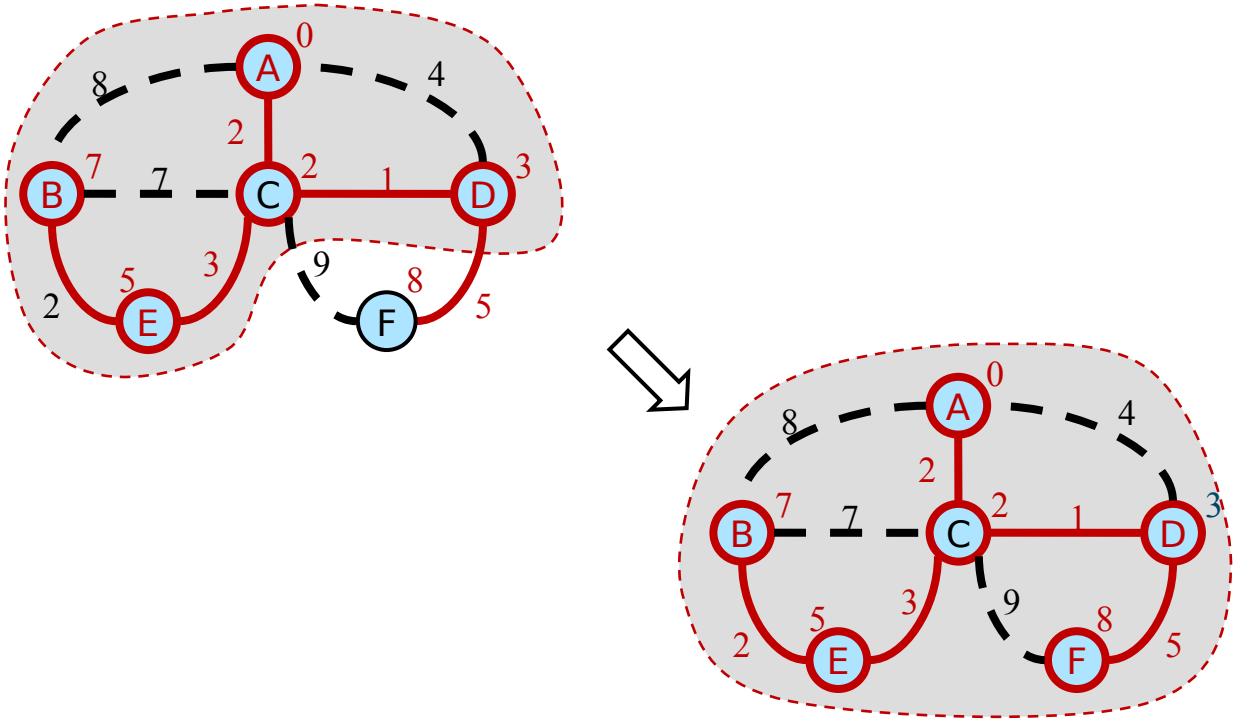
- The relaxation of edge e updates distance $d(z)$ as follows:

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

Example



Example (cont.)



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
- We store two labels with each vertex:
  - Distance
  - Entry in priority queue

```
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()
                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)
```

Analysis of Dijkstra's Algorithm

- Graph operations
  - Method *incidentEdges* is called 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 structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

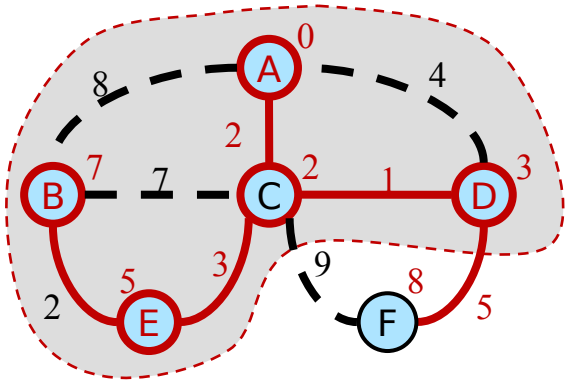
Shortest Paths Tree

- Using the template method pattern, we can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- We store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

```
Algorithm DijkstraShortestPathsTree(G, s)
        …
        for all v ∈ G.vertices()
        …
                v.setParent(⊘)
        …
        for all e ∈ u.incidentEdges()
                { relax edge e }
                z ← e.opposite(u)
                r ← u.getDistance() +
e.weight()
                if r < z.getDistance()
z.setDistance(r)
z.setParent(e)
Q.replaceKey(z.getEntry(),r)
```

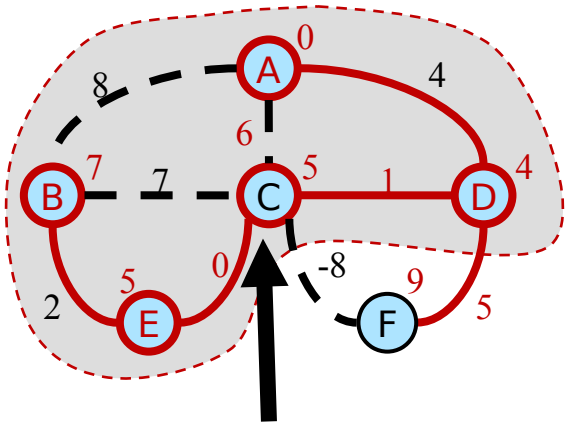
Why Dijkstra's Algorithm Works

- 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



Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!