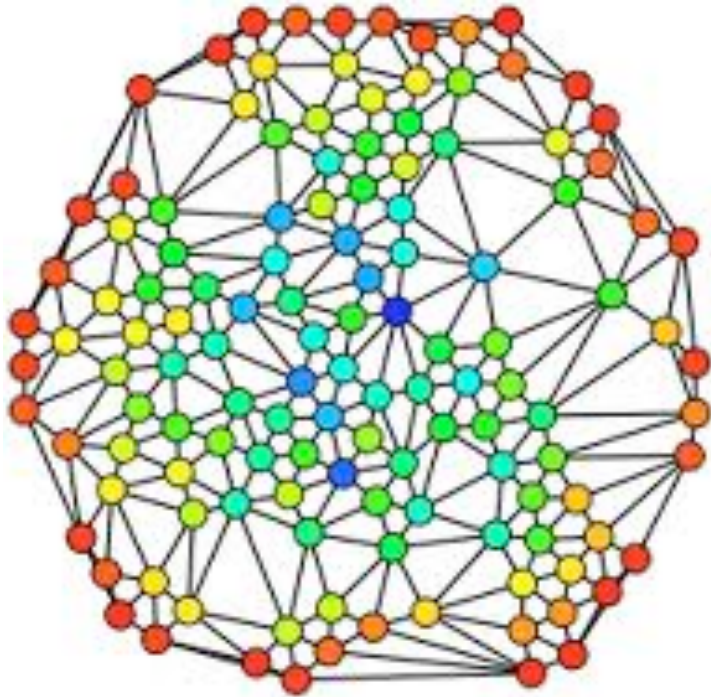


Fall 2010

Data Structures

Lecture 14

Fang Yu
Department of Management Information Systems
National Chengchi University

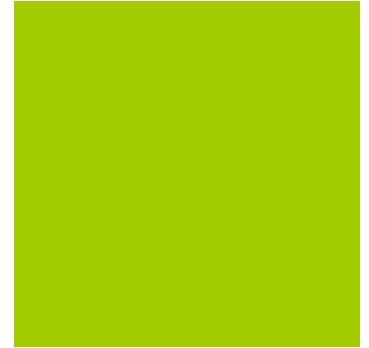


Graphs

Definition, Implementation and Traversal

Graphs

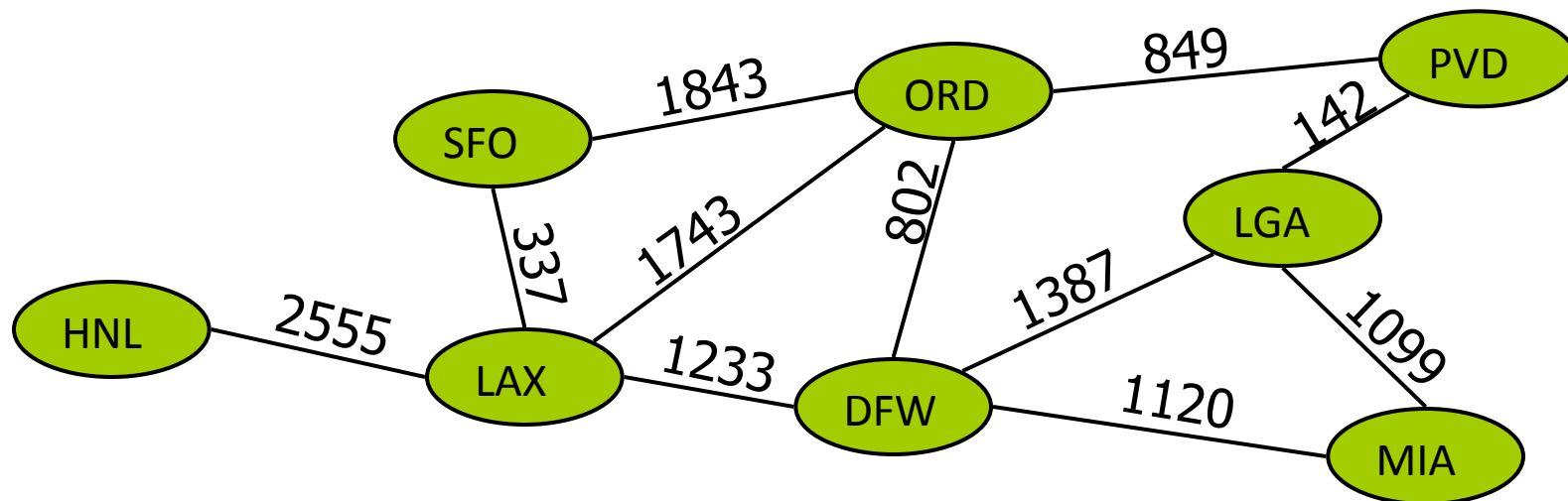
- Formally speaking, a graph is a pair (V, E) , where
 - V is a set of nodes, called vertices
 - E is a collection of pairs of vertices, called edges
 - Vertices and edges are positions and store elements



Graphs

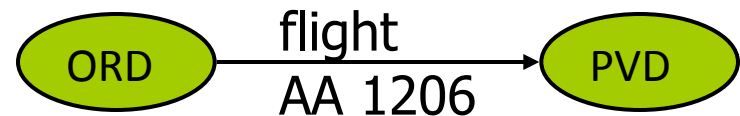
■ Example:

- 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



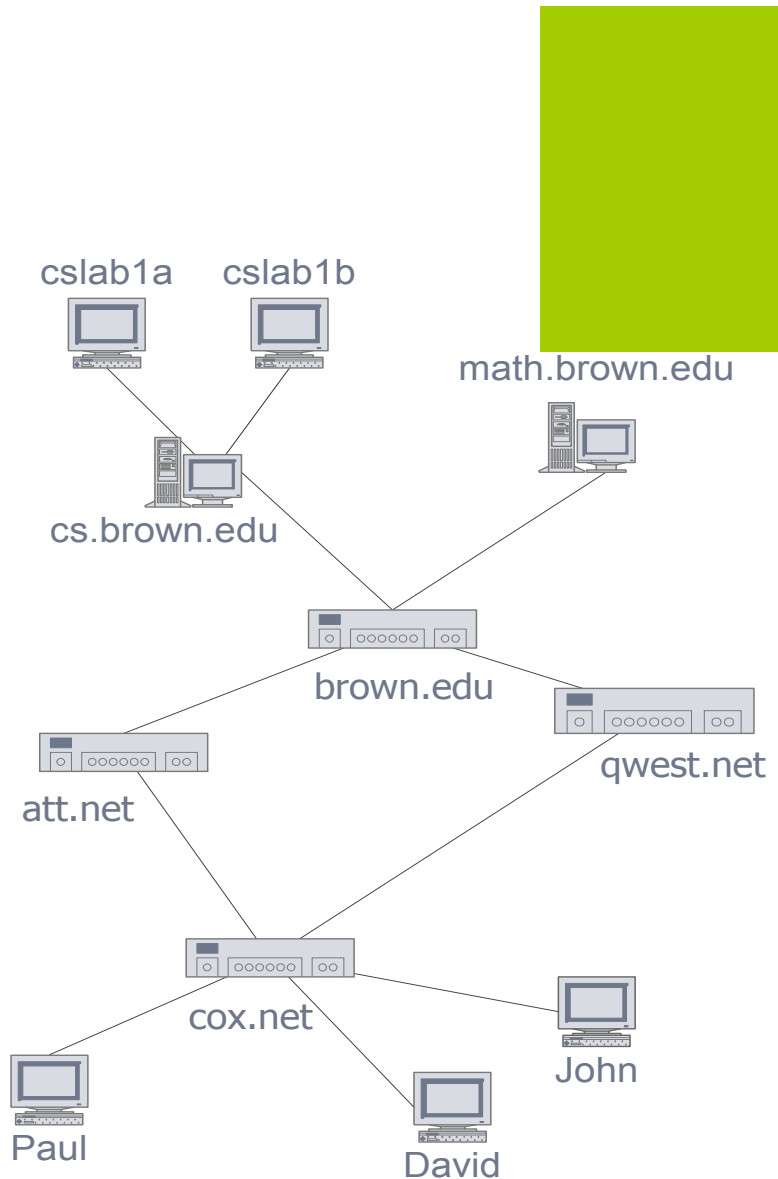
Edge Types

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



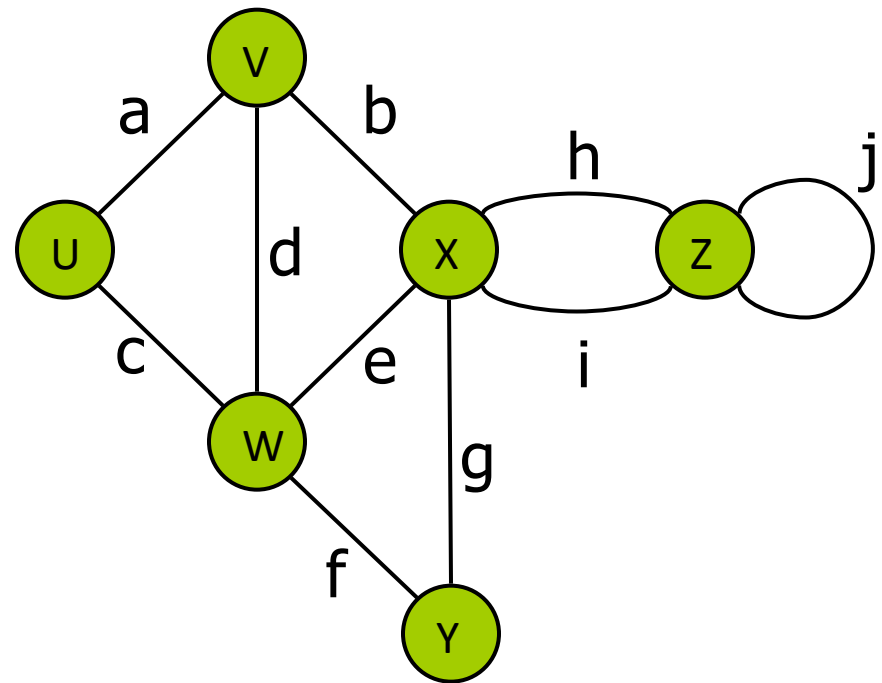
Applications

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



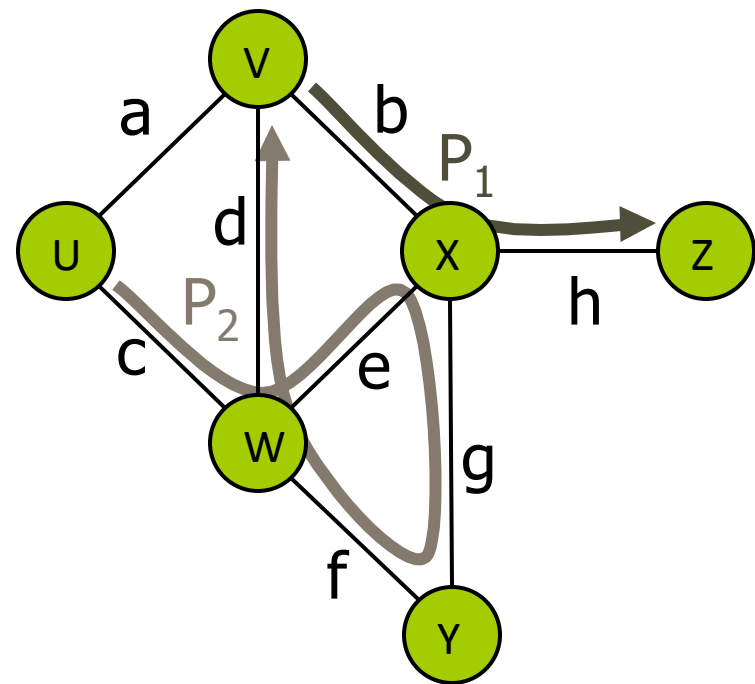
Terminology

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



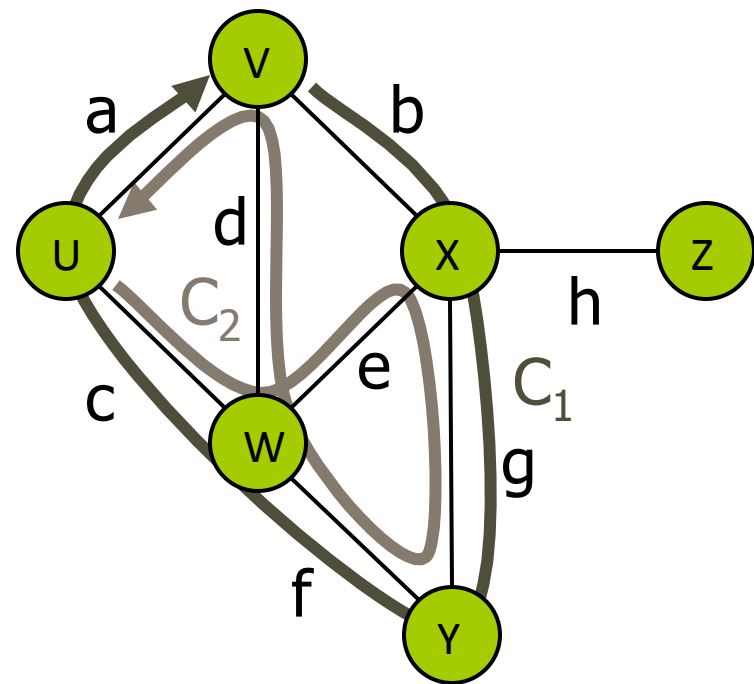
Terminology (cont.)

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



Terminology (cont.)

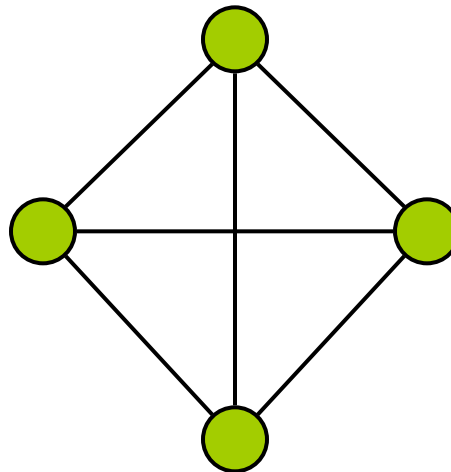
- Cycle
 - circular sequence of alternating vertices and edges
 - each edge is preceded and followed by its endpoints
- Simple cycle
 - cycle such that all its vertices and edges are distinct
- Examples
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, \hookrightarrow)$ is a simple cycle
 - $C_2 = (U, c, W, e,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



Properties

Notation

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



Example

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

Properties

- Property 1

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

- Proof: each edge is counted twice

- Property 2

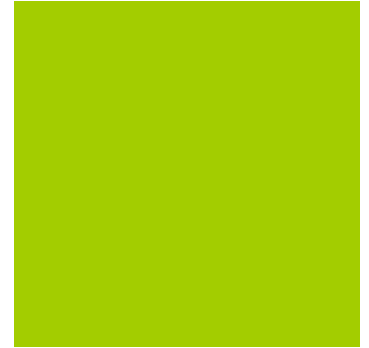
- In an undirected graph with no self-loops and no multiple edges

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

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

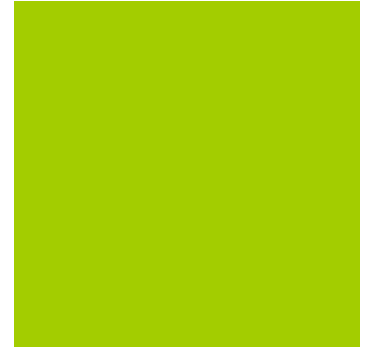
- What is the bound for a directed graph?

Main Methods of the Graph ADT



- Vertices and edges
 - are positions
 - store elements
- Accessor methods
 - `endVertices(e)`: an array of the two endvertices of `e`
 - `opposite(v, e)`: the vertex opposite of `v` on `e`
 - `areAdjacent(v, w)`: true iff `v` and `w` are adjacent
 - `replace(v, x)`: replace element at vertex `v` with `x`
 - `replace(e, x)`: replace element at edge `e` with `x`

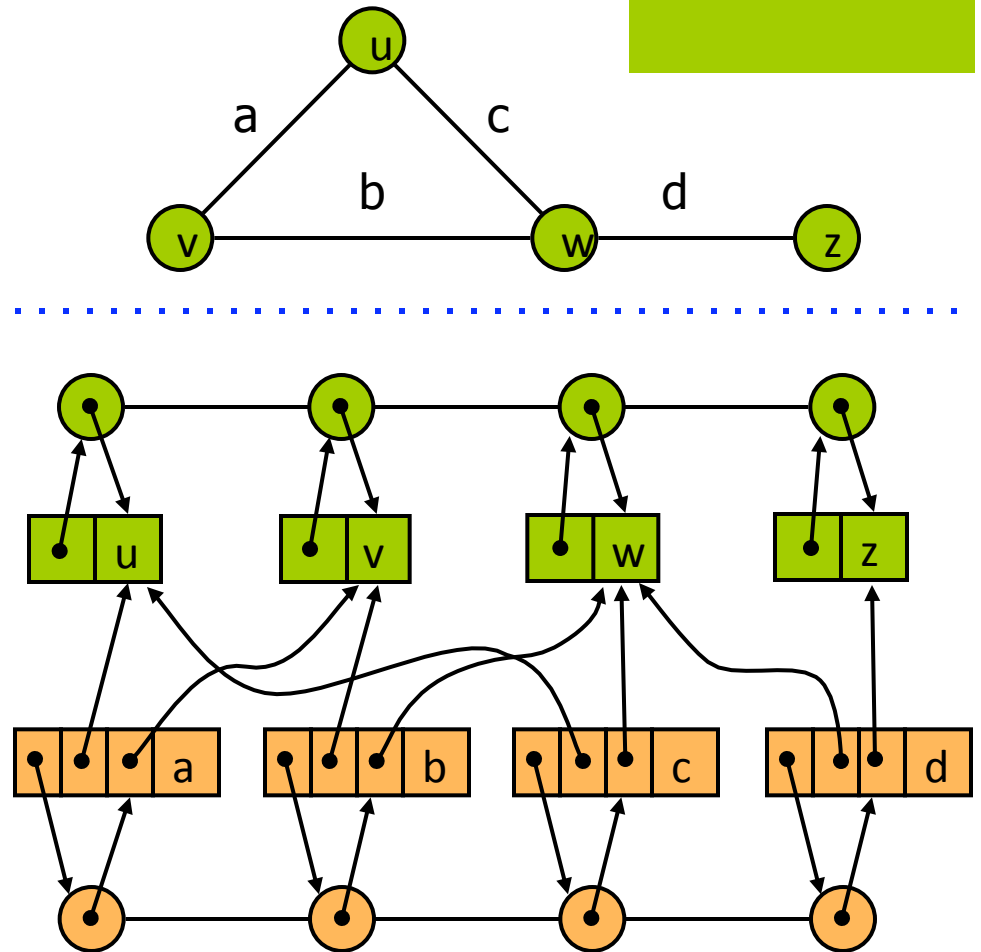
Main Methods of the Graph ADT



- Update methods
 - `insertVertex(o)`: insert a vertex storing element `o`
 - `insertEdge(v, w, o)`: insert an edge (v, w) storing element `o`
 - `removeVertex(v)`: remove vertex `v` (and its incident edges)
 - `removeEdge(e)`: remove edge `e`
- Iterable collection methods
 - `incidentEdges(v)`: edges incident to `v`
 - `vertices()`: all vertices in the graph
 - `edges()`: all edges in the graph

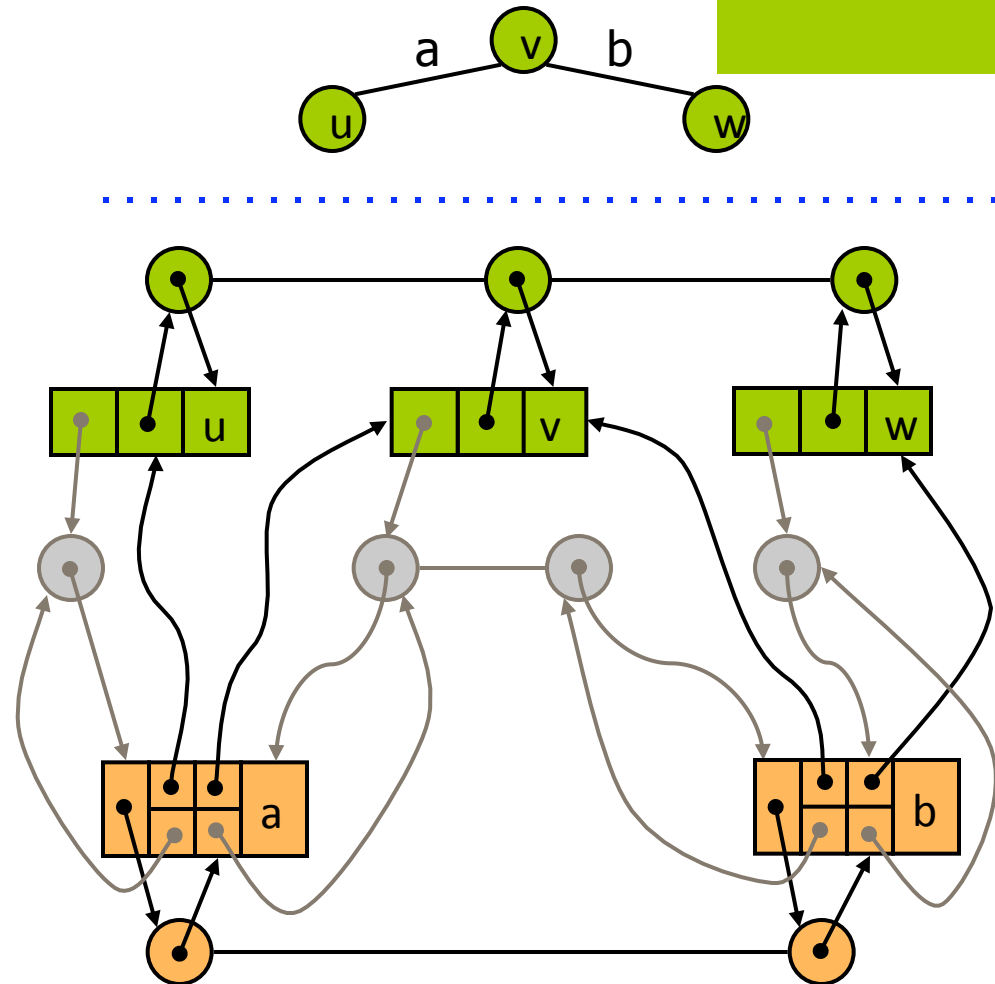
Edge List Structure

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



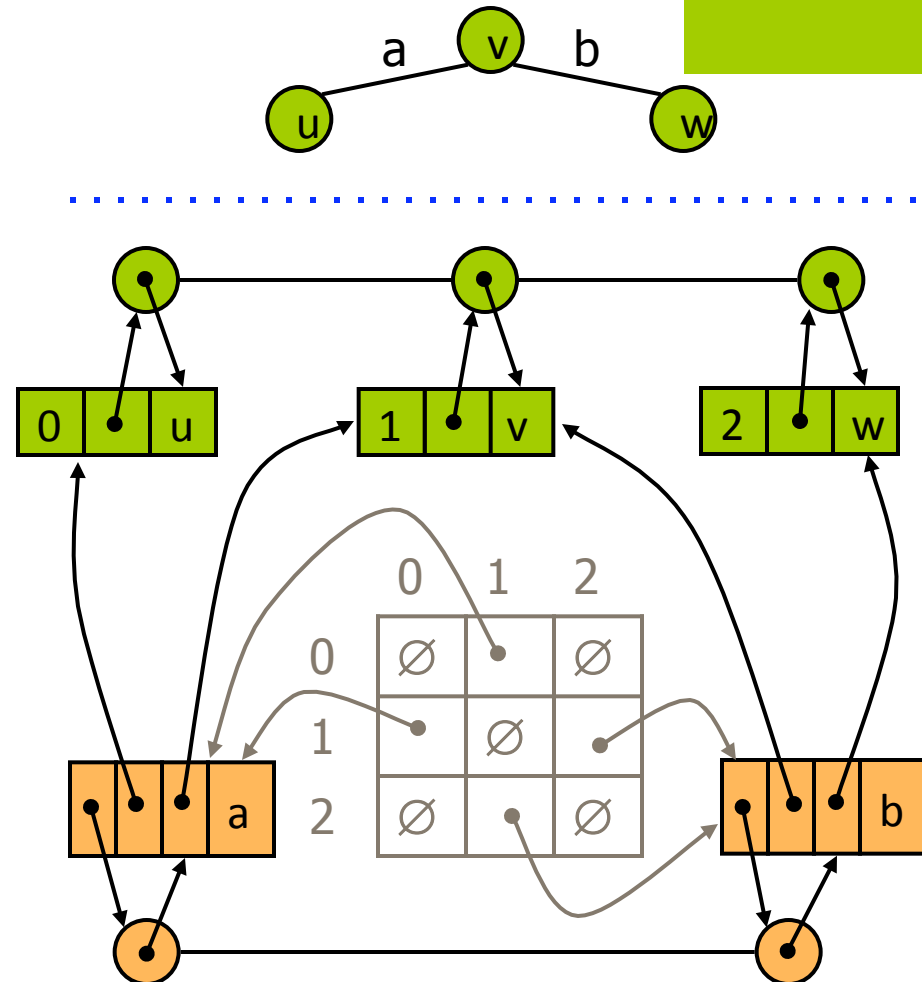
Adjacency List Structure

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



Adjacency Matrix Structure

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

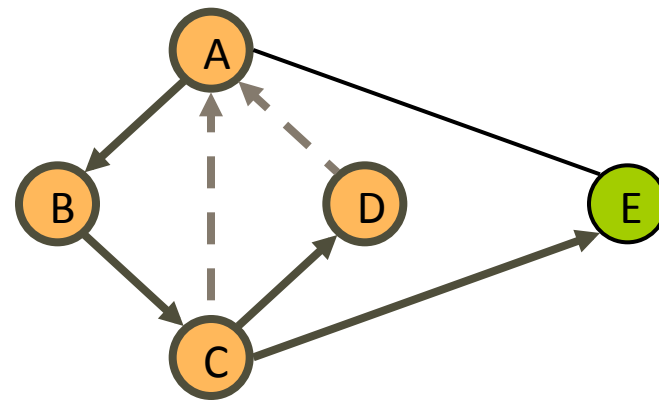


Performance

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops 	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Graph Traversal

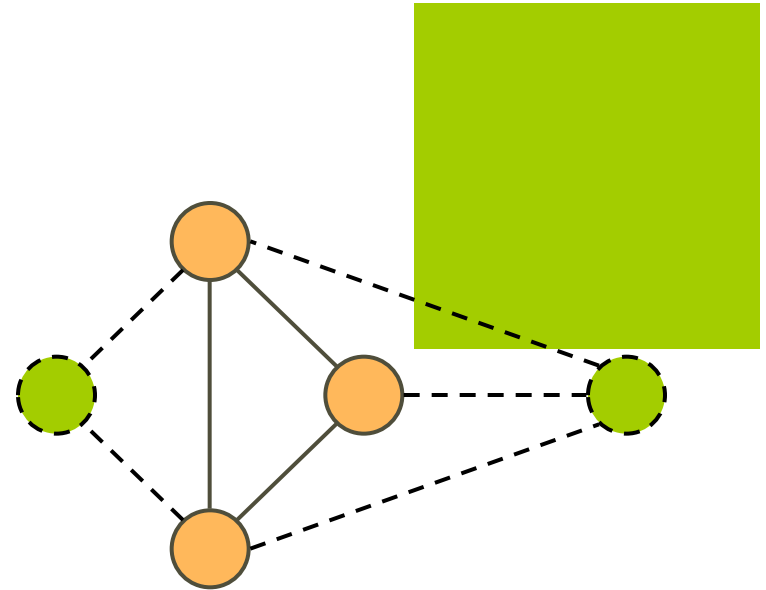
- How to visit all vertices?



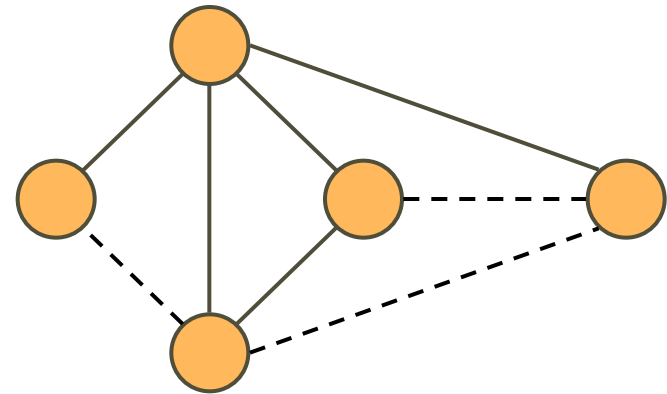
Depth-First Search

Subgraphs

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



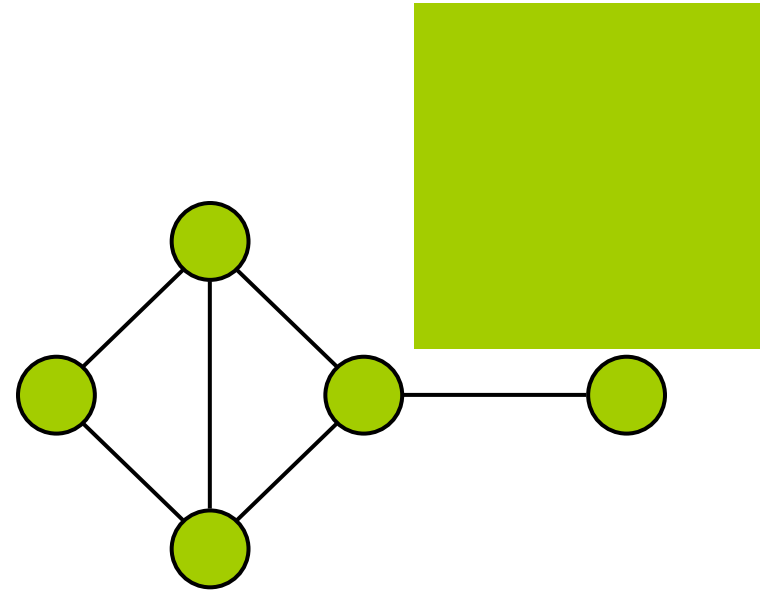
Subgraph



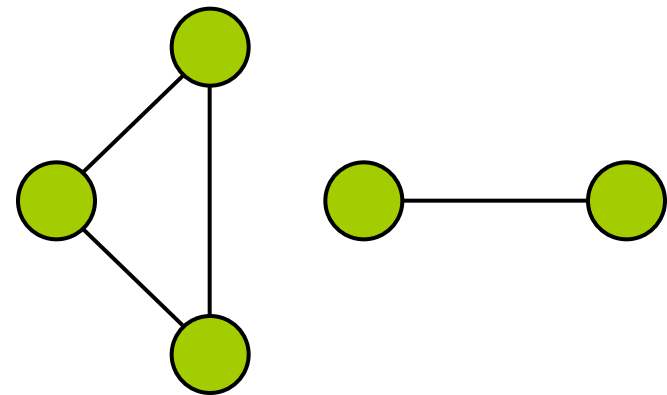
Spanning subgraph

Connectivity

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



Connected graph



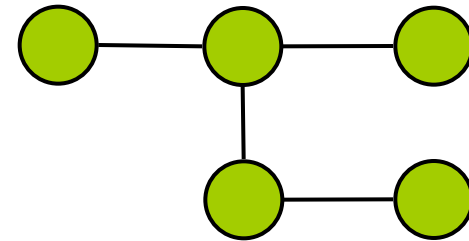
Non connected graph with two connected components

Trees and Forests

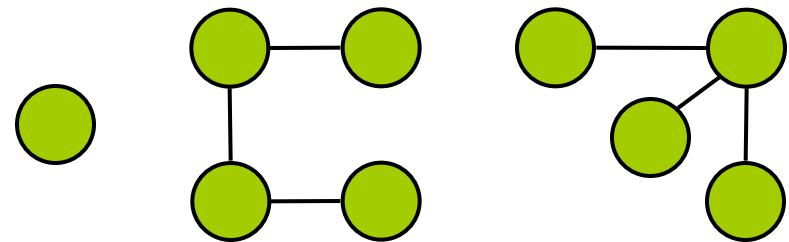
- A (free) tree is an undirected graph T such that
 - T is connected
 - T has no cycles

This definition of tree is different from the one of a rooted tree

- A forest is an undirected graph without cycles
- The connected components of a forest are trees



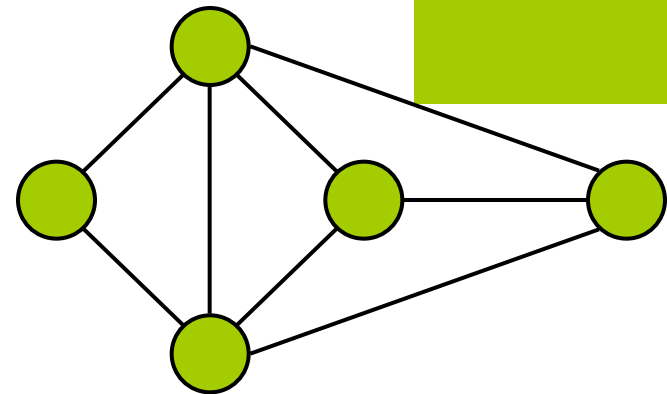
Tree



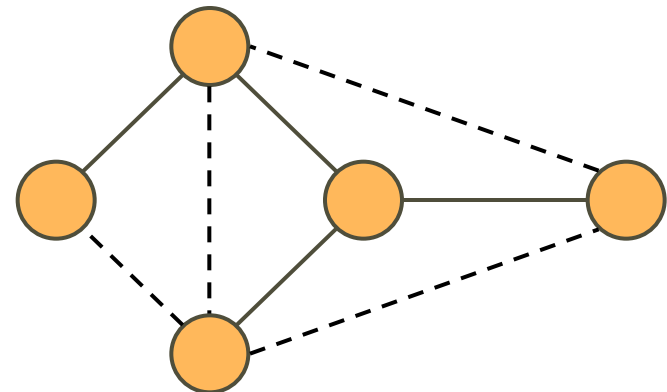
Forest

Spanning Trees and Forests

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



Graph



Spanning tree

Depth-First Search



- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
 - Visits all the vertices and edges of G
 - Determines whether G is connected
 - Computes the connected components of G
 - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
 - Find and report a path between two given vertices
 - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm

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

Algorithm *DFS*(*G*)

Input graph *G*

Output labeling of the edges of *G*
as discovery edges and
back edges

```
for all u ∈ 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*

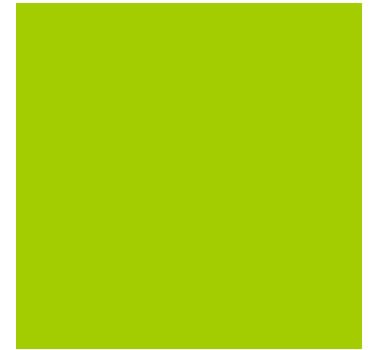
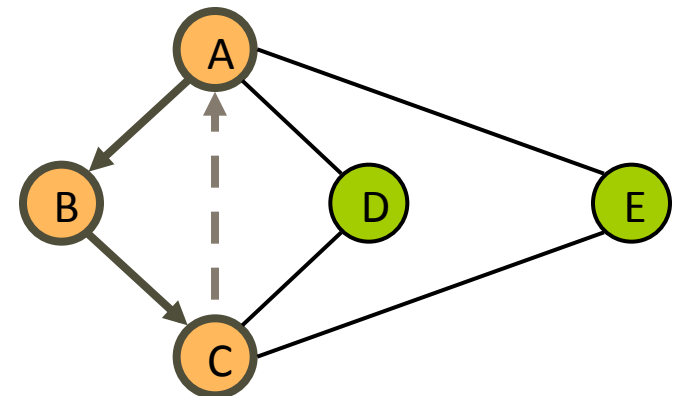
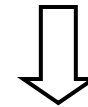
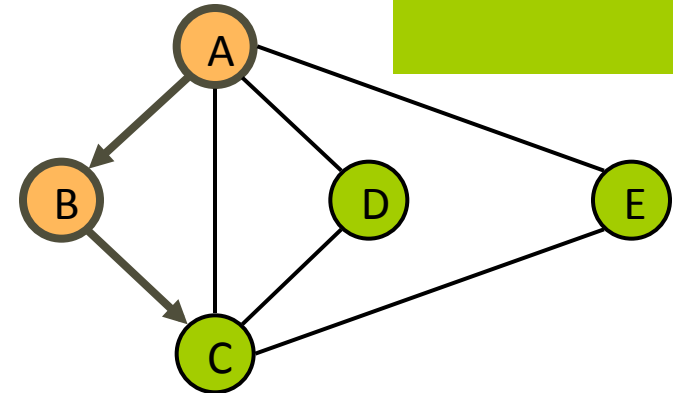
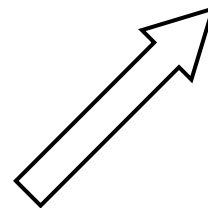
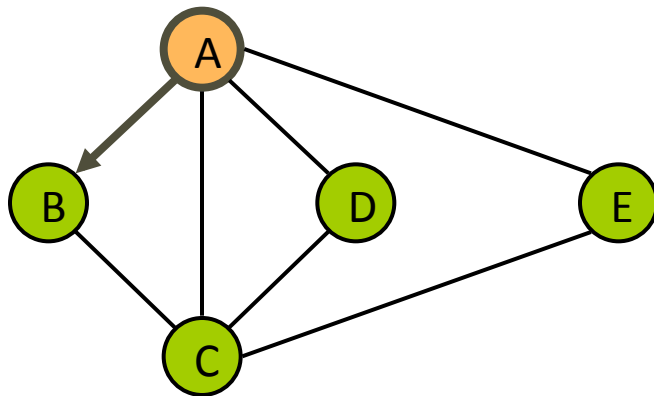
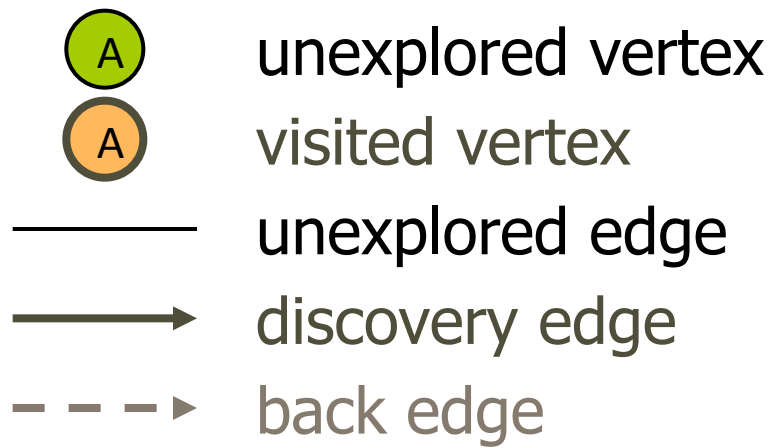
setLabel(*e*, *DISCOVERY*)

DFS(*G*, *w*)

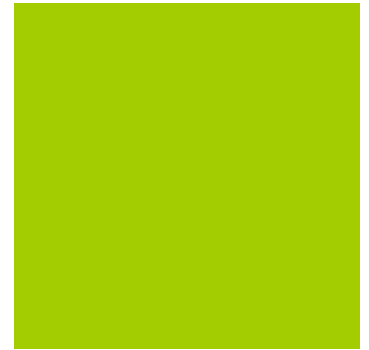
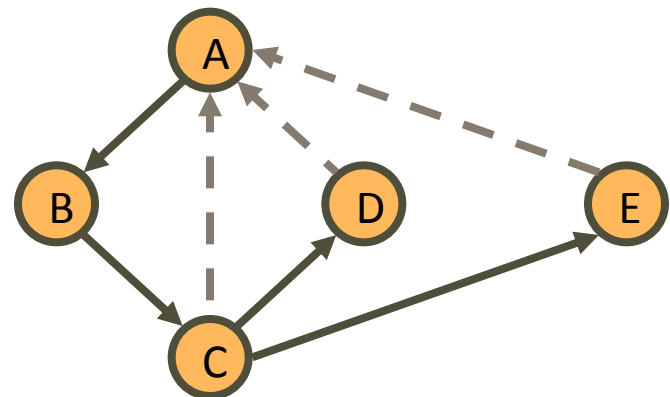
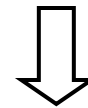
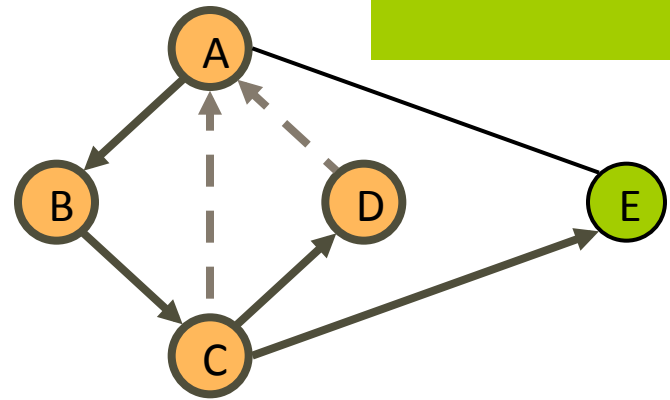
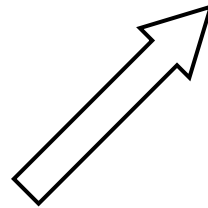
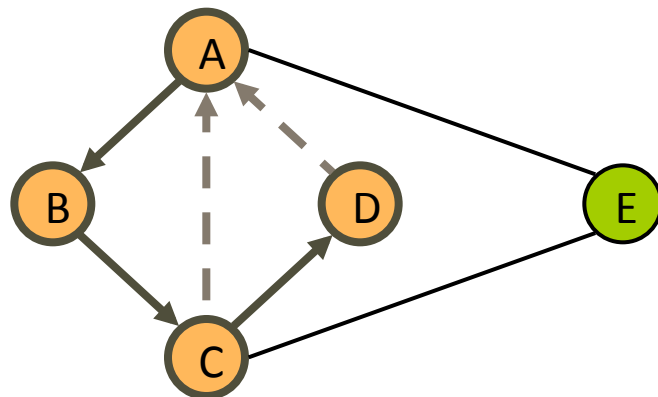
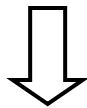
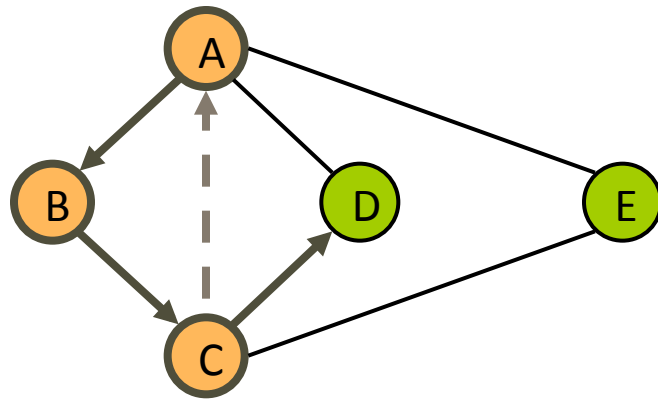
else

setLabel(*e*, *BACK*)

Example

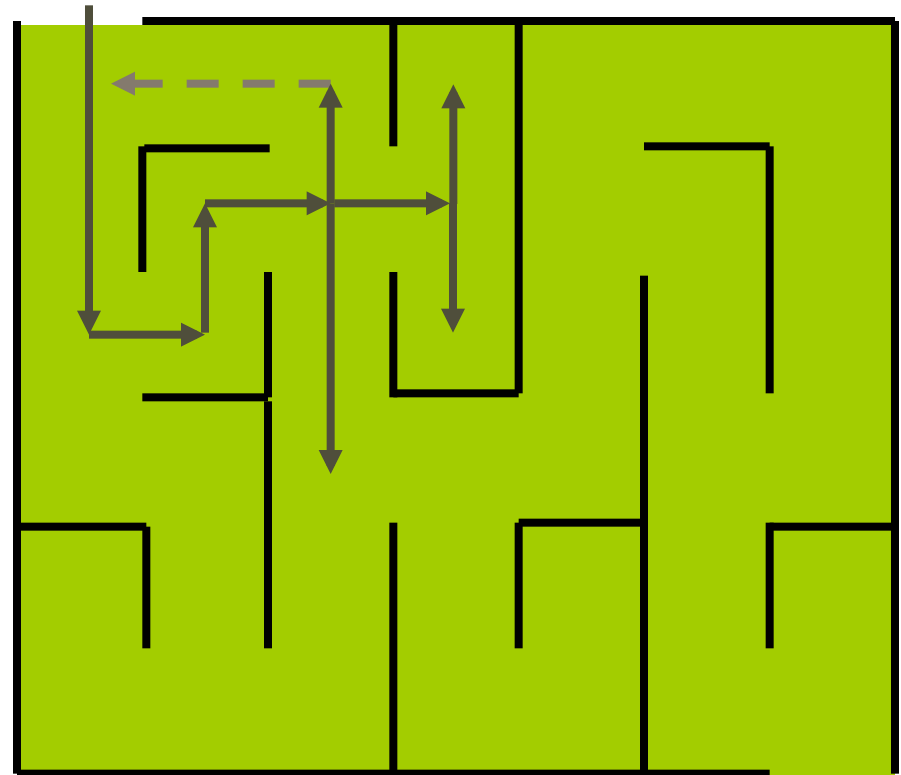


Example (cont.)



1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 2680, 26

-



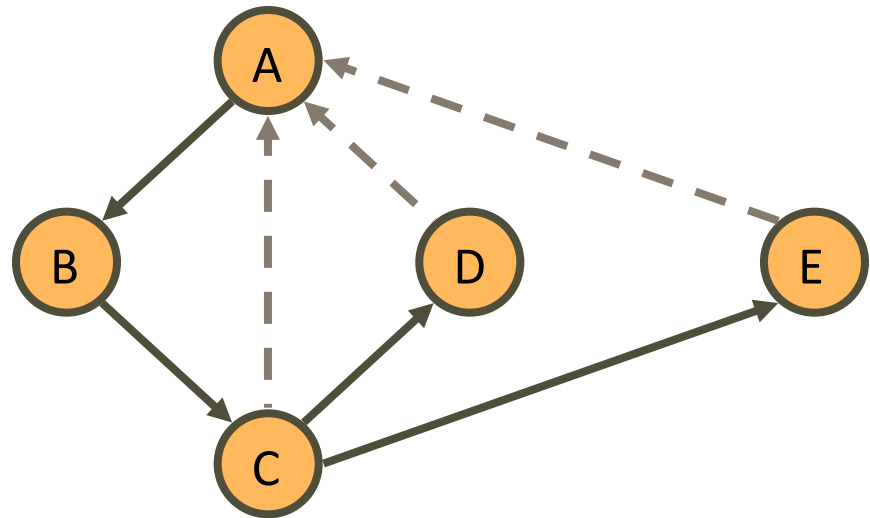
Properties of DFS

Property 1

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

Property 2

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



Analysis of DFS

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

Path Finding

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

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

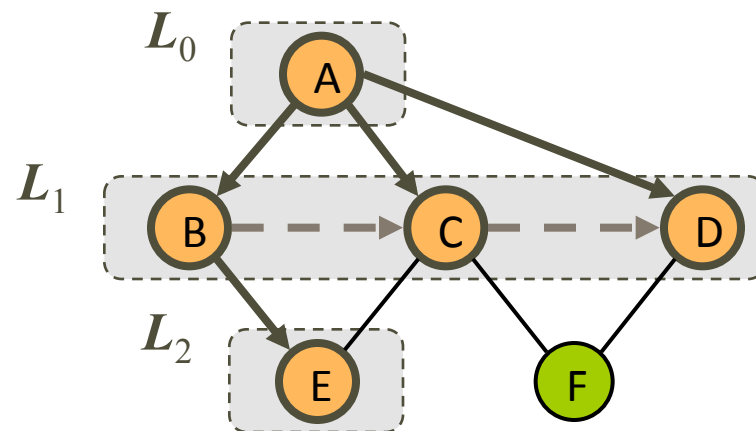
Cycle Finding

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

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

Breadth-First Search

- Traverse the graph level by level



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

BFS Algorithm

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

Algorithm *BFS*(*G*)

Input graph *G*

Output labeling of the edges
and partition of the
vertices of *G*

```
for all u ∈ G.vertices()  
    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.*addLast*(*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*)

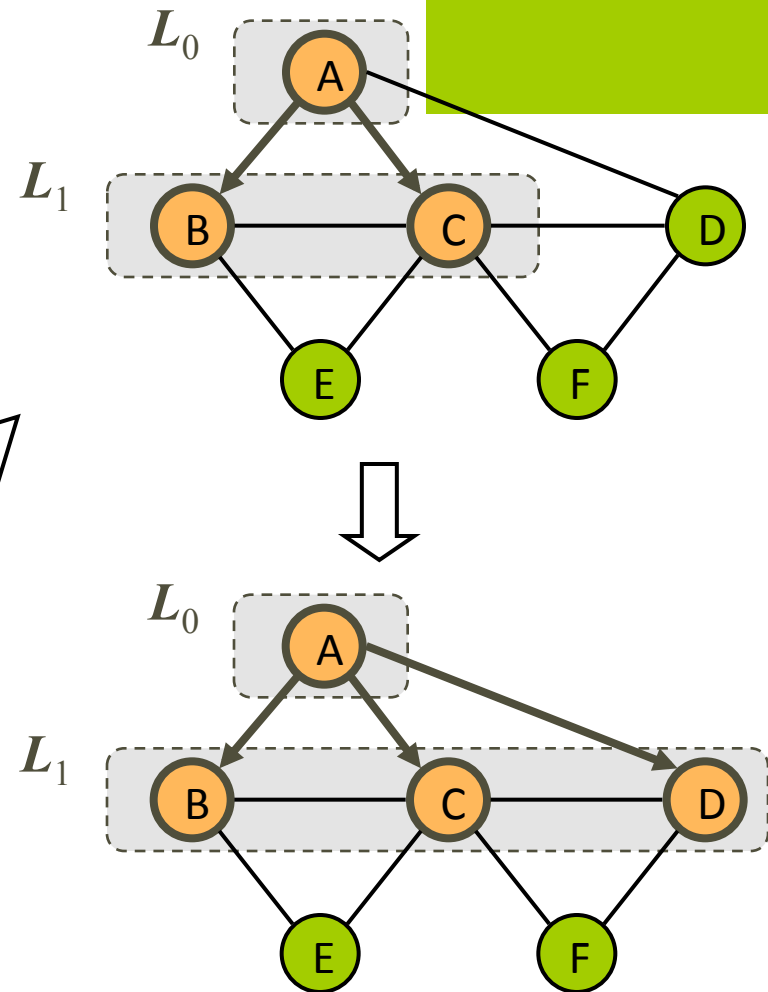
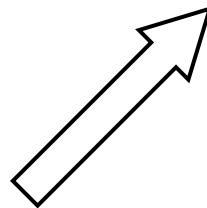
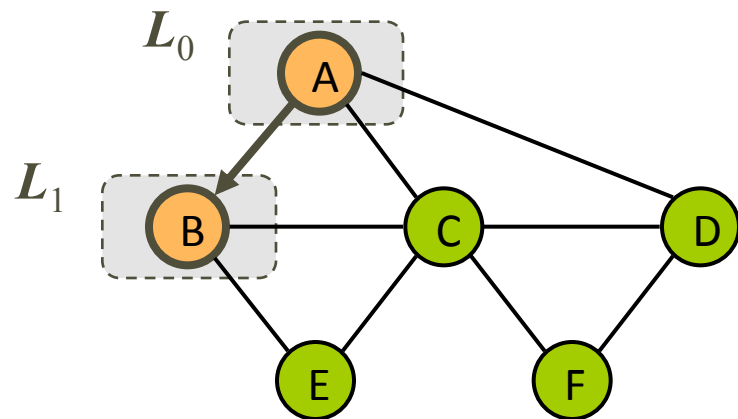
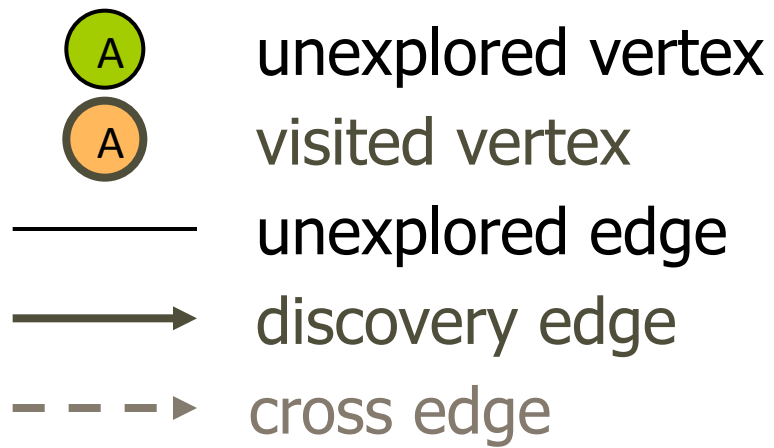
*L*_{*i*+1}.*addLast*(*w*)

else

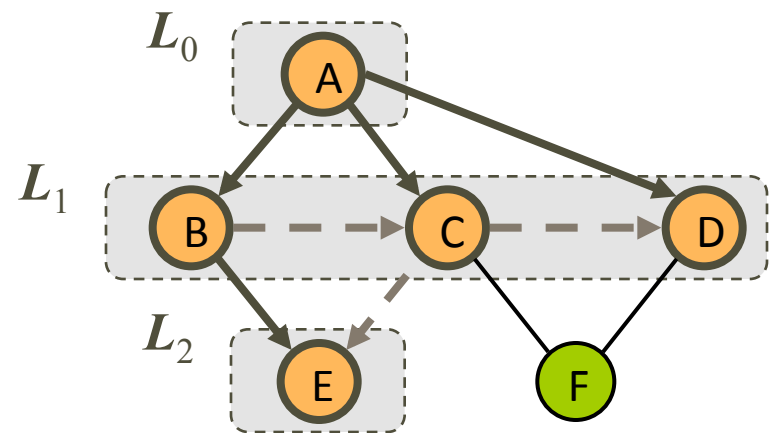
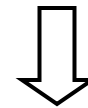
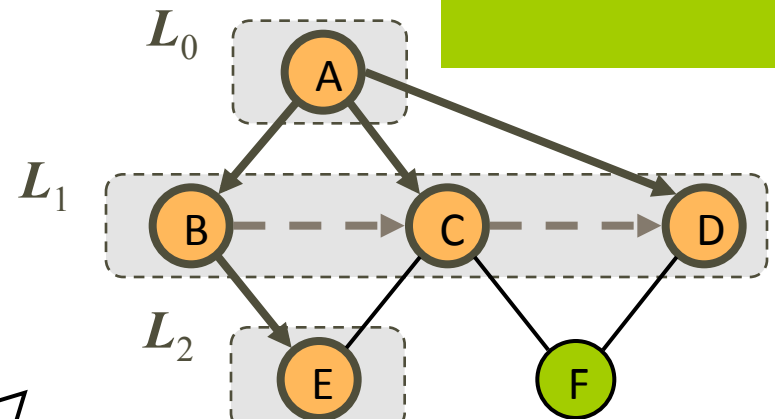
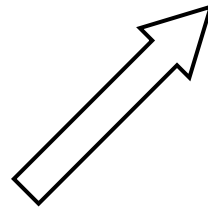
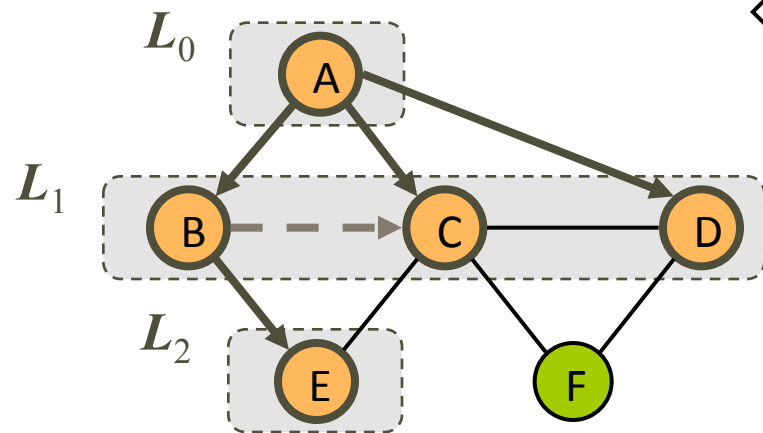
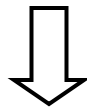
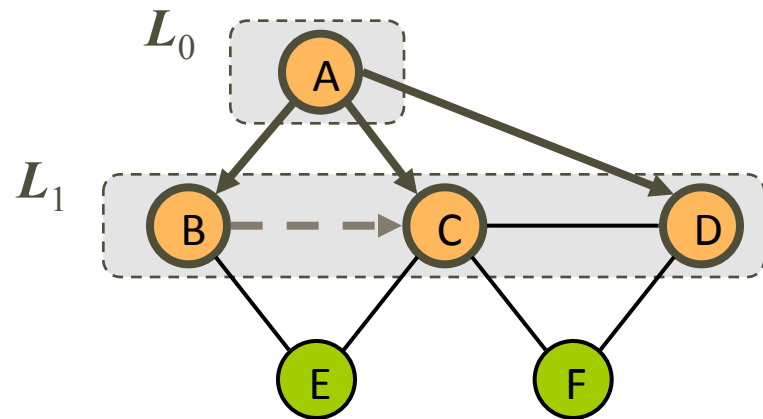
setLabel(*e*, *CROSS*)

i ← *i* + 1

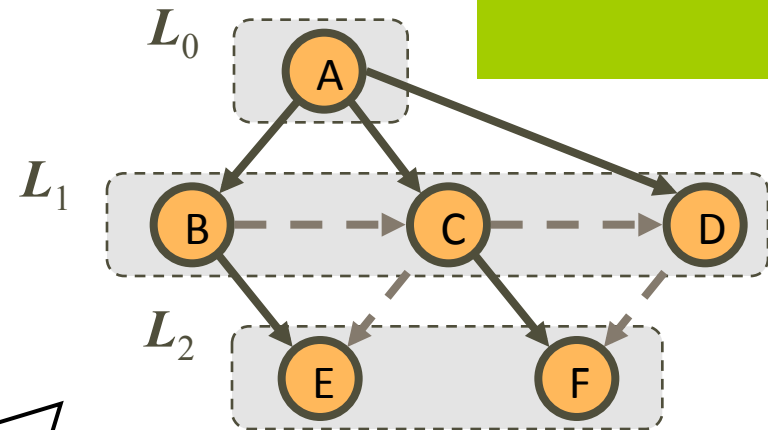
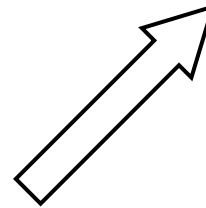
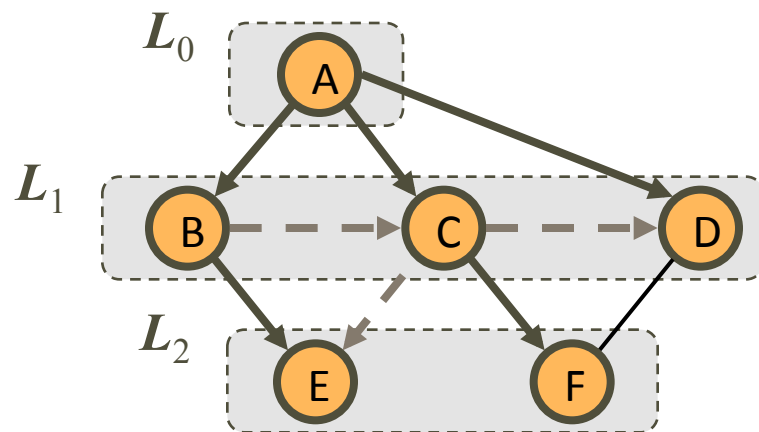
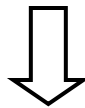
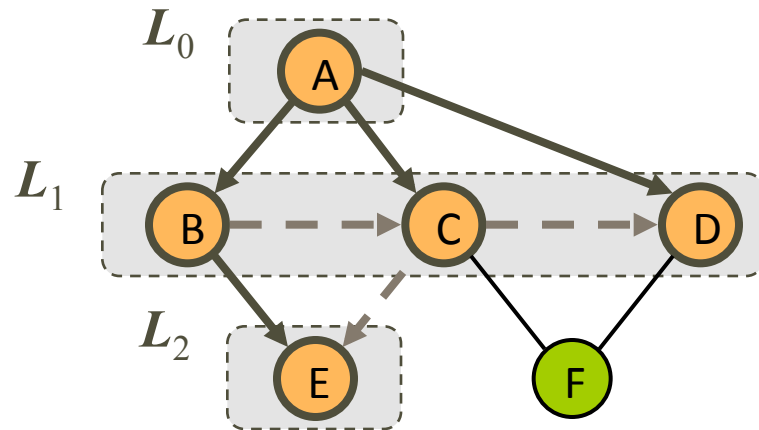
Example



Example (cont.)



Example (cont.)



Properties

Notation

G_s : connected component of s

Property 1

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

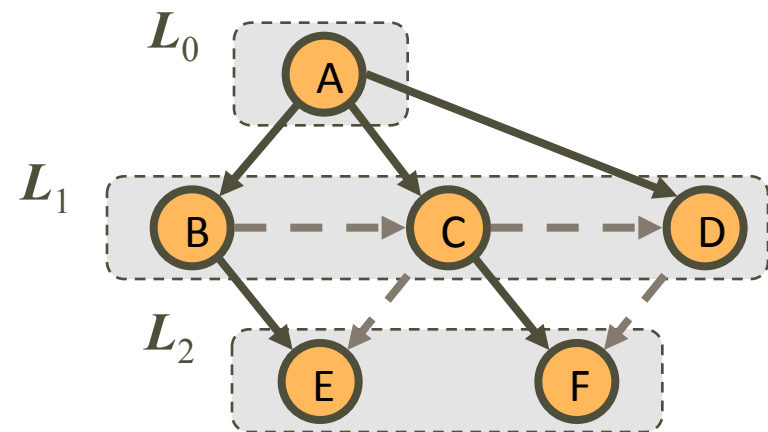
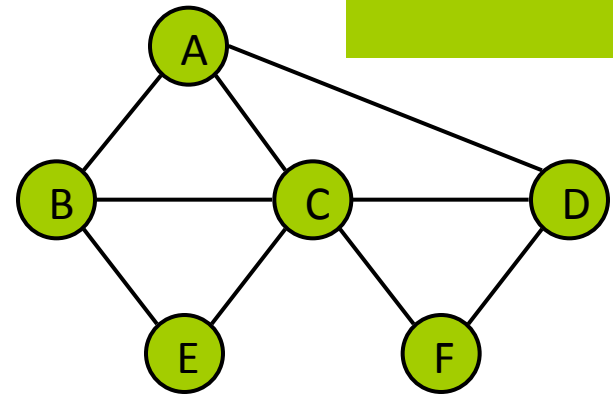
Property 2

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

Property 3

For each vertex v in L_i

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



Analysis

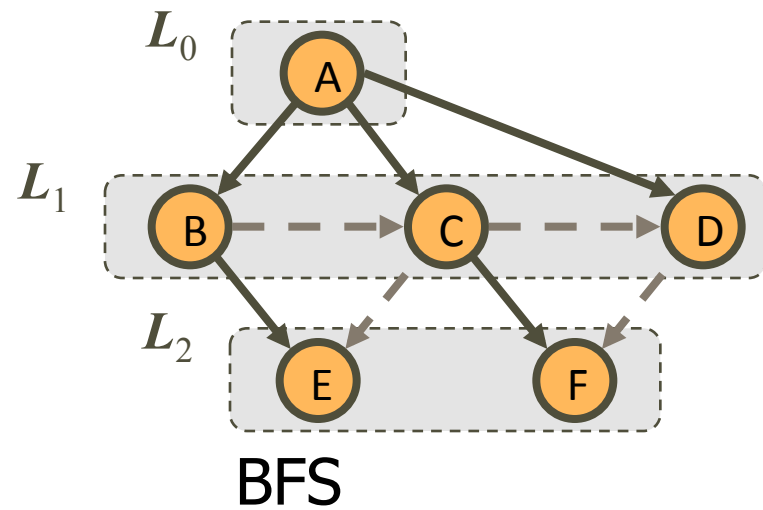
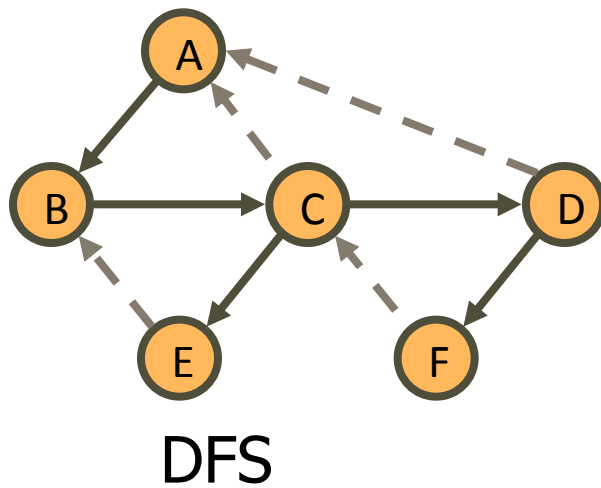
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
 - once as UNEXPLORED
 - once as VISITED
- Each edge is labeled twice
 - once as UNEXPLORED
 - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for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

Applications

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

DFS vs. BFS

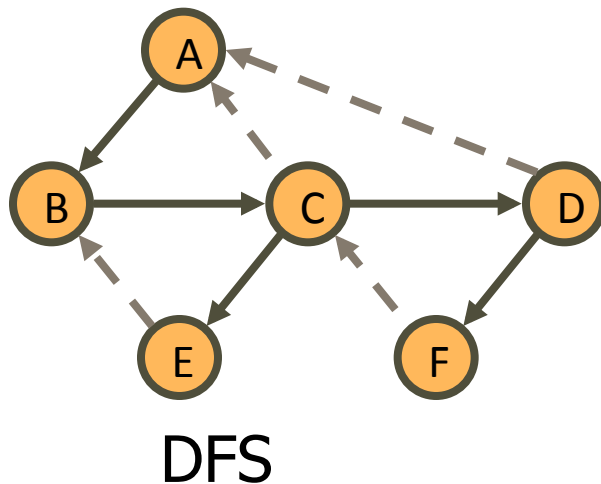
Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	



DFS vs. BFS (cont.)

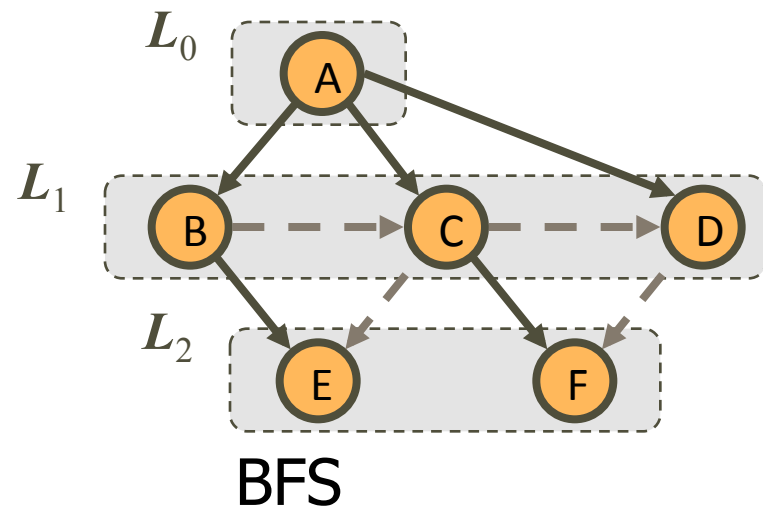
Back edge (v, w)

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



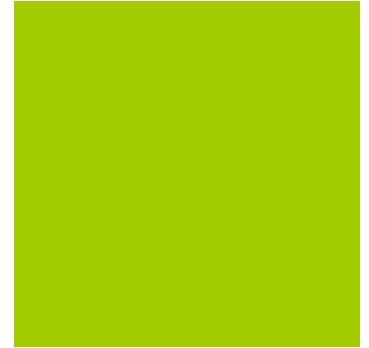
Cross edge (v, w)

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



Java Graph Library

- No standard library
- JGraphT
 - An open source library
 - <http://www.jgrapht.org/>
 - Supports most mentioned Graph functions
 - You can simply download the file and use the library to create your graph



No Homework

- Project demo on Dec. 23

Smart Ranking:

- Stage 1 : Rank your web pages by keywords
- Stage 2 : Rank your websites by keywords
- Stage 3 : Re-rank google websites by keywords
- Stage 4 : Derive relative keywords by top-ranked websites

Schedule

- Each team gives 10 minutes PPT presentation focusing on the project interests, key ideas, and achievements + 10 minutes system demo
- Lets draw the schedule:
- Dec. 23

	I	II	III
9:00~10:00			
10:00~11:00			
11:00~12:00			