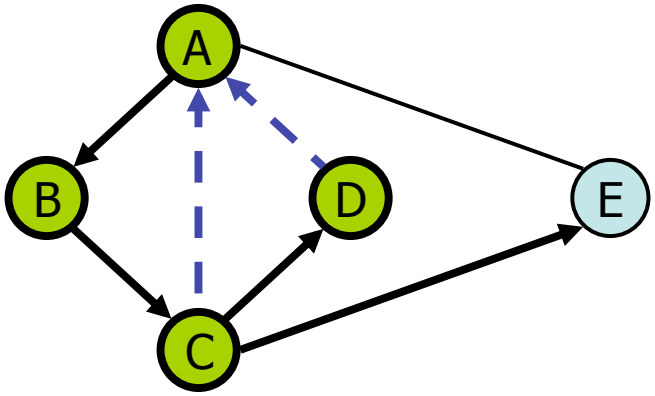
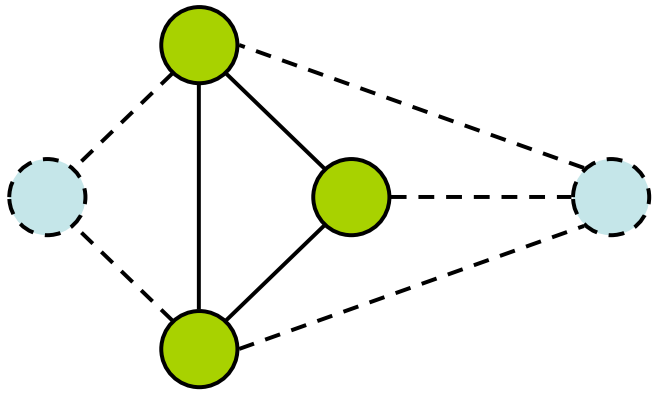
# Graph Traversals

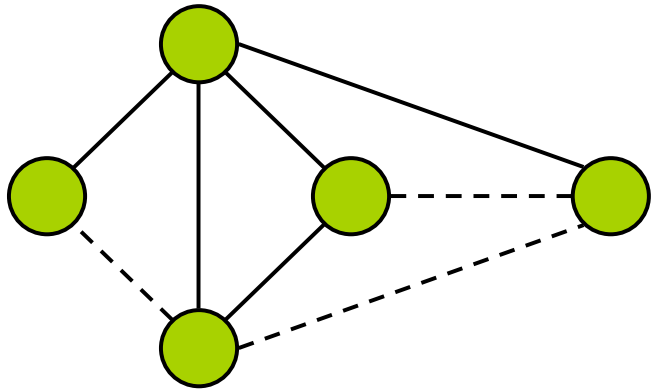


# Subgraphs

- 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



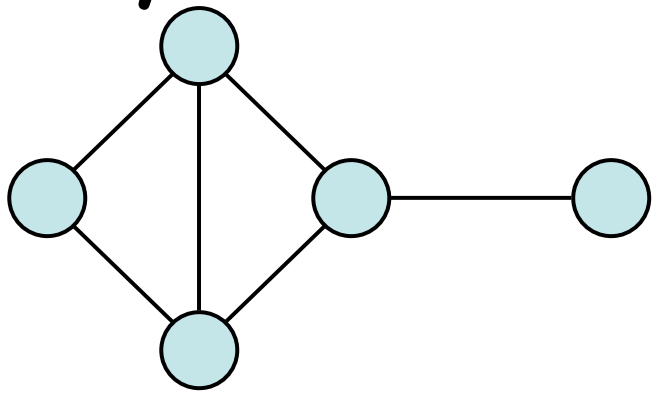
Subgraph



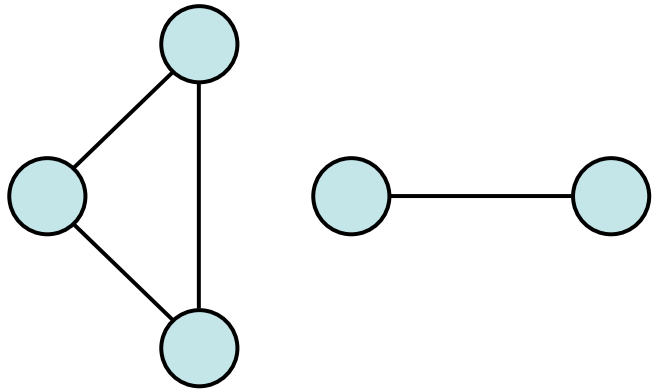
Spanning subgraph

# 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



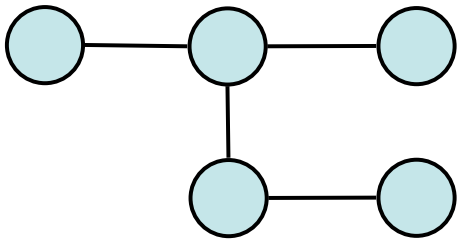
Connected graph



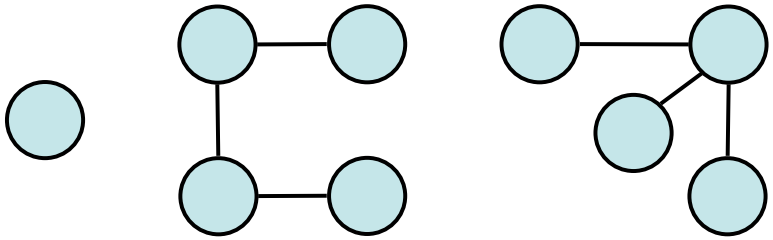
Non connected graph with two connected components

# Trees and Forests

- A tree is an undirected graph T such that
  - T is connected
  - T has no cycles
- A forest is an undirected graph without cycles (a collection of trees).
- The connected components of a forest are trees



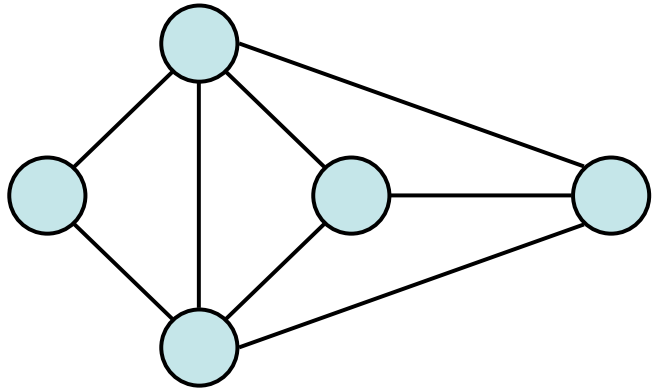
Tree



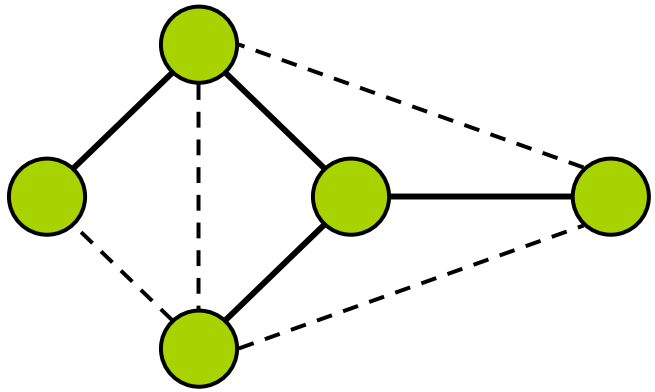
Forest

# 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



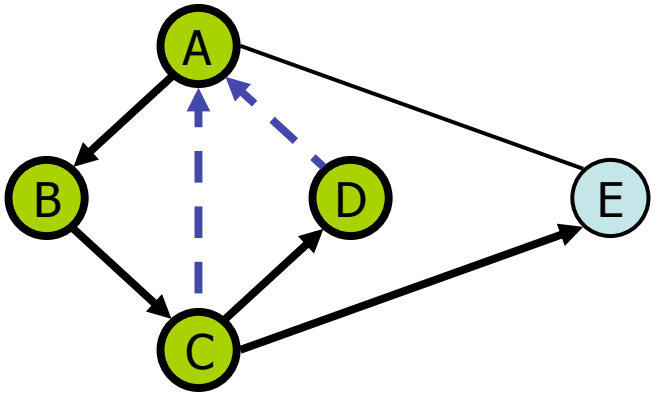
Spanning tree

# Graph Traversals

A 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 $G$
- Build a spanning tree in a connected graph

# Depth-First Search



# Outline and Reading

- Depth-first search
  - With a Stack
  - Recursive
  - Examples
  - Properties
  - Complexity
- Applications of DFS
  - Path finding
  - Cycle finding

# Depth-First Search

Depth-First Search is a graph traversal technique that:

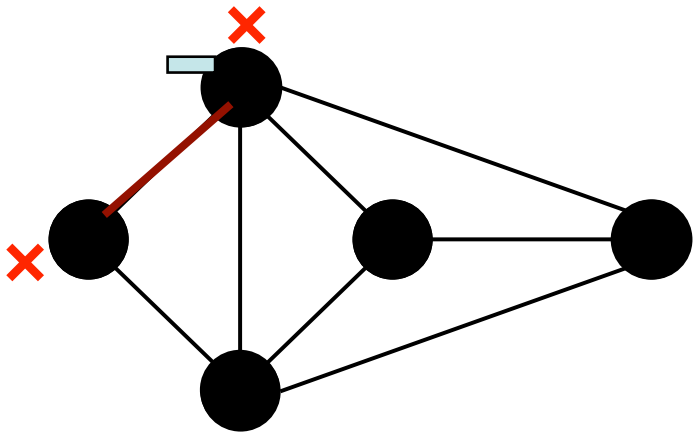
- on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time (which is O(m) )

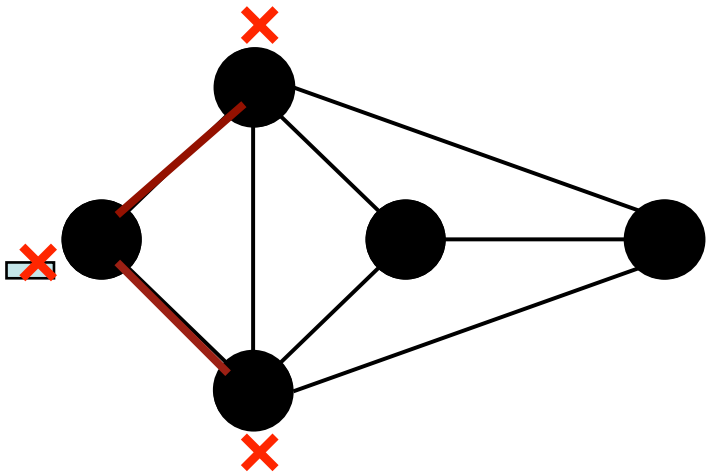
- can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph

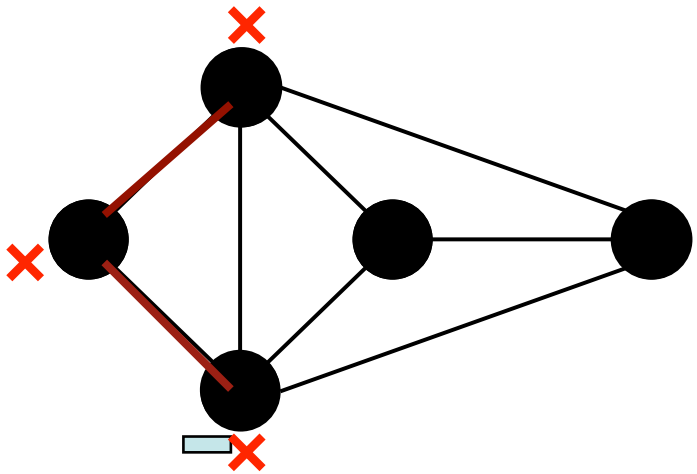
The idea:

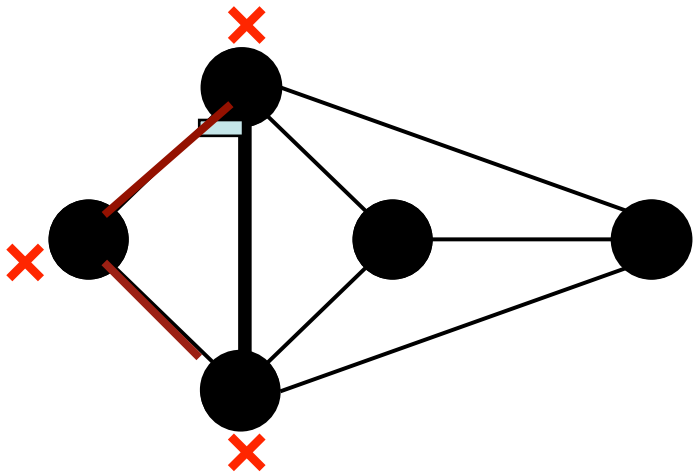
Starting at an arbitrary vertex, follow along a simple path until you get to a vertex which has no unvisited adjacent vertices.

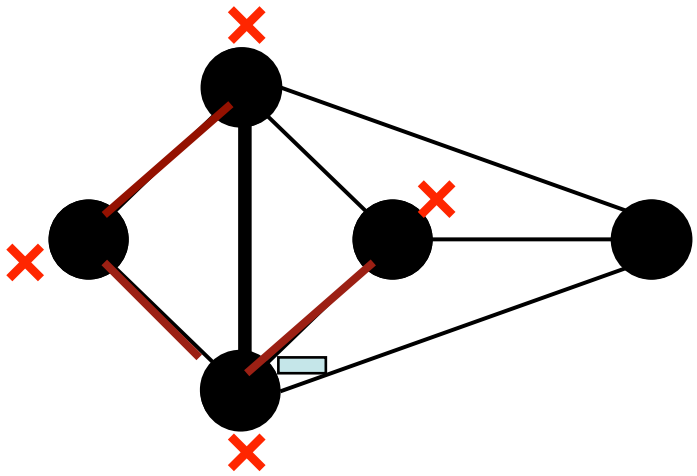
Then start tracing back up the path, one vertex at a time, to find a vertex with unvisited adjacent vertices.

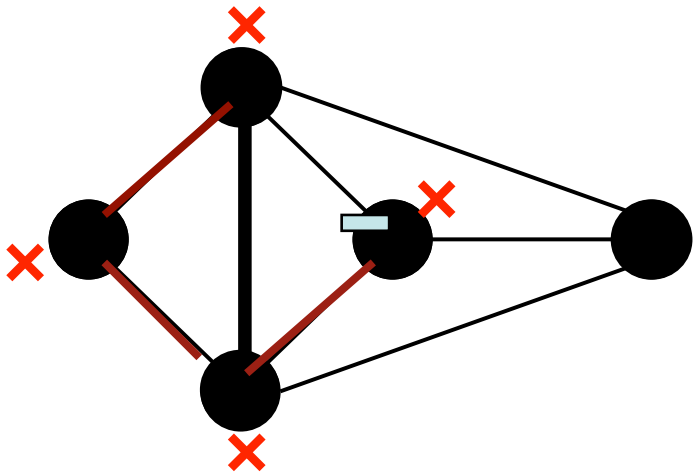


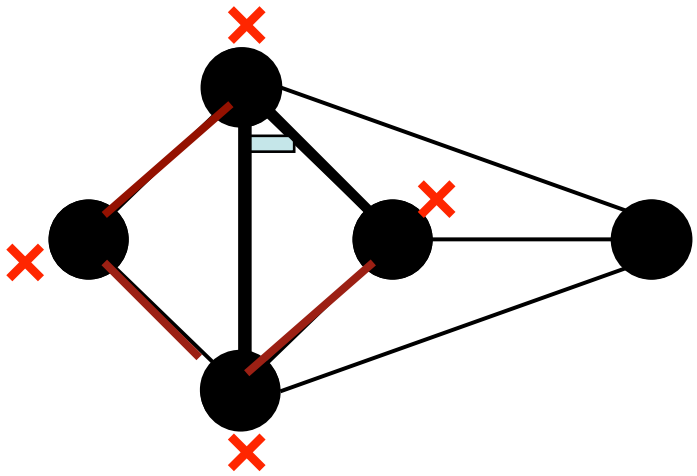


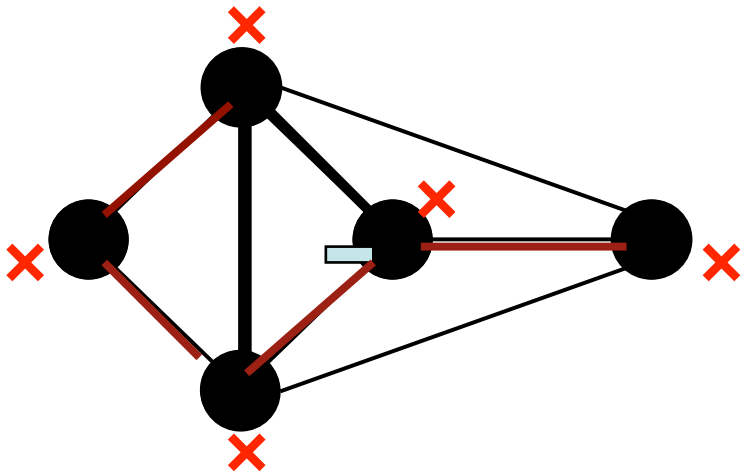


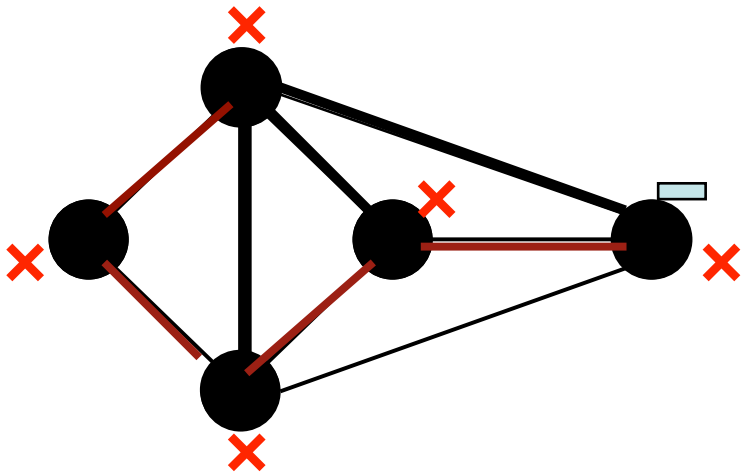


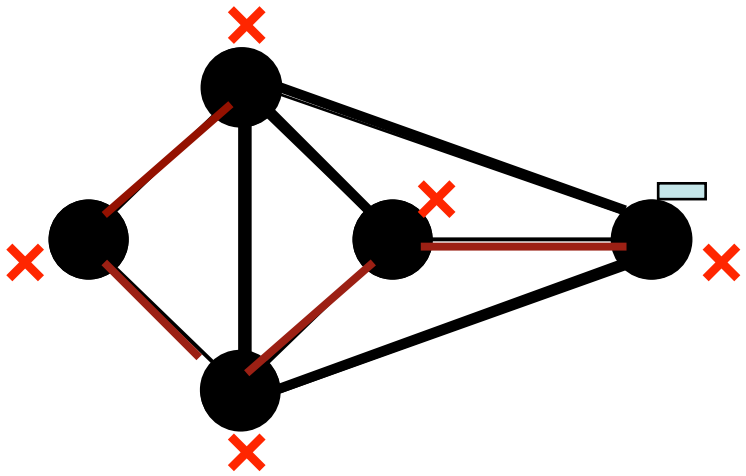


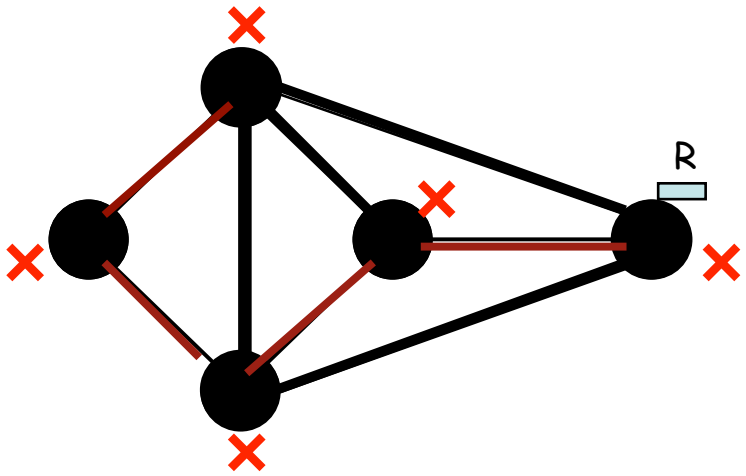


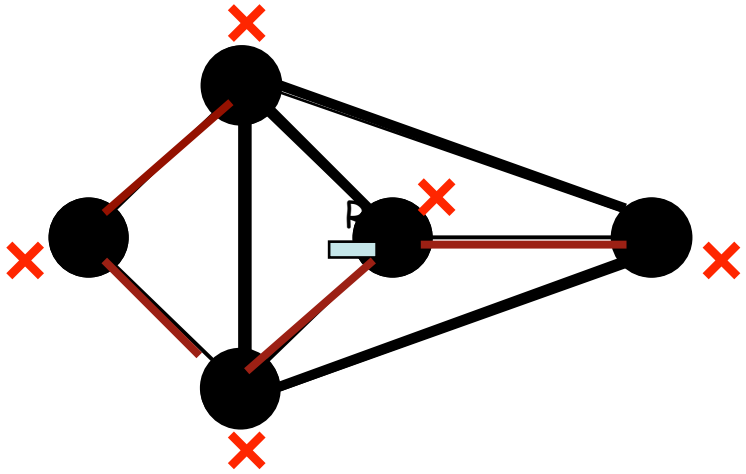




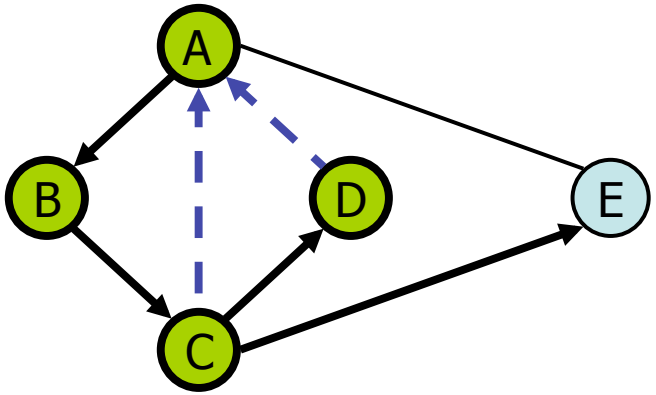


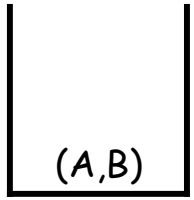


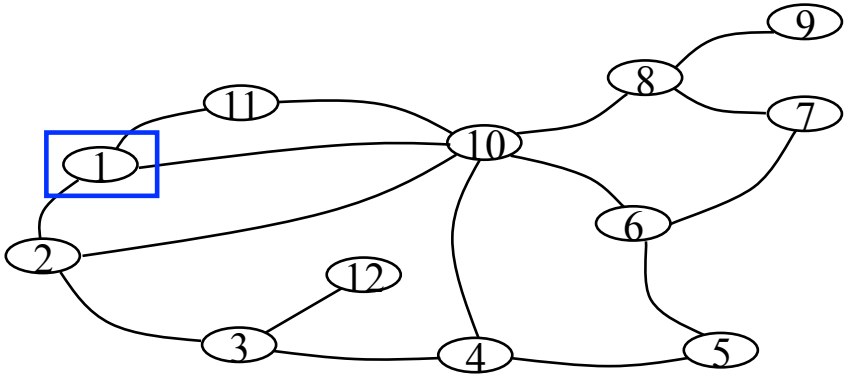

R



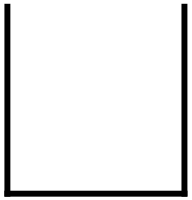
# DFS Algorithm – With a Stack



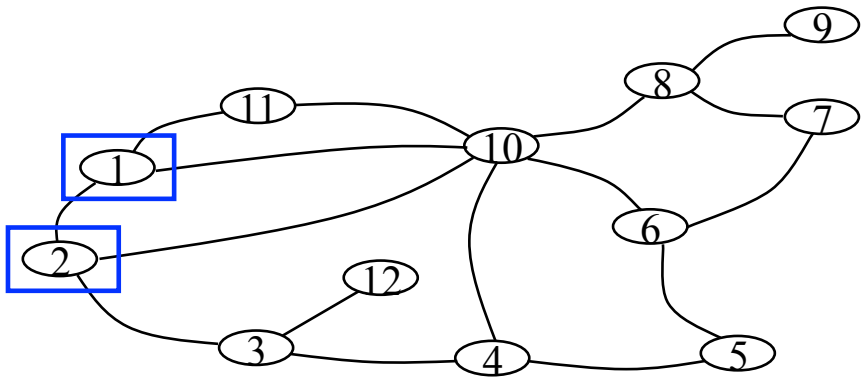






Visited : $\{1\}$



to visit



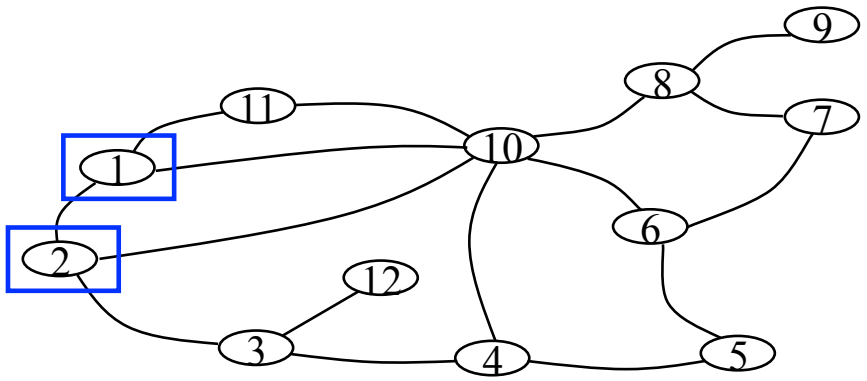

Visited : {1}

POP: (1,2)
Visited : {1,2}
T={(1,2)}

**PUSH**

(1,2)
(1,11)
(1,10)

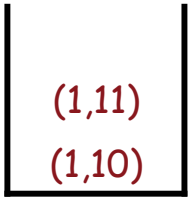
to visit



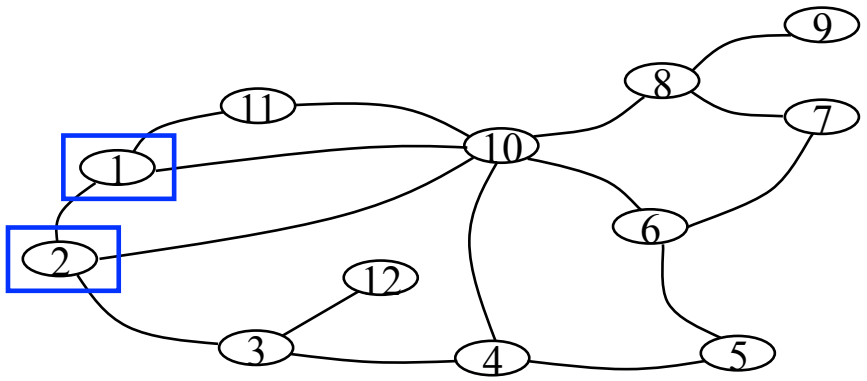

Visited : {1}

POP: (1,2)
Visited : {1,2}
T={(1,2)}

**PUSH**




to visit




POP: (2,3)

Visited : {1,2,3}

T={(1,2), (2,3)}

**PUSH**

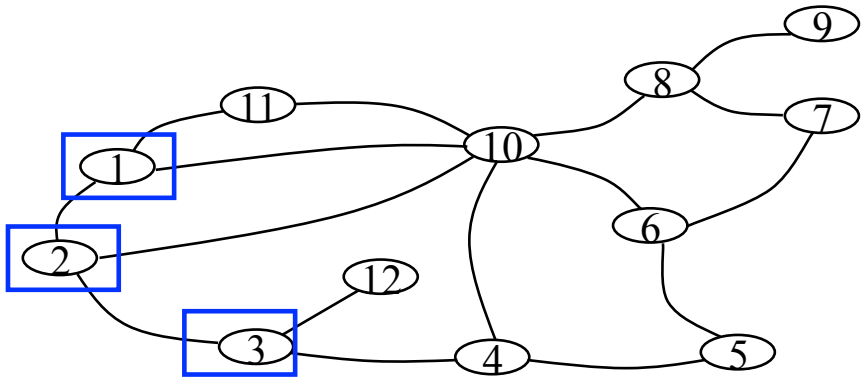
(2,3)

(2,10)

(1,11)

(1,10)

to visit




POP: (2,3)

Visited : {1,2,3}

T={(1,2), (2,3)}

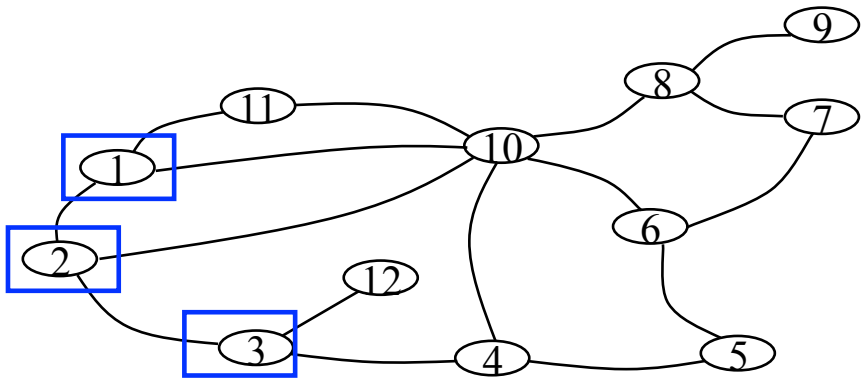
**PUSH**

(2,10)

(1,11)

(1,10)

to visit

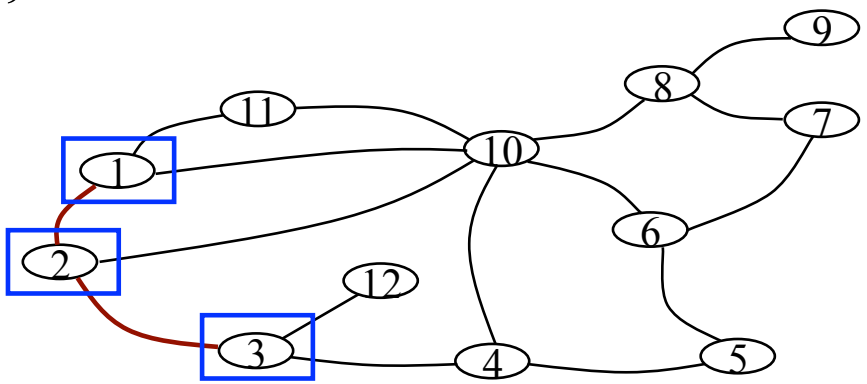



**PUSH**

(3,12)
(3,4)
(2,10)
(1,11)
(1,10)

to visit

T={(1,2), (2,3)}




(3,12)

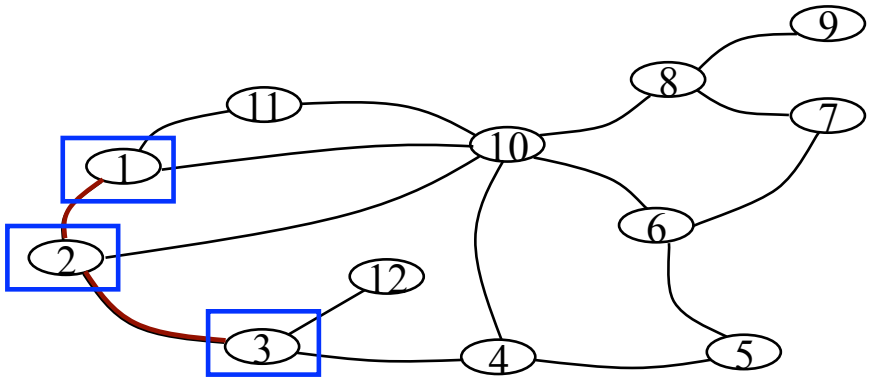
(3,4)

(2,10)

(1,11)

(1,10)

| |
|---|
| (3,12) |
| (3,4) |
| (2,10) |
| (1,11) |
| (1,10) |




POP → (3,12)

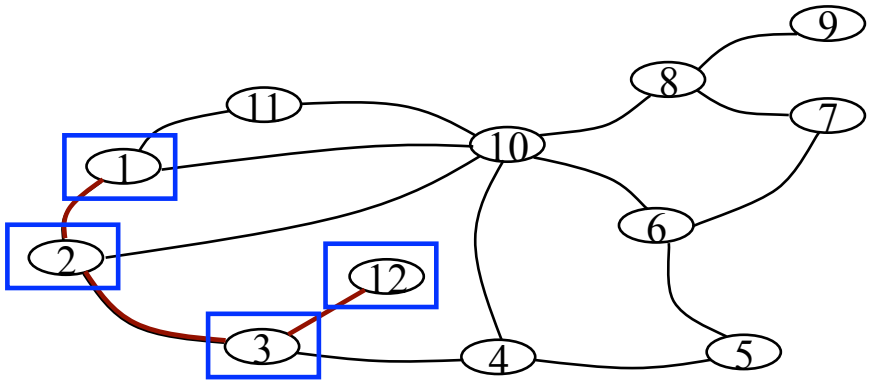
Visited : {1,2,3,12}

T={(1,2), (2,3), (3,12)}

**PUSH**
(nothing
in this case)

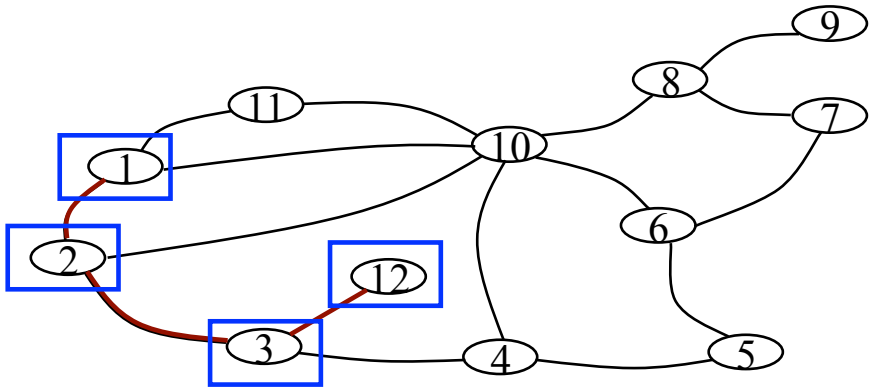
| |
|---|
| (3,4) |
| (2,10) |
| (1,11) |
| (1,10) |

(3,4)
(2,10)
(1,11)
(1,10)




T={(1,2), (2,3), (3,12)}

(3,4)
(2,10)
(1,11)
(1,10)




POP → (3,4)

Visited : {1,2,3,12,4}

T={(1,2), (2,3), (3,12), (3,4)}

PUSH
(4,4) and (4,10)

(4,10)
(4,5)
(2,10)
(1,11)
(1,10)

POP → (4,10)

Visited : {1,2,3,12,4,10}

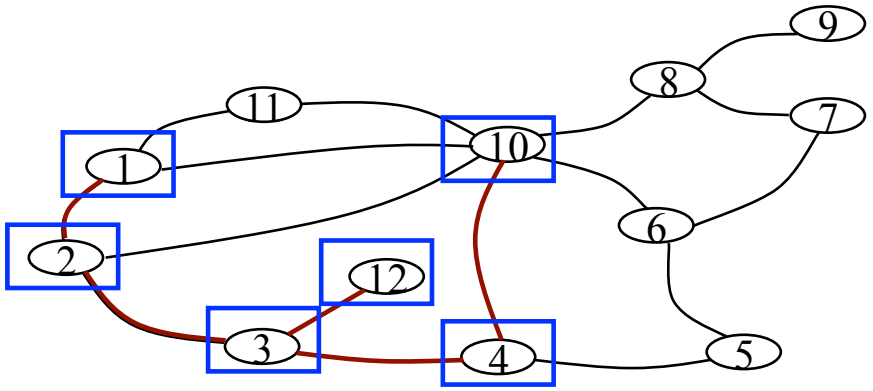
T={(1,2), (2,3), (3,12), (3,4), (4,10)}

PUSH
(10,6) and (10,8)

(10,8)
(10,6)
(4,5)
(2,10)
(1,11)
(1,10)

• • •

(10,8)
(10,6)
(4,5)
(2,10)
(1,11)
(1,10)




T={(1,2), (2,3), (3,12), (3,4), (4,10)}

• • •

# Complexity

Elementary operations: Pop, Push, and visits

Number of PUSH:

$$\sum_{v \in V} d(v) = 2m$$

Number of POP:

$$\sum_{v \in V} d(v) = 2m$$

Visit of a node: n

$O(n+m) = O(m)$

# DFS Algorithm – Recursive version

```
DFS(v)
Mark v visited
 ∀w ∈ Adjacent(v)
        if w not visited
                visit w
                DFS(w)
```

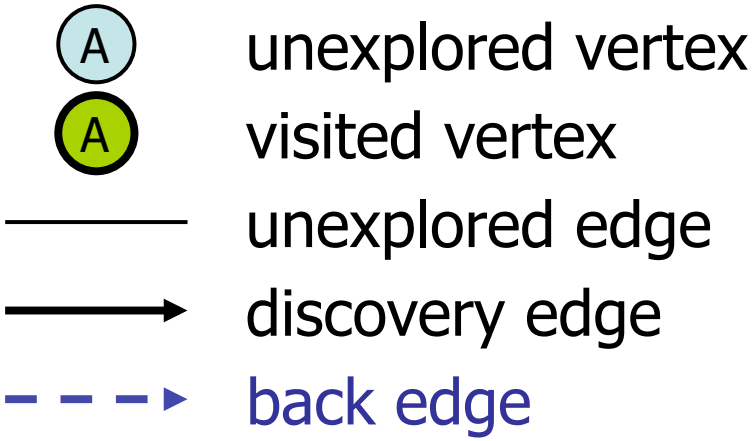
# DFS Again - More Detail…

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

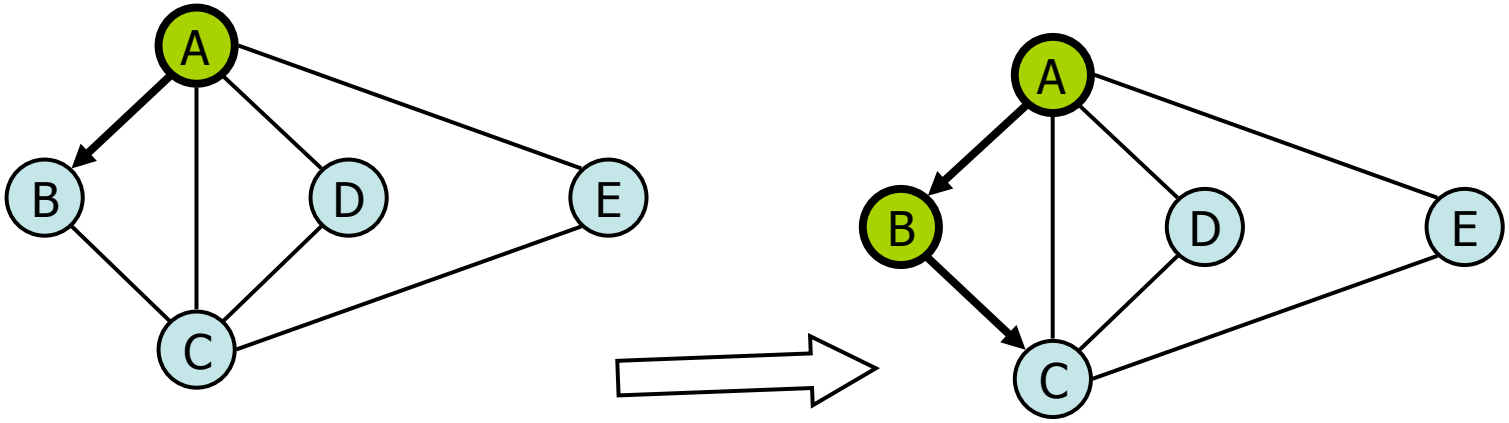
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G,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

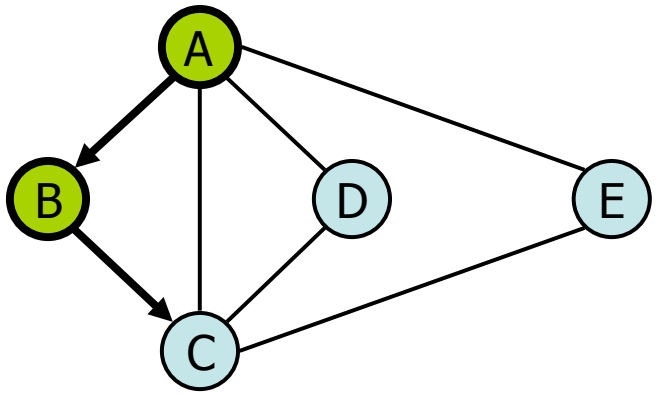
```
Algorithm DFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
    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

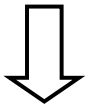
# Example

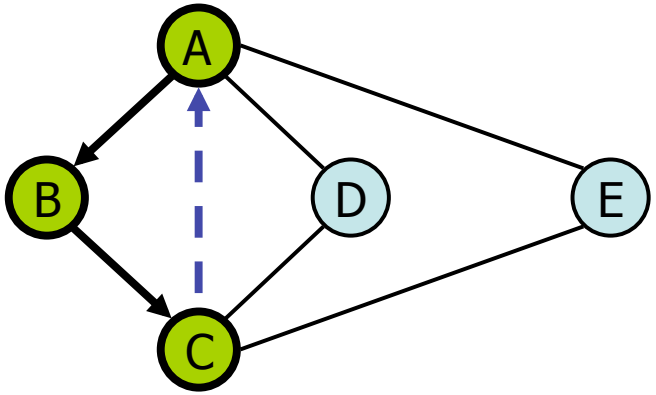


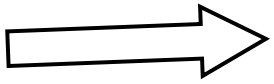

```
Algorithm DFS(G, v)
    setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)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

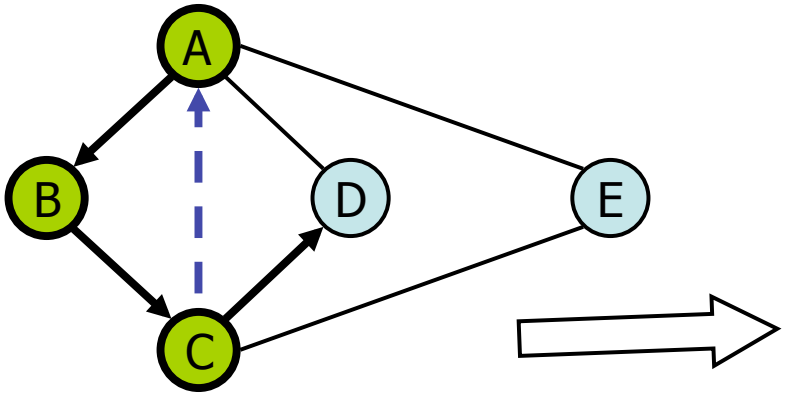




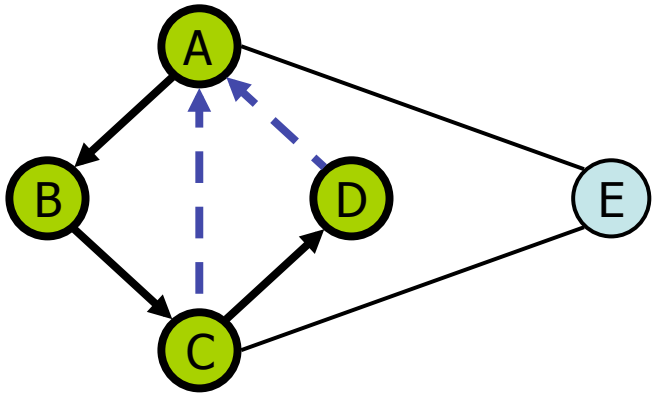


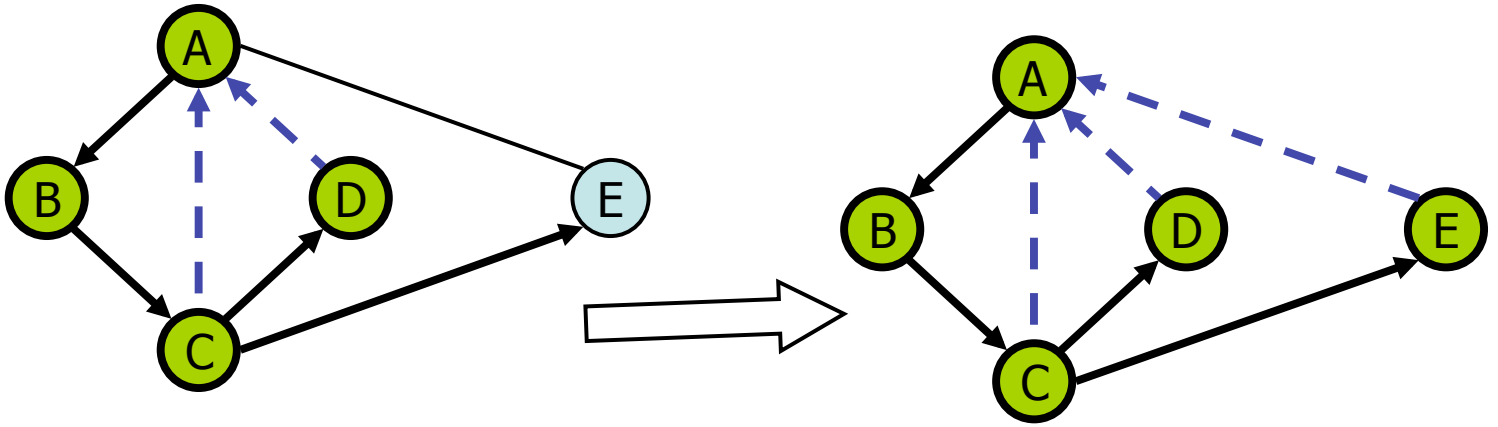





```
Algorithm DFS(G, v)
    setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)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

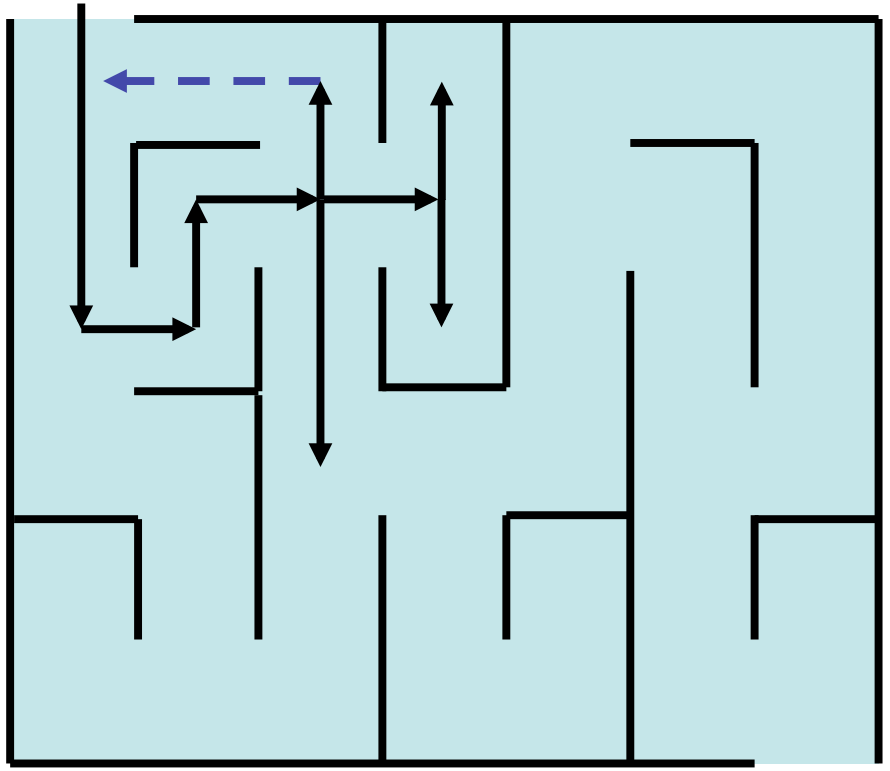



```
Algorithm DFS(G, v)
    setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)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



# 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)



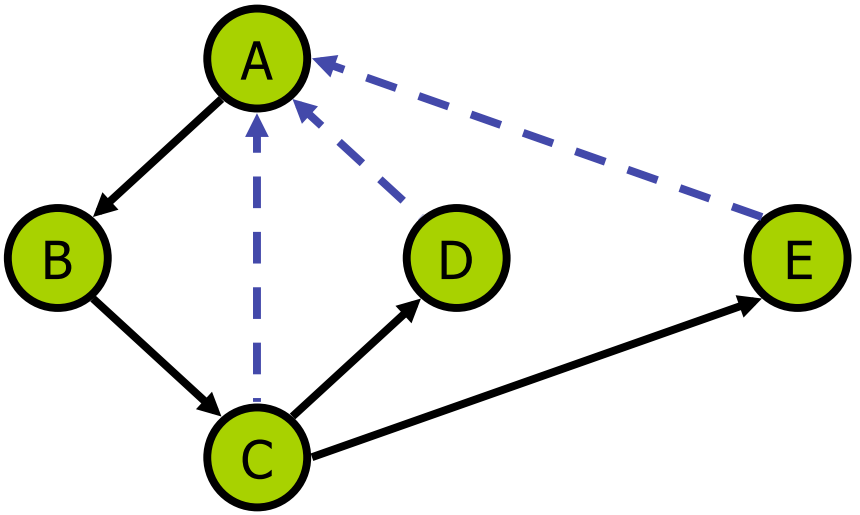
# Properties of DFS

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 + labeling

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED

2n

- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK

2m

- Method incidentEdges is called once for each vertex

$$\sum_{v \in V} d(v) = 2m$$

If the graph is implemented with adjacency list

- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

$$O(n + m) = O(m)$$

## Conclusion

If we represent the graph with an adjacency list

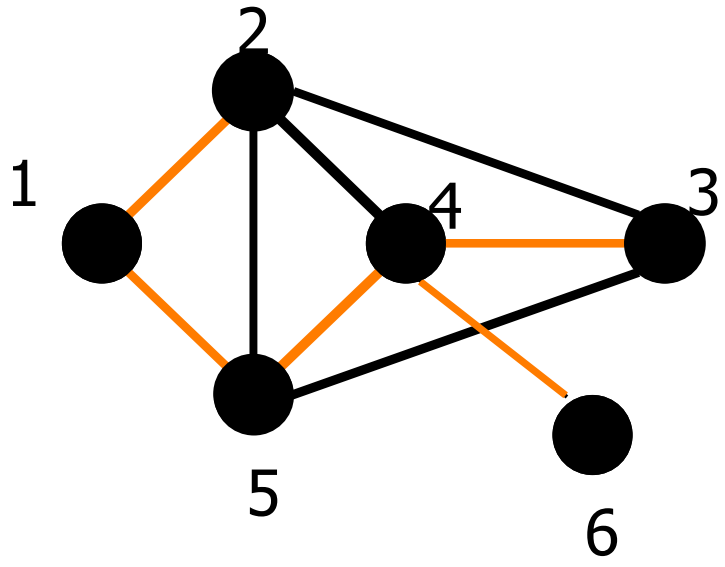
Complexity of DFS is O(m)

WORST CASE: m = $O(n^2)$, when …

**Question:** With adjacency matrix ?

# Path Finding

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




2 -- 6

$2_{(2,1)}\ 1\ {}_{(1,5)}5_{(5,4)}\ 4\ \cancel{3}\ {}_{(4,3)}\ {}_{(4,6)}\ 6$

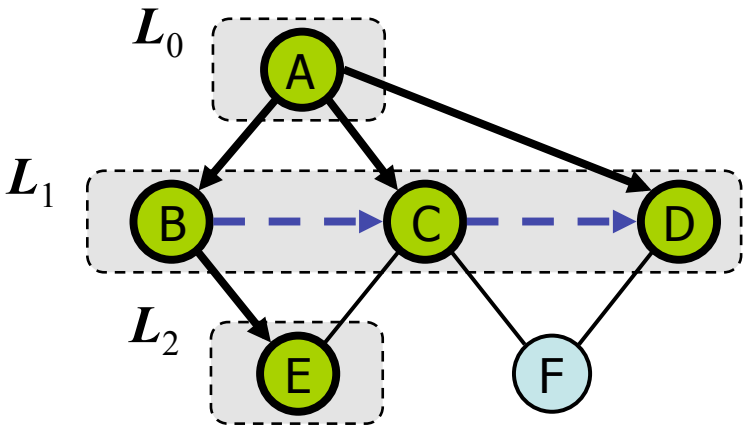
```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
         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

# Cycle Finding

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

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

# Breadth-First Search



# Outline and Reading

- Breadth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
  - Applications
- DFS vs. BFS
  - Comparison of applications
  - Comparison of edge labels

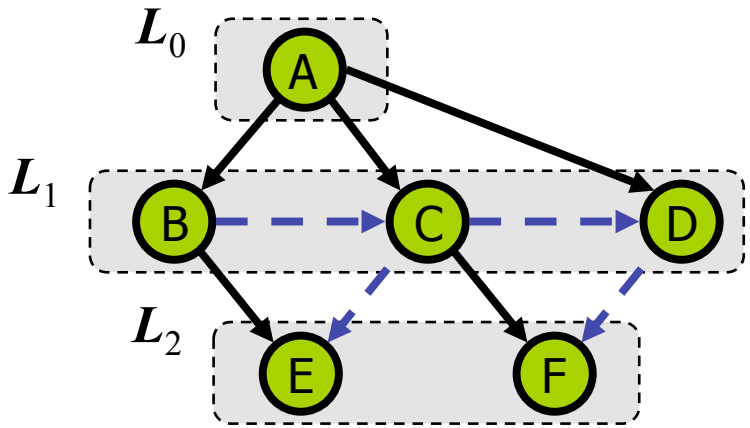
# Breadth-First Search

Breadth-First Search (BFS) is a graph traversal technique that:

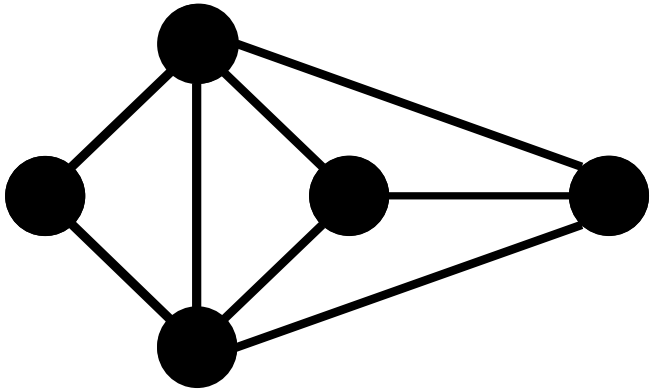
- on a graph with $n$ vertices and $m$ edges, takes $O(n + m)$ time
- can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

The idea:

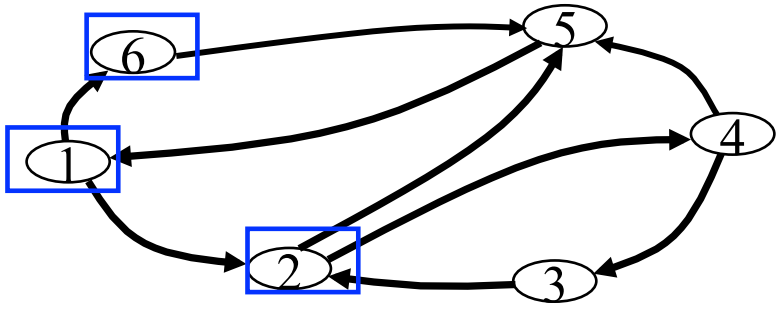
Visit a vertex and then visit all unvisited vertices that are adjacent to it before visiting a vertex which is 2 away from it.




level by level



# Breath First Search with a Queue




visited : {1}
T = ϕ

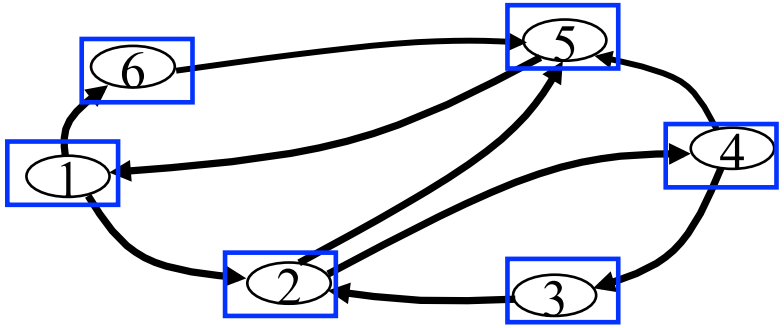
to visit: {(1,2), (1,6)}

(1,2) – 2 visited?
visited : {1,2}
T = {(1,2)}

to visit: {(1,6), (2,4), (2,5)}

(1,6) - 6 visited ?
visited : {1,2,6}
T = {(1,2), (1,6)}

to visit: {(2,4), (2,5), (6,5)}




(2,4) - 4 visited?

visited: {1,2,6,4}

to visit: {(2,5), (6,5), (4,5), (4,3) }

T = {(1,2), (1,6), (2,4)}

(2,5) -5 visited ?

visited: {1,2,6,4,5}

to visit: {(6,5), (4,5), (4,3)}

T = {(1,2), (1,6), (2,4), (2,5)}

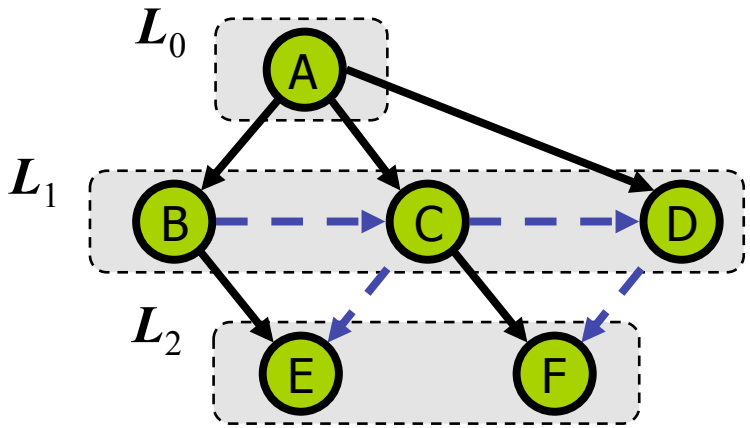
(6,5) - 5? already visited!

(4,5) - 5? already visited!

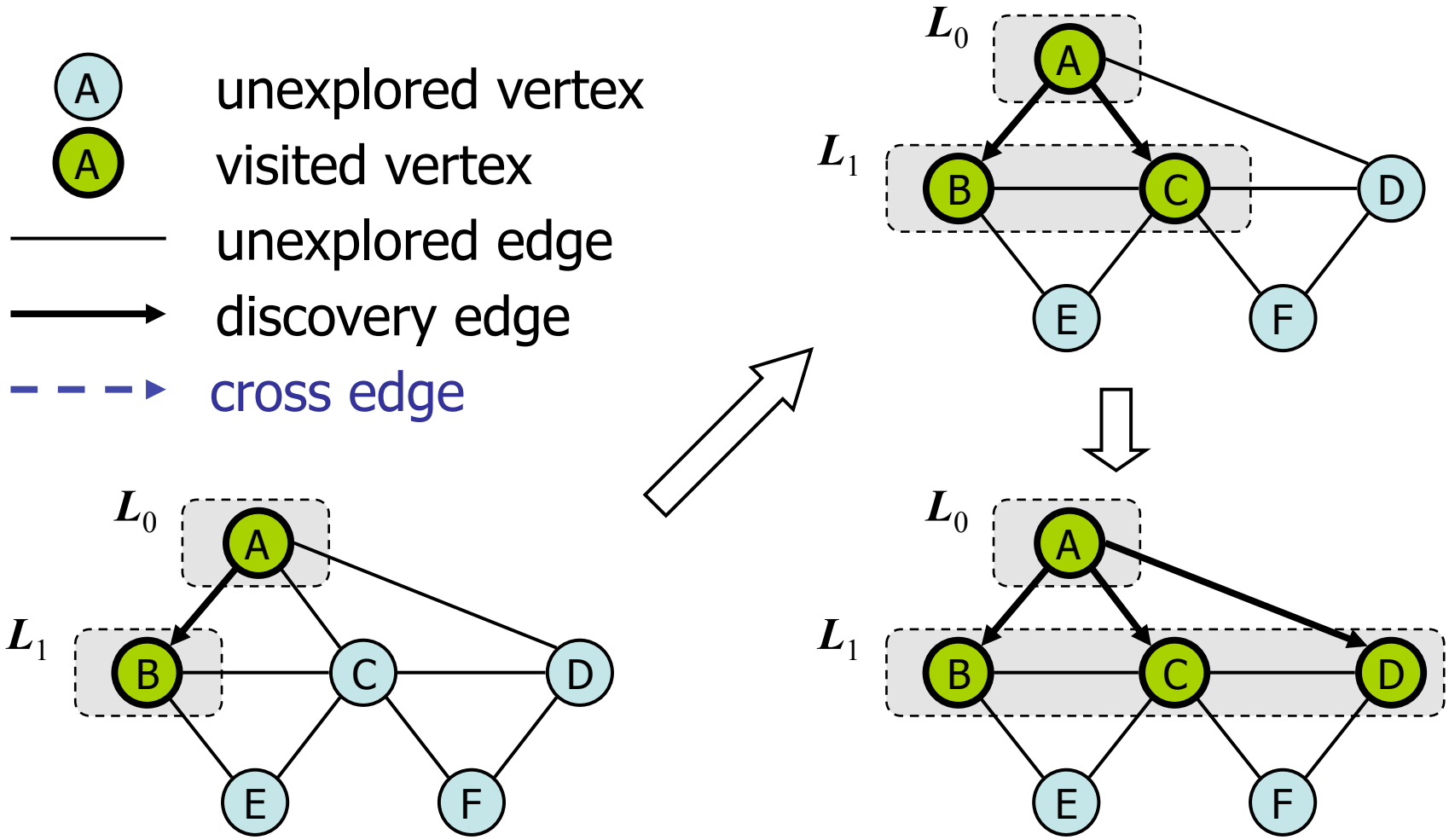
(4,3) - 3?

# BFS with labeling

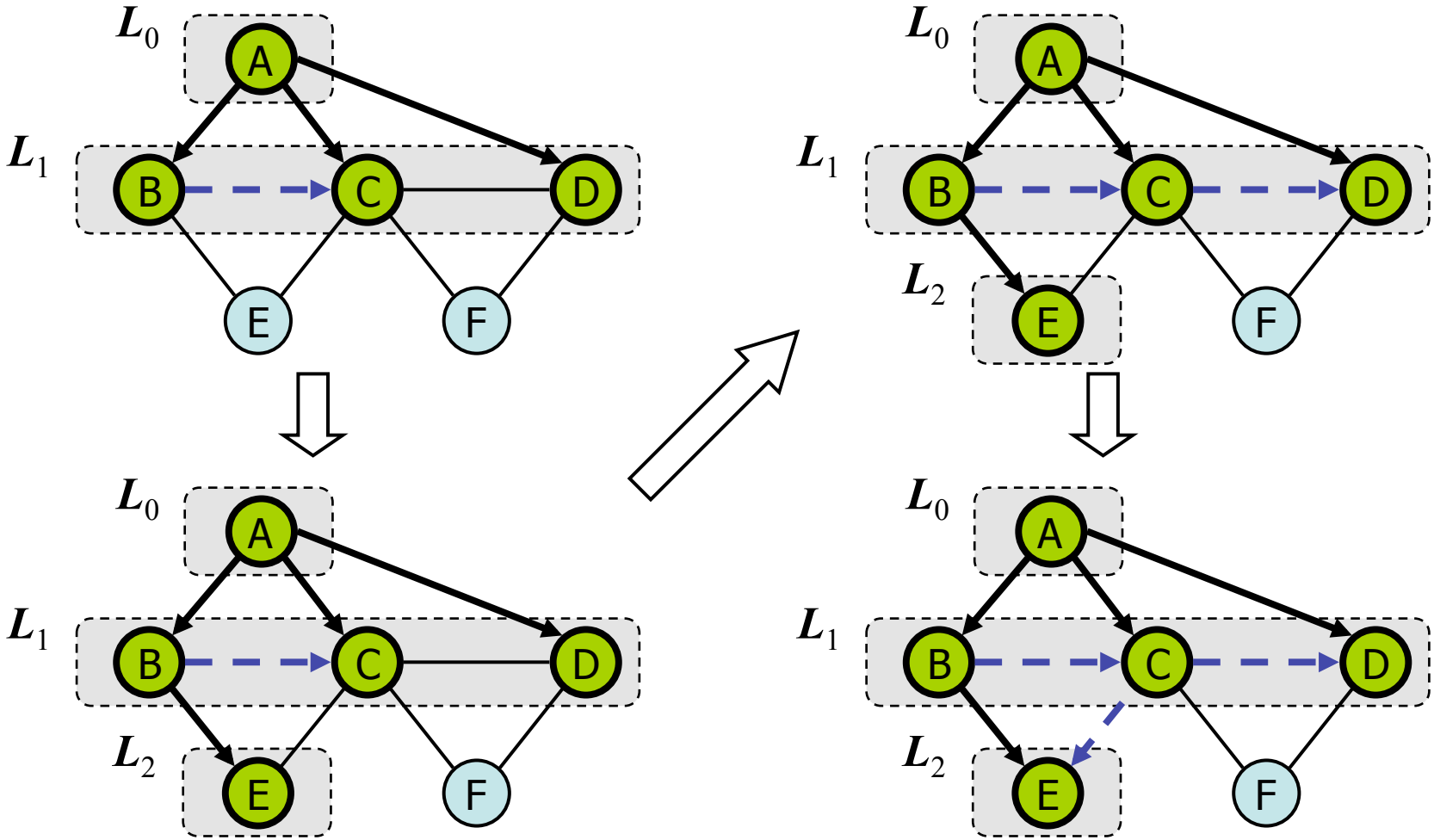
Using a sequence for each level



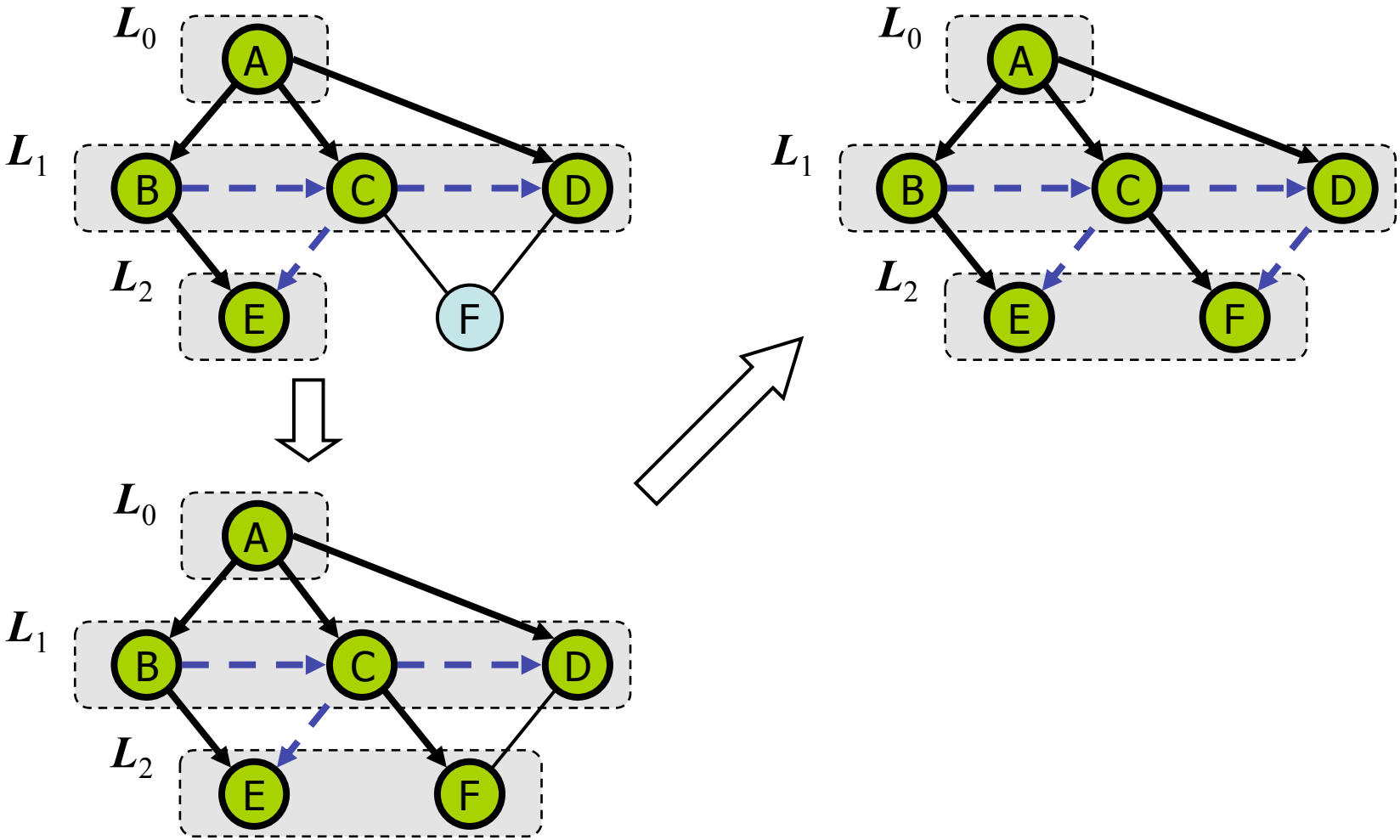
# Example


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

# Example (cont.)



# Example (cont.)



This version uses several queues instead of just one

# BFS Again - more details

- 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()
    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)
    L_0 ← new empty sequence
    L_0.insertLast(s)
    setLabel(s, VISITED)
    i ← 0
    while ! L_i.isEmpty()
      L_{i+1} ← new empty sequence
      for all v ∈ L_i.elements()
        for all e ∈ G.incidentEdges(v)
          if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
              setLabel(e, DISCOVERY)
              setLabel(w, VISITED)
              L_{i+1}.insertLast(w)
            else
              setLabel(e, CROSS)
      i ← i +1
```

# Properties

Notation

$G_s$: connected component of $s$

Property 1

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

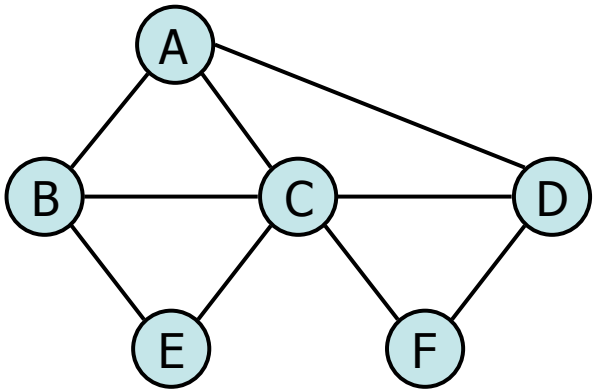
Property 2

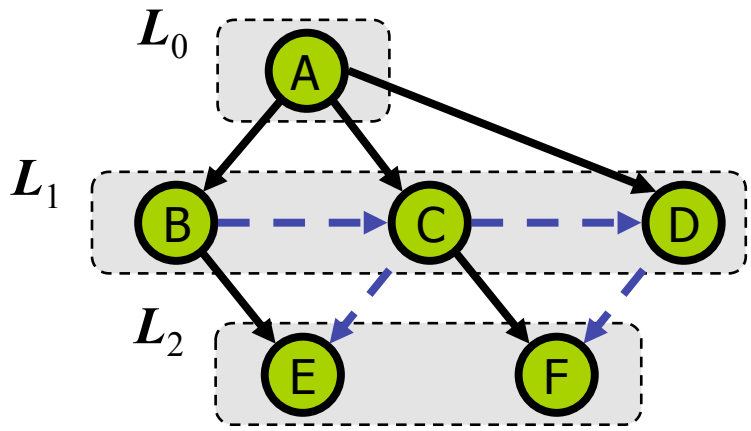
The discovery edges labeled by $BFS(G, s)$ form a spanning tree $T_s$ of $G_s$

Property 3

For each vertex $v$ in $L_i$

- The path of $T_s$ from $s$ to $v$ has $i$ edges
- Every path from $s$ to $v$ in $G_s$ has at least $i$ edges



# Analysis

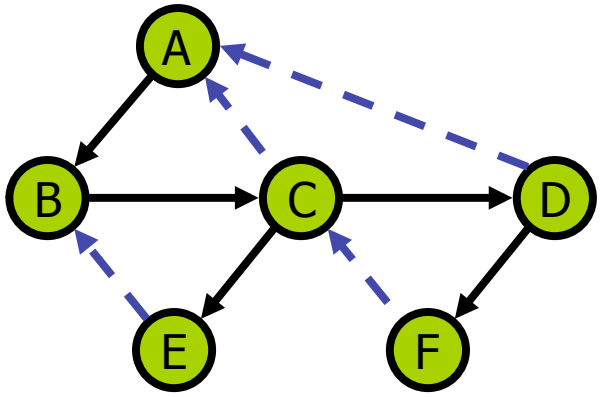
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence $L_i$
- Method incidentEdges is called once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

# Applications

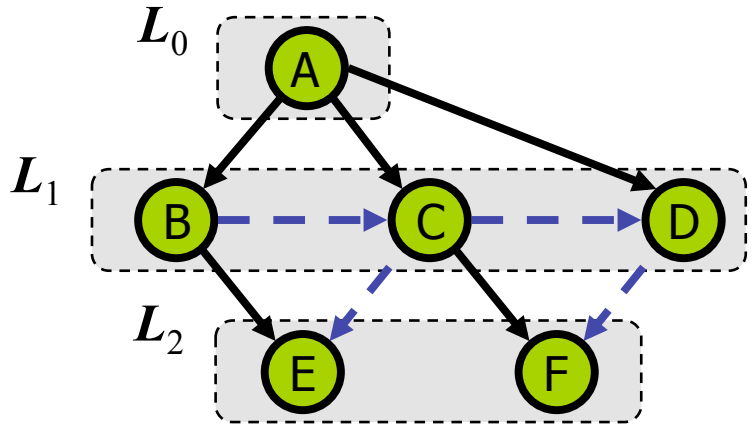
- Using the template method pattern, we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Given two vertices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

# DFS vs. BFS

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




DFS




BFS

*) "A connected graph is *biconnected* if the removal of any single vertex (and all edges incident on that vertex) can not disconnect the graph."

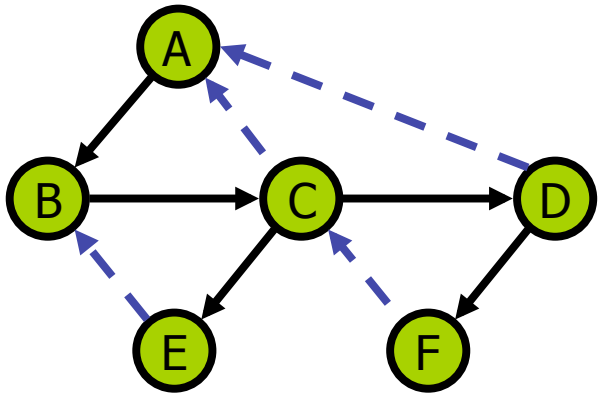
# DFS vs. BFS (cont.)

Back edge $(v,w)$

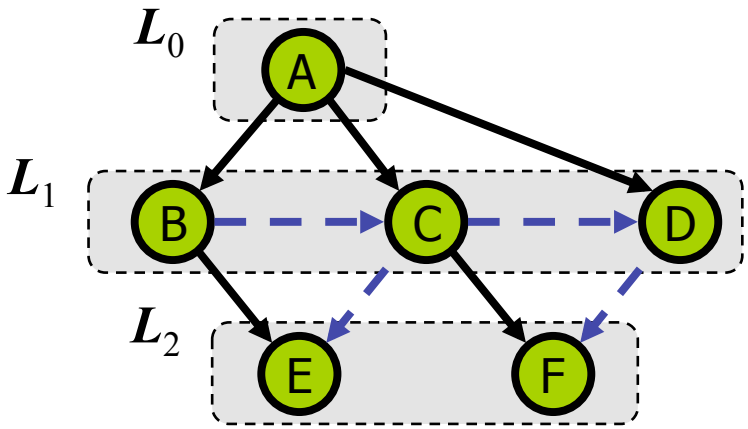
- $w$ is an ancestor of $v$ in the tree of discovery edges

Cross edge $(v,w)$

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

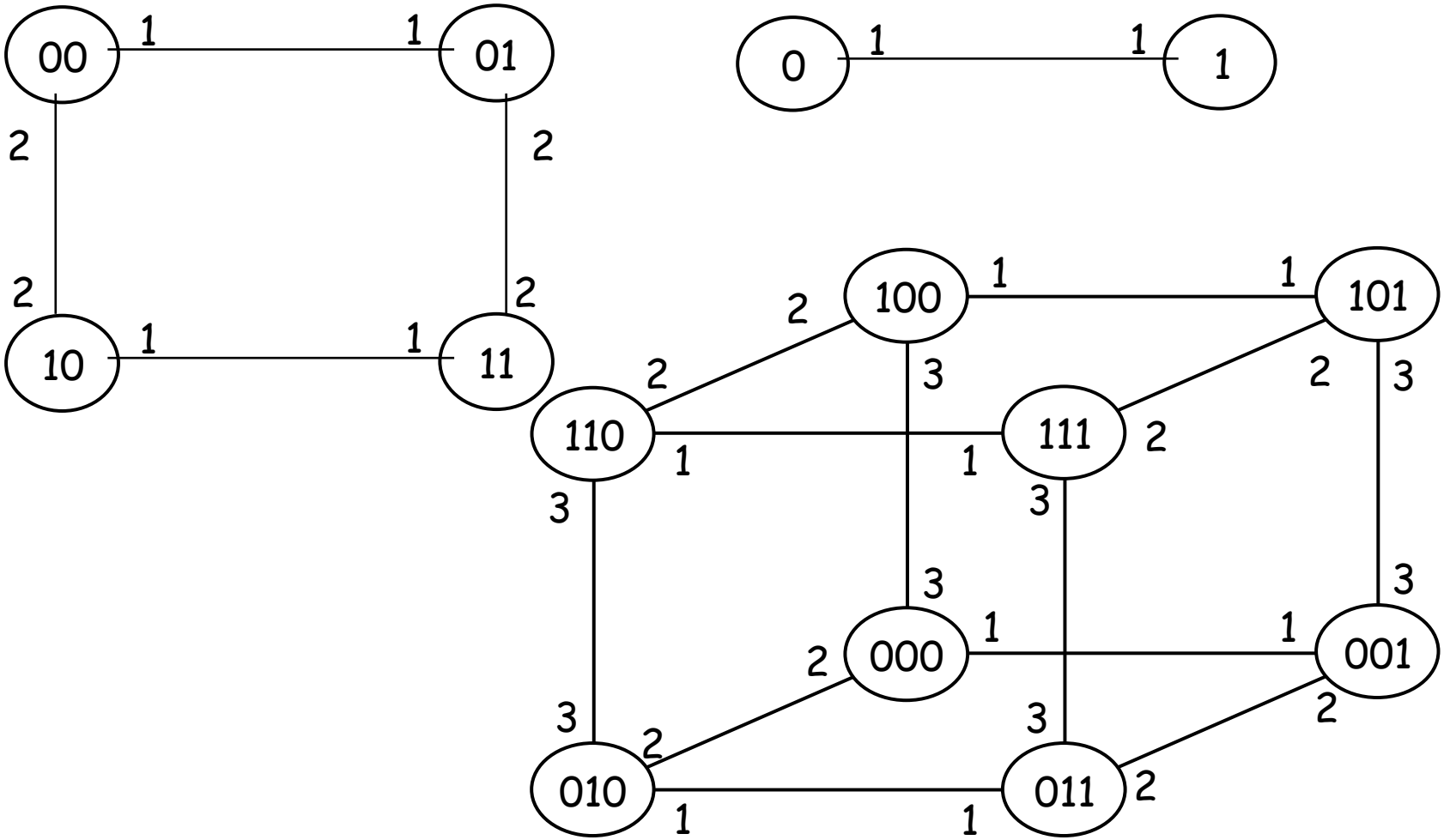



DFS

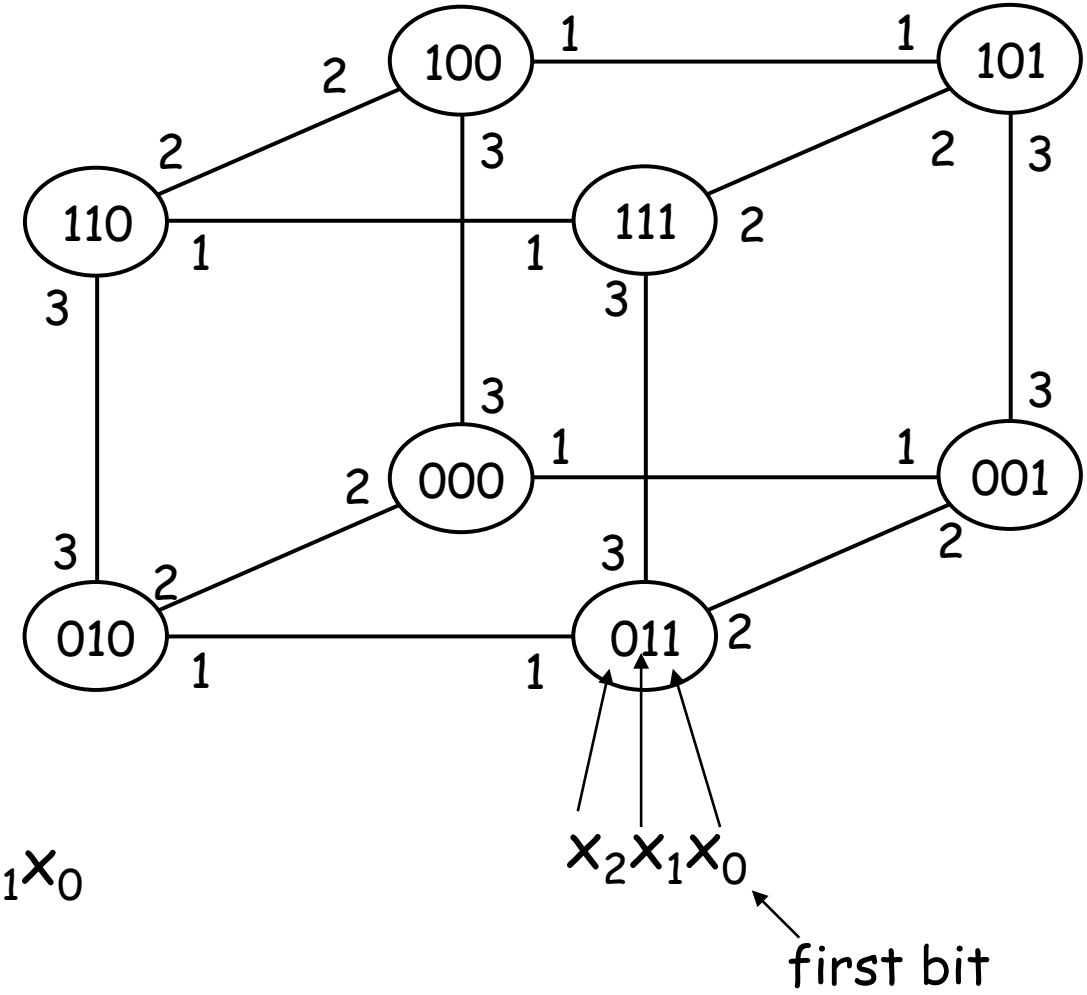



BFS

# Example: DFS/BFS in the hypercube

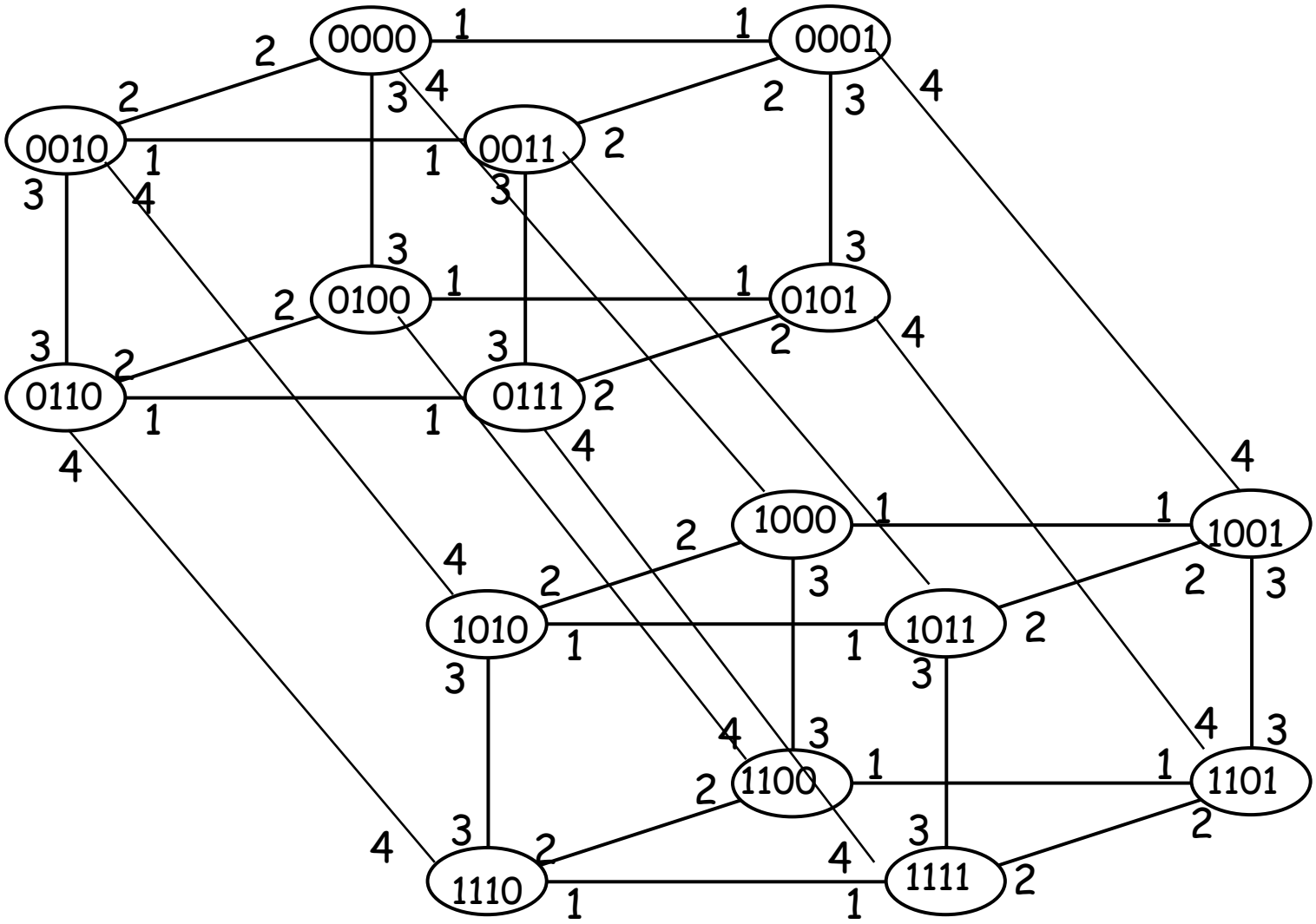


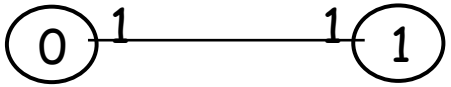

Each link between two nodes is labeled by the dimension of the bit by which the nodes ' name differ.



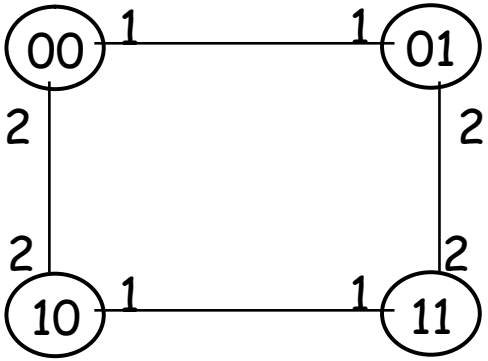

$X = x_k x_{k-1} \ldots x_1 x_0$

K-bit name

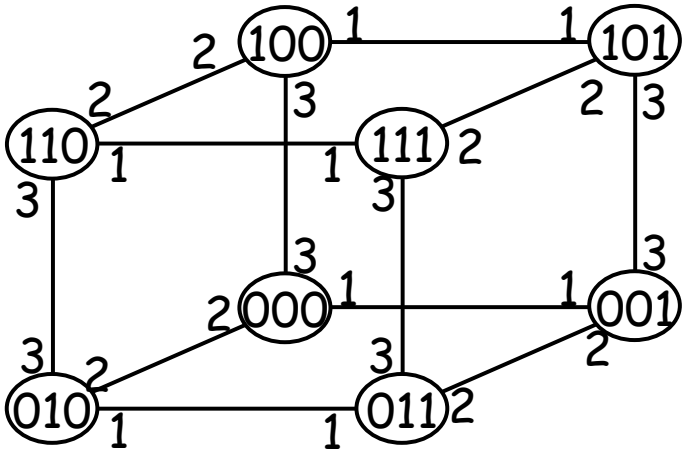

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111




n=2 dimensions=1




n=4 dimensions=2




n=8 dimensions=3

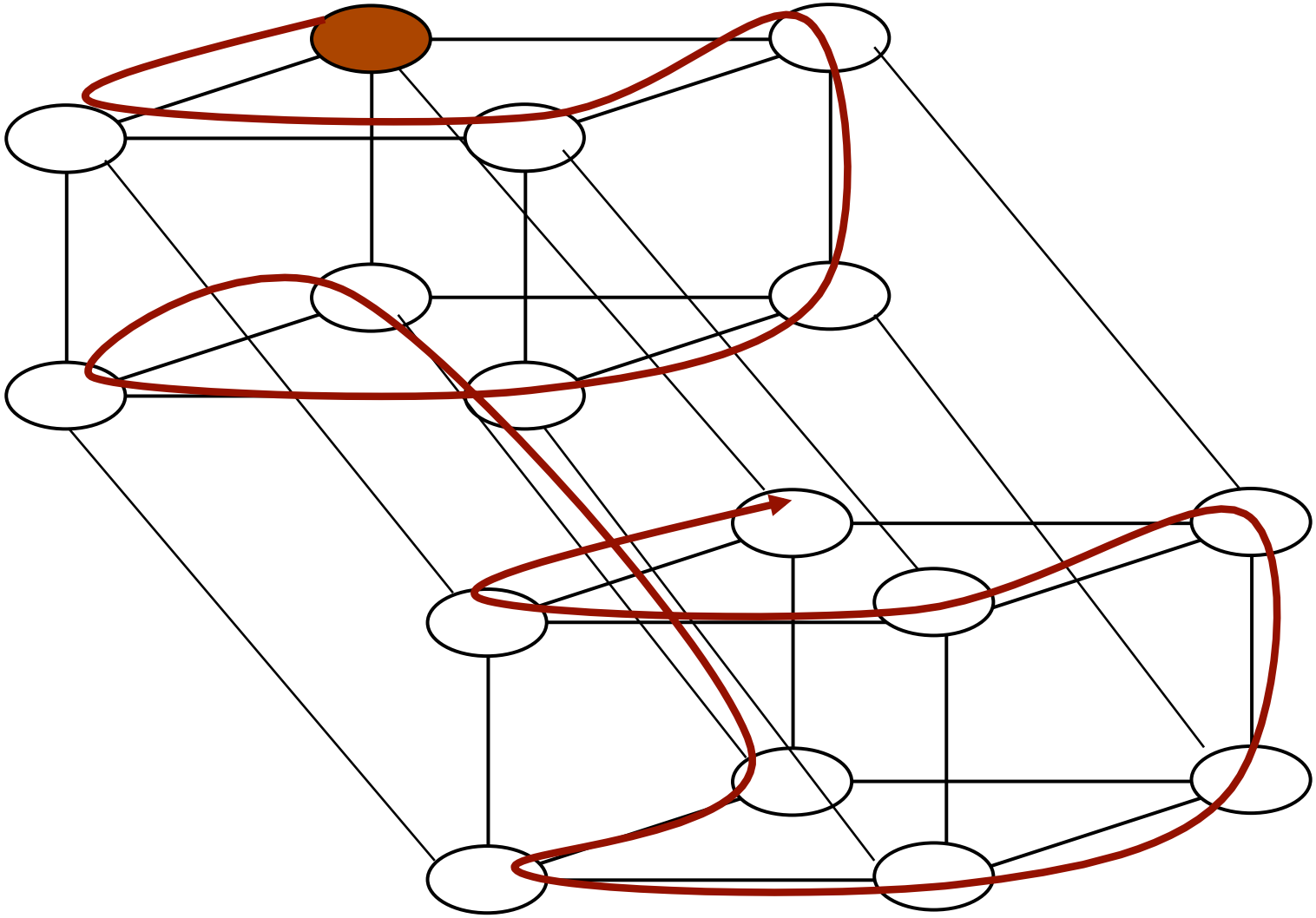
dimensions = log n

A hypercube of dimension d has $n = 2^d$ nodes

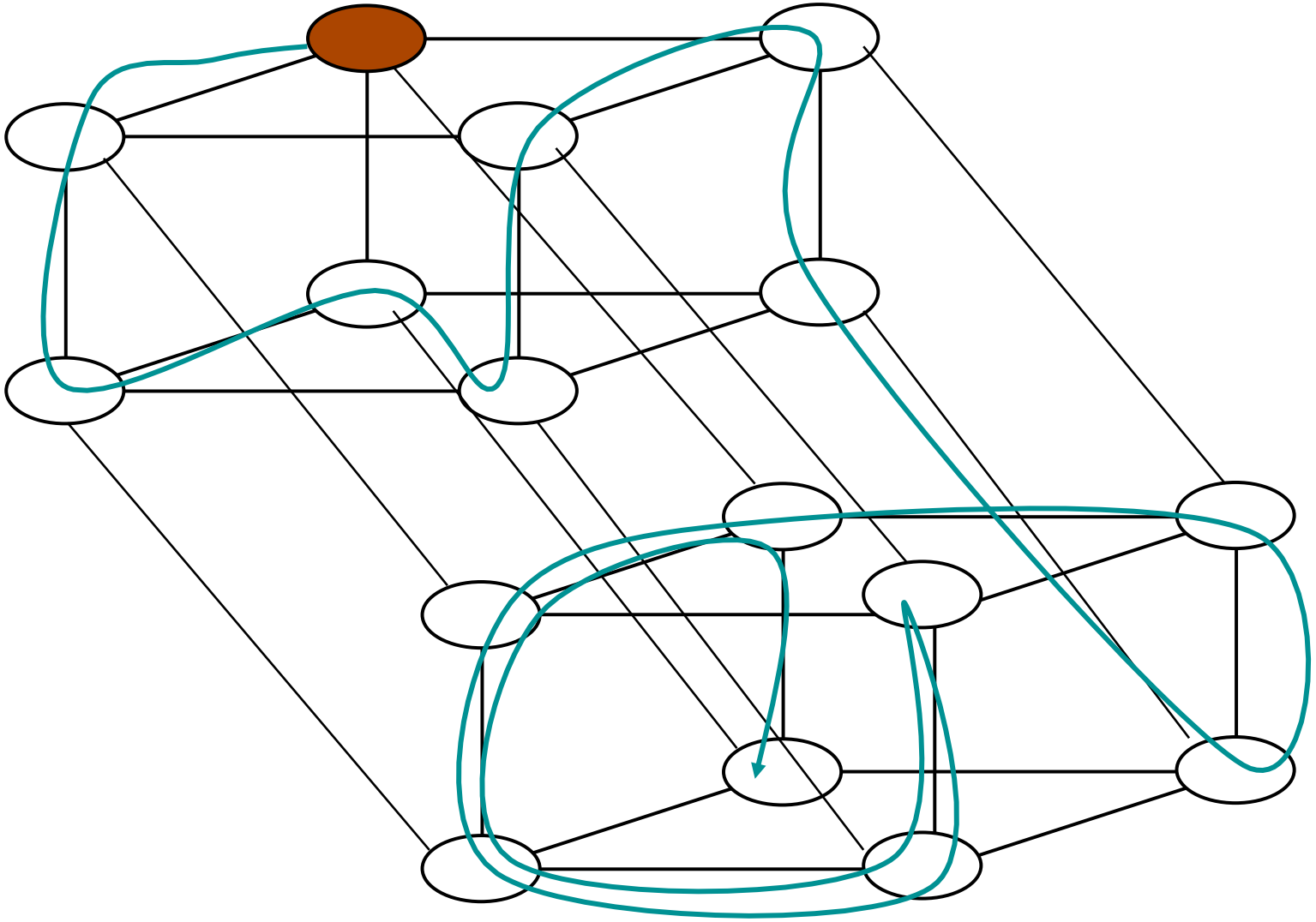
Each node has d links

➔ $m = n\,d/2 =$ **$O(n \log n)$**

# A Depth-first traversal



# Another Depth-first traversal



# A Breadth-first traversal