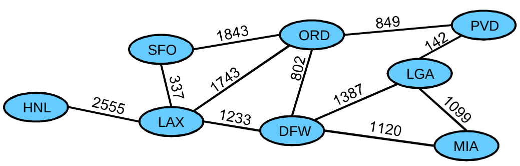
Non-Linear Data Structures

Graphs

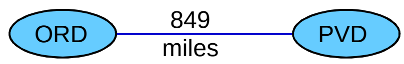
Graphs
The Graph ADT Definition

- A graph **G** is a pair (**V**, **E**), representing many-to-many relationships between pairs of objects, where:
  - **V** is a ***set*** of nodes representing the objects, called **vertices**
  - **E** is a ***collection*** of pairs of vertices from **V** representing their connection, called arcs or **edges**
  - Vertices and edges have **Entries** that store their respective **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.



Graphs
Edge Types

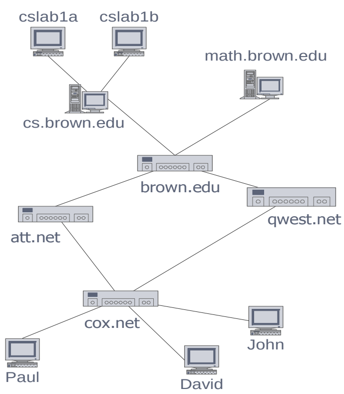
- Directed edge
  - Has **ordered** pair of vertices (**u**,**v**)
  - First vertex **u** is the origin
  - Second vertex **v** is the destination
  - e.g., a flight.
- Undirected edge
  - Has **unordered** pair of vertices (**u**,**v**)
  - e.g., a flight route
- Directed graph
  - All the edges are **directed**
  - e.g., route network
- Undirected graph
  - All the edges are **undirected**
  - e.g., flight network



- Graphical Representation
  - **Vertices** → *Oval* or *Rectangle* shapes
  - **Edges** → *Curves* or *Lines* connecting pairs of vertex shapes
  - **Directed** edge → *Arrow* at destn.
  - **Undirected** edge → No arrows.

Graphs
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



Non-Linear Data Structures

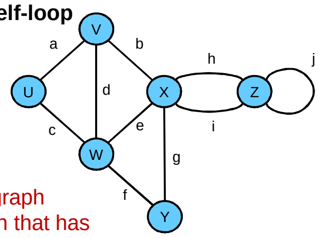
The Undirected Graph

Graphs
Terminology

- Any two vertices joined by an edge are called the end vertices (or endpoints) of that edge.
- An edge is incident on a vertex if the vertex is one of the edge's endpoints.
- Two vertices **u** and **v** are adjacent if they are endpoints of an existing edge.
- The degree of a vertex, deg(**v**), is the number of incident edges of **v**.
- Any two undirected edges with the same origin and the same destination are called parallel edges.
- An edge that connects a vertex to itself is called a self-loop.

Examples:

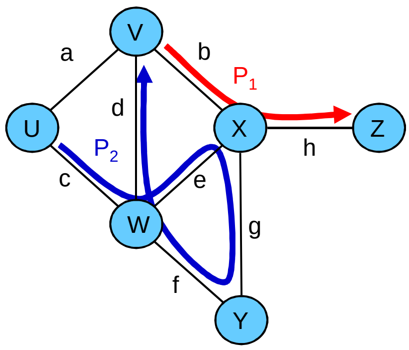
- **U** and **V** are the **endpoints** of **a**
- Edges: **a**, **d**, and **b** are **incident** on **V**
- **U** and **V** are **adjacent**
- **X** has **degree 5**
- **h** and **i** are **parallel** edges
- **j** is a **self-loop**



A **simple** graph is the graph that has no parallel edges or self-loops

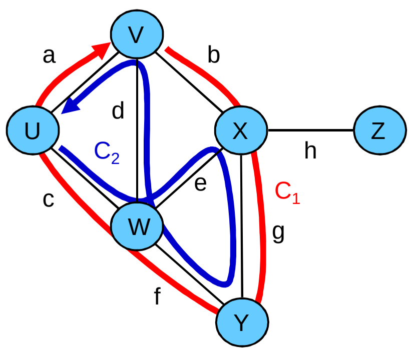
Graphs
Terminology (cont.)

- Path
  - A sequence of alternating vertices and edges
  - It begins with a vertex
  - It ends with a vertex
  - Each edge is preceded and followed by its endpoints
- Simple path
  - A 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



Graphs
Terminology (cont.)

- Cycle
  - A circular sequence of alternating vertices and edges
  - Each edge is preceded and followed by its endpoints
- Simple cycle
  - A cycle such that all its vertices and edges are distinct except the first and last vertices
- 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



Graphs
Properties

Property 1

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

Proof: Each edge is counted twice

Property 2

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

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

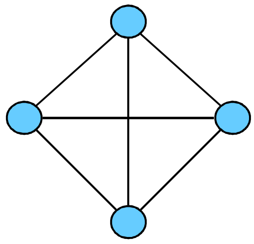
Proof: Each vertex has degree at most $(n-1)$

Notation

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

Example

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



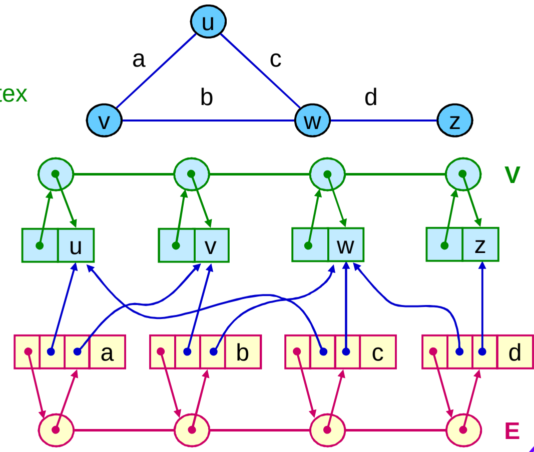
Graphs
Main Operations of the Graph ADT

Note that, vertices and edges have Entries that store elements

- Accessor methods:
  - numVertices(): Return no. vertices
  - numEdges(): Return no. edges
  - degree(v): Returns the number of incident edges to vertex v
  - getEdge(u, v): Return the edge from vertex u to vertex v, or null if none exist
  - endVertices(e): Return an array storing the endvertices of edge e
  - opposite(v, e): Return the vertex opposite of v on edge e
  - incidentEdges(v): all edges incident to v
  - vertices(): all vertices in the graph
  - edges(): all edges in the graph
- Mutator methods:
  - insertVertex(x): Insert and return a new vertex storing element x
  - insertEdge(v, w, x): Insert and return a new undirected edge (v,w) storing element x
  - removeVertex(v): Remove vertex v, its incident edges; and return its element
  - removeEdge(e): remove edge e and return its element
  - replace(v, x): Replace element at vertex v with x
  - replace(e, x): Replace element at edge e with x

Graph Representation as 1. Edge List Structure

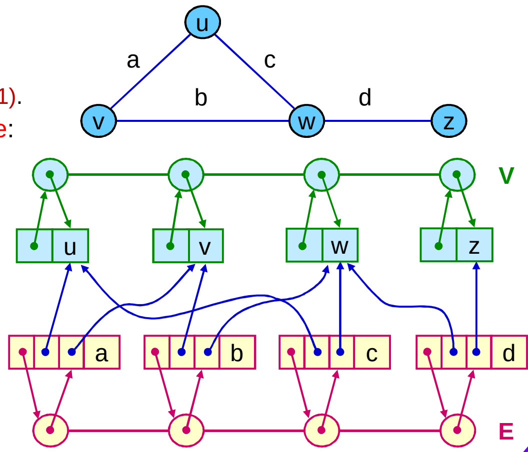
- The vertex object for **v** has:
  - A reference to its element **o**
  - A reference to its entry in vertex sequence **V**.
- The edge object for **e** has:
  - A reference to its element **o**
  - Its origin vertex object
  - Its destination vertex object
  - A reference to its entry in edge sequence **E**.
- The vertex sequence **V** is:
  - A sequence of vertex objects
- The edge sequence **E** is:
  - A sequence of edge objects



Note: **V** and **E** may be implemented as lists or as dictionaries.

Graph Method Implementation Using Edge List Structure

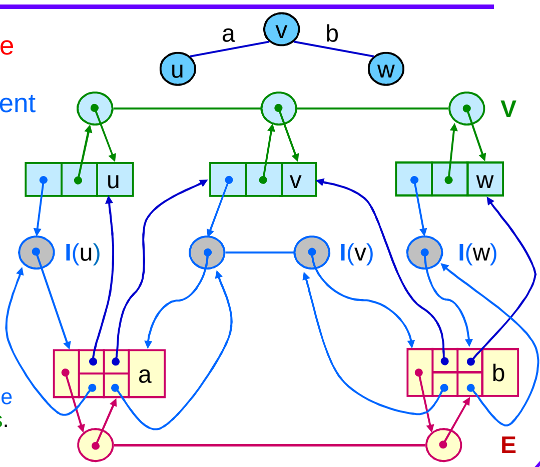
- vertices(): Call **V**.iterator(). O(n).
- edges(): Call **E**.iterator(). O(m).
- numVertices(); numEdges(): O(1).
- incidentEdges; getEdge; degree:
  - We have to inspect all edges. O(m).
- insertVertex; insertEdge:
  - Use the insertion method for the doubly linked list. O(1).
- removeVertex(**v**):
  - Find and remove all incident edges on **v** from **E**. O(m).
  - Remove vertex **v** from **V**. O(1).
- removeEdge(**e**):
  - Remove edge **e** from **E**. O(1).



Suppose **V** and **E** are implemented as doubly linked lists.

Graph Representation as 2. Adjacency List Structure

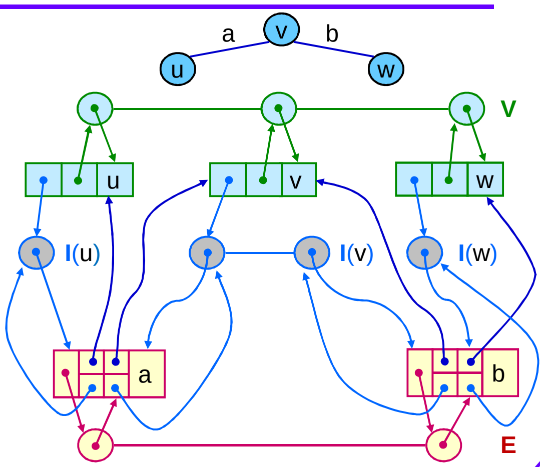
- Extends the edge list structure with **extra** information that support direct access to incident edges and adjacent vertices:
  1. A reference to incidence sequence **I** stored in each vertex object **v**:
     - Elements of **I** hold references to the edges incident on **v**.
  2. Augmented edge objects:
     - Each edge object **e** holds additional references to the associated entries in incidence sequences of **e**'s end vertices.



Note: **V**, **E** and the **I**s may be implemented as doubly linked lists.

Graph Method Implementation Using Adjacency List Structure

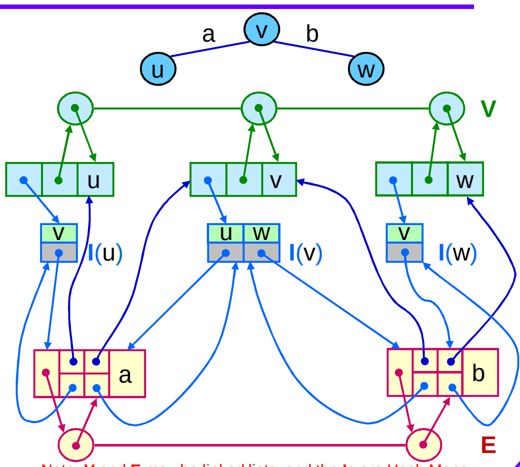
- vertices; edges; removeEdge; insertVertex; insertEdge; numVertices; numEdges:
  - Same as before.
- incidentEdges(**v**):
  - Get all elements of **I**(**v**)
  - O(deg(**v**)).
- getEdge(**u**,**v**):
  - Inspect **I**(**u**) or **I**(**v**); the shorter
  - O(min(deg(**u**),deg(**v**))).
- removeVertex(**v**):
  - Use **I**(**v**) to remove all incident edges on **v** from **E**. O(deg(**v**))
  - Remove vertex **v** from **V**. O(1).
- degree(**v**): O(1).



Note: **V**, **E** and the **I**s are implemented as doubly linked lists.

Graph Representation as 3. Adjacency Map Structure

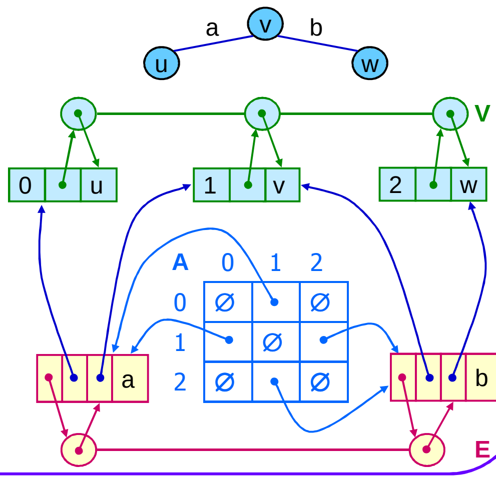
- Replaces the un-ordered linked list representation of the incidence sequence **I(v)** with a hash-based map **I(v)** for each vertex **v**, where:
  - The opposite endpoint of each incident edge serves as a **key** in the map, and a reference to that edge serves as the **value**.
  - The space usage for an adjacency map remains O(n+m).
  - The advantage over the adjacency list, is that method getEdge(**u**, **v**) can take expected O(1) time, by searching for vertex **u** as a key in **I**(**v**).



Note: **V** and **E** may be linked lists, and the **I**s are Hash Maps

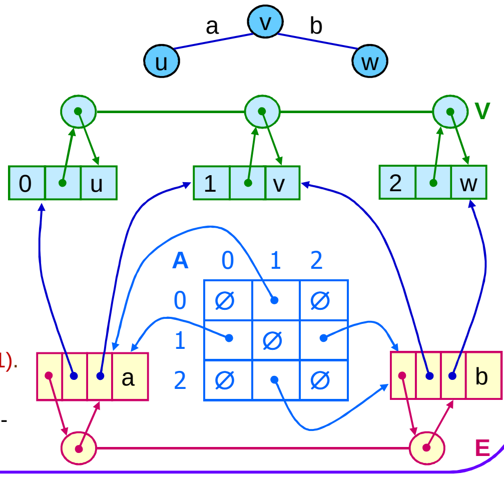
Graph Representation as 4. Adjacency Matrix Structure

- Extends the edge list structure by augmenting the edge sequence with a matrix **A** that allows fast determination of adjacencies between pairs of vertices:
  1. Augmented vertex objects:
     - Each vertex object **v** holds a distinct additional integer key **i** in the range 0, 1, …, n-1; called the index of vertex **v**.
  2. A 2D-adjacency **n** x **n** array **A**:
     - Each cell holds a reference to an edge object for adjacent vertices
     - Null for non-adjacent vertices.
     - Space needed is O($\mathbf{n}^2$).



Graph Method Implementation Using Adjacency Matrix Structure

- vertices; edges; insertEdge; removeEdge; numVertices; numEdges: Same as before.
- incidentEdges(**v**):
  - Examine a row or a column of **A**,
  - Get incident edges. O(n+deg(**v**)).
- getEdge(**u**,**v**):
  - Find indices i, j of **u**, **v** from **V**
  - Test if **A**[i,j] is **null** or not. O(1).
- insertVertex; removeVertex(**v**):
  - Insert **v** in **V**. O(1). Or, find and remove all incident edges on **v** from **E**. O(n). Then, remove **v** from **V**. O(1).
  - Create a new array **A**. O(n^2).
- degree(**v**): Find index **i** of **v**; Count non-null values in row **i** of **A**. O(n).



Graphs
Operation Performance

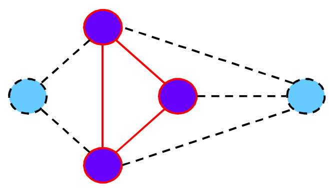
▪ **n** vertices, **m** edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Map	Adjacency Matrix
Space	O(n + m)	O(n + m)	O(n + m)	O(n^2)
vertices()	O(n)	O(n)	O(n)	O(n)
edges()	O(m)	O(m)	O(m)	O(m)
endVertices(**e**), opposite(**v**, **e**), replace(**v**, **o**), replace(**e**, **o**), numVertices(), numEdges()	O(1)	O(1)	O(1)	O(1)
degree(**v**)	O(m)	O(1)	O(1)	O(n)
incidentEdges(**v**)	O(m)	O(deg(**v**))	O(deg(**v**))	O(n + deg(**v**))
getEdge(**v**, **w**)	O(m)	O(min(deg(**v**), deg(**w**)))	O(1) exp.	O(1)
insertVertex(**o**)	O(1)	O(1)	O(1)	O(n^2)
insertEdge(**v**, **w**, **o**)	O(1)	O(1)	O(1) exp.	O(1)
removeVertex(**v**)	O(m)	O(deg(**v**))	O(deg(**v**))	O(n^2)
removeEdge(**e**)	O(1)	O(1)	O(1) exp.	O(1)

Graph Traversals

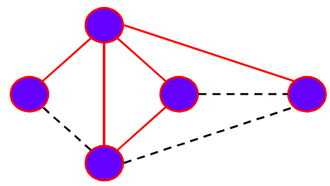
Depth-First Search

Definitions: Subgraphs

- **Traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.
- 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**.



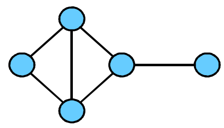
Subgraph



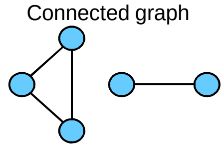
Spanning subgraph

Definitions:
Connectivity

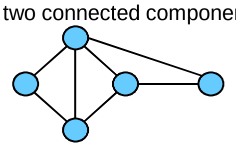
- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph **G** is a maximal connected subgraph of **G**
- A graph **G** is biconnected if it contains no vertex whose removal would divide **G** into two or more connected components



Connected graph



Non-connected graph with two connected components

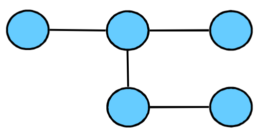


Bi-connected graph

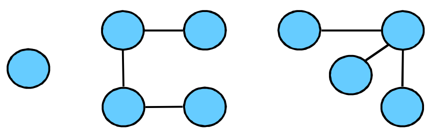
Definitions:
Trees and Forests

- A 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



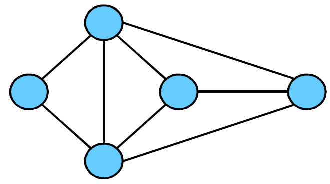
Tree



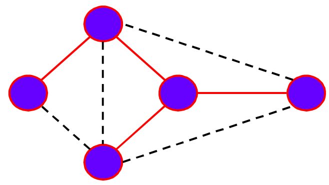
Forest

Definitions:
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

Benefits of
Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph **G**
  1. Visits all the vertices and edges of **G**
  2. Determines whether **G** is connected
  3. Computes the connected components of **G**
  4. Computes a spanning forest of **G**
- DFS on a graph with $\mathbf{n}$ vertices and $\mathbf{m}$ edges takes $O(\mathbf{n}+\mathbf{m})$ time
- DFS can be further extended to solve other graph problems:
  5. Finds and reports a path between two given vertices
  6. Finds a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm

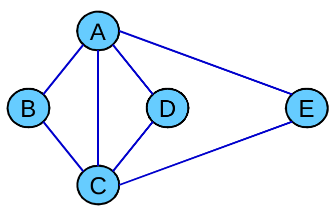
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges. Pseudo code:

```
Algorithm DFS(G)
    Input    graph G
    Output   labeling of the edges of G as
             discovery edges and back edges

    for all  u ∈ G.vertices()
        setLabel(u, UNEXPLORED)

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

    for all  v ∈ G.vertices()
        if  getLabel(v) = UNEXPLORED
            DFS(G, v)
```

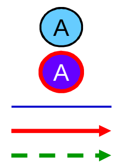
```
Algorithm DFS(G, v)        Recursive Algorithm
    Input    graph G and a start vertex v of G
    Output   labeling of the edges of G  in the
             connected component of v  as
             discovery edges and back edges

    setLabel(v, VISITED)

    for all  e ∈ G.incidentEdges(v)
        if  getLabel(e) = UNEXPLORED
            w ← G.opposite(v, e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)
```

Example: DFS(G, A)

```
DFS(G, v)        // called at level 0 with v = A.

setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
        w ← G.opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else
            setLabel(e, BACK)
```



graph G

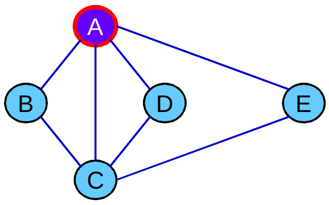


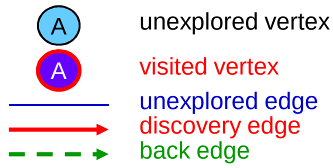
unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge

Example: DFS(G, A) ... (cont.)

DFS(G, v) // called at level 0 with v = A.

setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
if getLabel(e) = UNEXPLORED
w ← G.opposite(v, e)
if getLabel(w) = UNEXPLORED
setLabel(e, DISCOVERY)
DFS(G, w)
else
setLabel(e, BACK)

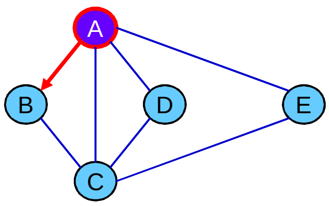


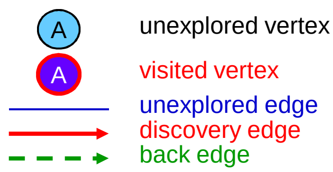


Example: DFS(G, A) ... (cont.)

DFS(G, v) // called at level 0 with v = A.

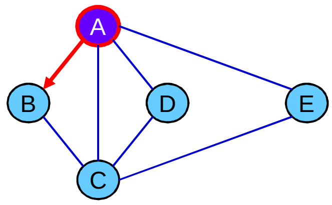
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
if getLabel(e) = UNEXPLORED
w ← G.opposite(v, e)
if getLabel(w) = UNEXPLORED
setLabel(e, DISCOVERY)
DFS(G, w)
else
setLabel(e, BACK)





Example: DFS(G, A) ... (cont.)

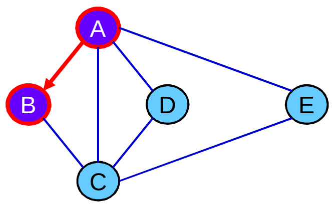
```
DFS(G, v)        // called at level 0 with v = A.

setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
        w ← G.opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else
            setLabel(e, BACK)
```



A unexplored vertex
A visited vertex
unexplored edge
discovery edge
back edge

Example: DFS(G, A) ... (cont.)

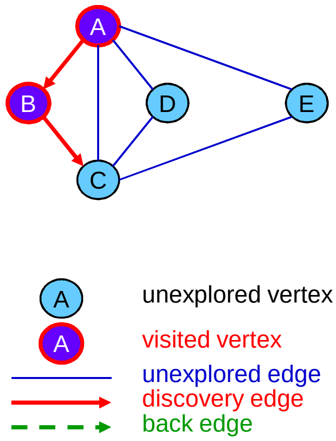
```
DFS(G, v)        // called at level 1 with v = B.

setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
        w ← G.opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else
            setLabel(e, BACK)
```



A unexplored vertex
A visited vertex
unexplored edge
discovery edge
back edge

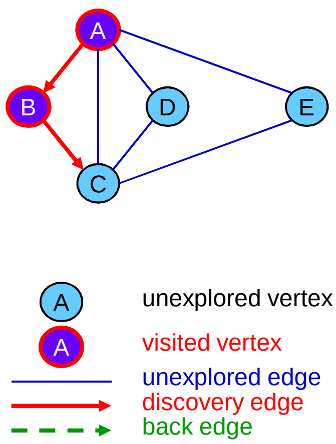
Example: DFS(G, A) ... (cont.)

```
DFS(G, v)        // called at level 1 with v = B.

setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)
     if  getLabel(e) = UNEXPLORED
          w ← G.opposite(v, e)
          if getLabel(w) = UNEXPLORED
               setLabel(e, DISCOVERY)
               DFS(G, w)
          else
               setLabel(e, BACK)
```



Example: DFS(G, A) ... (cont.)

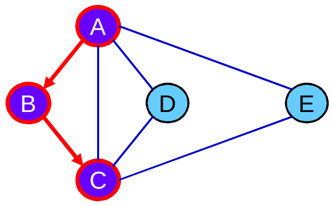
```
DFS(G, v)        // called at level 1 with v = B.

setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)
     if  getLabel(e) = UNEXPLORED
          w ← G.opposite(v, e)
          if getLabel(w) = UNEXPLORED
               setLabel(e, DISCOVERY)
               DFS(G, w)
          else
               setLabel(e, BACK)
```

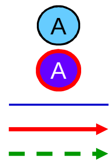


Example: DFS(G, A) ... (cont.)

DFS(G, v) // called at level 2 with v = C.

```
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← G.opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else
            setLabel(e, BACK)
```

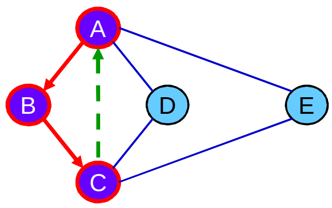


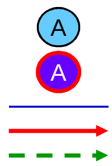


Example: DFS(G, A) ... (cont.)

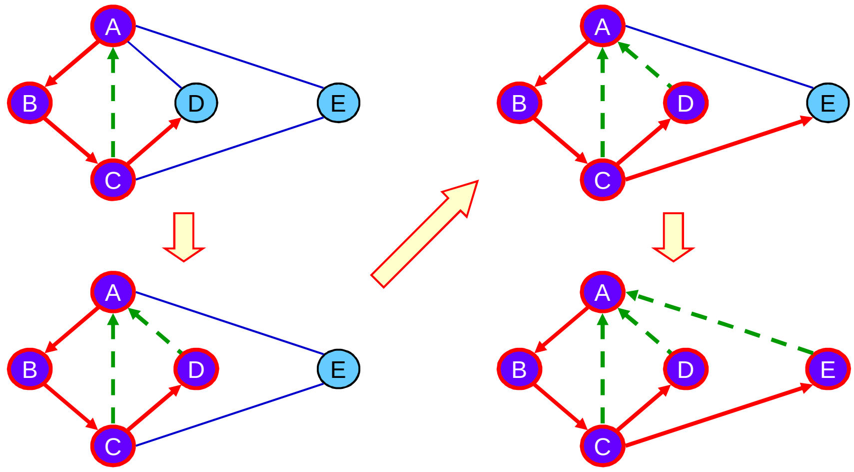
DFS(G, v) // called at level 2 with v = C.

```
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← G.opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else
            setLabel(e, BACK)
```

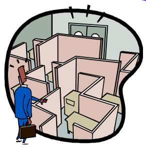




Example: DFS(G, A) ... (cont.)

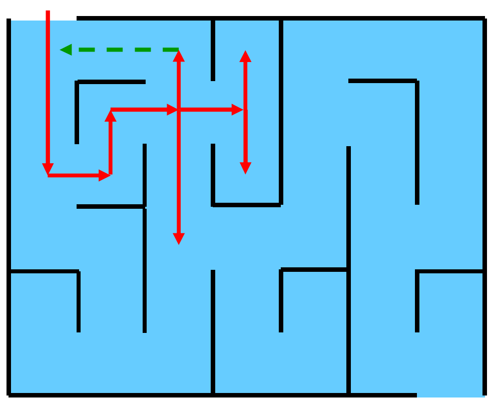


DFS and Maze Traversal



The DFS algorithm is similar to a classic strategy for exploring a maze:

- We mark each intersection, corner and dead-end (vertex) as visited
- We mark each corridor (edge) as traversed
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



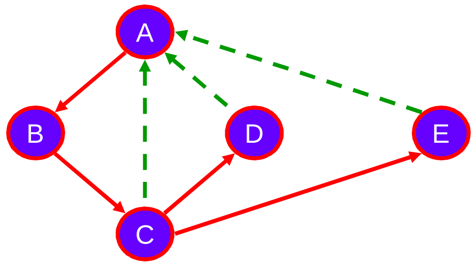
Properties of DFS

Property 1

$\mathbf{DFS}(\mathbf{G}, \mathbf{v})$ visits all the vertices and edges in the connected component of $\mathbf{v}$

Property 2

The discovery edges labeled by $\mathbf{DFS}(\mathbf{G}, \mathbf{v})$ form a spanning tree of the connected component of $\mathbf{v}$



Analysis of DFS ... $O(n + m)$

- Setting or getting a vertex or an edge label takes $\mathbf{O}(1)$ 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 $\mathbf{O}(\mathbf{n} + \mathbf{m})$ time provided the graph is represented by the adjacency list structure. [or $\mathbf{O}(\mathbf{n}^2)$ if adjacency matrix is used]
  - Recall that $\Sigma_v \deg(\mathbf{v}) = 2\mathbf{m}$

Path Finding Algorithm

- We can specialize the DFS algorithm to find a path between two given vertices **u** and **z** using the template method pattern (see sec.7.3.7):
- Call **DFS(G, u)** with **u** as the start vertex
- 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, return the path, which is the contents of the stack.

```
Algorithm pathDFS(G, v, z) Recursive Algorithm
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()    //A list of path elements
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← G.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 Algorithm

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern (see sec.7.3.7):
- 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 ← G.opposite(v, e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                cycleDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()  //Cycle elements
    S.pop(v)
```

Graph Traversals

Breadth-First Search

Benefits of Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph **G**
  1. Visits all the vertices and edges of **G**
  2. Determines whether **G** is connected
  3. Computes the connected components of **G**
  4. Computes a spanning forest of **G**
- BFS on a graph with $\mathbf{n}$ vertices and $\mathbf{m}$ edges takes $O(\mathbf{n}+\mathbf{m})$ time
- BFS can be further extended to solve other graph problems
  5. Finds and reports a path with the minimum number of edges between two given vertices
  6. Finds a simple cycle, if there is one.

BFS Algorithm

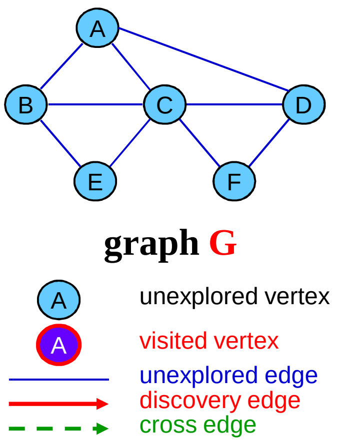
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges. Pseudo code:

```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges and
           partition of the vertices of G

    for all u ∈ G.vertices()
        setLabel(u, UNEXPLORED)

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

    for all v ∈ G.vertices()
        if getLabel(v) = UNEXPLORED
            BFS(G, v)
```

```
Algorithm BFS(G, s)     Iterative Algorithm
    Input   graph G and a start vertex s of G
    Output  label the edges of G in the connected
            component of s as discovery & cross

    L0 ← new empty list
    L0.addLast(s)
    setLabel(s, VISITED)
    for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
        Li+1 ← new empty list
        for all v ∈ Li.elements()
            for all e ∈ G.incidentEdges(v)
                if getLabel(e) = UNEXPLORED
                    w ← G.opposite(v, e)
                    if getLabel(w) = UNEXPLORED
                        setLabel(e, DISCOVERY)
                        setLabel(w, VISITED)
                        Li+1.addLast(w)
                else
                    setLabel(e, CROSS)
```

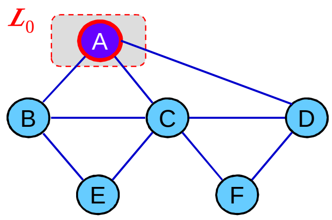
Example: BFS(G, A)

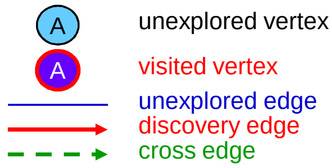
```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
                w ← G.opposite(v, e)
                if getLabel(w) = UNEXPLORED
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    Li+1.addLast(w)
            else
                setLabel(e, CROSS)
```



Example: BFS(G, A) … (cont.)

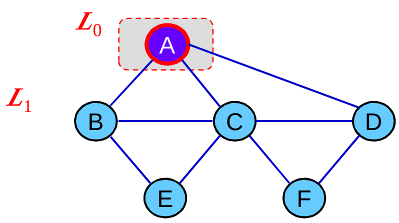
```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```

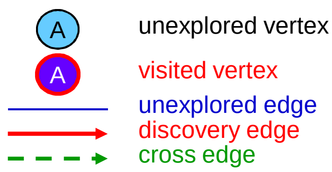




Example: BFS(G, A) … (cont.)

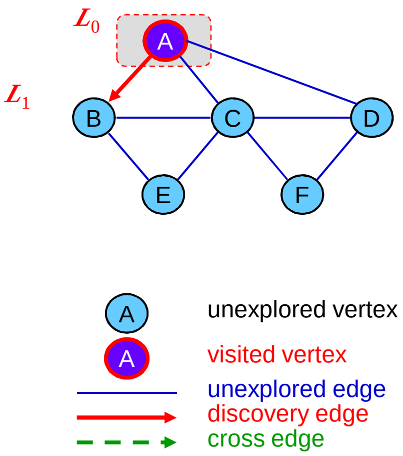
```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```





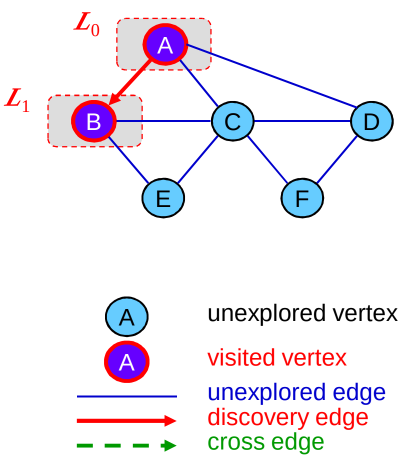
Example: BFS(G, A) ... (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



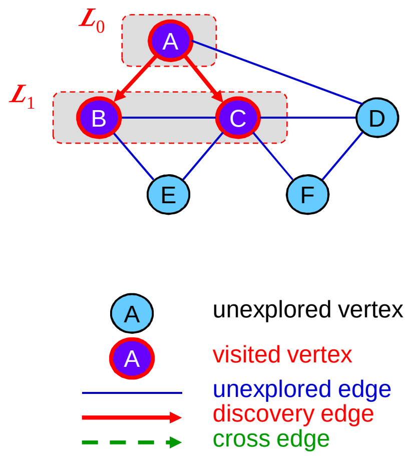
Example: BFS(G, A) ... (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



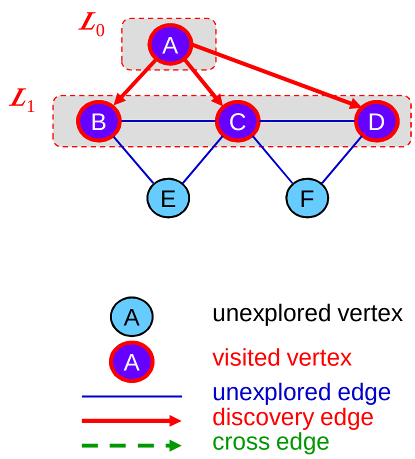
Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



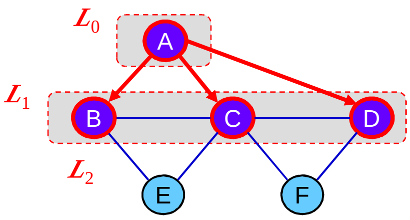
Example: BFS(G, A) … (cont.)

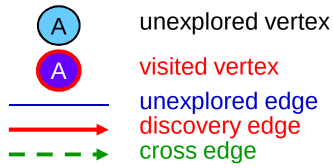
```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



Example: BFS(G, A) ... (cont.)

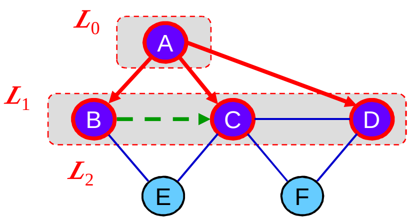
```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```

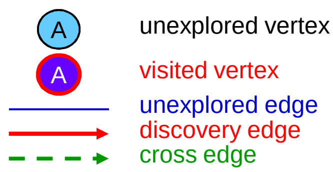




Example: BFS(G, A) ... (cont.)

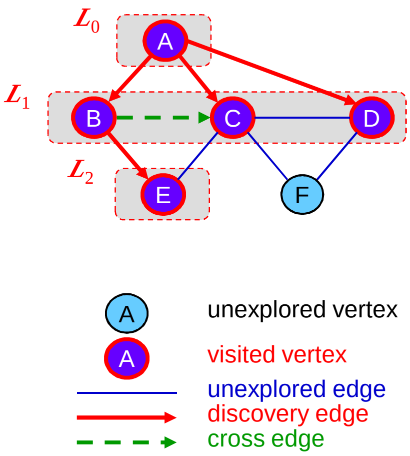
```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```





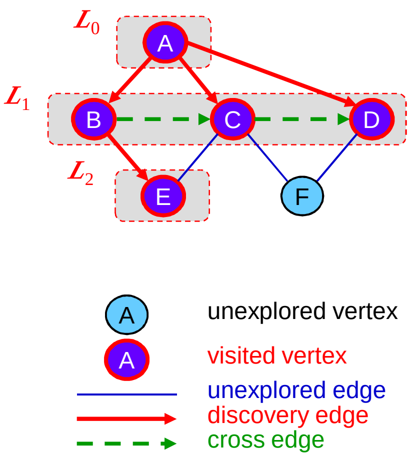
Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



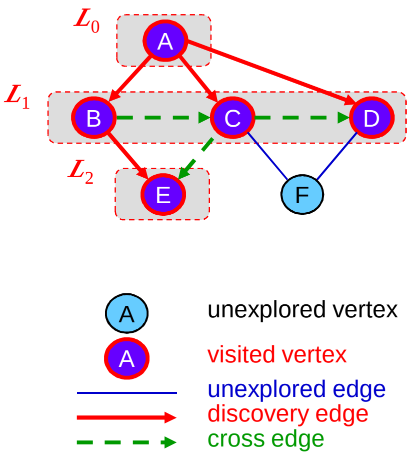
Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



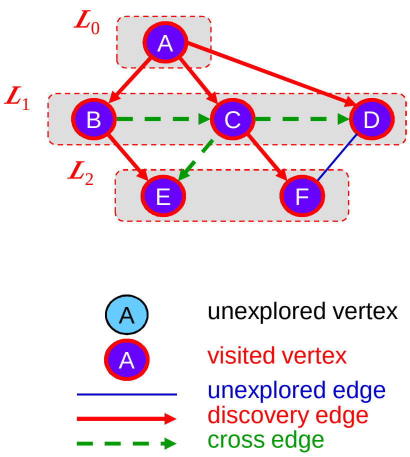
Example: BFS(G, A) ... (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



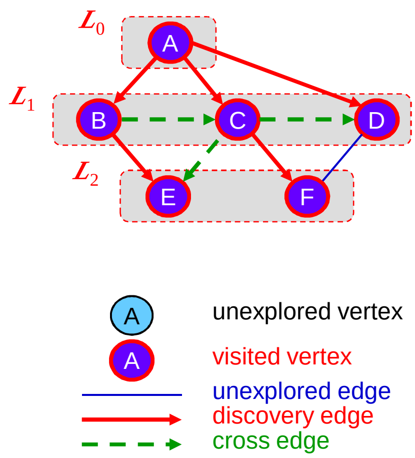
Example: BFS(G, A) ... (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



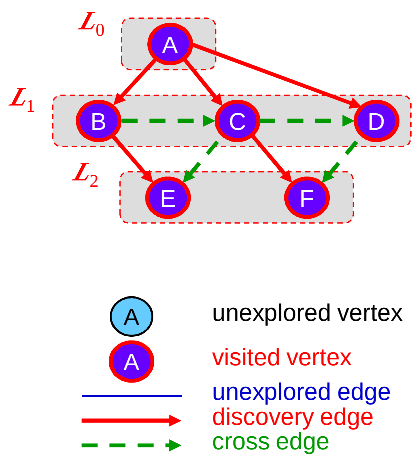
Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



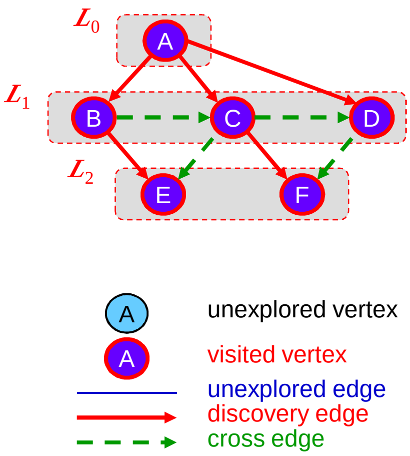
Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
              w ← G.opposite(v, e)
              if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
              else
                  setLabel(e, CROSS)
```



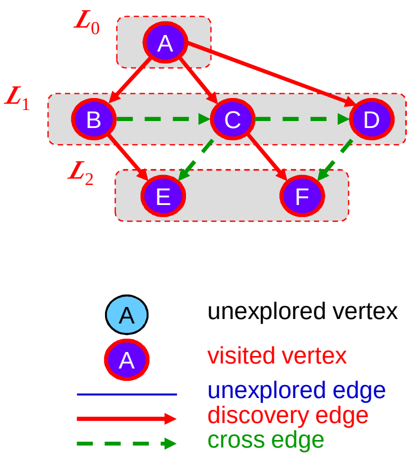
Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
                w ← G.opposite(v, e)
                if getLabel(w) = UNEXPLORED
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    Li+1.addLast(w)
                else
                    setLabel(e, CROSS)
```



Example: BFS(G, A) … (cont.)

```
L0 ← new empty list
L0.addLast(s)
setLabel(s, VISITED)
for (i ← 0 ; ¬Li.isEmpty(); i ← i +1)
    Li+1 ← new empty list
    for all  v ∈ Li.elements()
        for all  e ∈ G.incidentEdges(v)
            if  getLabel(e) = UNEXPLORED
                w ← G.opposite(v, e)
                if getLabel(w) = UNEXPLORED
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    Li+1.addLast(w)
                else
                    setLabel(e, CROSS)
```

Properties of BFS

Notation

G_s: is a connected component of s

Property 1

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

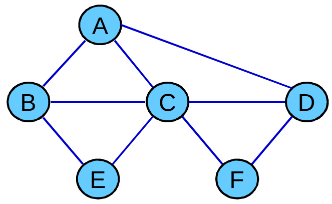
Property 2

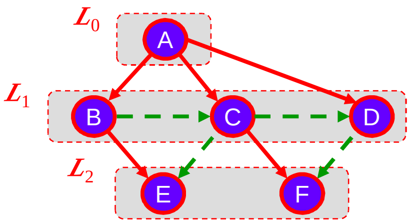
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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



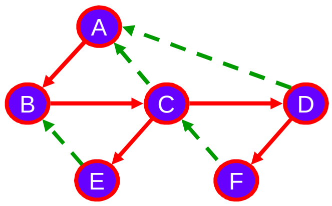


Analysis of BFS … $O(n + m)$

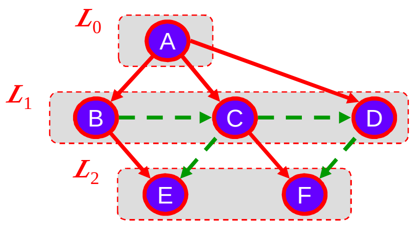
- Setting or getting a vertex or an edge label takes $O(1)$ time
- Each vertex is labeled twice
  - Once as UNEXPLORED
  - Once as VISITED
- Each edge is labeled twice
  - Once as UNEXPLORED
  - Once as DISCOVERY or CROSS
- Each vertex is inserted once into a list L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure. [or $O(n^2)$ if adjacency matrix is used]
  - Recall that $\Sigma_v deg(v) = 2m$

DFS vs. BFS

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



DFS



BFS

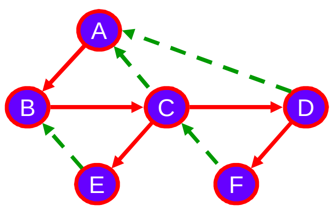
DFS vs. BFS (cont.)

Back edge (v, w)

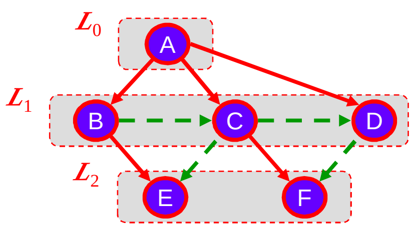
- **w** is an ancestor of **v** in the tree of discovery edges

Cross edge (v, w)

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



DFS



BFS