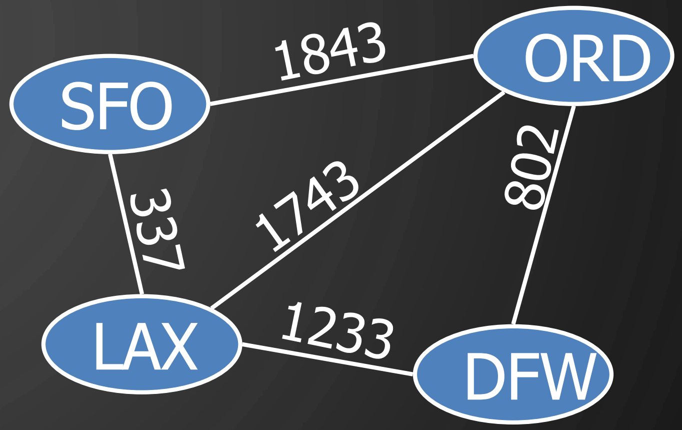
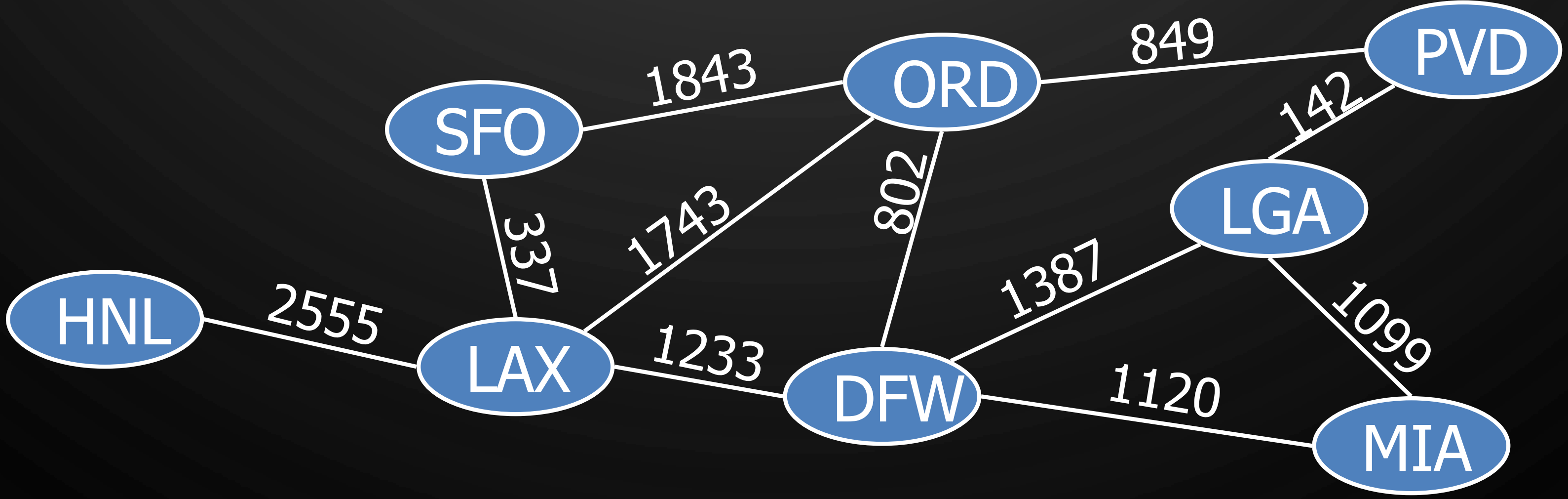
CHAPTER 13
GRAPH ALGORITHMS



ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO

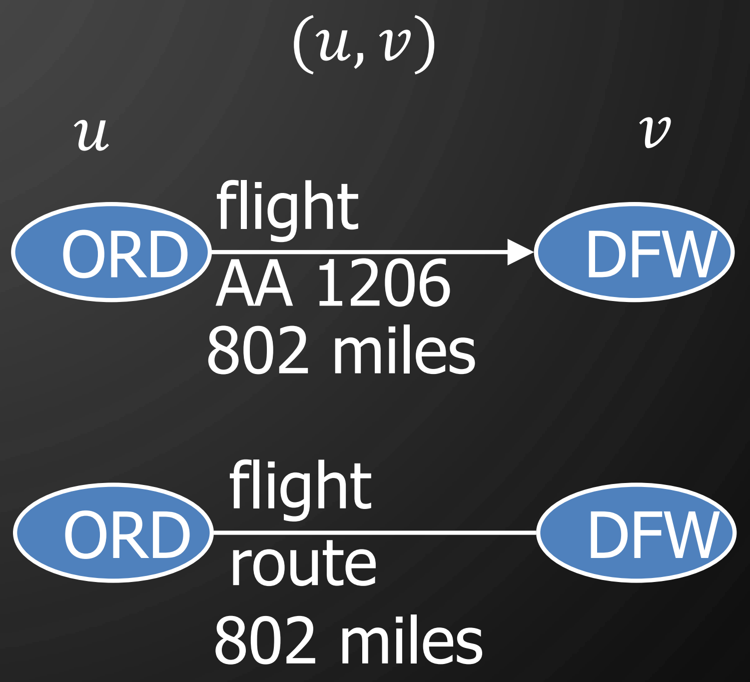
GRAPH

- A graph is a pair $G = (V, E)$, where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges can store arbitrary 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



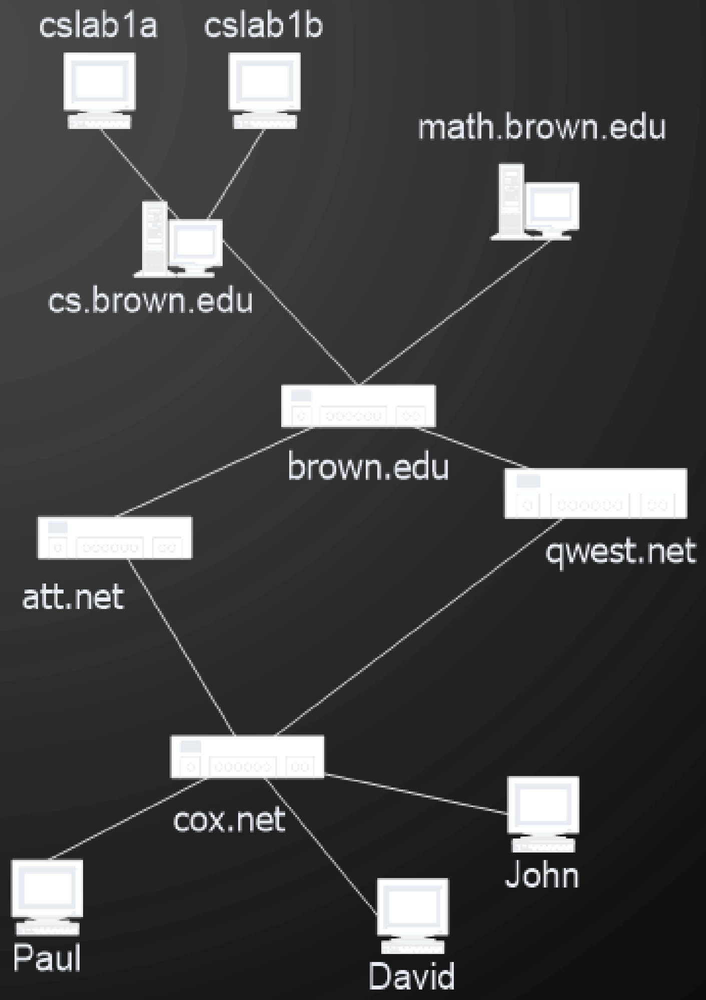
EDGE & GRAPH TYPES

- Edge Types
  - Directed edge
    - ordered pair of vertices (u, v)
    - first vertex u is the origin/source
    - second vertex v is the destination/target
    - e.g., a flight
  - Undirected edge
    - unordered pair of vertices (u, v)
    - e.g., a flight route
  - Weighted edge
- Graph Types
  - Directed graph (Digraph)
    - all the edges are directed
    - e.g., route network
  - Undirected graph
    - all the edges are undirected
    - e.g., flight network
  - Weighted graph
    - all the edges are weighted



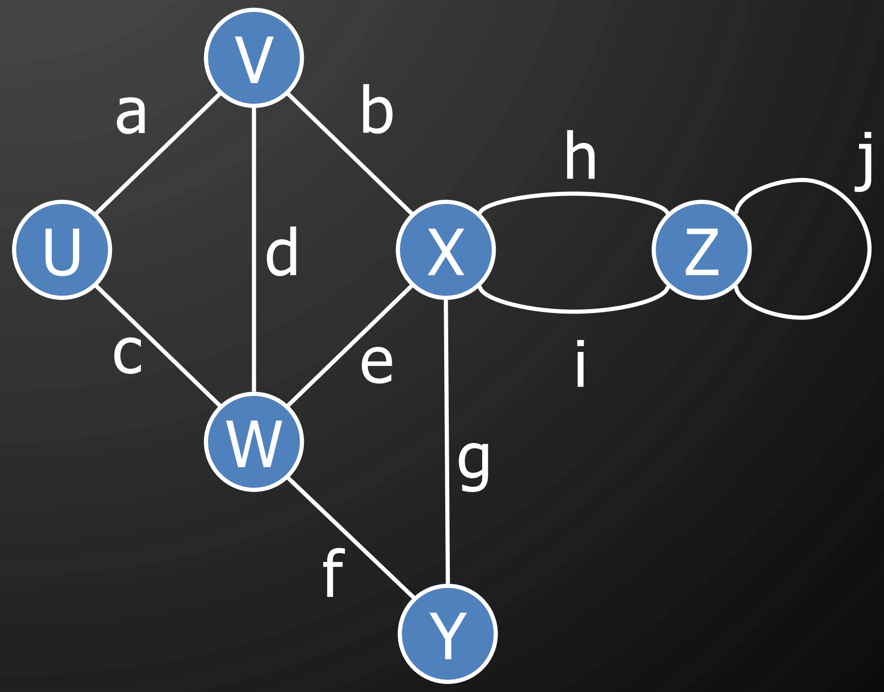
APPLICATIONS

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



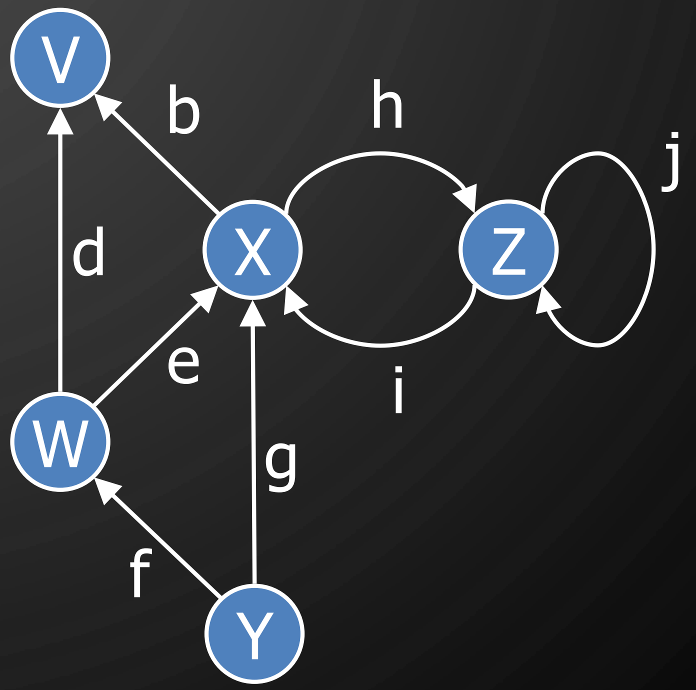
TERMINOLOGY

- End points (or end vertices) 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 (multiple) edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



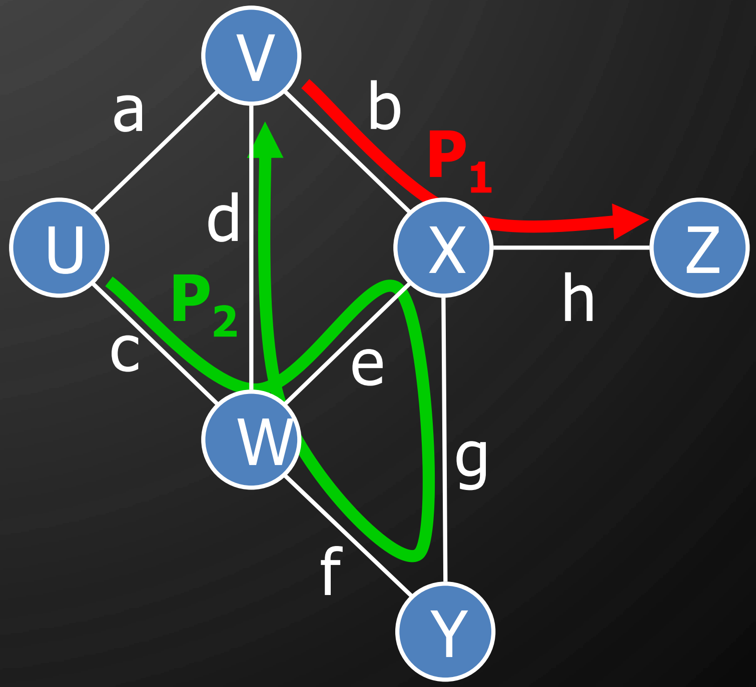
TERMINOLOGY

- Outgoing edges of a vertex
  - h and b are the outgoing edges of X
- Incoming edges of a vertex
  - e, g, and i are incoming edges of X
- In-degree of a vertex
  - X has in-degree 3
- Out-degree of a vertex
  - X has out-degree 2



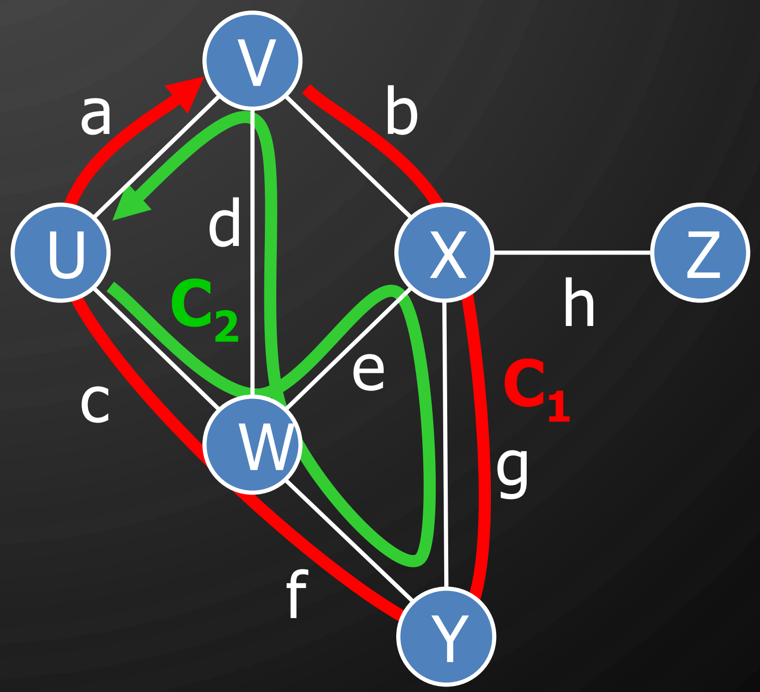
TERMINOLOGY

- 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



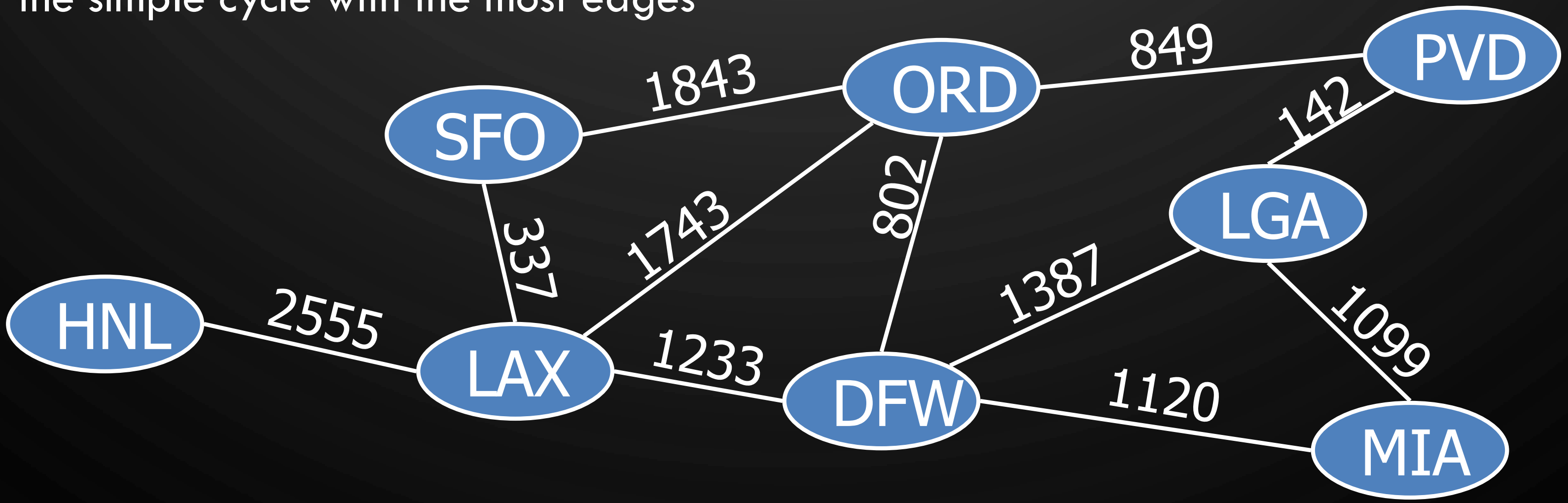
TERMINOLOGY

- 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
  - $C_1 = \{V, b, X, g, Y, f, W, c, U, a, V\}$ is a simple cycle
  - $C_2 = \{U, c, W, e, X, g, Y, f, W, d, V, a, U\}$ is a cycle that is not simple



EXERCISE ON TERMINOLOGY

1. Number of vertices?
2. Number of edges?
3. What type of the graph is it?
4. Show the end vertices of the edge with largest weight
5. Show the vertices of smallest degree and largest degree
6. Show the edges incident to the vertices in the above question
7. Identify the shortest simple path from HNL to PVD
8. Identify the simple cycle with the most edges



EXERCISE
PROPERTIES OF UNDIRECTED GRAPHS

- Property 1 – Total degree

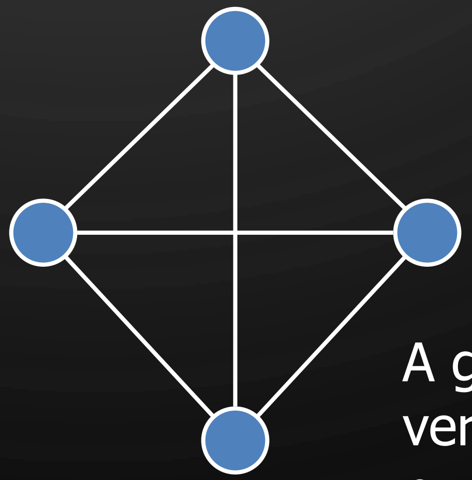
  $$\Sigma_v deg(v) = ?$$

- Property 2 – Total number of edges
  - In an undirected graph with no self-loops and no multiple edges

    $$m \leq Upper\ Bound?$$

    $$Lower\ Bound? \leq m$$

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



Example
- $n = ?$
- $m = ?$
- $\deg(v) = ?$

A graph with given number of vertices (4) and maximum number of edges

EXERCISE
PROPERTIES OF UNDIRECTED GRAPHS

- Property 1 – Total degree

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

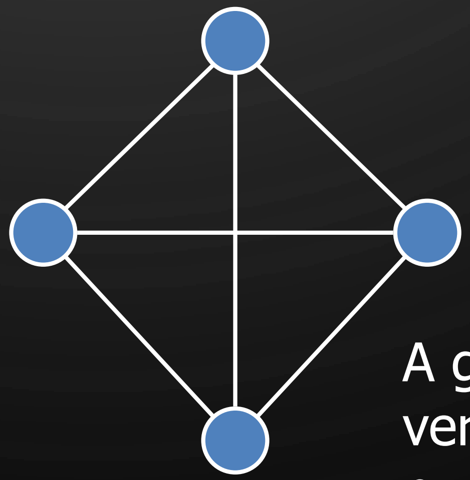
- Property 2 – Total number of edges
  - In an undirected graph with no self-loops and no multiple edges

$$m \leq \frac{n(n-1)}{2}$$

$$0 \leq m$$

Proof: Each vertex can have degree at most $(n-1)$

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



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

A graph with given number of vertices (4) and maximum number of edges

EXERCISE
PROPERTIES OF DIRECTED GRAPHS

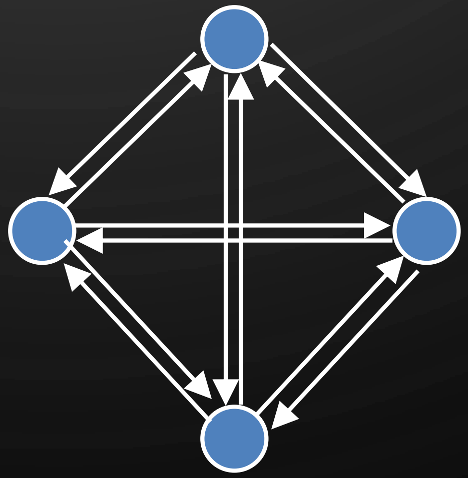
- Property 1 – Total in-degree and out-degree

$$\Sigma_v in - \deg(v) = ?$$
$$\Sigma_v out - \deg(v) = ?$$

- Property 2 – Total number of edges
  - In an directed graph with no self-loops and no multiple edges

$$m \leq UpperBound?$$
$$LowerBound? \leq m$$

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



Example
- $n = ?$
- $m = ?$
- $\deg(v) = ?$

A graph with given number of vertices (4) and maximum number of edges

EXERCISE
PROPERTIES OF DIRECTED GRAPHS

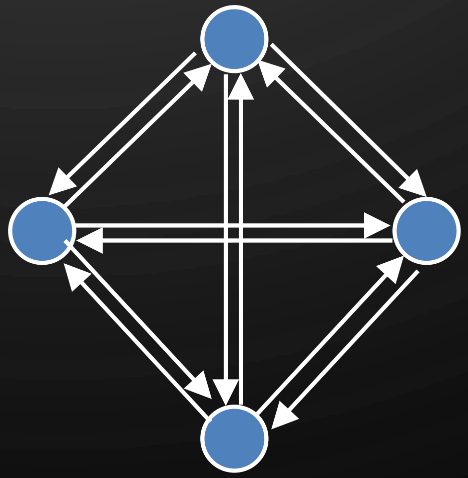
- Property 1 – Total in-degree and out-degree

$$\Sigma_v in - \deg(v) = m$$
$$\Sigma_v out - \deg(v) = m$$

- Property 2 – Total number of edges
  - In an directed graph with no self-loops and no multiple edges

$$m \leq n(n-1)$$
$$0 \leq m$$

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

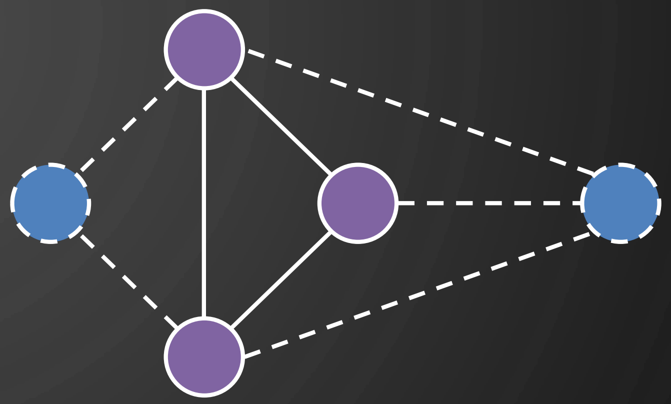


Example
- $n = 4$
- $m = 12$
- $\deg(v) = 6$

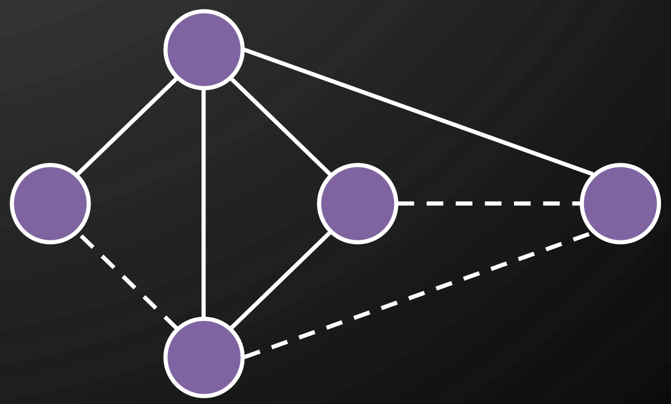
A graph with given number of vertices (4) and maximum number of edges

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



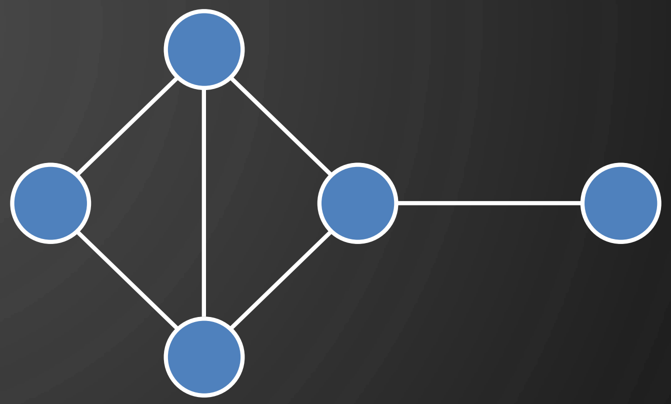
Subgraph



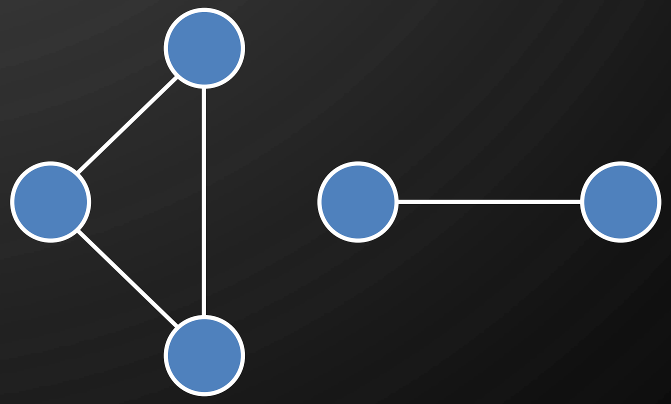
Spanning subgraph

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



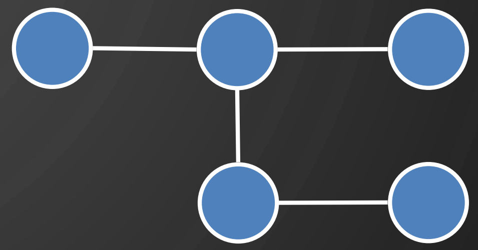
Connected graph



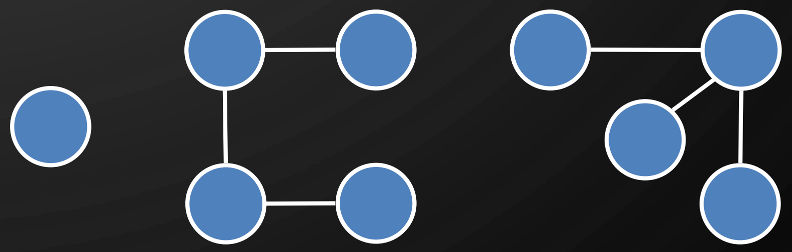
Non connected graph with two connected components

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



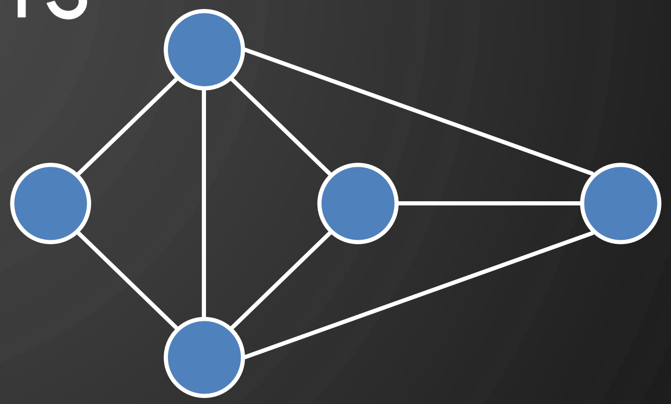
Tree



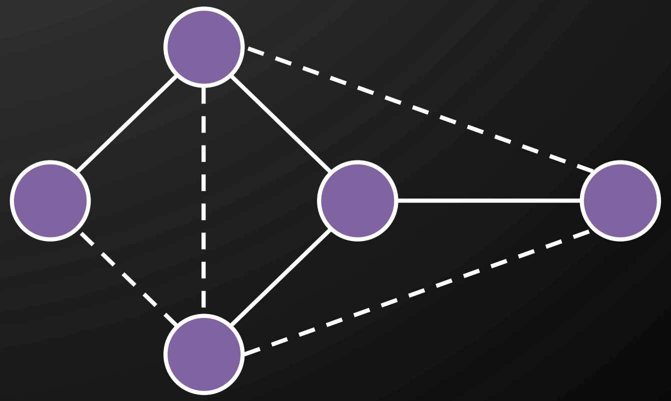
Forest

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



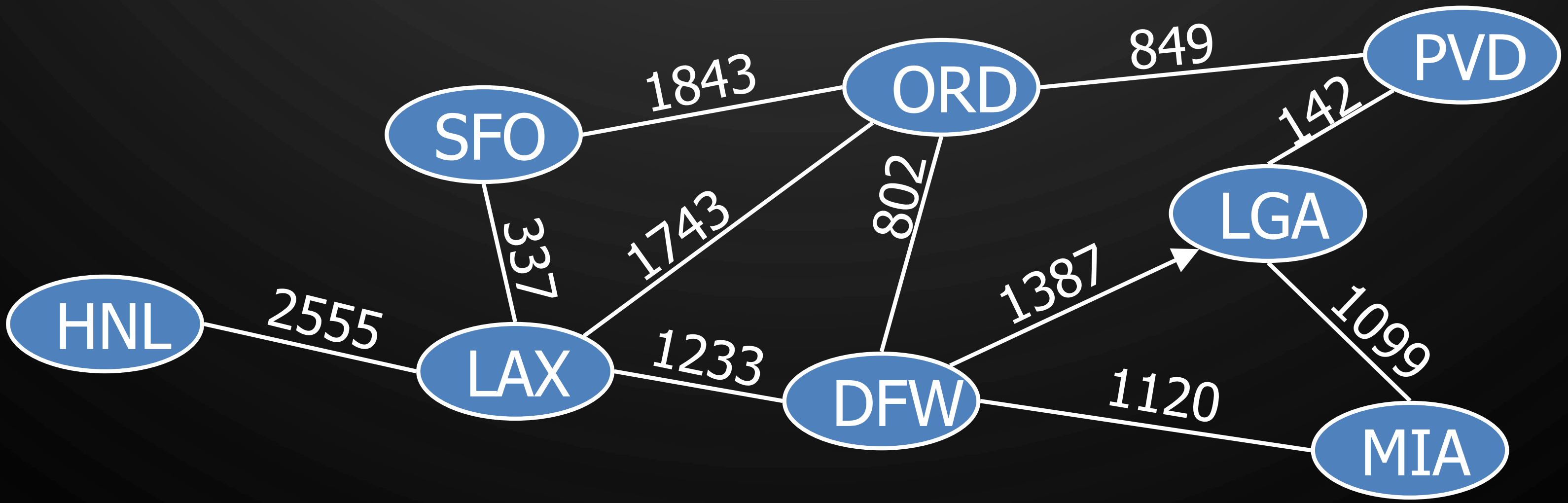
Spanning tree

GRAPH ADT

- Vertices and edges are positions and store elements
- Vertex ADT
  - $\text{operator} * (\quad)$
  - $\text{incidentEdges}(\quad)$
  - $\text{isAdjacentTo}(v)$
- Edge ADT
  - $\text{operator} * (\quad)$
  - $\text{endVertices}(\quad)$
  - $\text{opposite}(v)$
  - $\text{isAdjacentTo}(f)$
  - $\text{isIncidentOn}(v)$
  - $\text{isDirected}(\quad)$
  - $\text{origin}(\quad)$
  - $\text{dest}(\quad)$
- Graph ADT
  - $\text{vertices}(\quad)$
  - $\text{edges}(\quad)$
  - $\text{insertVertex}(x)$
  - $\text{insertEdge}(v, w, x)$
  - $\text{insertDirectedEdge}(v, w, x)$
  - $\text{eraseVertex}(v)$
  - $\text{eraseEdge}(e)$
- Many more generic/accessor methods
- Lists of entities provide iterators

EXERCISE ON ADT

1. $ord.\text{incidentEdges}()$
2. $ord.\text{adjacentVertices}()$
3. $ord.\text{degree}()$
4. $(lga, mia).\text{endVertices}()$
5. $(dfw, lga).\text{opposite}(dfw)$
6. $dfw.\text{isAdjacentTo}(sfo)$
7. $\text{insertVertex}(iah)$
8. $\text{insertEdge}(mia, pvd, 1200)$
9. $\text{eraseVertex}(ord)$
10. $\text{eraseEdge}(dfw, ord)$
11. $(dfw, lga).\text{isDirected}()$
12. $(dfw, lga).\text{origin}()$
13. $(dfw, lga).\text{dest}()$



EDGE LIST STRUCTURE

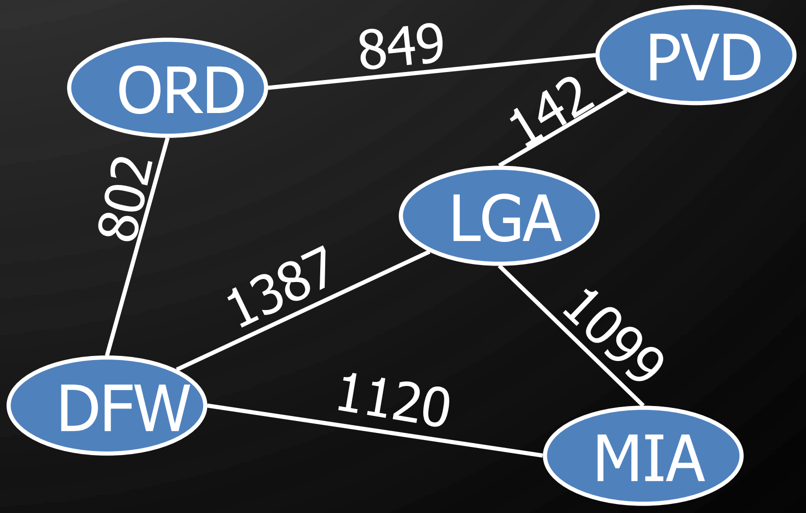
Edge List

(ORD, PVD)	849
(ORD, DFW)	802
(LGA, PVD)	142
(LGA, MIA)	1099
(DFW, LGA)	1387
(DFW, MIA)	1120

Vertex Sequence

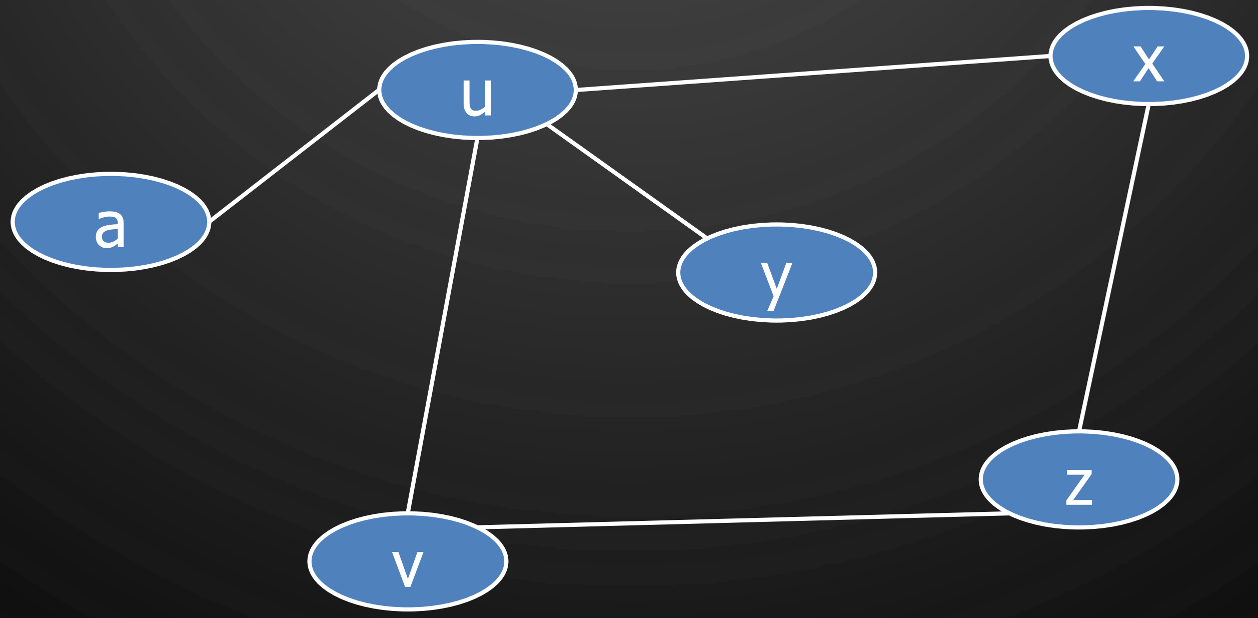
- ORD
- LGA
- PVD
- DFW
- MIA

- An edge list can be stored in a sequence, a vector, a list or a dictionary such as a hash table



EXERCISE
EDGE LIST STRUCTURE

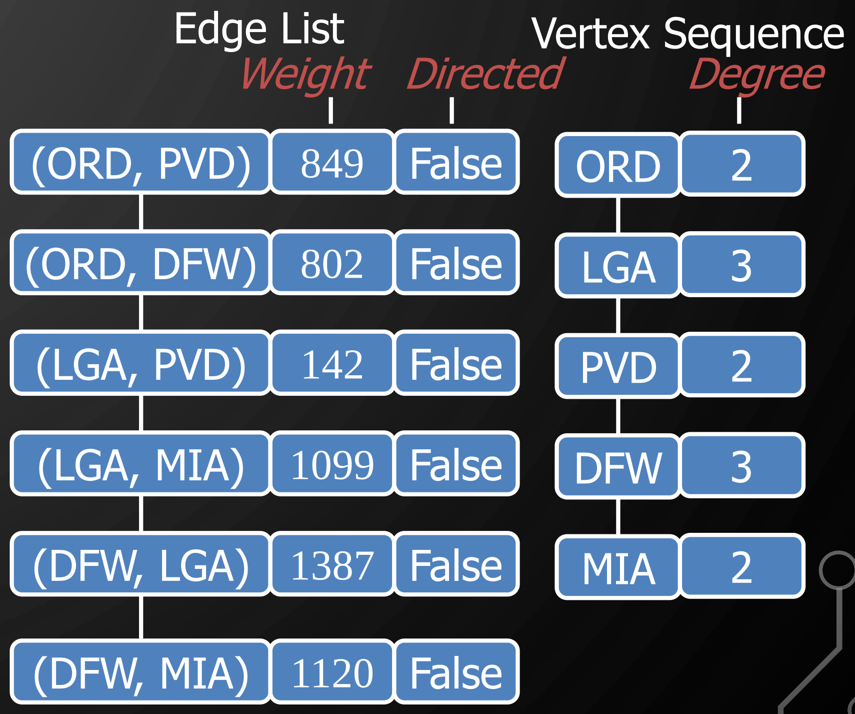
- Construct the edge list for the following graph



ASYMPTOTIC PERFORMANCE

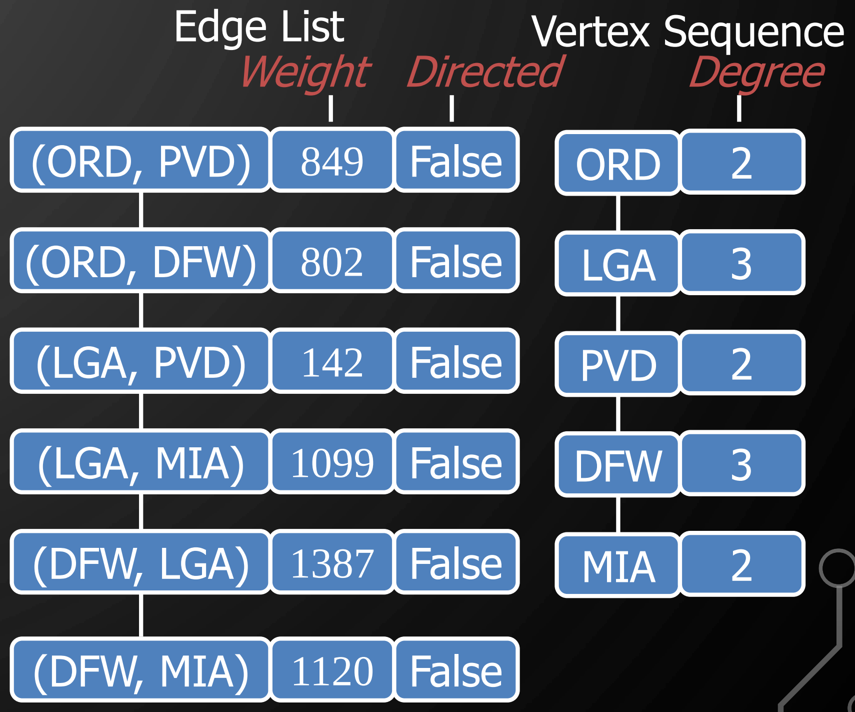
EDGE LIST STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Edge List
Space	?
$\text{endVertices}()$, $\text{opposite}()$, $\text{isIncidentOn}(v)$	?
$v.\text{incidentEdges}()$, $v.\text{isAdjacentTo}(w)$	?
$\text{insertVertex}(x)$, $\text{insertEdge}(u, v, w)$, $\text{eraseEdge}(e)$	?
$\text{eraseVertex}(v)$	?



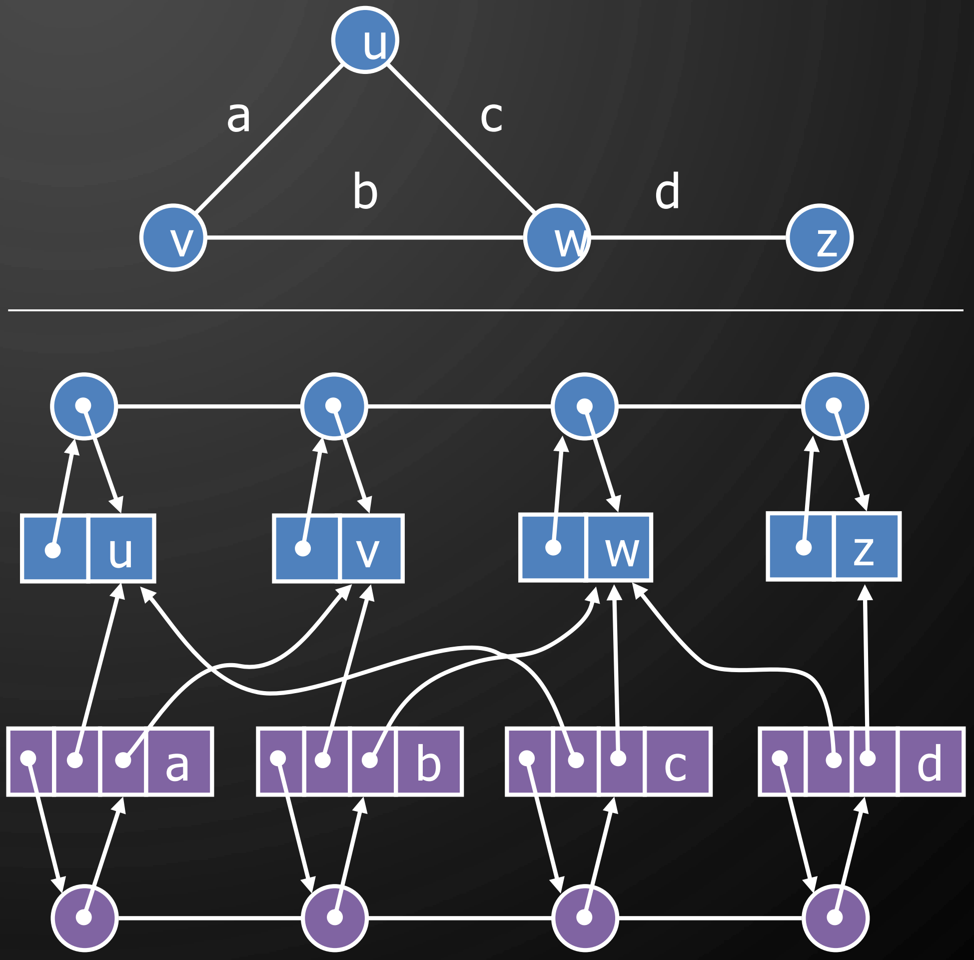
ASYMPTOTIC PERFORMANCE
EDGE LIST STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Edge List
Space	$O(n+m)$
endVertices(), opposite(), isIncidentOn(v)	$O(1)$
v.incidentEdges(), v.isAdjacentTo(w)	$O(m)$
insertVertex(x), insertEdge(u, v, w), eraseEdge(e)	$O(1)$
eraseVertex(v)	$O(m)$

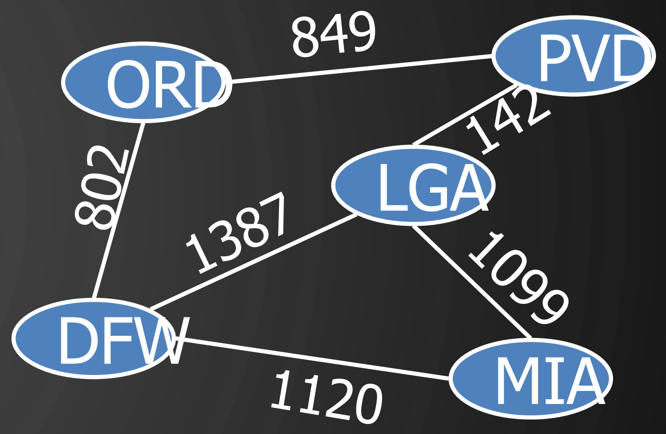


EDGE LIST STRUCTURE

- 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

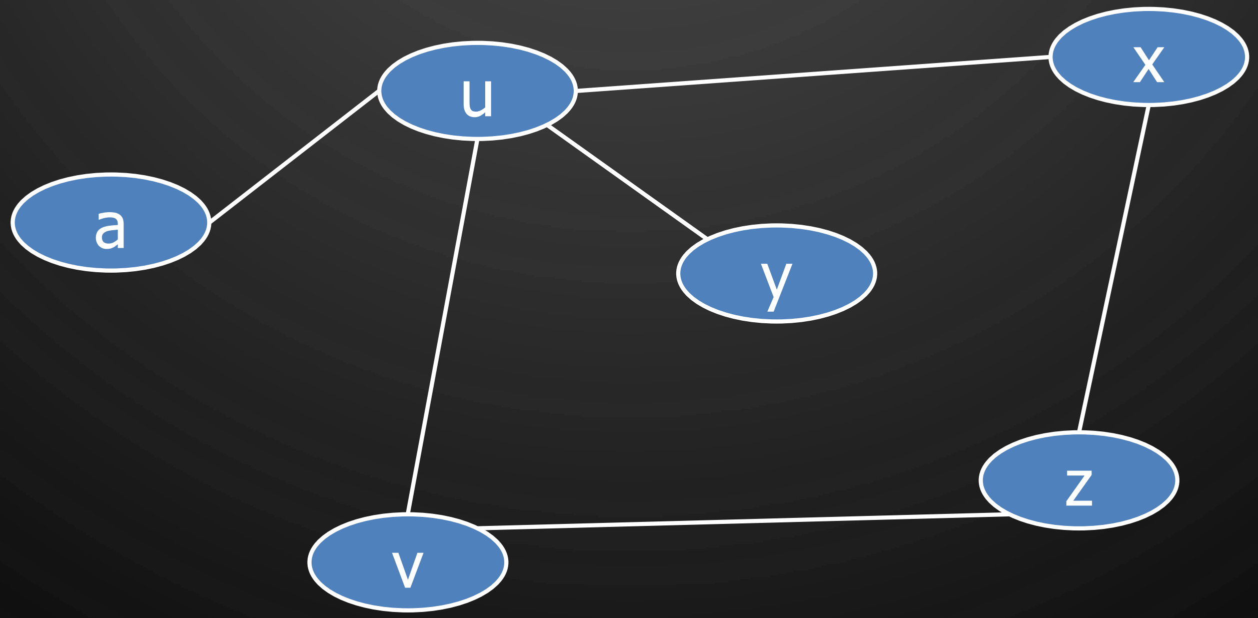


Adjacency List

- Adjacency Lists associate edges with their end vertices
- Each vertex stores a list of incident edges

EXERCISE
ADJACENCY LIST STRUCTURE

- Construct the adjacency list for the following graph

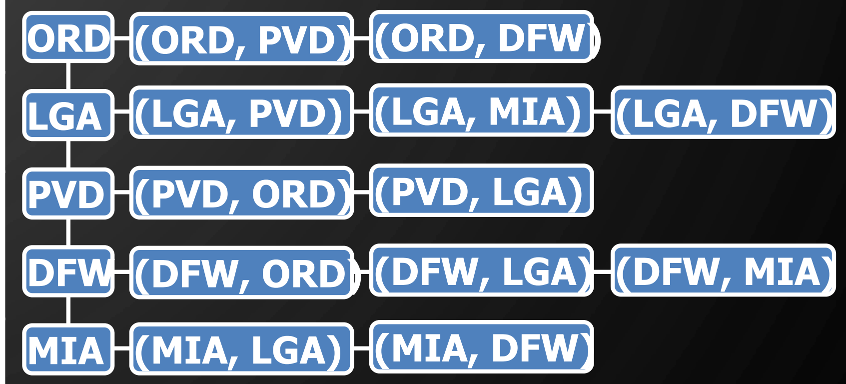


ASYMPTOTIC PERFORMANCE

ADJACENCY LIST STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Adjacency List
Space	?
endVertices(), opposite(), isIncidentOn(v)	?
v.incidentEdges(), v.isAdjacentTo(w)	?
insertVertex(x), insertEdge(u, v, w), eraseEdge(e)	?
eraseVertex(v)	?

Adjacency List

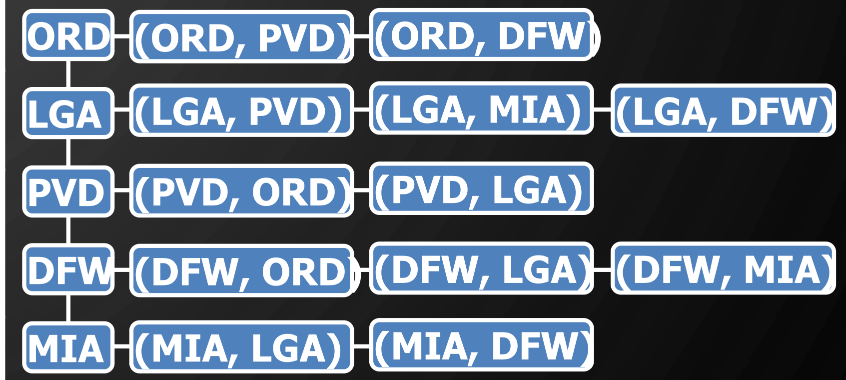


ASYMPTOTIC PERFORMANCE

ADJACENCY LIST STRUCTURE

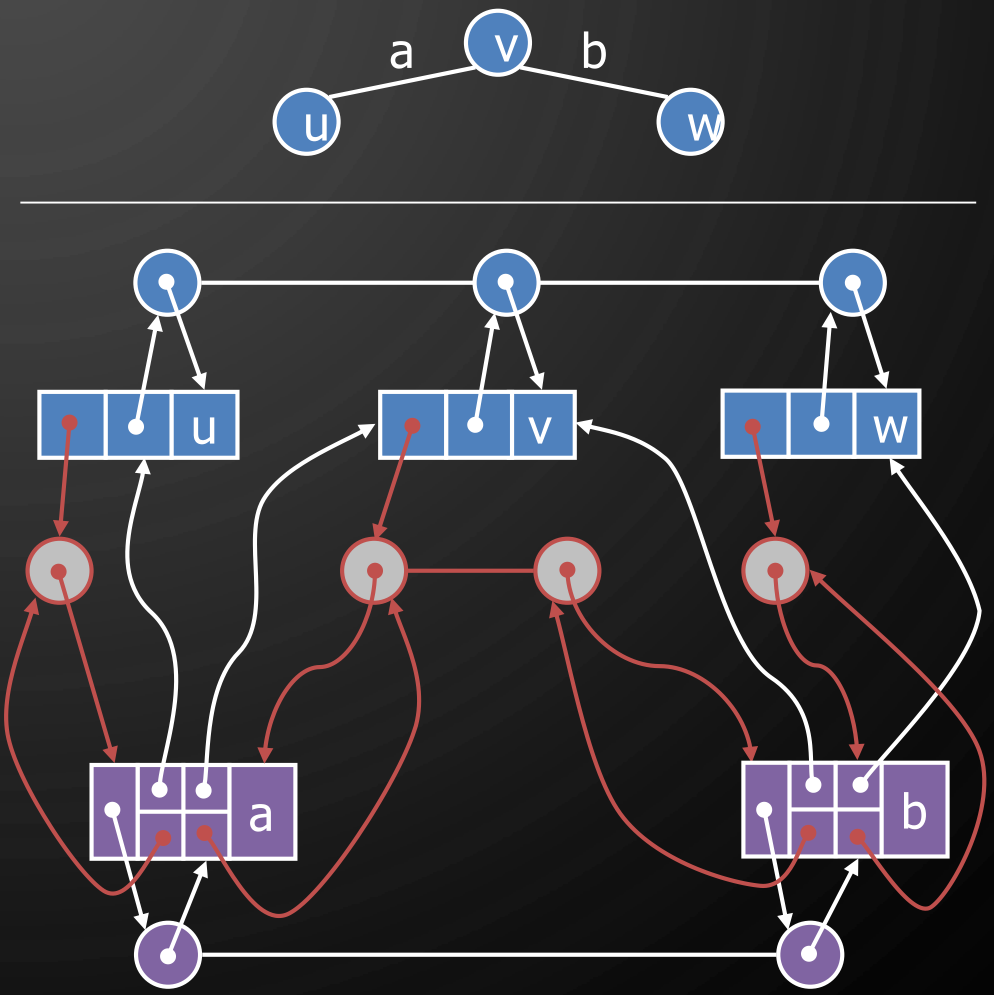
• n vertices, m edges • No parallel edges • No self-loops	Adjacency List
Space	$O(n+m)$
endVertices(), opposite(), isIncidentOn(v)	$O(1)$
v.incidentEdges(), v.isAdjacentTo(w)	$O(\deg(v))$ $O(\min(\deg(v), \deg(w)))$
insertVertex(x), insertEdge(u, v, w), eraseEdge(e)	$O(1)$
eraseVertex(v)	$O(\deg(v))$

Adjacency List



ADJACENCY LIST STRUCTURE

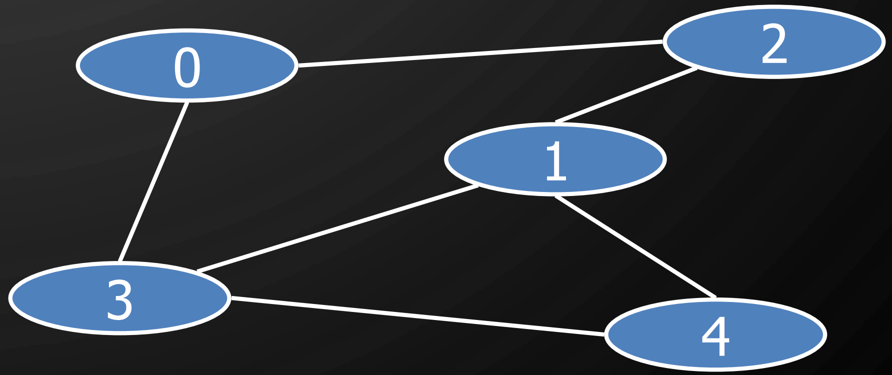
- Store vertex sequence and edge sequence
- Each vertex stores a sequence of incident edges
    - Sequence of references to edge objects of incident edges
- Augmented edge objects
    - References to associated positions in incidence sequences of end vertices



ADJACENCY MATRIX STRUCTURE

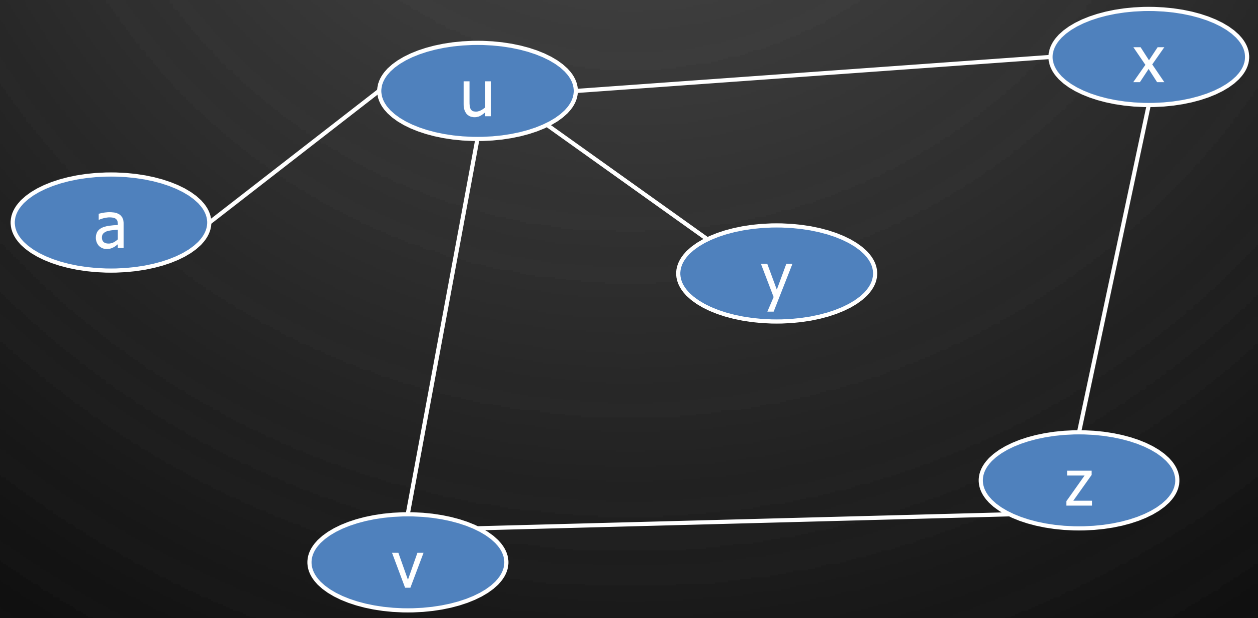
	0	1	2	3	4
0	**0**	**0**	**1**	**1**	**0**
1	**0**	**0**	**1**	**1**	**1**
2	**1**	**1**	**0**	**0**	**0**
3	**1**	**1**	**0**	**0**	**1**
4	**0**	**1**	**0**	**1**	**0**

- Adjacency matrices store edges in a table, indexed by the vertex



EXERCISE
ADJACENCY MATRIX STRUCTURE

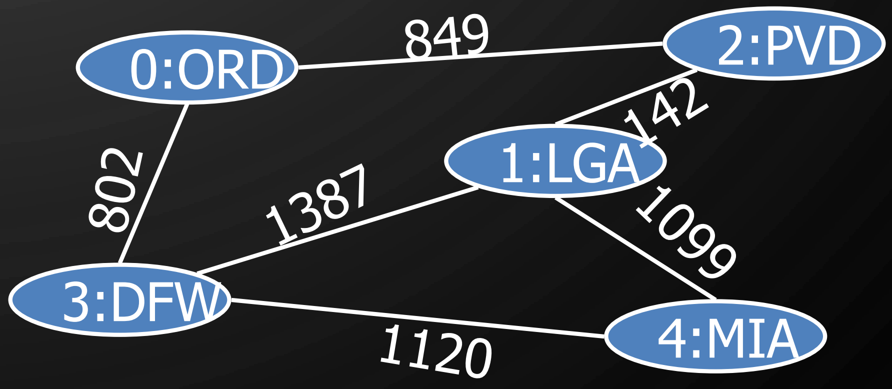
- Construct the adjacency matrix for the following graph



ADJACENCY MATRIX STRUCTURE IN A WEIGHTED GRAPH

	0 ORD	1 LGA	2 PVD	3 DFW	4 MIA
0 ORD	0	0	849	802	0
1 LGA	0	0	142	1387	1099
2 PVD	849	142	0	0	0
3 DFW	802	138	0	0	1120
4 MIA	0	1099	0	1120	0

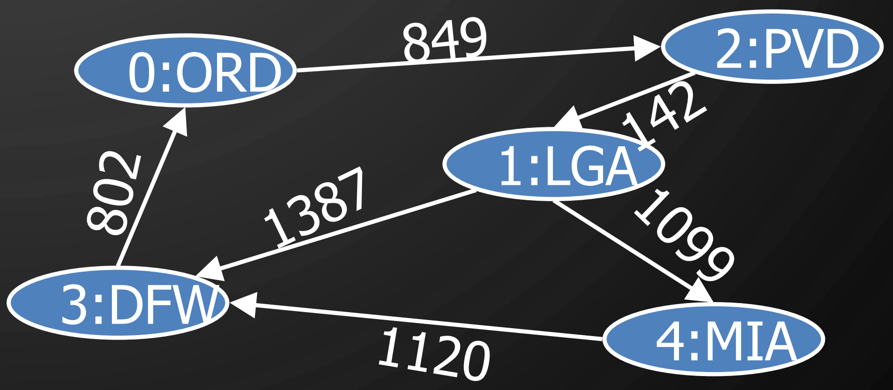
- Store edge object/property in table, or include a pointer to it inside of the table



EXERCISE

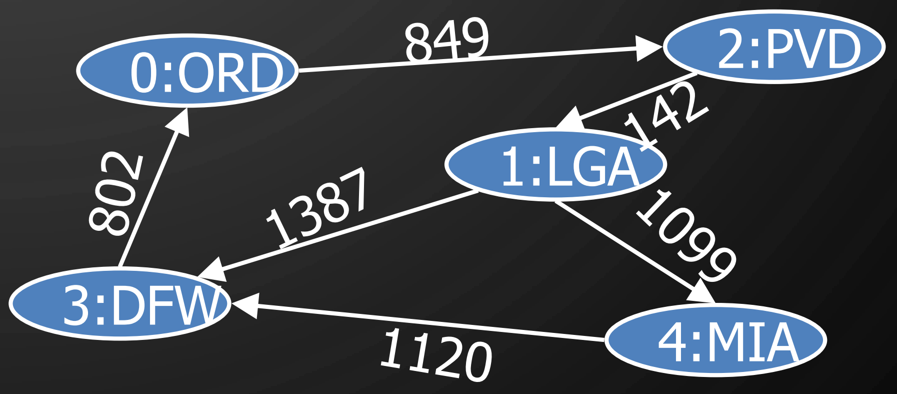
ADJACENCY MATRIX STRUCTURE: WEIGHTED DIGRAPH

	0 ORD	1 LGA	2 PVD	3 DFW	4 MIA
0 ORD					
1 LGA					
2 PVD					
3 DFW					
4 MIA					



EXERCISE
ADJACENCY MATRIX STRUCTURE: WEIGHTED DIGRAPH

	0 ORD	1 LGA	2 PVD	3 DFW	4 MIA
0 ORD	0	0	849	0	0
1 LGA	0	0	0	1387	1099
2 PVD	0	142	0	0	0
3 DFW	802	0	0	0	0
4 MIA	0	0	0	1120	0



ASYMPTOTIC PERFORMANCE OF ADJACENCY MATRIX STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Adjacency Matrix
Space	?
endVertices(), opposite(), isIncidentOn(v), v.isAdjacentTo(w)	?
v.incidentEdges()	?
insertEdge(u, v, w), eraseEdge(e)	?
insertVertex(x), eraseVertex(v)	?

	0	1	2	3	4
0	0	0	1	1	0
1	0	0	1	1	1
2	1	1	0	0	0
3	1	1	0	0	1
4	0	1	0	1	0

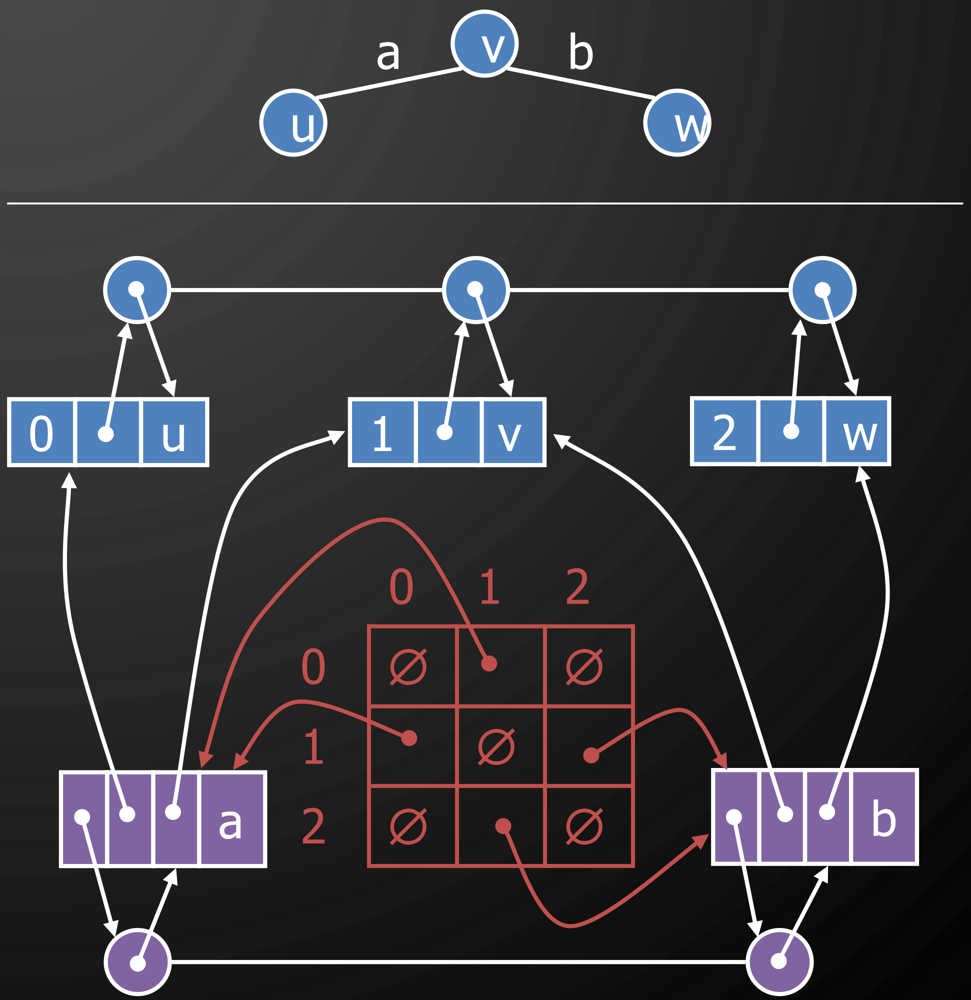
ASYMPTOTIC PERFORMANCE OF ADJACENCY MATRIX STRUCTURE

• n vertices, m edges • No parallel edges • No self-loops	Adjacency Matrix
Space	$O(n^2)$
endVertices(), opposite(), isIncidentOn(v), v.isAdjacentTo(w)	$O(1)$
v.incidentEdges()	$O(n)$
insertEdge(u, v, w), eraseEdge(e)	$O(1)$
insertVertex(x), eraseVertex(v)	$O(n^2)$

	0	1	2	3	4
0	0	0	1	1	0
1	0	0	1	1	1
2	1	1	0	0	0
3	1	1	0	0	1
4	0	1	0	1	0

ADJACENCY MATRIX STRUCTURE

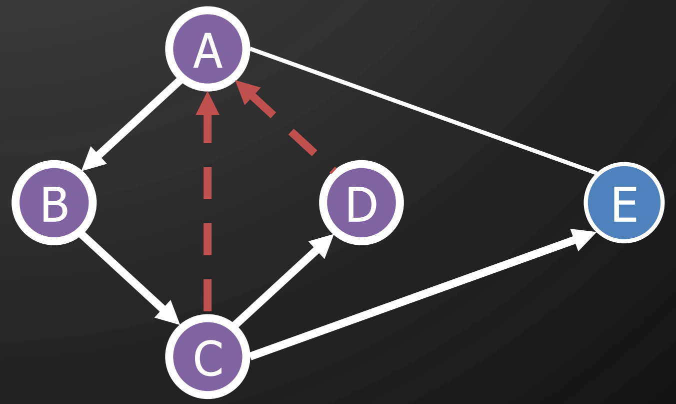
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



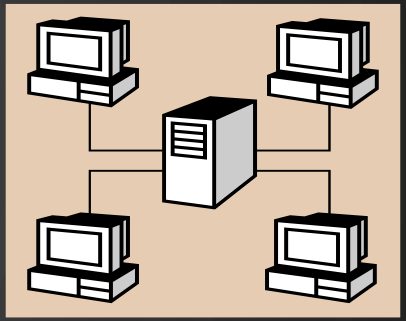
ASYMPTOTIC PERFORMANCE

• n vertices, m edges • No parallel edges • No self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n+m)$	$O(n+m)$	$O(n^2)$
$\text{endVertices}()$, $\text{opposite}()$, $\text{isIncidentOn}(v)$	$O(1)$	$O(1)$	$O(1)$
$v.\text{incidentEdges}()$	$O(m)$	$O(\deg(v))$	$O(n)$
$v.\text{isAdjacentTo}(w)$	$O(m)$	$O(\min(\deg(v), \deg(w)))$	$O(1)$
$\text{insertEdge}(u, v, w)$, $\text{eraseEdge}(e)$	$O(1)$	$O(1)$	$O(1)$
$\text{insertVertex}(x)$	$O(1)$	$O(1)$	$O(n^2)$
$\text{eraseVertex}(v)$	$O(m)$	$O(\deg(v))$	$O(n^2)$

DEPTH-FIRST SEARCH



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

unexplored vertex

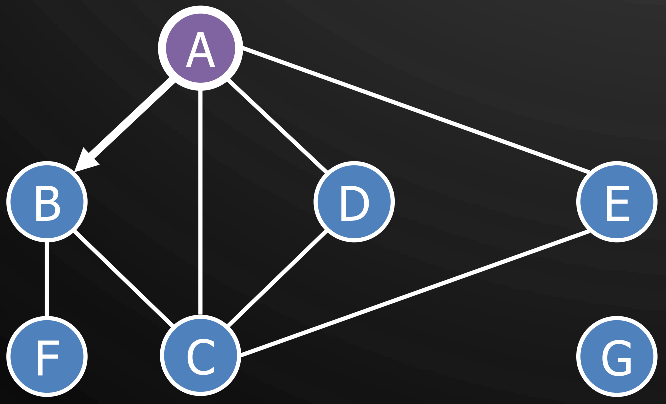
visited vertex

unexplored edge

discovery edge

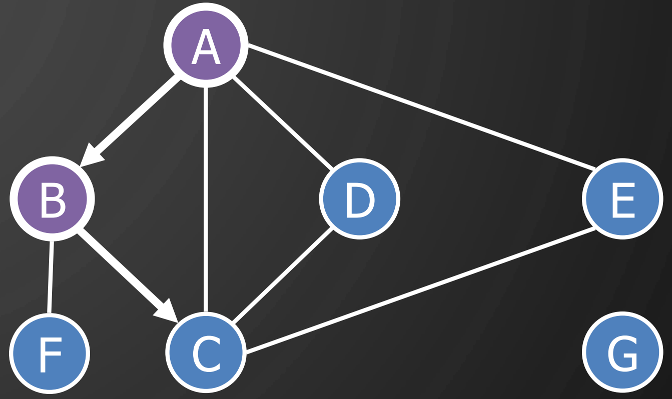
back edge

$A(A) = \{\boldsymbol{B}, C, D, E\}$

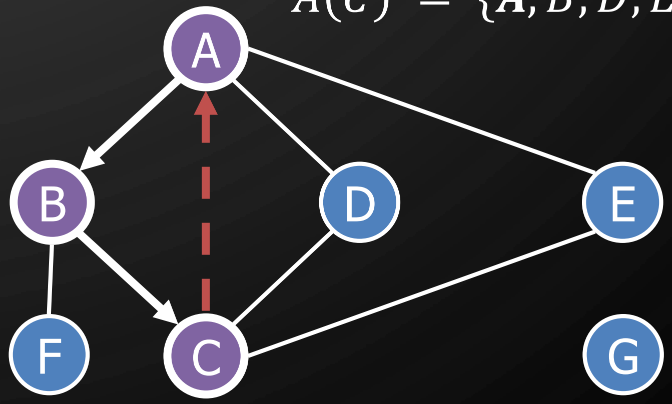


$A(B) = \{\boldsymbol{A}, C, F\}$

$A(B) = \{A, \boldsymbol{C}, F\}$



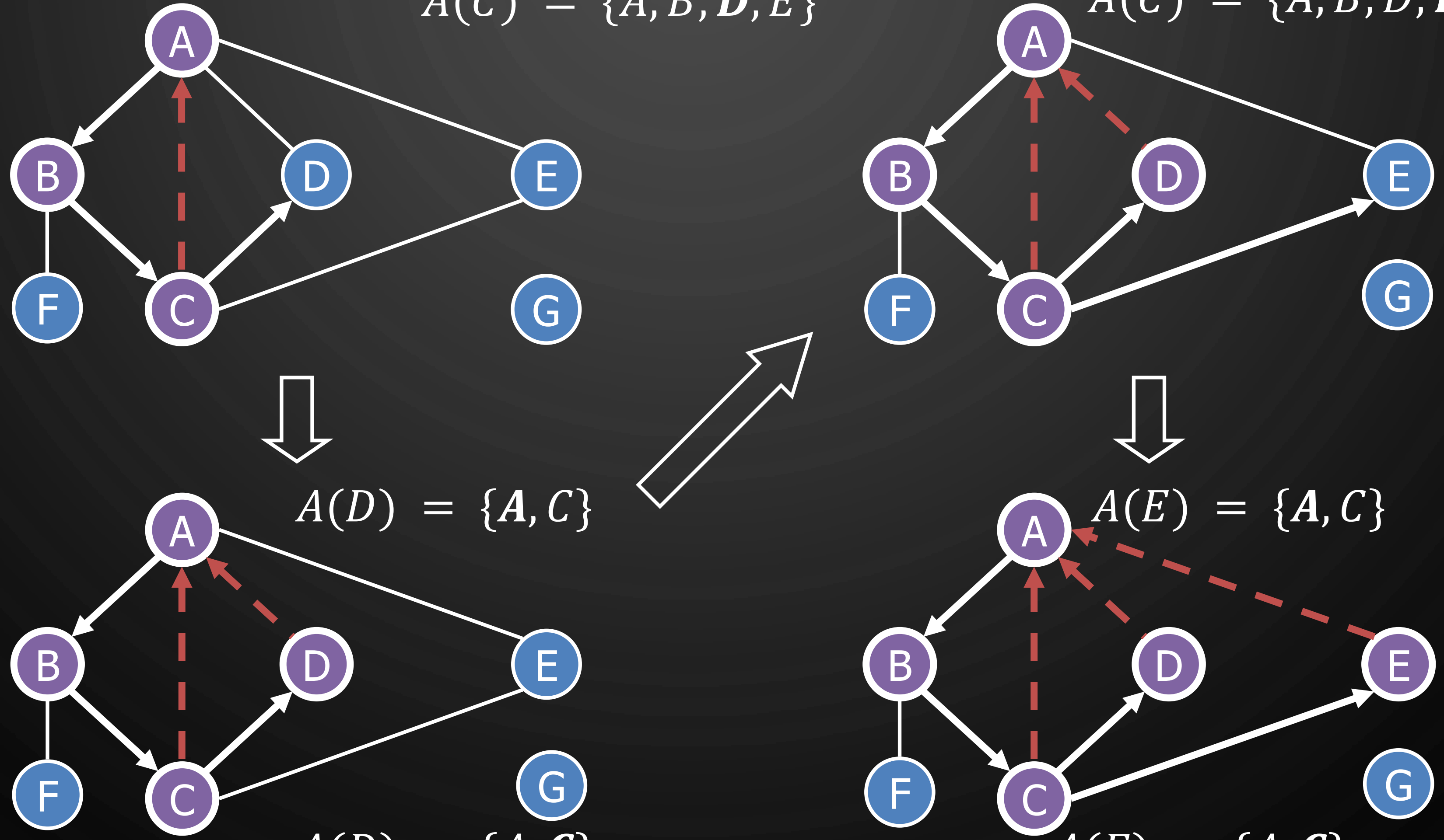
$A(C) = \{\boldsymbol{A}, B, D, E\}$



$A(C) = \{A, \boldsymbol{B}, D, E\}$

$A(C) = \{A, B, \boldsymbol{D}, E\}$

EXAMPLE



$A(C) = \{A, B, \boldsymbol{D}, E\}$

$A(D) = \{\boldsymbol{A}, C\}$

$A(D) = \{A, \boldsymbol{C}\}$

$A(D) = \{A, C\}$

$A(C) = \{A, B, D, \boldsymbol{E}\}$

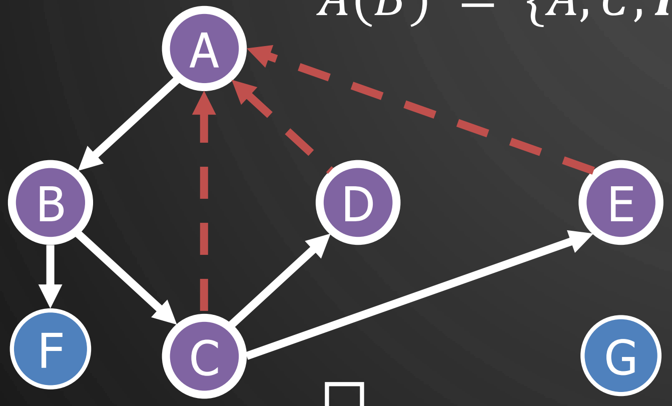
$A(E) = \{\boldsymbol{A}, C\}$

$A(E) = \{A, \boldsymbol{C}\}$

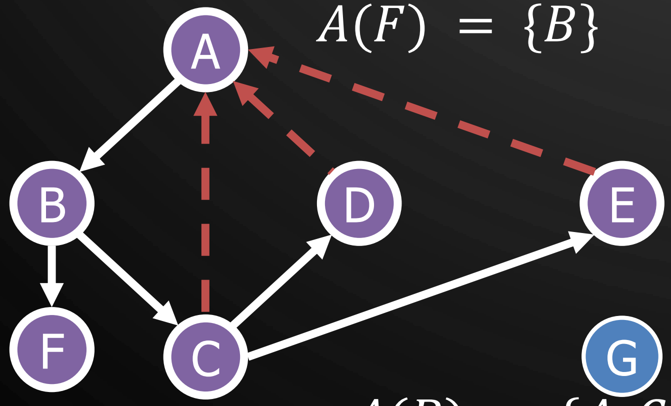
$A(E) = \{A, C\}$

EXAMPLE

$$A(C) = \{A, B, D, E\}$$
$$A(B) = \{A, C, \boldsymbol{F}\}$$

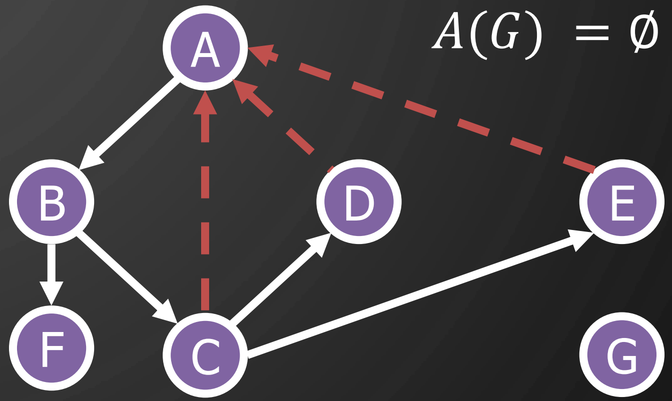


$$A(F) = \{B\}$$

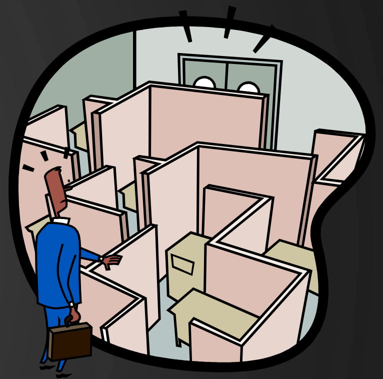


$$A(B) = \{A, C, F\}$$
$$A(A) = \{A, B, C, D\}$$

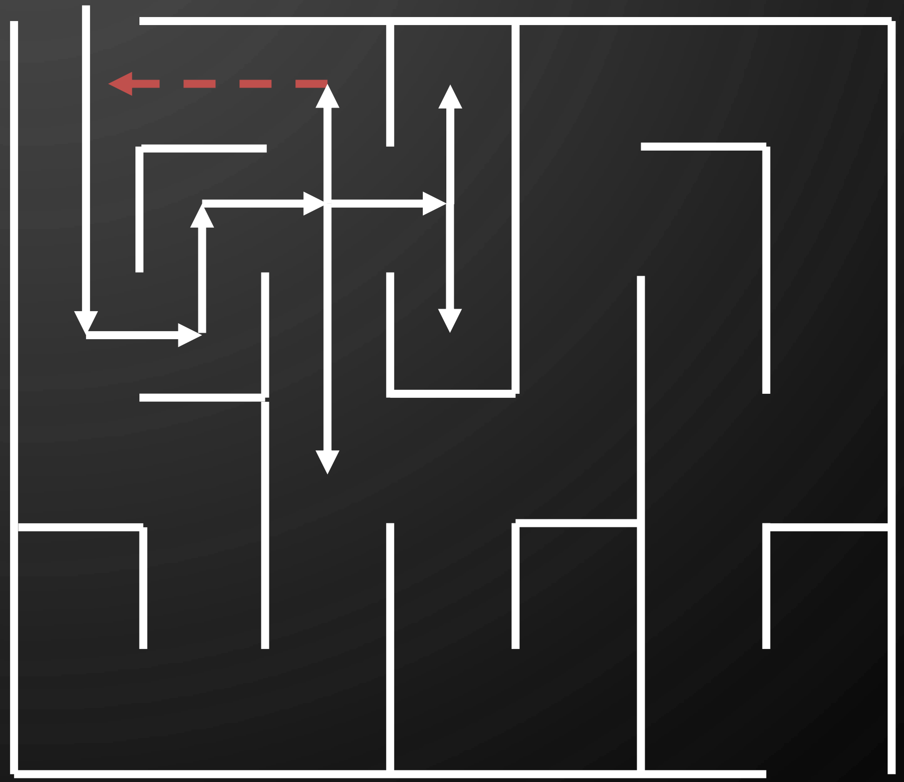
$$A(G) = \emptyset$$



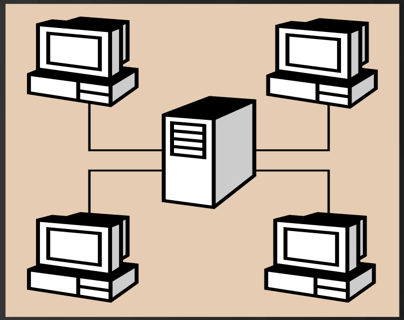
DFS AND MAZE TRAVERSAL



- 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)



DFS ALGORITHM



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

Algorithm $\mathrm{DFS}(G)$
Input: Graph G
Output: Labeling of the edges of G as discovery edges
and back edges

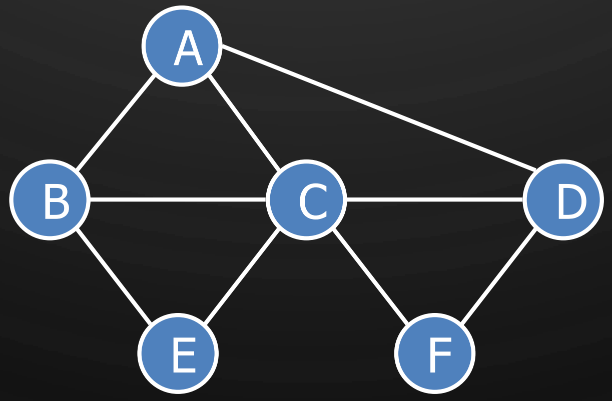
1. **for each** $v \in G.\mathrm{vertices}()$ **do**
2. $v.\mathrm{setLabel}(UNEXPLORED)$
3. **for each** $e \in G.\mathrm{edges}()$ **do**
4. $e.\mathrm{setLabel}(UNEXPLORED)$
5. **for each** $v \in G.\mathrm{vertices}()$ **do**
6. **if** $v.\mathrm{getLabel}() = UNEXPLORED$
7. $\mathrm{DFS}(G, v)$

Algorithm $\mathrm{DFS}(G, v)$
Input: Graph G and a start vectex v
Output: Labeling of the edges of G in the connected component of v as discovery edges and back edges

1. $v.\mathrm{setLabel}(VISITED)$
2. **for each** $e \in v.\mathrm{incidentEdges}()$ **do**
3. **if** $e.\mathrm{getLabel}() = UNEXPLORED)$
4. $w \leftarrow e.\mathrm{opposite}(v)$
5. **if** $w.\mathrm{getLabel}() = UNEXPLORED$
6. $e.\mathrm{setLabel}(DISCOVERY)$
7. $\mathrm{DFS}(G, w)$
8. **else**
9. $e.\mathrm{setLabel}(BACK)$

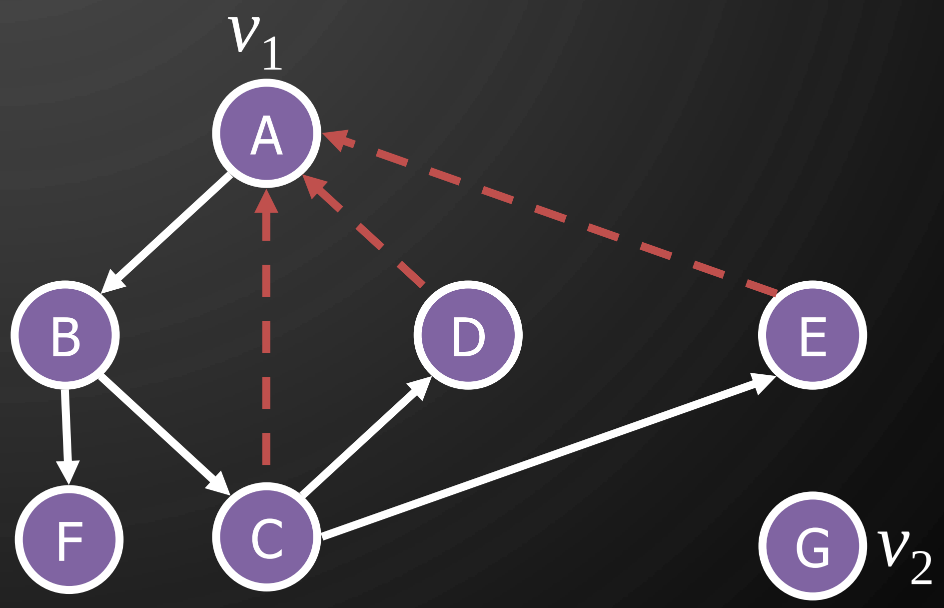
EXERCISE
DFS ALGORITHM

- Perform DFS of the following graph, start from vertex A
  - Assume adjacent edges are processed in alphabetical order
  - Number vertices in the order they are visited
  - Label edges as discovery or back edges

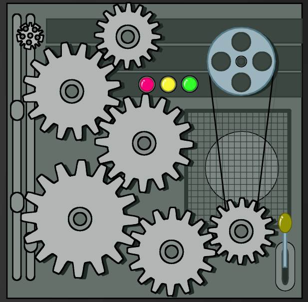


PROPERTIES OF DFS

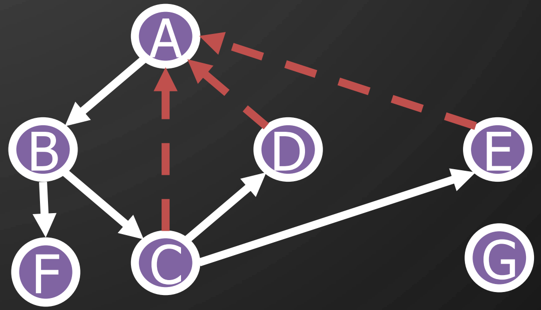
- Property 1
  - $\mathrm{DFS}(G, v)$ visits all the vertices and edges in the connected component of v
- Property 2
  - The discovery edges labeled by $\mathrm{DFS}(G, v)$ form a spanning tree of the connected component of v



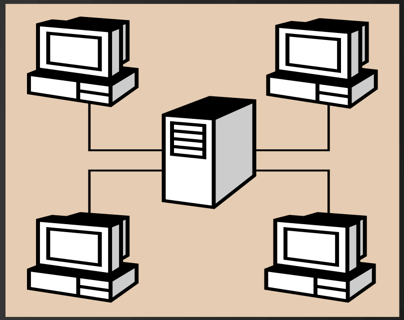
ANALYSIS OF DFS



- 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 $BACK$
- Function $\mathrm{DFS}(G, v)$ and the method $\mathrm{incidentEdges}(\,)$ are called once for each vertex



ANALYSIS OF DFS



- DFS runs in $O(n+m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Algorithm $\text{DFS}(G)$
Input: Graph G
Output: Labeling of the edges of G as discovery edges
and back edges

for each $v \in G.\text{vertices}(\,)$ **do** $O(n)$
  $v.\text{setLabel}(UNEXPLORED)$
for each $e \in G.\text{edges}(\,)$ **do** $O(m)$
  $e.\text{setLabel}(UNEXPLORED)$
for each $v \in G.\text{vertices}(\,)$ **do** $O(n+m)$
  if $v.\text{getLabel}(\,) = UNEXPLORED$
    $\text{DFS}(G, v)$

Algorithm $\text{DFS}(G, v)$
Input: Graph G and a start vectex v
Output: Labeling of the edges of G in the connected component of v as discovery edges and back edges

$v.\text{setLabel}(VISITED)$
for each $e \in v.\text{incidentEdges}(\,)$ **do** $O(deg(v))$
  if $e.\text{getLabel}(\,) = UNEXPLORED)$
    $w \leftarrow e.\text{opposite}(v)$
    if $w.\text{getLabel}(\,) = UNEXPLORED$
      $e.\text{setLabel}(DISCOVERY)$
      $\text{DFS}(G, w)$
    else
      $e.\text{setLabel}(BACK)$

APPLICATION

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 $\mathrm{DFS}(G, u)$ 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)

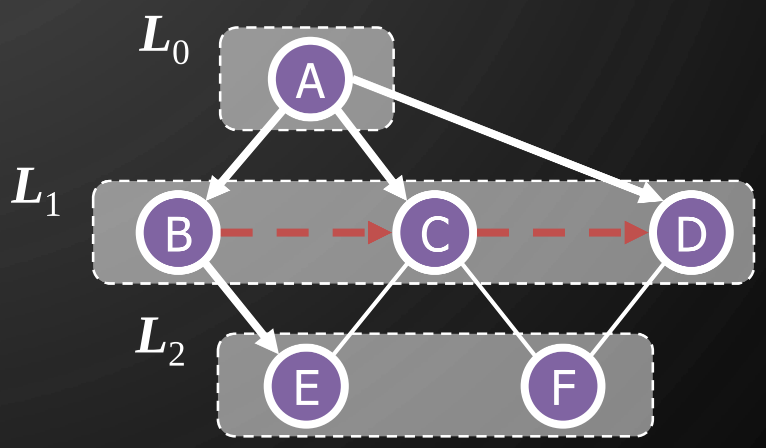
```
v.setLabel(VISITED)
S.push(v)
if v = z
  return S.elements()
for each e ∈ v.incidentEdges() do
  if e.getLabel() = UNEXPLORED)
    w ← e.opposite(v)
    if w.getLabel() = UNEXPLORED
      e.setLabel(DISCOVERY)
      S.push(e)
      pathDFS(G, w)
      S.pop()
    else
      e.setLabel(BACK)
S.pop()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

APPLICATION
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)
  v.setLabel(VISITED)
  S.push(v)
  for each e ∈ v.incidentEdges() do
    if e.getLabel() = UNEXPLORED)
      w ← e.opposite(v)
      S.push(e)
      if w.getLabel() = UNEXPLORED
        e.setLabel(DISCOVERY)
        cycleDFS(G, w)
        S.pop()
      else
        T ← empty stack
        repeat
          T.push(S.top())
          S.pop()
        until T.top() = w
        return T.elements()
  S.pop()
```

BREADTH-FIRST SEARCH

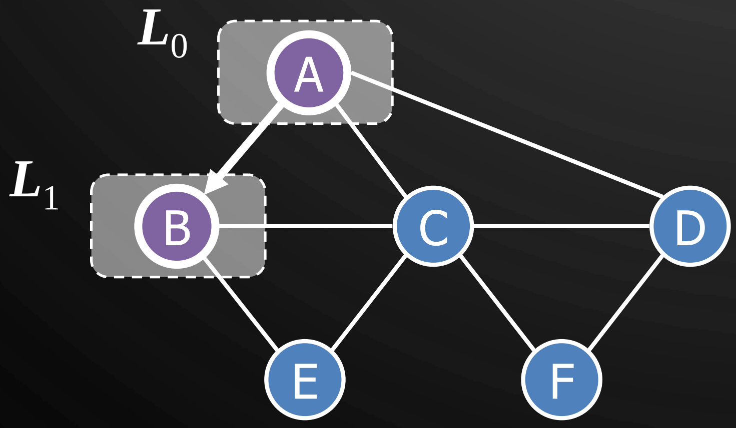


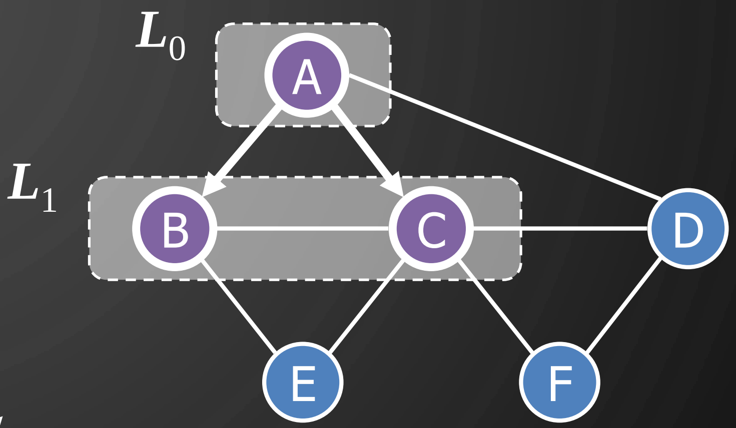
BREADTH-FIRST SEARCH

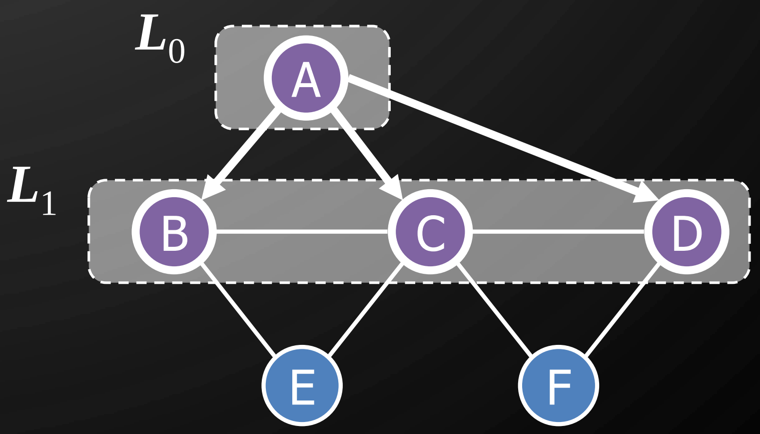
- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and 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

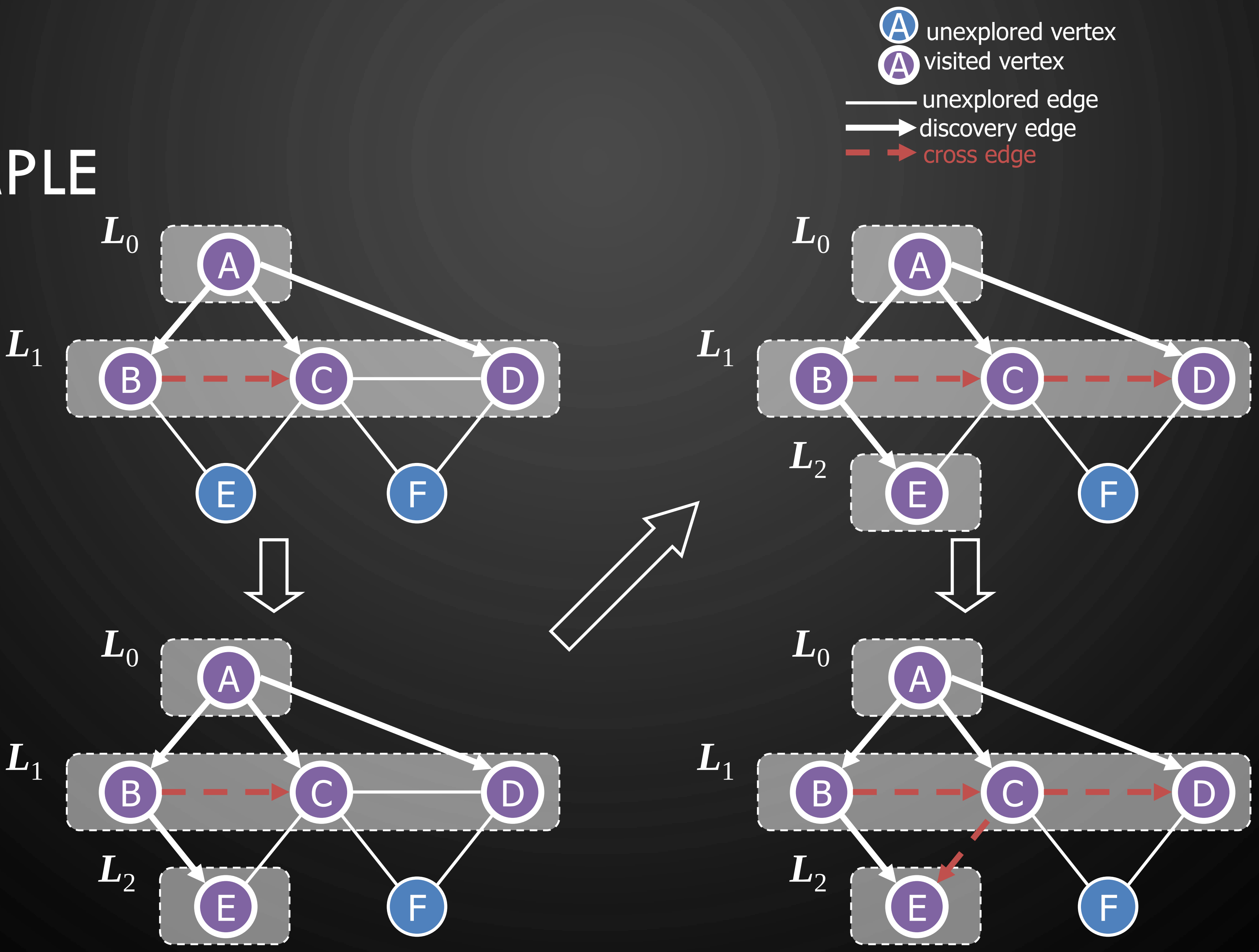
A unexplored vertex
A visited vertex
unexplored edge
discovery edge
cross edge



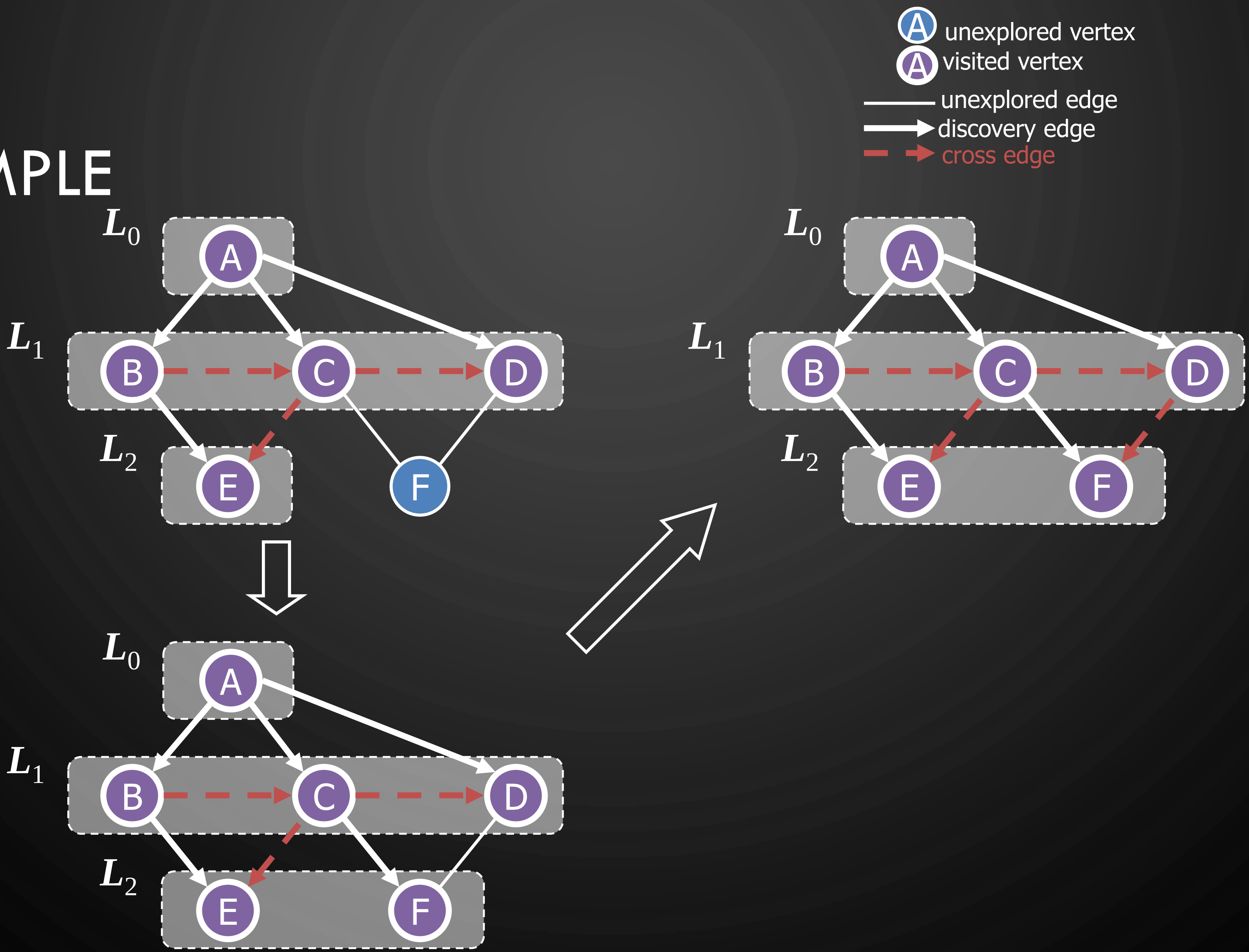




EXAMPLE


A unexplored vertex
A visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
L_2
A
B
C
D
E
F

EXAMPLE



A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
L_2
A
B
C
D
E
F

BFS ALGORITHM

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

Algorithm $\mathrm{BFS}(G)$
Input: Graph G
Output: Labeling of the edges and partition of the vertices of G

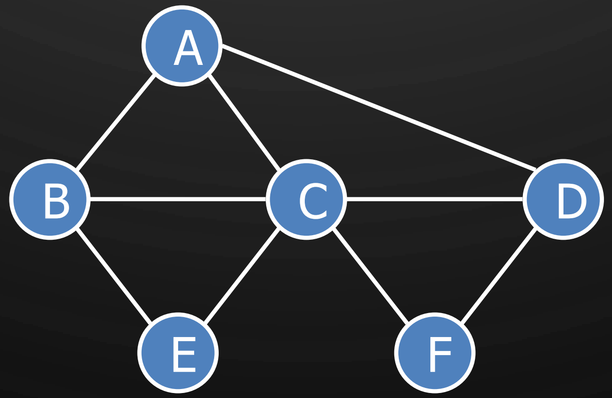
for each $v \in G.\mathrm{vertices}()$ **do**
  $v.\mathrm{setLabel}(UNEXPLORED)$
for each $e \in G.\mathrm{edges}()$ **do**
  $e.\mathrm{setLabel}(UNEXPLORED)$
for each $v \in G.\mathrm{vertices}()$ **do**
  if $v.\mathrm{getLabel}() = UNEXPLORED$
    $\mathrm{BFS}(G, v)$

Algorithm $\mathrm{BFS}(G, s)$

$L_0 \leftarrow \{s\}$
$s.\mathrm{setLabel}(VISITED)$
$i \leftarrow 0$
while $\neg L_i.\mathrm{empty}()$ **do**
  $L_{i+1} \leftarrow \emptyset$
  for each $v \in L_i$ **do**
    for each $e \in v.\mathrm{incidentEdges}()$ **do**
      if $e.\mathrm{getLabel}() = UNEXPLORED$
        $w \leftarrow e.\mathrm{opposite}(v)$
        if $w.\mathrm{getLabel}() = UNEXPLORED$
          $e.\mathrm{setLabel}(DISCOVERY)$
          $w.\mathrm{setLabel}(VISITED)$
          $L_{i+1} \leftarrow L_{i+1} \cup \{w\}$
        else
          $e.\mathrm{setLabel}(CROSS)$
  $i \leftarrow i + 1$

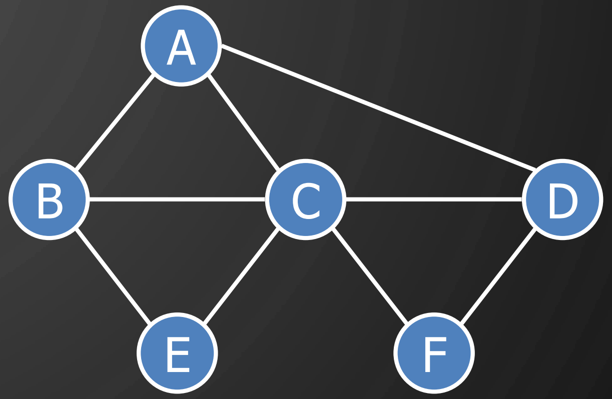
EXERCISE
BFS ALGORITHM

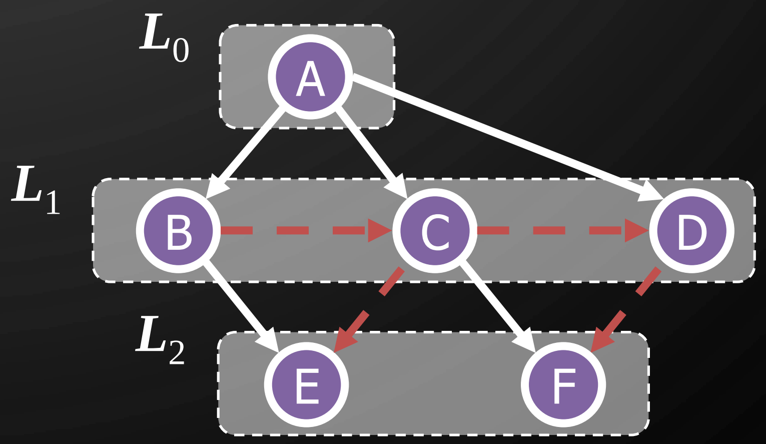
- Perform BFS of the following graph, start from vertex A
  - Assume adjacent edges are processed in alphabetical order
  - Number vertices in the order they are visited and note the level they are in
  - Label edges as discovery or cross edges



PROPERTIES

- Notation
  - G_s: connected component of s
- Property 1
  - $\mathrm{BFS}(G, s)$ visits all the vertices and edges of G_s
- Property 2
  - The discovery edges labeled by $\mathrm{BFS}(G, s)$ 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 $\mathrm{incidentEdges}(\,)$ 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 $\Sigma_v \deg(v) = 2m$

ANALYSIS OF BFS

- 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 each $v \in G.\text{vertices}()$ **do** $O(n)$
  $v.\text{setLabel}(UNEXPLORED)$
for each $e \in G.\text{edges}()$ **do** $O(m)$
  $e.\text{setLabel}(UNEXPLORED)$
for each $v \in G.\text{vertices}()$ **do** $O(n+m)$
  if $v.\text{getLabel}() = UNEXPLORED$
    $\text{BFS}(G, v)$

Algorithm BFS(G, s)

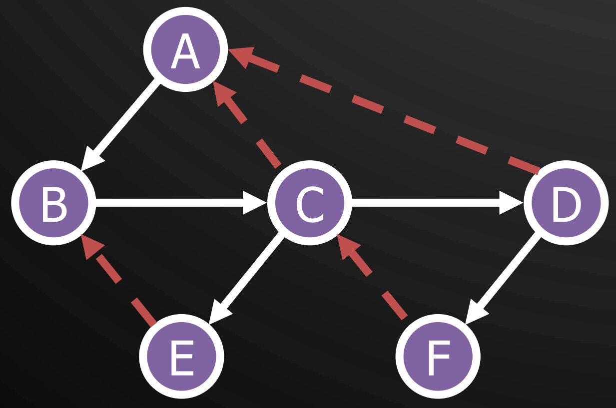
$L_0 \leftarrow \{s\}$
$s.\text{setLabel}(VISITED)$
$i \leftarrow 0$
while $\neg L_i.\text{empty}()$ **do**
  $L_{i+1} \leftarrow \emptyset$
  for each $v \in L_i$ **do** $O(deg(v))$
    for each $e \in v.\text{incidentEdges}()$ **do**
      if $e.\text{getLabel}() = UNEXPLORED$
        $w \leftarrow e.\text{opposite}(v)$
        if $w.\text{getLabel}() = UNEXPLORED$
          $e.\text{setLabel}(DISCOVERY)$
          $w.\text{setLabel}(VISITED)$
          $L_{i+1} \leftarrow L_{i+1} \cup \{w\}$
        else
          $e.\text{setLabel}(CROSS)$
  $i \leftarrow i + 1$

APPLICATIONS

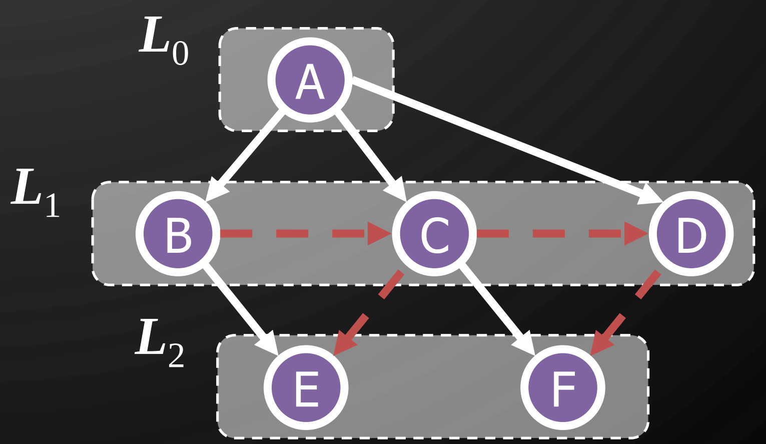
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n+m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find 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	√	



DFS

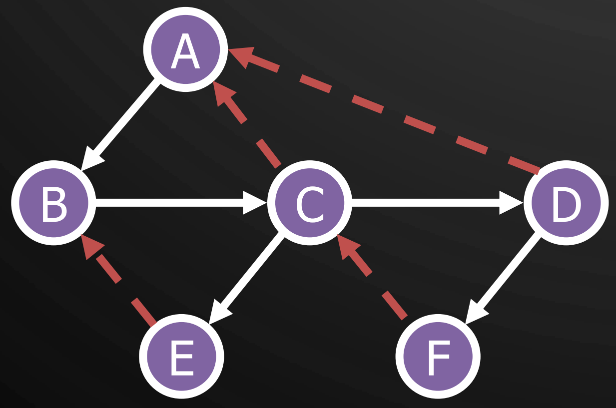


BFS

DFS VS. BFS

Back edge (v, w)

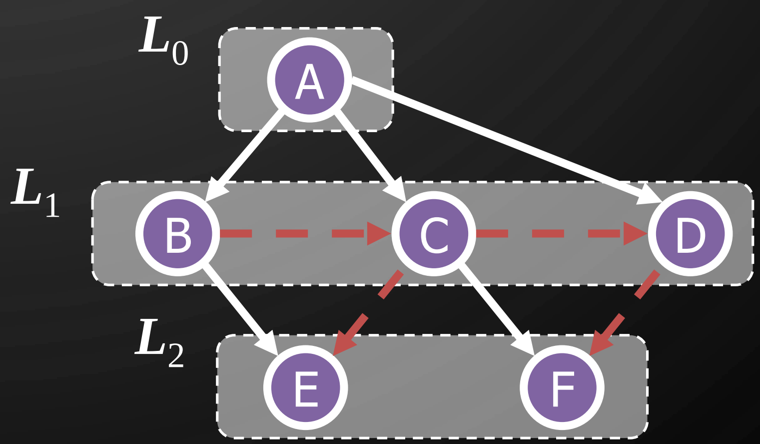
- w is an ancestor of v in the tree of discovery edges



DFS

Cross edge (v, w)

- w is in the same level as v or in the next level in the tree of discovery edges



BFS