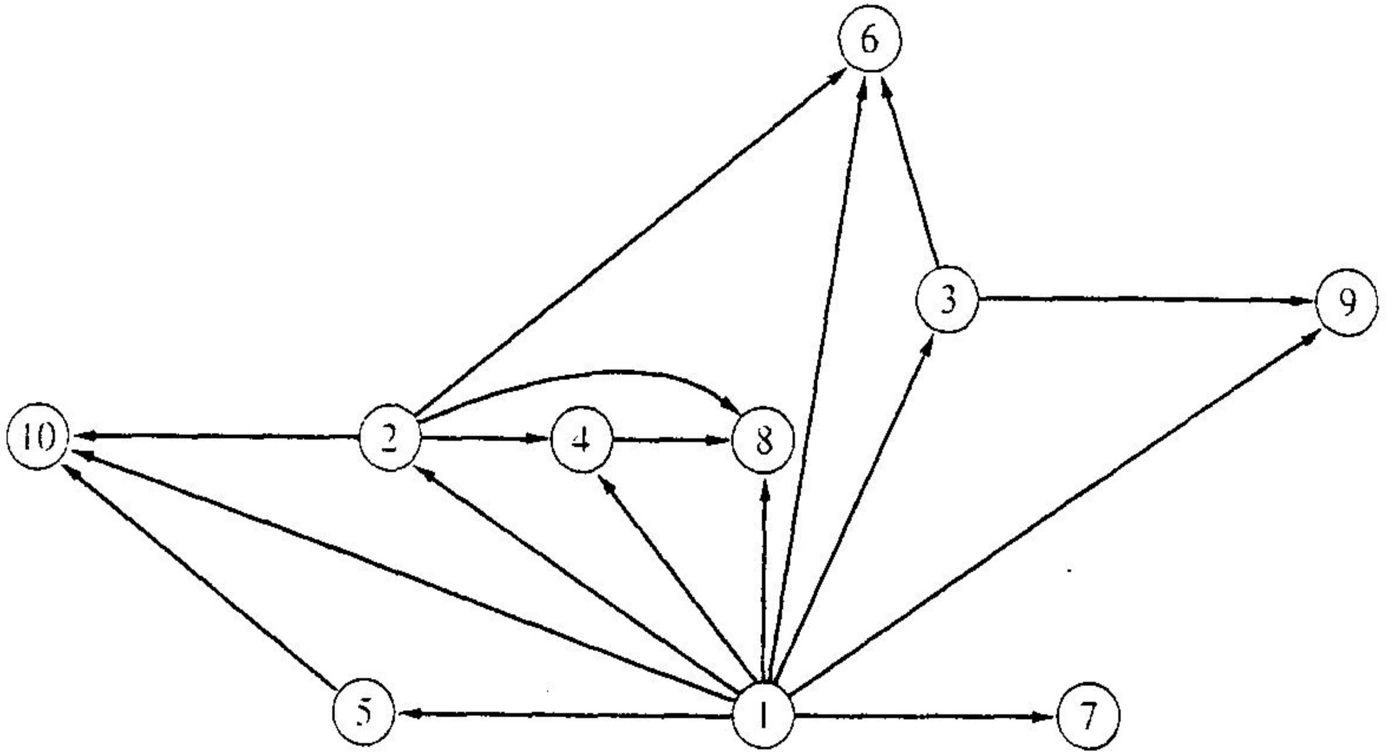
CS4102 Algorithms
Spring 2022

Warm up

If the "nodes" here represent whole numbers, when is there a connection from value *i* to value *j* ?



Graphs – Basic Review, BFS, and Intro. to DFS

Tom Horton, Robbie Hott

CLRS Chapter 22.1 and 22.2

Definition: Directed graph

- **Directed Graph**
  - A *directed graph*, or *digraph*, is a pair
  - G = (V, E)
  - where V is a set whose elements are called *vertices*, and
  - E is a set of *ordered pairs* of elements of V.

    - Vertices are often also called *nodes*.
    - Elements of E are called edges, or directed edges, or arcs.
    - For directed edge *(v, w)* in E, *v* is its *tail* and *w* its *head*;
    - *(v, w)* is represented in the diagrams as the arrow, *v -> w*.
    - In text we often simply write *vw*.
    - *vw* is said to be *incident* upon the vertices v and w

Definition: Undirected graph

- **Undirected Graph**
  - A undirected graph is a pair
  - G = (V, E)
  - where V is a set whose elements are called vertices, and
  - E is a set of ***unordered* pairs of *distinct*** elements of V.
    - Vertices are often also called nodes.
    - Elements of E are called edges, or undirected edges.
    - Each edge may be considered as a subset of V containing two elements,
    - *{v, w}* denotes an undirected edge
    - In diagrams this edge is the line *v*---*w*.
    - In text we simple write *vw*, or *wv*
    - *vw* is said to be *incident* upon the vertices v and w

Terms You Should Know

- **Edge Weights:** A graphs can be
  - Weighted or not weighted
    - Every edge has a numeric weight value. (Weights can be reals, integers, etc.)
    - Weight often represent things like cost, length, distance, capacity,…
    - Often indicated by *wt(vw) or W(vw)* for edge *vw*, or *wt(e)* for edge e (or sometimes *e.wt*)
  - Directed or undirected graphs can be weighted or unweighted

- **Degree of a vertex:** how many adjacent vertices
  - Digraph: **in-degree** (num. of incoming edges) vs. **out-degree**

- Can an edge connect a vertex to itself?
  - Undirected graphs: Normally an edge can't connect a vertex to itself
    - If so, we call this a *multigraph* and these are *parallel edges*
  - Directed graph (digraph): an edge may connect a vertex to itself

Formal Definition: Weighted Graph

A weighted graph is a triple (V, E, W)

- where (V, E) is a graph (directed or undirected) and
- W is a function from E into R, the reals (or integers or rational numbers).
- For an edge e, W(e) is called the weight of e.

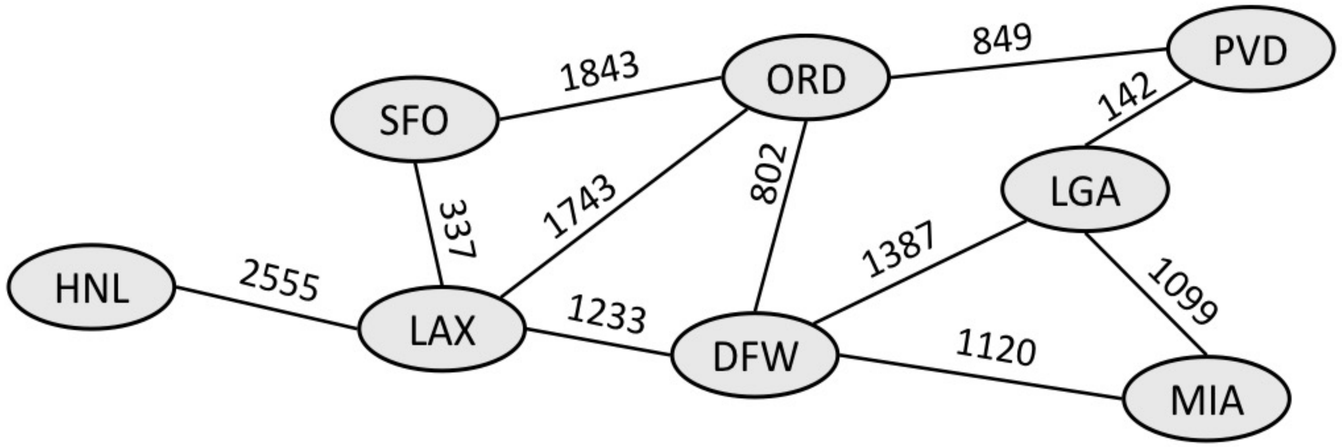


Image from Stanford's cs106b course website

Terms You Should Know or Learn Now

- **Size of graph?** Two measures:
  - Number of nodes: usually ‘V’ (some people use *n*)
    - Note: we really mean |V|, size of set V. But we drop the | | here.
  - Number of edges: usually ‘E’ (some people use *m*)
- **Dense graph:** many edges
  - Maximally dense?
  - Undirected: each node connects to all others, so
    E = V(V-1)/2
    Called a *complete graph*
  - Directed: E = V(V-1) *why?*
- **Sparse graph:** fewer edges
  - Could be zero edges…

Terms You Should Know or Learn Now

- Path and simple path
  - One vertex is *reachable* from another vertex
  - ”Simple” means path contains an edge at most once
- A *connected graph*
  - undirected graph, where each vertex is reachable from all others
- A *strongly connected* *di*<u>*di*</u>*graph:*
  - direction affects what this means!
  - node u may be reachable from v, but not v from u
  - Strongly connected means both directions
- If an undirected graph is not connected, then connected subgraphs are called *connected components*

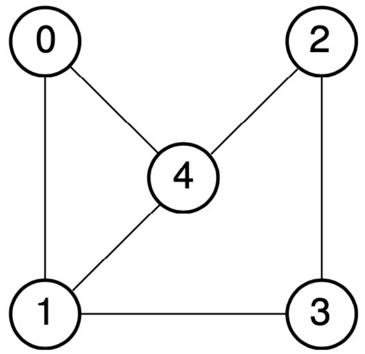
Terms You Should Know or Learn Now

- **Cycle**
  - Directed graph: non-empty path with same starting and ending node
  - An edge may appear more than once (but why?)
    - **Simple cycle**: no node repeated except start and end
  - Undirected graph: same idea
    - If an edge appears more than once (I.e. non-simple) then we traverse it in the same direction
- **Acyclic**: no-cycles
- A connected, acyclic undirected graph: *free tree*
  - If we specify a root, it's a *rooted tree*
  - Acyclic but not connected? an undirected *forest*
- **Directed acyclic graph:** a DAG

Self-test: Understand these Terms?

- Subgraph
- Symmetric digraph
- complete graph
- Adjacency relation
- Path, simple path, reachable
- Connected, Strongly Connected
- Cycle, simple cycle
- acyclic
- undirected forest
- free tree, undirected tree
- rooted tree
- Connected component

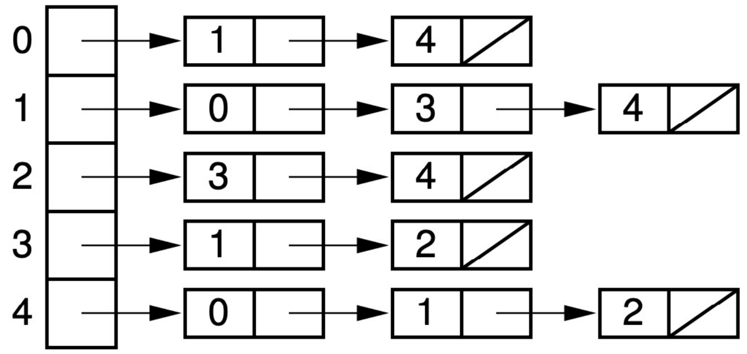
Data Structures for Undirected Graphs



(a)

$$\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & & 1 & & & 1 \\
1 & 1 & & & 1 & 1 \\
2 & & & & 1 & 1 \\
3 & & 1 & 1 & & \\
4 & 1 & 1 & 1 & & \\
\hline
\end{array}$$

(b)



(c)

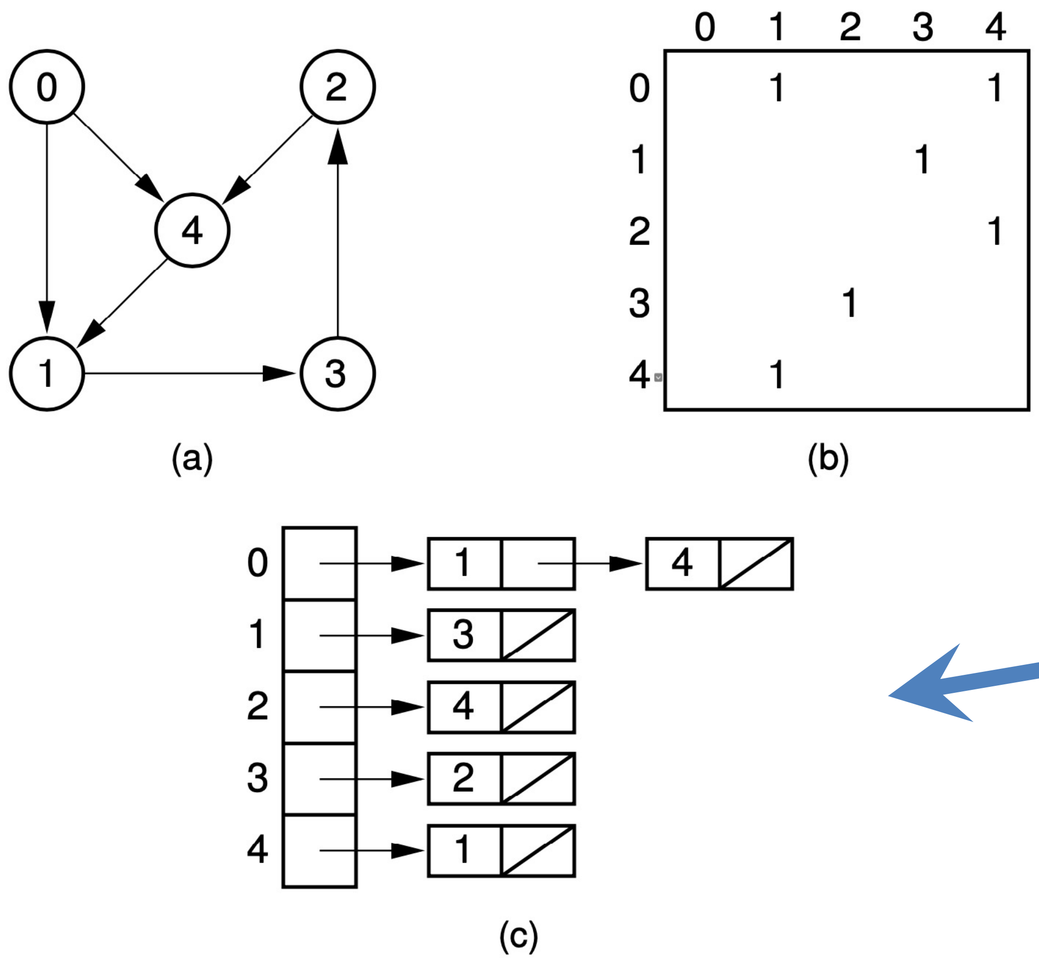
Adjacency Matrix:
A[u][v] is 1 if edge *(u,v)* exists.
Note symmetrical around diagonal. Could just store info in one half of matrix.

Adjacency List:
Note each edge *(u,v)* has an edge-node on *u*'s list and also *v*'s list.

Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Image of diagrams from https://people.cs.vt.edu/~shaffer/Book/

Data Structures for Digraphs



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

Figure 11.3 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Adjacency Matrix:
Not symmetrical around diagonal for digraph.

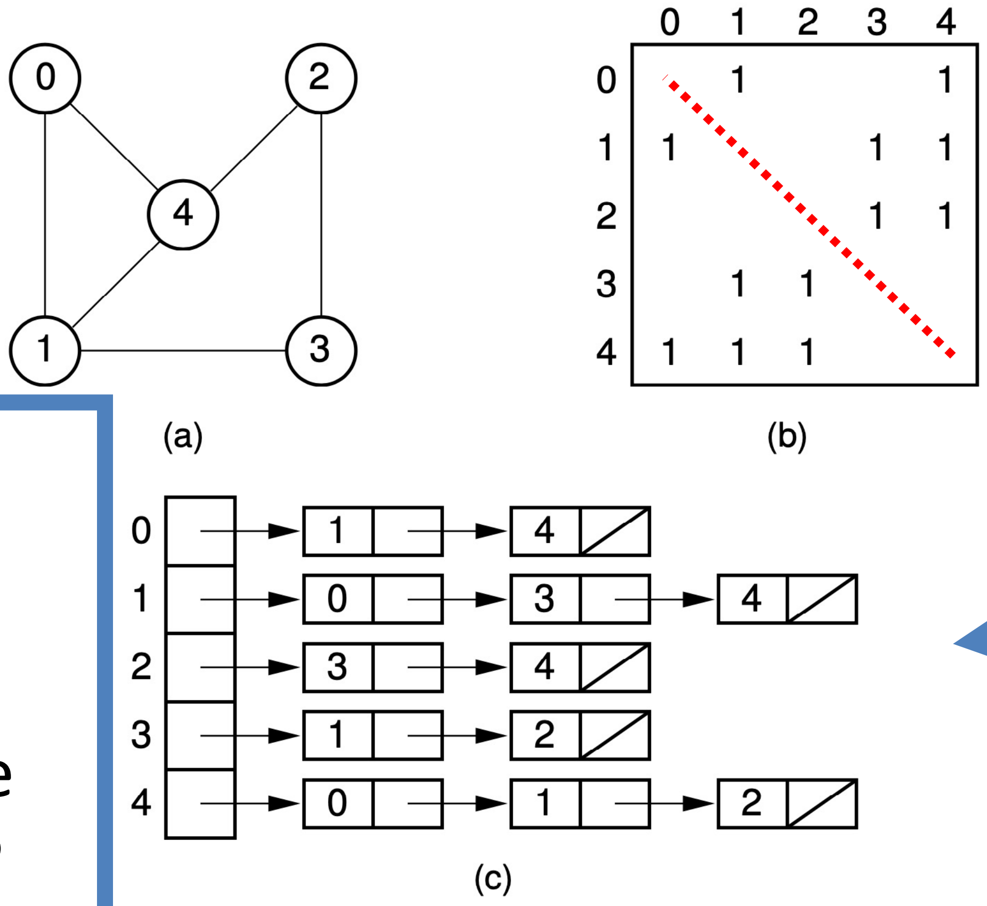
Adjacency List:
Note each directed edge *(u,v)* has an edge-node on just one vertex's list.

Image of diagrams from https://people.cs.vt.edu/~shaffer/Book/

Data Structures for Weighted Graphs

Images are of **unweighted** graphs.

How would we store weights?



Adjacency Matrix:
Store weight (u,v) in matrix cell. Use 0 or negative value if edge not in graph.

Adjacency List:
Add a field to the the edge node object to store the weight.

Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Image of diagrams from https://people.cs.vt.edu/~shaffer/Book/

Identifying Vertices as Strings

Vertices may be identified with strings not integers.

(1) Could use an adjacency map instead of an adjacency list, and also store strings in edge-nodes

(2) Programmers often have an index and/or lookup table to convert between int's and string IDs for vertices. Understand the example here?

There are other ways to do this. Use your programming skills!

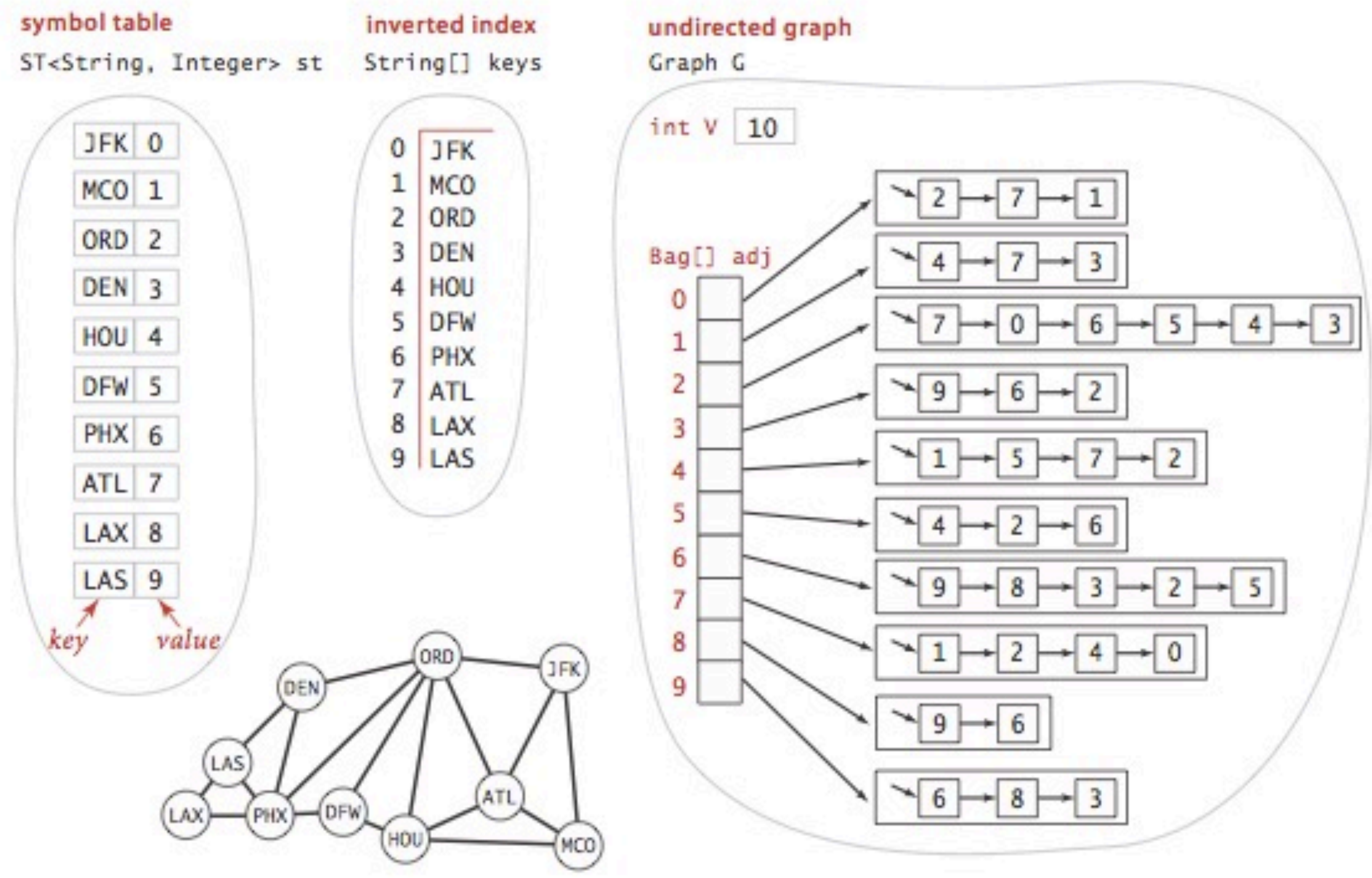


Image from https://algs4.cs.princeton.edu/home/

Breadth-First Search

Traversing Graphs

- "Traversing" means processing each vertex edge in some organized fashion by following edges between vertices
  - We speak of *visiting* a vertex. Might do something while there.
- Recall traversal of binary trees:
  - Several strategies: In-order, pre-order, post-order
  - Traversal strategy implies an <u>order</u> of visits
  - We used recursion to describe and implement these
- Graphs can be used to model interesting, complex relationships
  - Often traversal used just to process the set of vertices or edges
  - Sometimes traversal can identify interesting properties of the graph
  - Sometimes traversal (perhaps modified, enhanced) can answer interesting questions about the problem-instance that the graph models

BFS: Overall Strategy

- Breadth-first search’s strategy
  - Choose a starting vertex
  - Vertices are visited in order of increasing distance from the starting vertex
    - For starting vertex, distance d = 0
  - Examine all edges leading from vertices (at distance d) to adjacent vertices (at distance d+1)
  - Then, examine all edges leading from vertices at distance d+1 to distance d+2, and so on,
  - Until no new vertex is discovered

BFS: Specific Input/Output

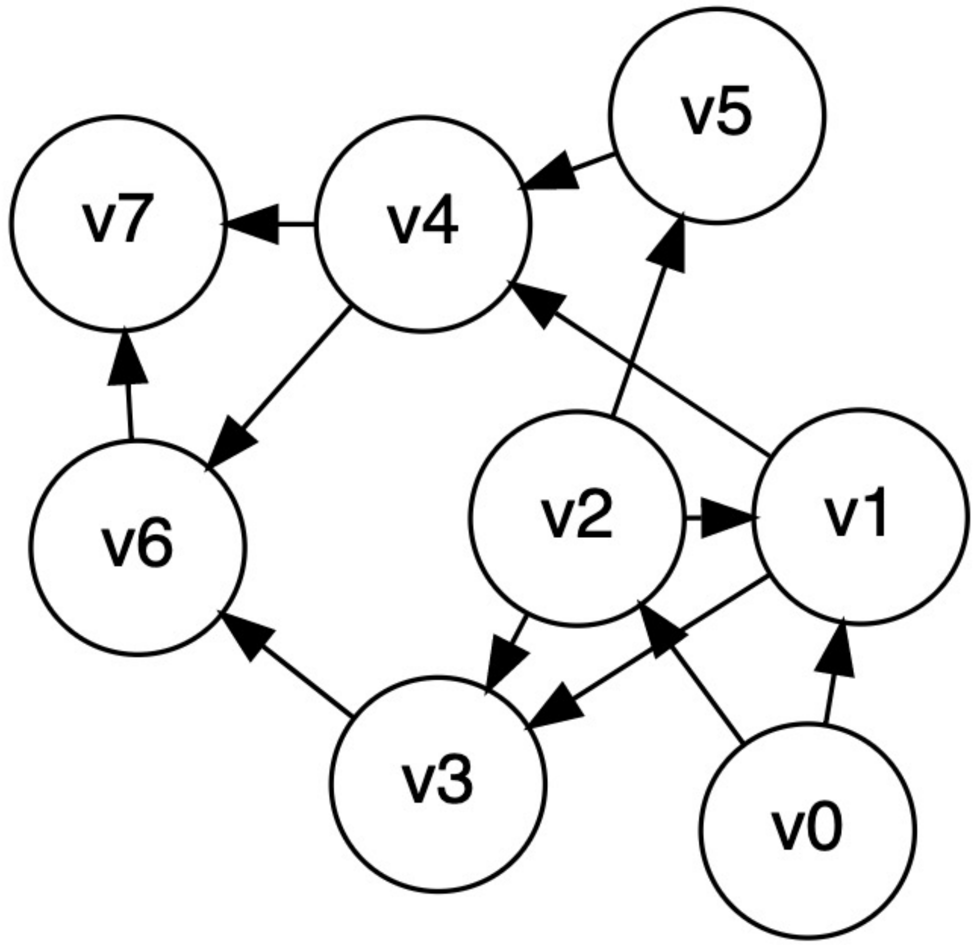
- Input:
  - A graph ***G***
  - single start vertex ***s***

- Output:
  - Distance from ***s*** to each node in ***G*** (distance = number of edges)
  - Breadth-First Tree of ***G*** with root ***s***

- **Important:** The paths in this BFS tree represent the **shortest paths** from s to each node in G
  - But edge weight's (if any) not used, so "short" is in terms of number of edges in path

Breadth-first search, quick example

- Let’s start at V0



Breadth-first search implementation

```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in G.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)
    $u.color$ = BLACK
```

- From CLRS
- Vertices here have some properties:
  - *color = white/gray/black*
  - *d = distance from start node*
  - *pi = parent in tree, i.e. v.pi is vertex by which v was* connected to BFS tree
- Color meanings here:
  - White: haven't seen this vertex yet
  - Gray: vertex has been seen and added to the queue for processing later
  - Black: vertex has been removed from queue and its neighbors seen and added to the queue

Breadth-first search: Analysis

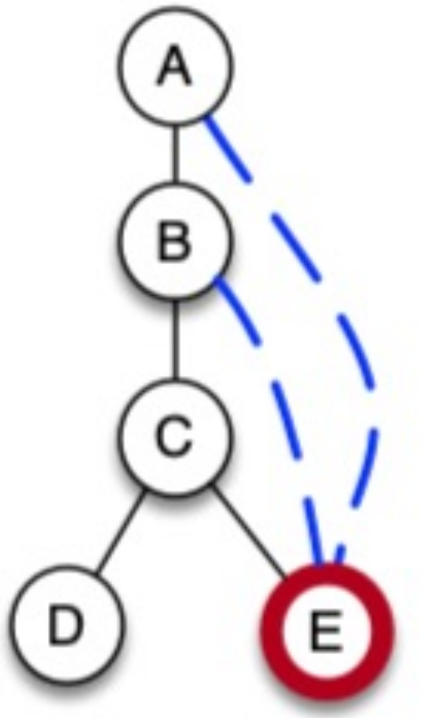
- For a digraph having V vertices and E edges
  - Each edge is processed once in the while loop for a cost of $\theta(E)$
  - Each vertex is put into the queue once and removed from the queue and processed once, for a cost $\theta(V)$
  - Total **time-complexity**: $\theta(V+E)$
    - For graph algorithm’s this is called “linear”
  - **Space complexity**: extra space is used for color array and queue, so $\theta(V)$

BFS Can be Useful

- Not all vertices are necessarily reachable from a selected starting vertex
  - E.g. for an undirected graph that's not connected
  - Could you use BFS to test if an undirected graph is connected? Count connected components?
- Earlier we said: The paths in this BFS tree represent the **shortest paths** from s to each node in G
  - The path in the tree from start vertex *s* to any vertex *v* contains the *minimum* number of edges of any path from *s* to *v*
  - BFS is the choice for "shortest path" problems where distance is in terms of edges or connections, and not edge-weights
- Next let's make an argument to justify this property of BFS.

Informal Argument on BFS Trees

- Assume there exists a vertex *v* with distance *v.d* from the BFS tree, but there's a path P2 with length < *v.d* from *s* to *v*
- Let *p* be the vertex that would be *v*'s "parent" on this shorter path
  - Our example: For shorter path A-E, *p* would be A
  - For the other shorter path A-B-E, *p* would be B
  - Vertex *p* is closer to start node than *v.pi*
- But wait! When BFS processes *p*, it would add unvisited vertex *v* to queue and assign *v*'s parent *v.pi* to be *p* and v.d to be p.d+1.
- If the path P2 is shorter than the path in the BFS tree, then *v* would have been attached to the tree at the same level as *v.pi* or higher
  - BFS adds a vertex to the tree as "early" as possible, as soon as it sees an edge connection



DFS

CLRS Section 22.3 on DFS

Readings

- CLRS:
  - Section 22.3 on DFS
  - Later/eventually:
    - Section 22.4 on Topological Sort
    - Section 22.5 on Strongly Connected Components

DFS: the Strategy in Words

- Depth-first search: Strategy
  - Go as deep as can visiting un-visited nodes
    - Choose any un-visited vertex when you have a choice
  - When stuck at a dead-end, backtrack as little as possible
    - Back up to where you could go to another unvisited vertex
  - Then continue to go on from that point
  - Eventually you'll return to where you started
    - Reach all vertices? Maybe, maybe not

Observations about the DFS Strategy

- Note: we must keep track of what nodes we've visited
- DFS traverses a subset of E (the set of edges)
  - Creates a tree, rooted at the starting point: the Depth-first Search Tree (DFS tree)
  - Each node in the DFS tree has a distance from the start. (We often don't care about this, but we could.)
- At any point, all nodes are either:
  - Un-discovered
  - Finished (you backed up from it), or
  - Discovered (i.e. visited) but not finished
    - On the path from the current node back to the root
    - We might back up to it
  - (Later we'll call these states: white, black and gray respectively)

DFS Strategy 1: Use a stack

- Maintain a Stack (Let’s call it S)

- Start at some node ‘s’ (push ‘s’ to S and mark as visited)
- While S not empty
  - Pop a node ‘n’ from S
  - Process ‘n’ if necessary (depending on problem you are solving)
  - For each non-visited neighbor of ‘n’
    - Mark neighbor as visited
    - Push neighbor onto S
  - Repeat

- Sound familiar? Same as BFS but uses stack instead of queue!
- Or we can implement recursively (see next slide)

DFS Strategy #2

- Use a recursive function to “visit” each node
  - Need a non-recursive function to initialize and make first call

- Before we look at this code… Important!
  - Best to think of DFS is a strategy as well as a single, particular bit of pseudo-code
  - We often add things to DFS code to solve problems
    - Code shown next is very minimal
  - We can use a DFS-based approach to solve many problems.
    - It’s kind of the ”Swiss Army Knife” of graph algorithms! ☺

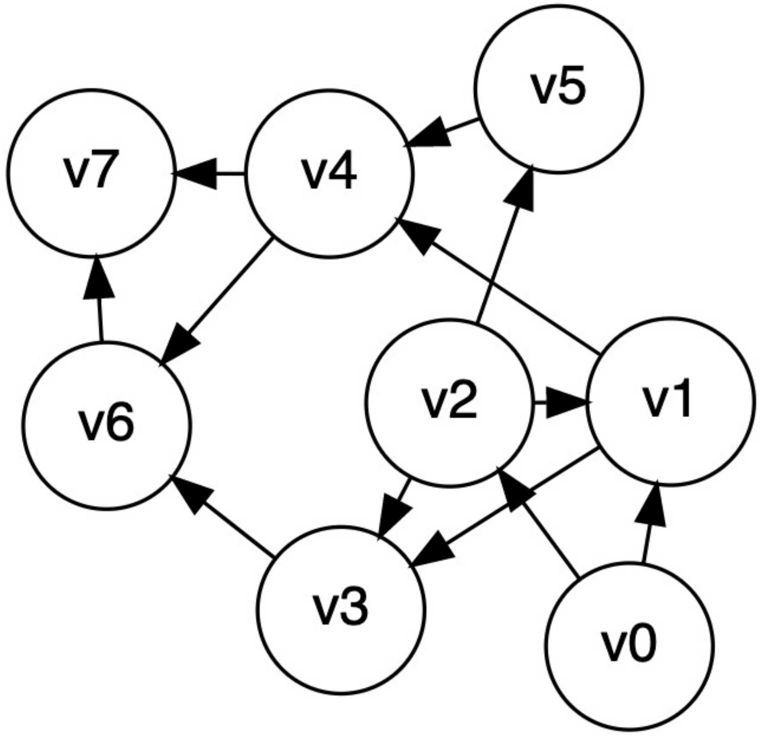
DFS Strategy 2: Recursion

```
def dfs(graph, start):                          //Main loop, inits and call on one start node
    visited = {}
    dfs_visit(graph, start, visited)

def dfs_visit(graph, curnode, visited):         //sometimes called dfs_recurse()
    visited[curnode] = True
    alist = graph.get_adjlist(curnode)          //get the neighbors of curnode
    for v in alist:
        if v not in visited:
            print("  dfs traversing edge:", curnode, v)
            dfs_visit(graph, v, visited)
    # end for-all adjacent vertices
    return
```

depth-first search, example

- Let’s start at V0



DFS to Process all Vertices in a Graph

- Purpose: do all required initializations, then call dfs_visit() as many times as needed to visit all nodes.
  - May create a DFS forest.
- Can be used to count connected components
  - Could remember which nodes are in each connected component
- CLRS text calls this version *dfs()* but it's really doing >1 DFS each from a different start node

```
def dfs_sweep(graph, start):
    visited = {}

     # loop repeats DFS on every unvisited node
    for v in graph:
        if v not in visited:
            dfs_visit(graph, v, visited)
```

Using DFS to Find if a Graphic is Acyclic

- Does a graph have a cycle?
  - DFS is great for this
  - But, slightly harder if graph is undirected

- Use DFS tree: classify edges and nodes as you process them
  - Nodes:
    - White: unvisited
    - Black: done with it, backed up from it (never to return)
    - Gray: Have reached it; exploring its adjacent nodes; but not done with it

CLRS's DFS Algorithm (non-recursive part)

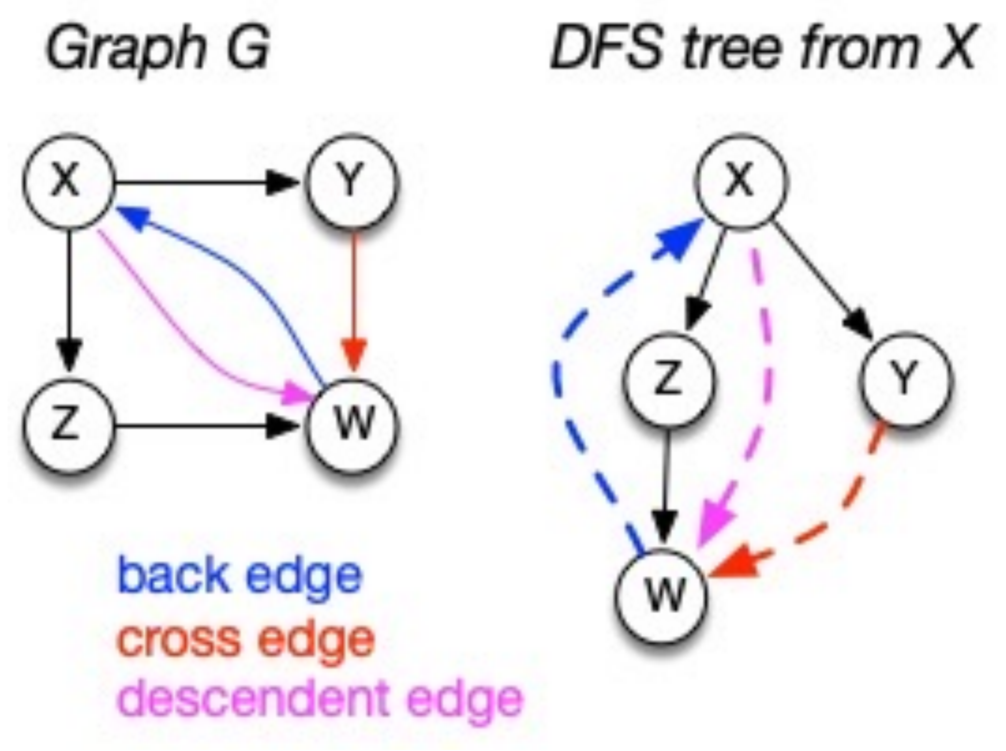
```
DFS_sweep(G)    // CLRS calls this just dfs()
1 for each vertex u in G.V
2     u.color = WHITE
3     u.π = NIL
4 time = 0
5 for each vertex u in G.V
6     if u.color == WHITE  // if unseen
7         DFS-VISIT(G, u)  // explore paths out of u
```

CLRS’s DFS Algorithm (recursive part)

```
DFS-VISIT(G, u)     // sometimes called this dfs_recurse()
1  time = time + 1  // white vertex u has just been discovered
2  u.d = time  // discovery time of u
3  u.color = GRAY  // mark as seen
4  for each v in G.Adj[u]  // explore edge (u, v)
5      if v.color == WHITE   // if unseen
6          v.π = u
7          DFS-VISIT(G, v)  // explore paths out of v (i.e., go “deeper”)
8  u.color = BLACK  // u is finished
9  time = time + 1
10 u.f = time  // finish time of u
```

Depth-first search tree

- As DFS traverses a digraph, edges classified as:
  - tree edge, back edge, descendant edge, or cross edge
  - If graph undirected, do we have all 4 types?



Using Non-Tree Edges to Identify Cycles

- From the previous graph, note that:
- Back edges (indicates a cycle)
  - dfs_visit() sees a vertex that is gray
  - This back edge goes back up the DFS tree to a vertex that is on the path from the current node to the root
- Cross Edges and Descendant Edges (not cycles)
  - dfs_visit() sees a vertex that is black
  - Descendant edge: connects current node to a descendant in the DFS tree
  - Cross edge: connects current node to a node in another subtree – not a descendant of current node

Non-tree Edges in DFS

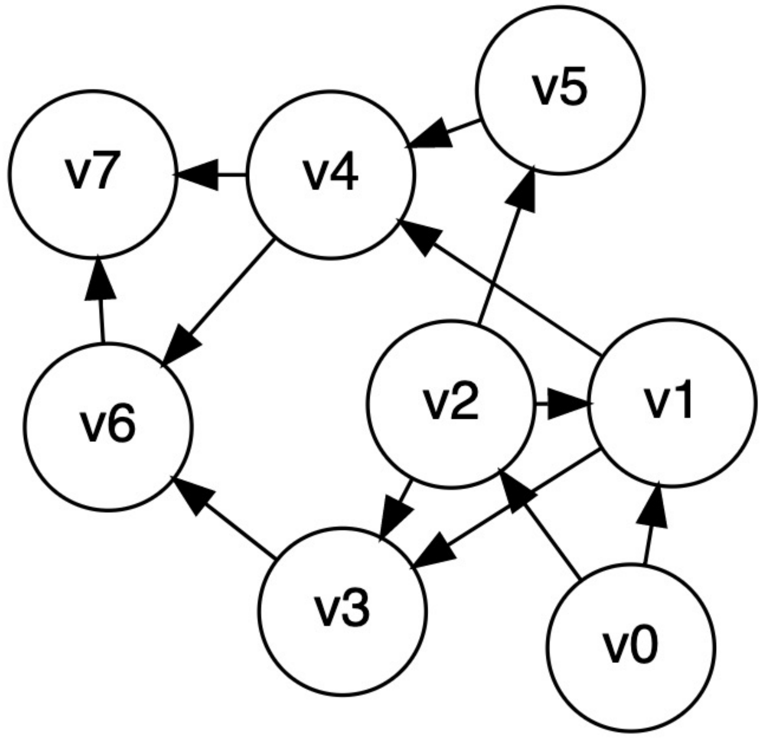
- Question 1: Finding back edges for an undirected graph is not **quite** this simple:
  - The parent node of the current node is gray
  - Not a cycle, is it? It's the same edge you just traversed
  - Question: how would you modify our code to recognize this?
- Question 2:
  - In digraph, how could you modify the code to distinguish cross edges from descendant edges?
  - Need to record the "time" at which a node was discovered (set to "gray") and finished (set to "black")
  - Also, have a "time counter", say, ctr
    - Set d[v] = ctr++ as discovery time
    - Set f[v] = ctr++ as finish time

Time Complexity of DFS

- For a digraph having V vertices and E edges
  - Each edge is processed once in the while loop of dfs_visit() for a cost of θ(E)
    - Think about adjacency list data structure.
    - Traverse each list exactly once. (Never back up)
    - There are a total of 2*E nodes in all the lists
  - The non-recursive dfs_sweep() algorithm will do θ(V) work even if there are no edges in the graph
  - Thus over all time-complexity is θ(V+E)
    - Remember: this means the larger of the two values
    - Note: This is considered "linear" for graphs since there are two size parameters for graphs.
  - Extra space is used for color array.
    - Space complexity is θ(V)

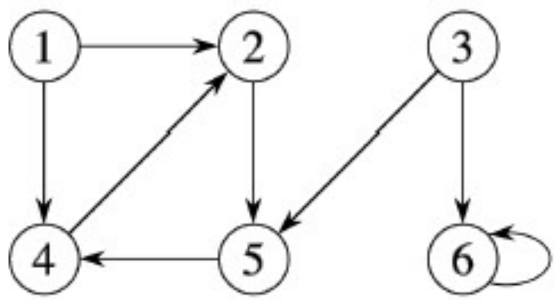
depth-first search, example

- Let's start at V0

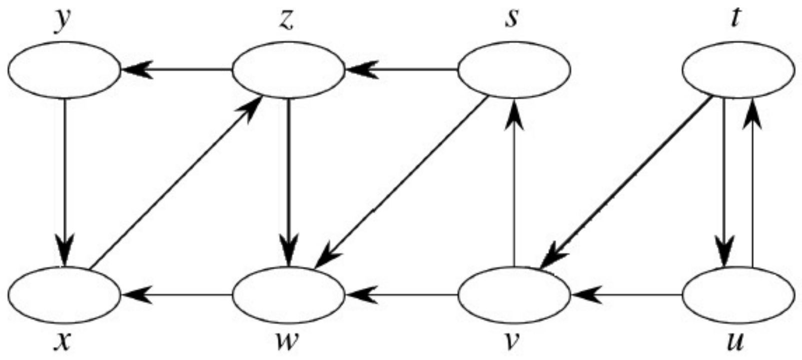


DFS Examples

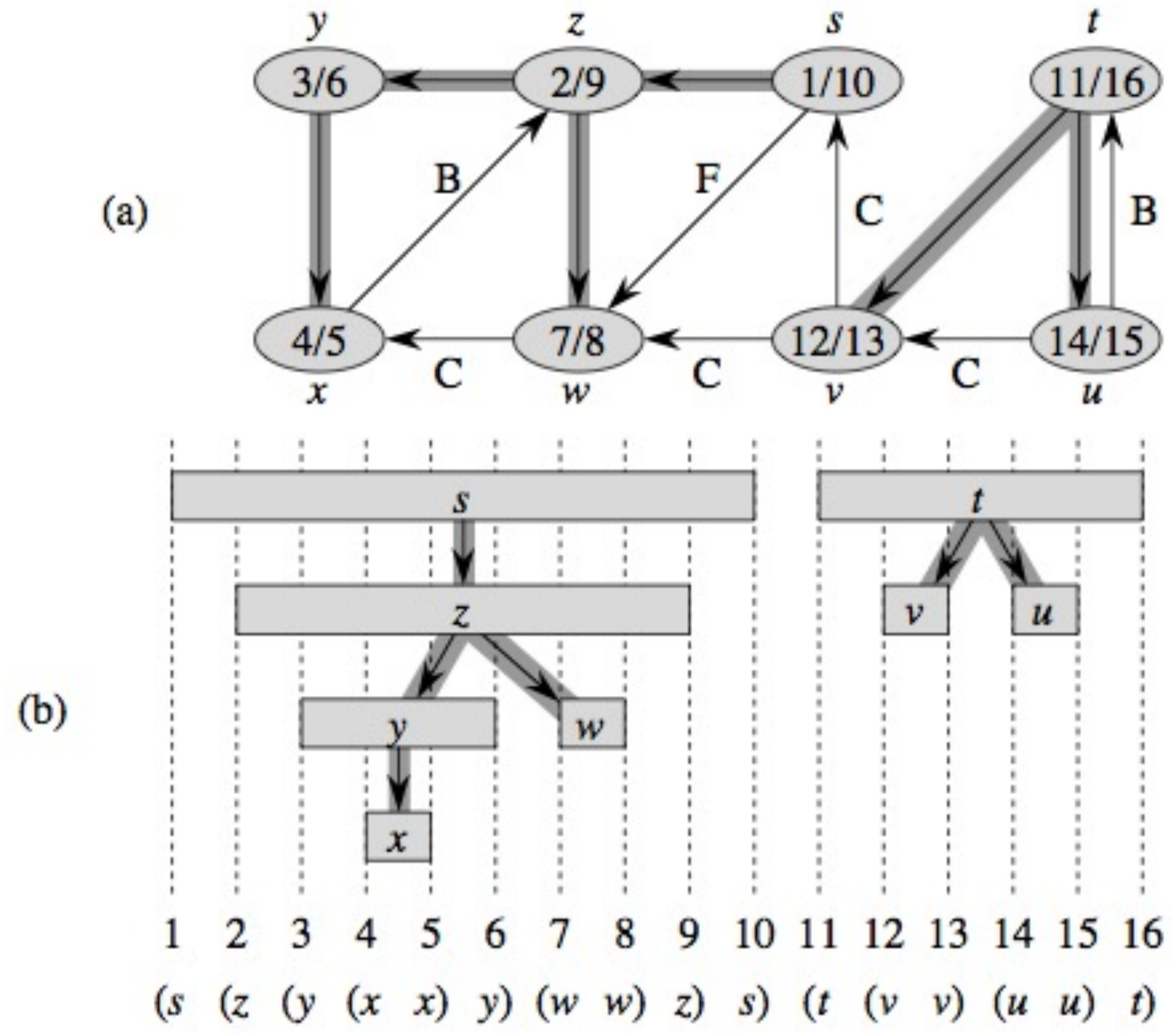
- Source vertex: 1



- Source vertex: s

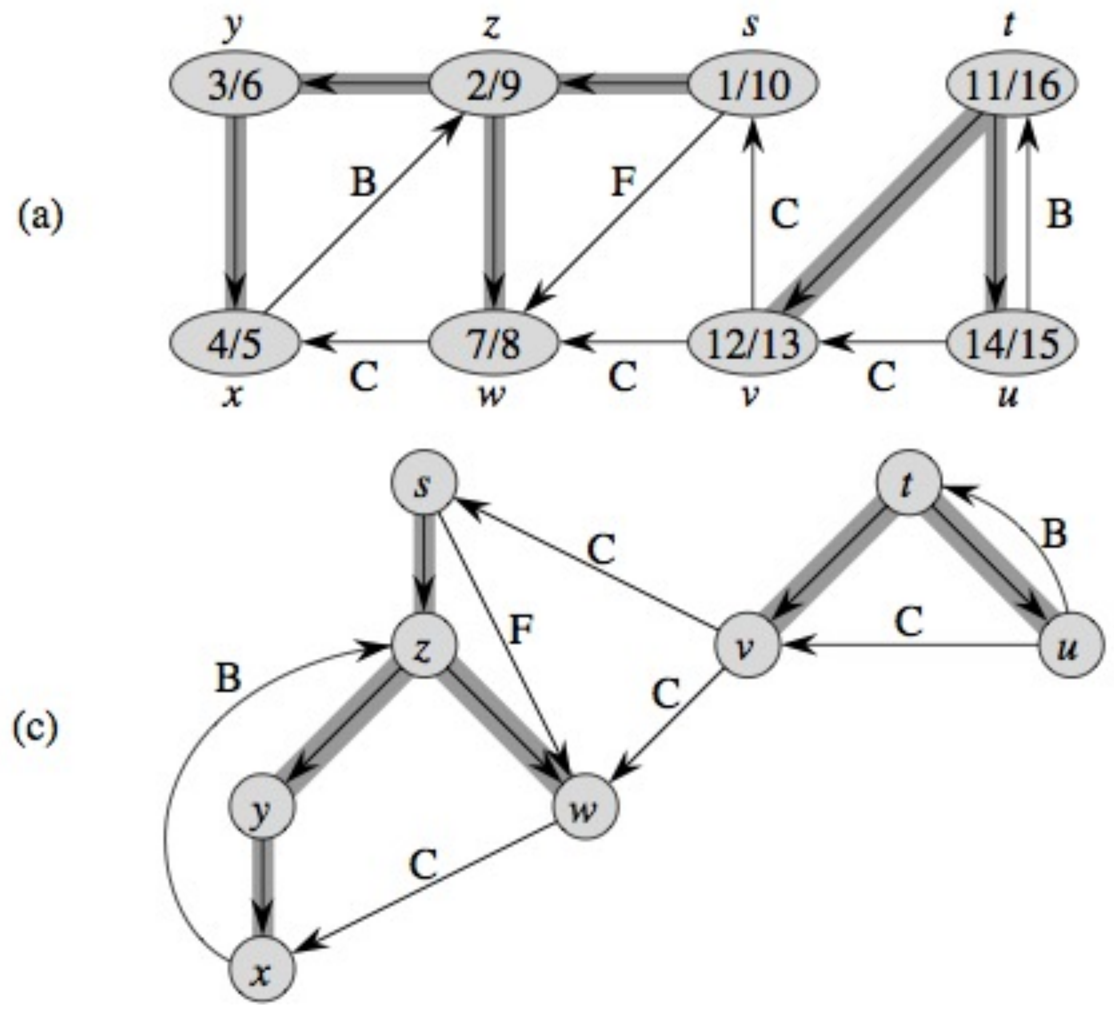


Properties of DFS Search, DFS Trees



- "Parentheses Structure". See pp. 606-609

Properties of DFS Search, DFS Trees



- Edge Classification. See pp. 606-609