Graphs

CS240 - Spring 2019

A New ADT

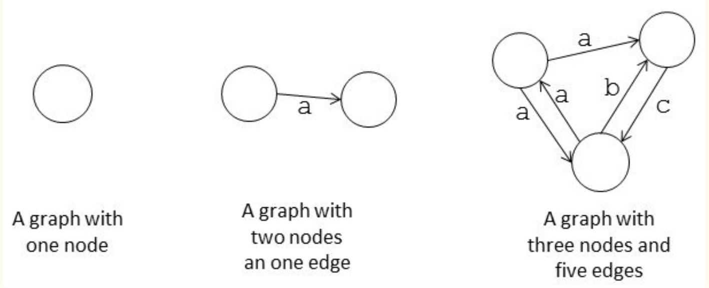
- Trees represent a hierarchy of data. What if we have relational data, similar to a tree, but no hierarchy.
  - For example, we go the RenFair, and there's lots of equally great stuff to do (tomato toss, turkey leg, jousting)
- Let's say our RenFair has paved paths that we are not allowed to leave. As good Computer Scientists, we want to come up with an algorithm for planning our day. What information do we need?

RenFair Paths

- Let’s call each activity a **vertex**, and the paths between each activity an **edge**. What information do we need to determine about our data set before we can form a plan?
  - Are all vertices connected to every other vertices *or* are there gaps?
  - Are all connected vertices the same distance from each other, *or* do the edges have **weight** (distance)?
  - Are the edges directional?
  - How many vertices are 1 stop away from any particular vertices?

What is a Graph?

- A data structure that consists of:
    - a finite set of vertices (nodes)
    - a set of edges that relate the vertices to each other
- A graph G is defined as follows:
    - G=(V,E)
        - *The set of edges describes relationships among the vertices*
    - Any subset of a graph's vertices and edges is a subgraph
- Graphs are essential in:
    - GPS, Database Design, Networks


a
a
a
a
b
c
A graph with one node
A graph with two nodes an one edge
A graph with three nodes and five edges

Directed vs. Undirected graphs

- When the edges in a graph have no direction, the graph is called undirected
  - Country roads with no traffic laws
- When the edges in a graph have a direction, the graph is called directed (or digraph)
  - One way streets in a big city

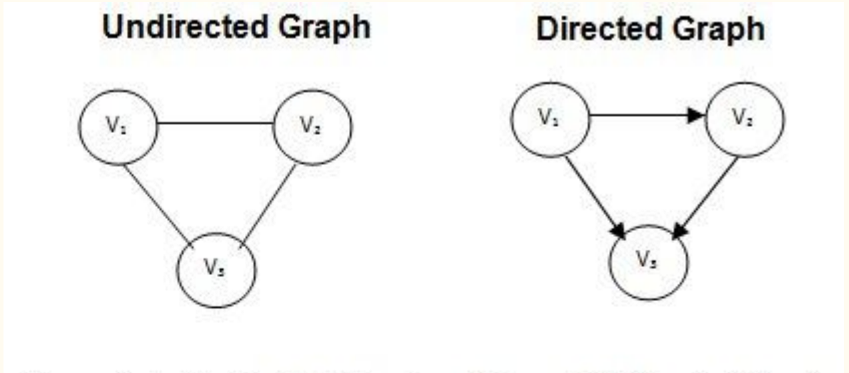
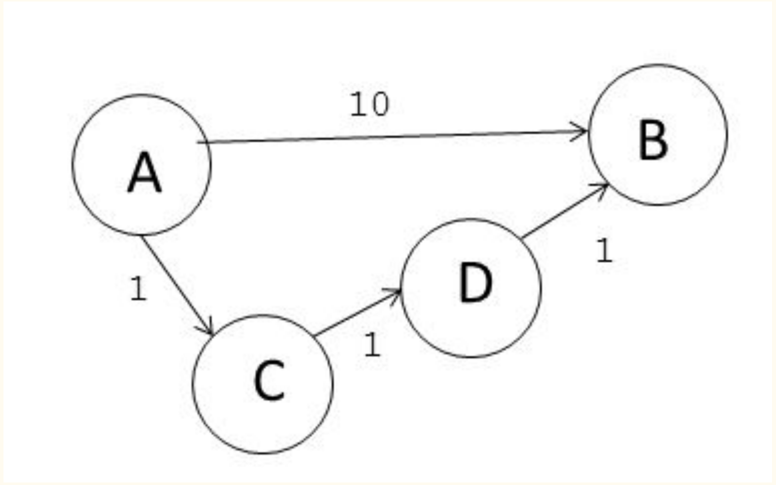
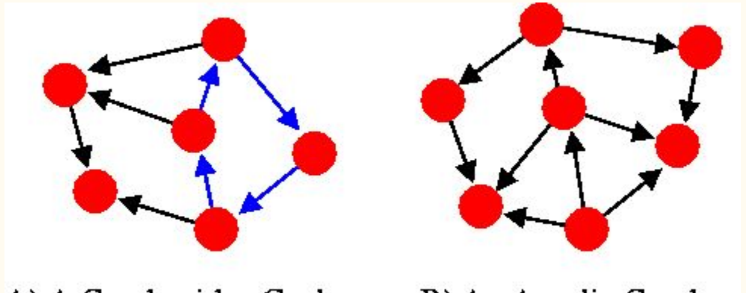


Figure 1: An Undirected Graph   Figure 2: A Directed Graph

Adjacent Vertices

- Connected by a single edge
- Degree (of a vertex)
  - # of adjacent vertices
  - if directed: Out-Degree vs In-Degree
- Path
  - sequence of vertices v1. . .vk such that consecutive vertices are adjacent.
  - **Simple Path** means no repeated v
- Cycle
  - The start and end vertices are the same




A) A Graph with a Cycle B) An Acyclic Graph

Connected vs Disconnected vs Complete

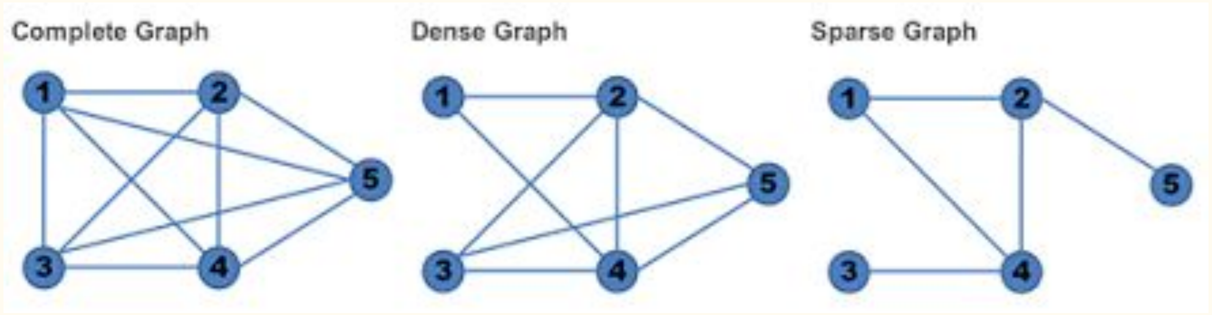
- Disconnected
    - Parts of the Graph are inaccessible from other parts of the graph through any path
- Connected
    - It is possible to build a path to any other part of the graph from any point in the graph
- Complete
    - All vertices of the graph are adjacent to all other vertices in the graph (connected by a single edge)

Kinds of Graphs

- Trees
    - Special cases of graphs
        - *connected graphs without cycles*
- Simple Graphs
    - There is only one edge between any two vertices and no self loops
        - *Formal Definition: A simple graph $G = (V,E)$ consists of a non-empty set V of vertices (or nodes) and a set E (possibly empty) of edges where each edge is a subset of V with cardinality 2 (an unordered pair).*
    - We are only considering simple graphs for this class

Sparse vs Dense

- A graph is sparse if $| E | \approx | V | - 1$.
  - If most V have a cardinality $<= 2$
- A graph is dense if $| E | \approx | V |(|V|-1)/2$.
  - If most V have a cardinality $=$ V-1



ADT Graph Operations

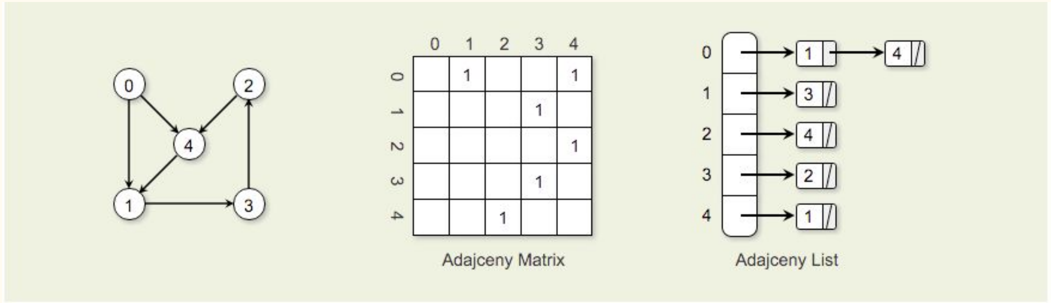
- What are some operations we need to perform on graphs?
  - Test whether graph is empty.
  - Get number of vertices in a graph.
  - Get number of edges in a graph.
  - Determine if an edge exists between two given vertices.
  - Insert
    - *Insert a new distinct vertex in graph*
    - *Insert edge between two given vertices in graph.*
  - Remove
    - *Remove specified vertex from graph and any edges between the vertex and other vertices.*
    - *Remove edge between two vertices in graph.*
  - Search for a vertex that contains given value.

How can we implement a graph?

- What information will need to be stored for each vertex?
  - Adjacent vertices
- What information do we need to store for each edge?
  - Weight (if directed)
  - What vertices it is connecting
- What data structure are we going to use to store the graph itself?
  - array or list

Representing Graphs in Memory

- Adjacency Matrix
  - 2 dimensional array contains entries for Vertex v_i.
    - *Column j in row i is marked with weight if there is an edge from v_i to v_j.*
  - Space requirements for the adjacency matrix are V^2.
- Adjacency List
  - Array of linked lists. The array is |V| items long, with position i storing a pointer to the linked list of edges for Vertex v_i.



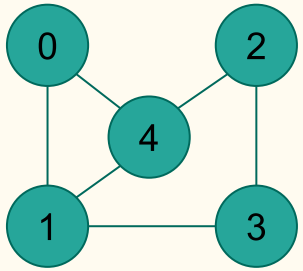
	0	1	2	3	4
0		1			1
1				1	
2					1
3				1	
4			1		

Adajceny Matrix

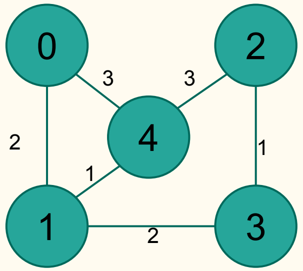
Adajceny List

Stored as Matrix and List

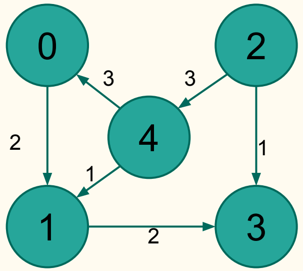
Undirected Graph



Undirected Weighted Graph



Directed Weighted Graph



Classwork

Implementing Graphs

- Adjacency Matrix: 50 bytes
  - 50 for edges
    - *5^2 * 2 bytes*
- Adjacency List : 120 bytes
  - 80 bytes for pointers
    - *4 bytes * 4 pointers * 5 nodes*
  - 40 bytes for edge weights
- Adjacency Matrix is good for a full graph,
- Adjacency List is good for sparse graph

Pros and Cons of Adjacency List

- Advantages:
  - Saves space for sparse graphs. In practice, most graphs are sparse.
  - "Visit" edges that start at v
    - *Only need to traverse linked list of v*
    - *Size of linked list of v is degree(v)*
- Disadvantages:
  - Check for existence of an edge (v, u)
    - *Must traverse linked list of v*
    - *Size of linked list of v is degree(v)*

Pros and Cons of Adjacency Matrix

- Advantage:
  - Save space on pointers and overhead for dense graphs.
  - Check for existence of an edge (i, j)
    - *Just check if A[i, j] = 1?*
      - So constant time complexity
- Disadvantage:
  - "Visit" all the edges that start at v
    - *Entire row v of the matrix must be traversed.*
      - So O(n) is best case

Graph Traversal

- Breadth First Traversal
- Depth First Traversal
- Shortest Path
- Minimum Spanning Tree
    - Kruskal
    - Prim

s-t Connectivity

- 2 algorithmic questions
  - Given two nodes s and t, is there a path between s and t?
    - *Called the s-t connectivity problem*
    - *Can be modified to ask: is there a path between all nodes?*
  - Given two nodes s and t, what is the length of the shortest path between s and t?
    - *Called the s-t shortest path problem*
    - *Can be modified to ask: what is the shortest path between one node and all other nodes?*

Search

Classwork

Searching Graphs

- BFS:
  - If you know a solution is not far from the root of the tree, a breadth first search (BFS) might be better.
  - finite number of children, but the height of the tree is infinite, DFS might never find the node you're looking for
    - *set of possible player moves in a game can be (effectively) infinite.*
- DFS:
  - If the tree is very deep and solutions are rare, depth first search (DFS) is likely to use less memory, and be faster in the average case.
  - If the tree is very wide, a BFS might need too much memory, so it might be completely impractical.

Traversing a Graph

Given a graph G = (V, E) and a source vertex, s, how can we systematically traverse the edges of G to “discover” (visit) vertices of G reachable from s?

Frontiers

- Discover all vertices ‘k’ vertices away from s before discovering any vertices $k + 1$ vertices from s
    - We expand the frontier between already discovered and undiscovered vertices one step at a time.
- Works similar to breadth first search of a tree
    - Use a First-In-First-Out (FIFO) queue to implement the Frontier.
        - *Need O(1) time to update*
- Produces a tree that contains all the reachable vertices from s with s as a root.

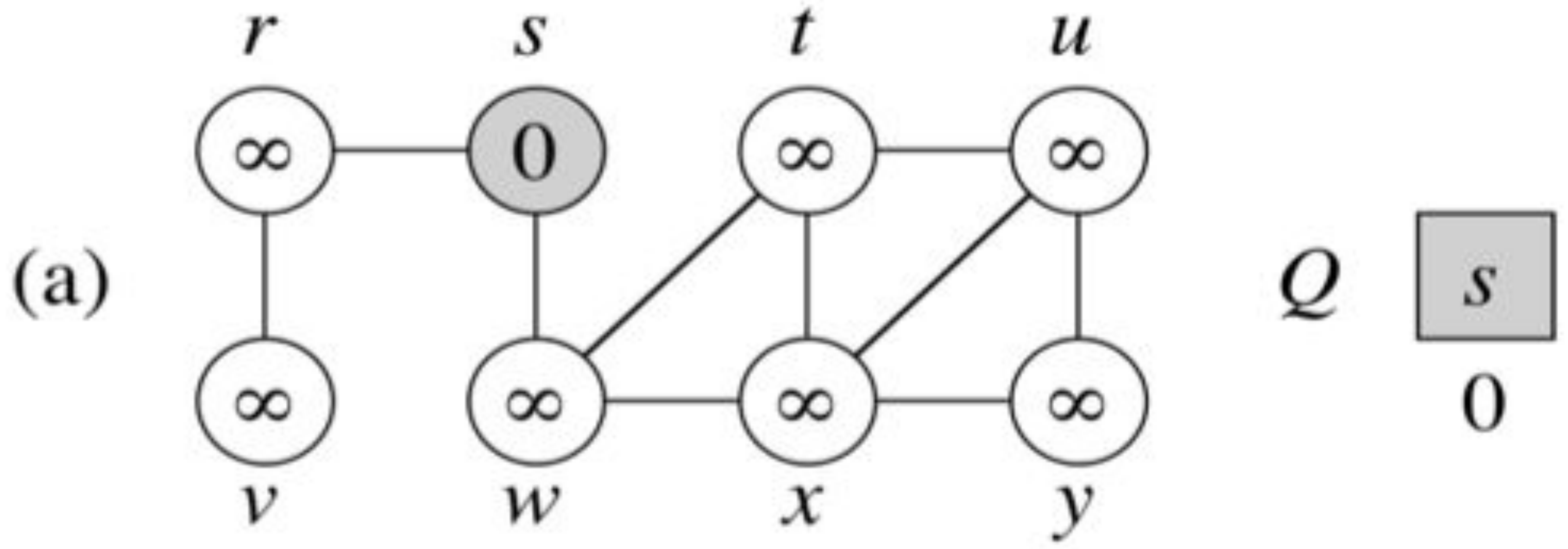
Breadth First

- On an **unweighted** graph, the algorithm also computes the shortest distance (dist) from s to any reachable node.
    - We are solving both the s-t connectivity problem and shortest path problem for unweighted graphs
- To get $O(|V| + |E|)$, we must use an adjacency list.
    - If we used an adjacency matrix, the running time would be $O(|V|^2)$. Why?
        - *For each node, we have to 'check' every other node to see if there is a connection*
- **All single source traversal methods may fail to reach all nodes for directed or unconnected graphs.**

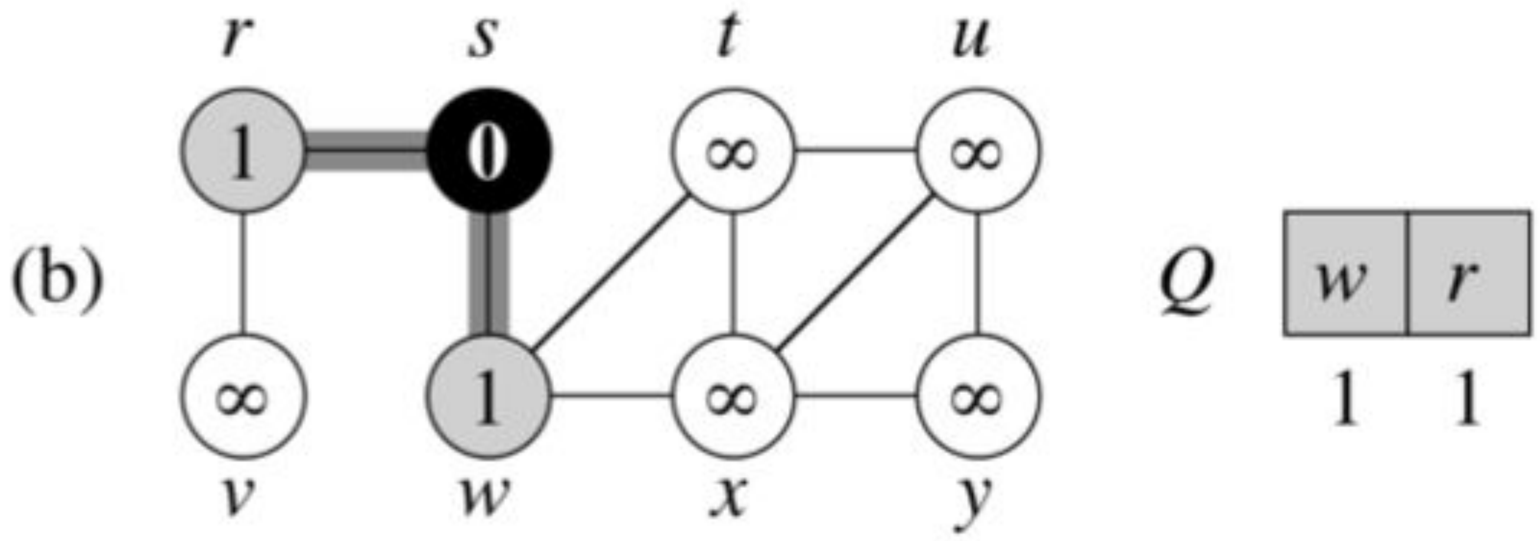
Going in Cycles

- How can we ensure we don’t go over the same path we already visited?
  - We tag the nodes to keep track of where we are in the process
- We can use colors to tag nodes
  - Nodes change color in order: White -> Gray -> Red
    - *White are undiscovered nodes*
    - *Grey are ‘frontier’ nodes*
    - ***Red** are fully explored nodes*

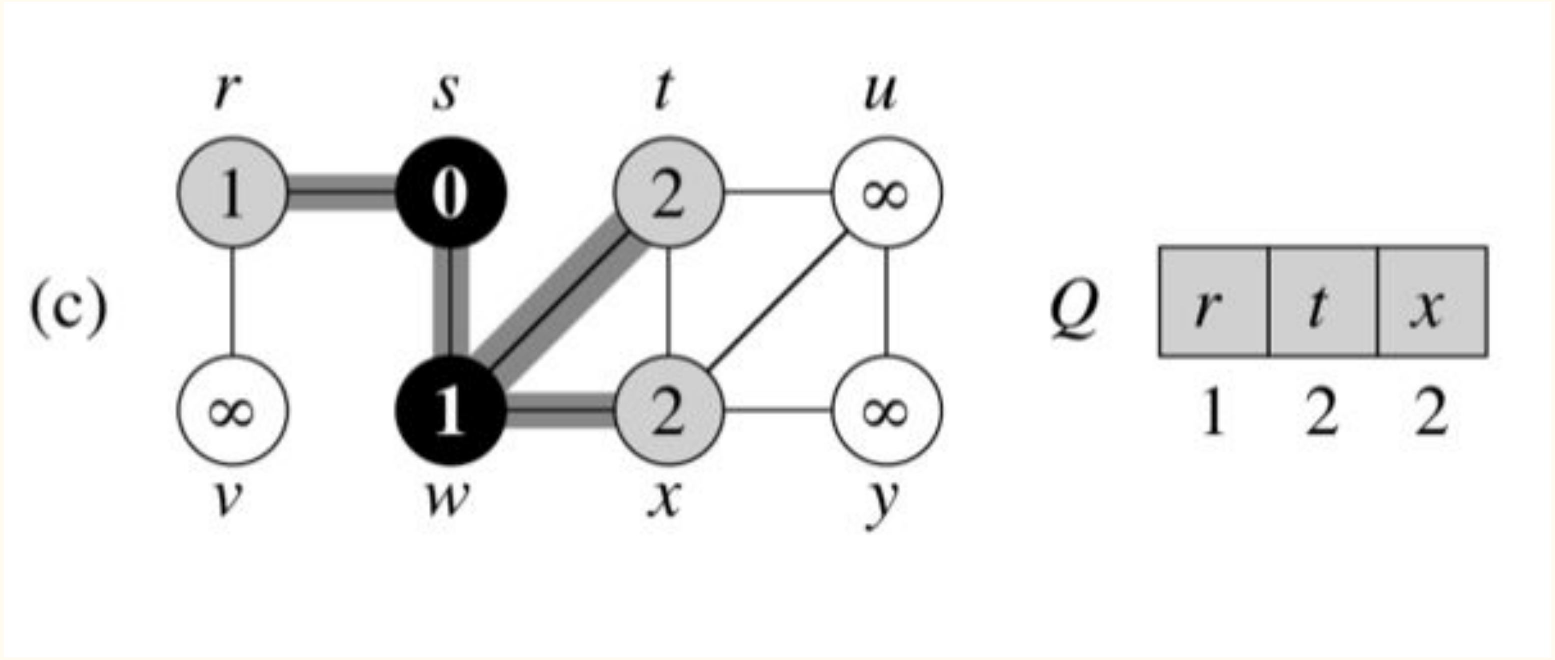
BFS: Example



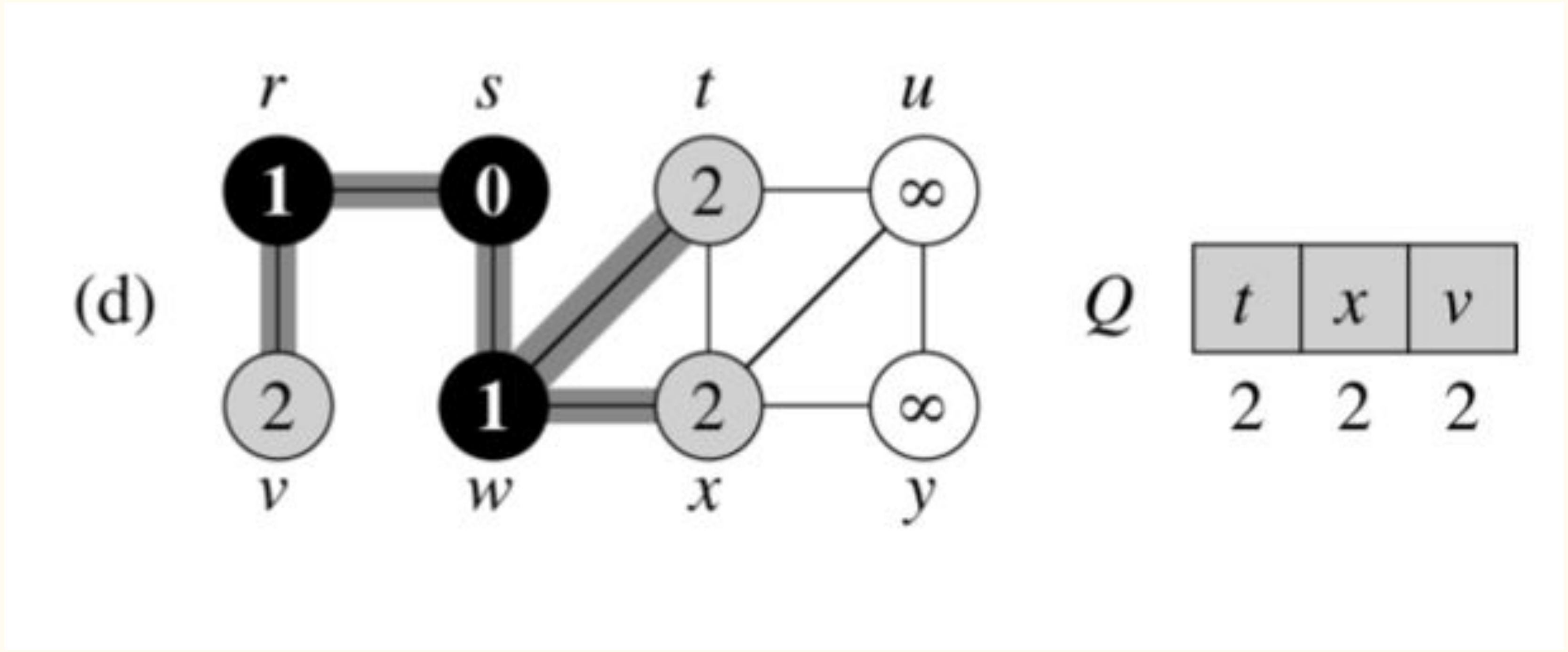
BFS: Example



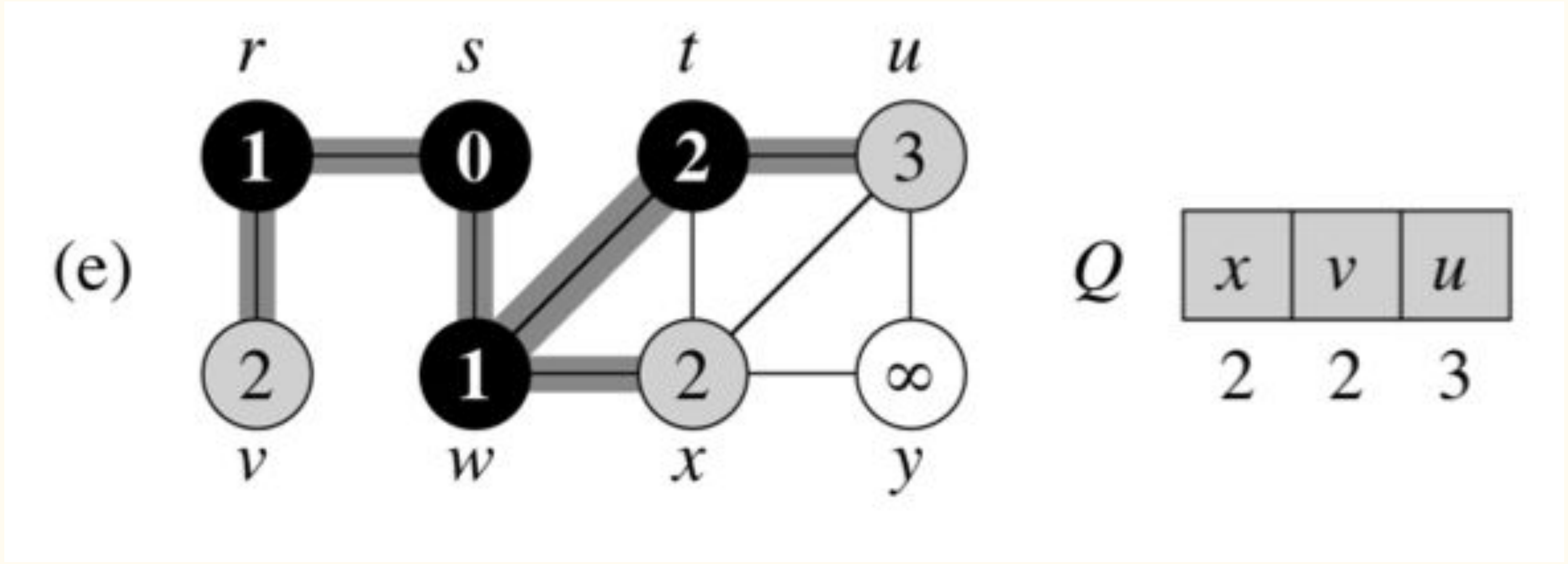
BFS: Example



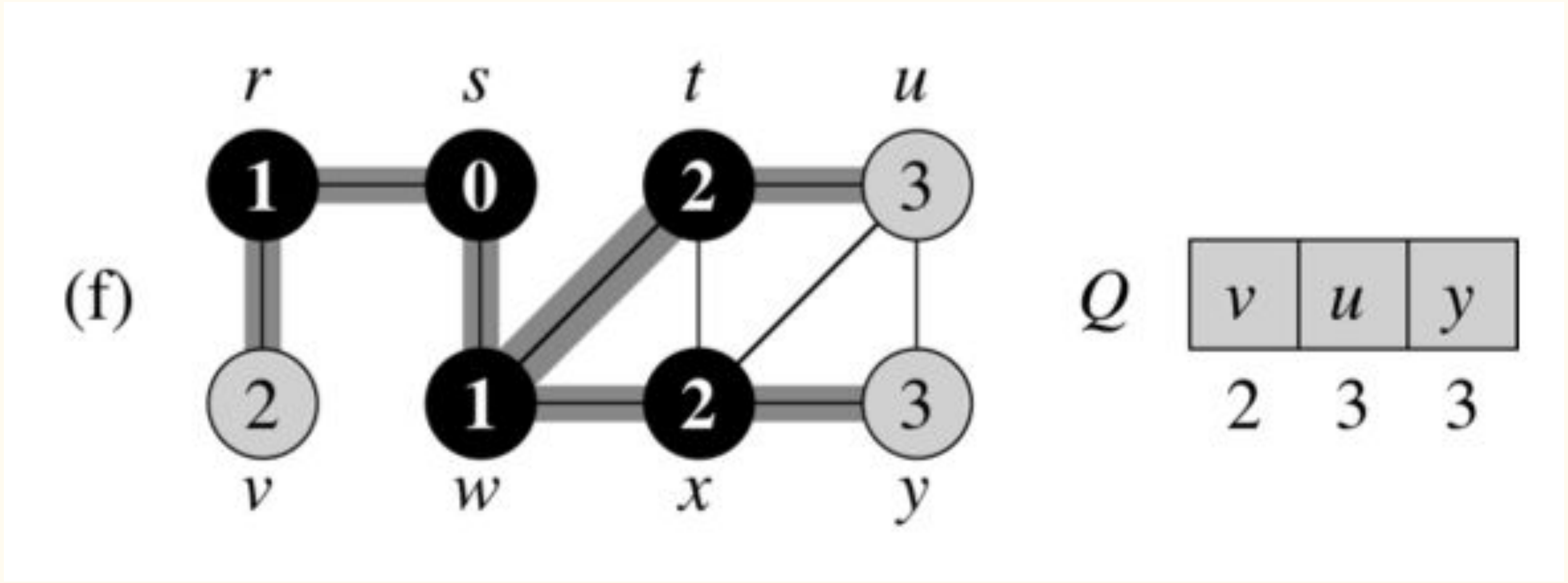
BFS: Example



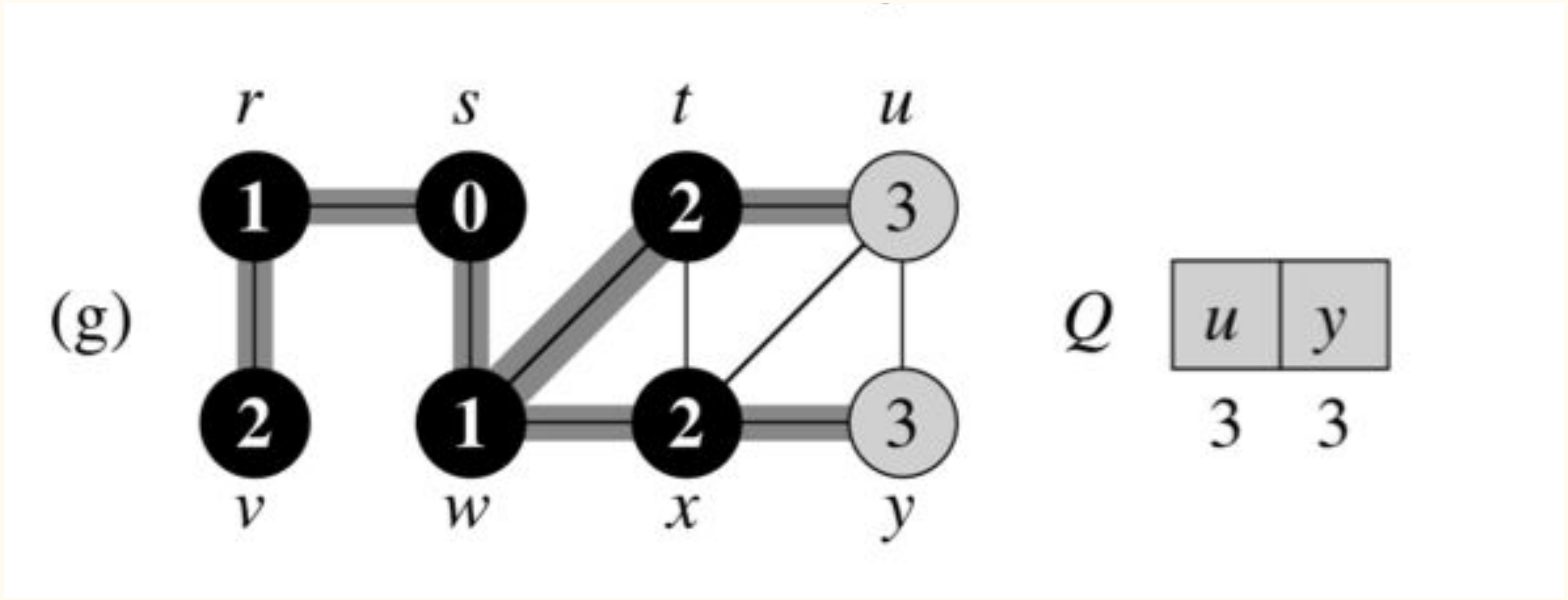
BFS: Example (continued)



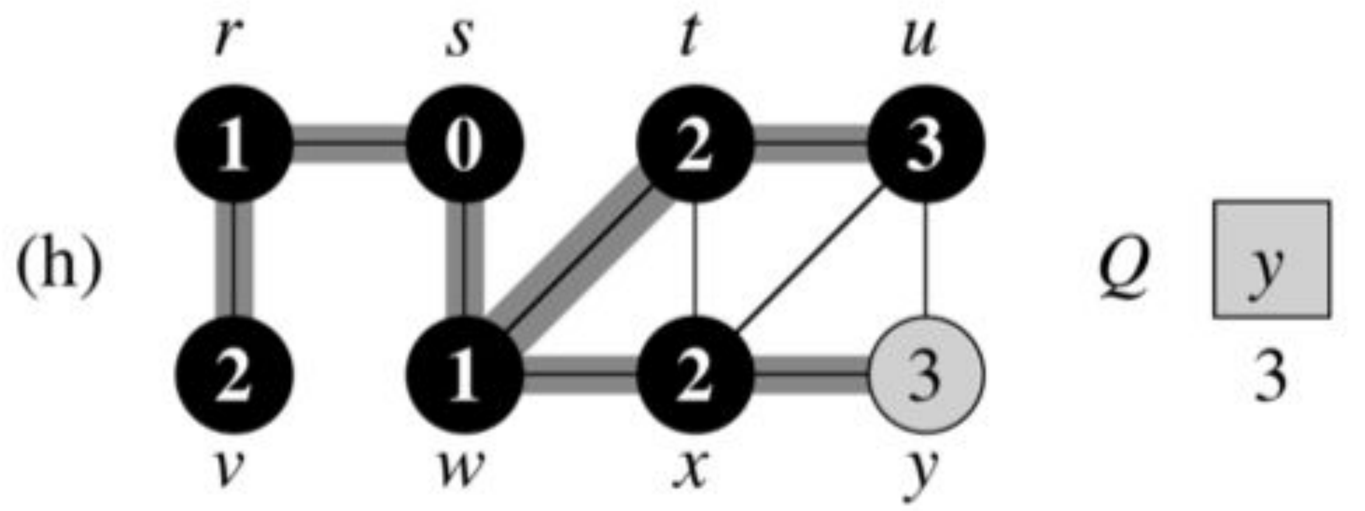
BFS: Example (continued)



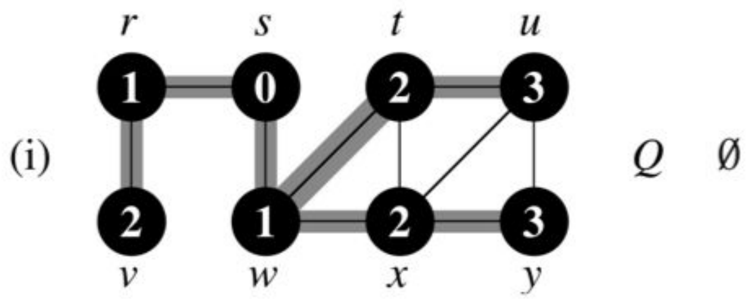
BFS: Example (continued)



BFS: Example (continued)



BFS: Example (continued)



The shaded edges are edges of the breadth first tree.

Implementing the BFS

```
BFS(G, root) //start the search from node s in graph G
    for each vertex u in G.V
        u.color = white
        u.distance = infinity // holds distance from s to u
        u.pre = NIL // u.pre denotes the predecessor of u
    root.color = gray
    root.distance = 0 // distance to itself is 0
    root.pre = NIL
    Q = // Q is an FIFO queue and it is initially empty
    Q.enqueue(root)
```

Implementing the BFS - Part 2

```
while Q is not empty:
    current = Q.dequeue()
    for each v adjacent to current:
        if v.color == white:
            v.color = gray
            v.distance = current.distance + n.edge(u)
            v.parent = current
            Q.enqueue(v)
    u.color = black
```

BFS and Connectivity

- How can we use the BFS to determine connectivity of our graph?
    - Once we have completed our BFS, we can check to see if there are any nodes still marked white
        - *If yes, unconnected graph*
        - *If no, connected graph*

Classwork

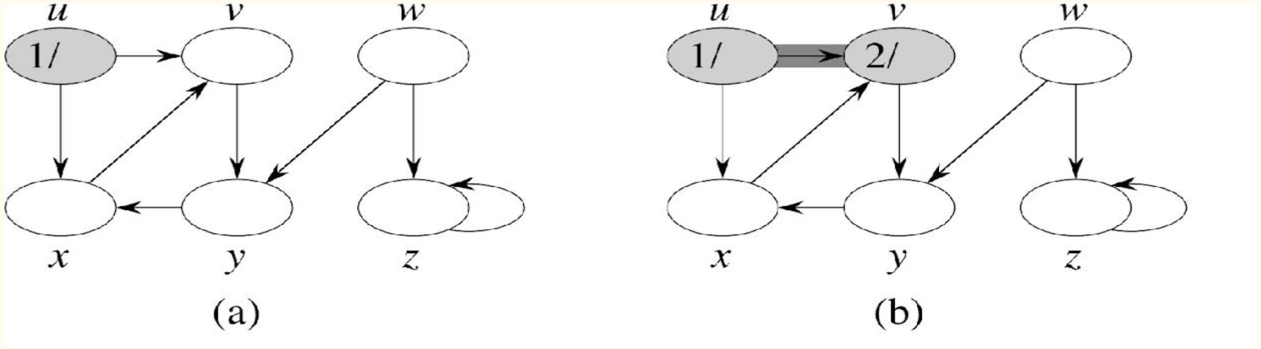
Traversing Graphs with BFS

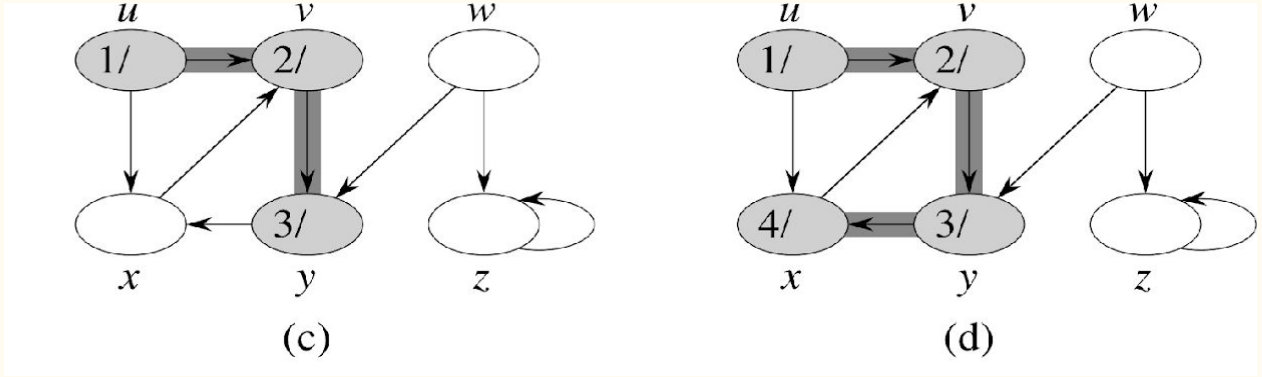
A, C, E, B, F, D
C, B, F, D
F

Depth First Search

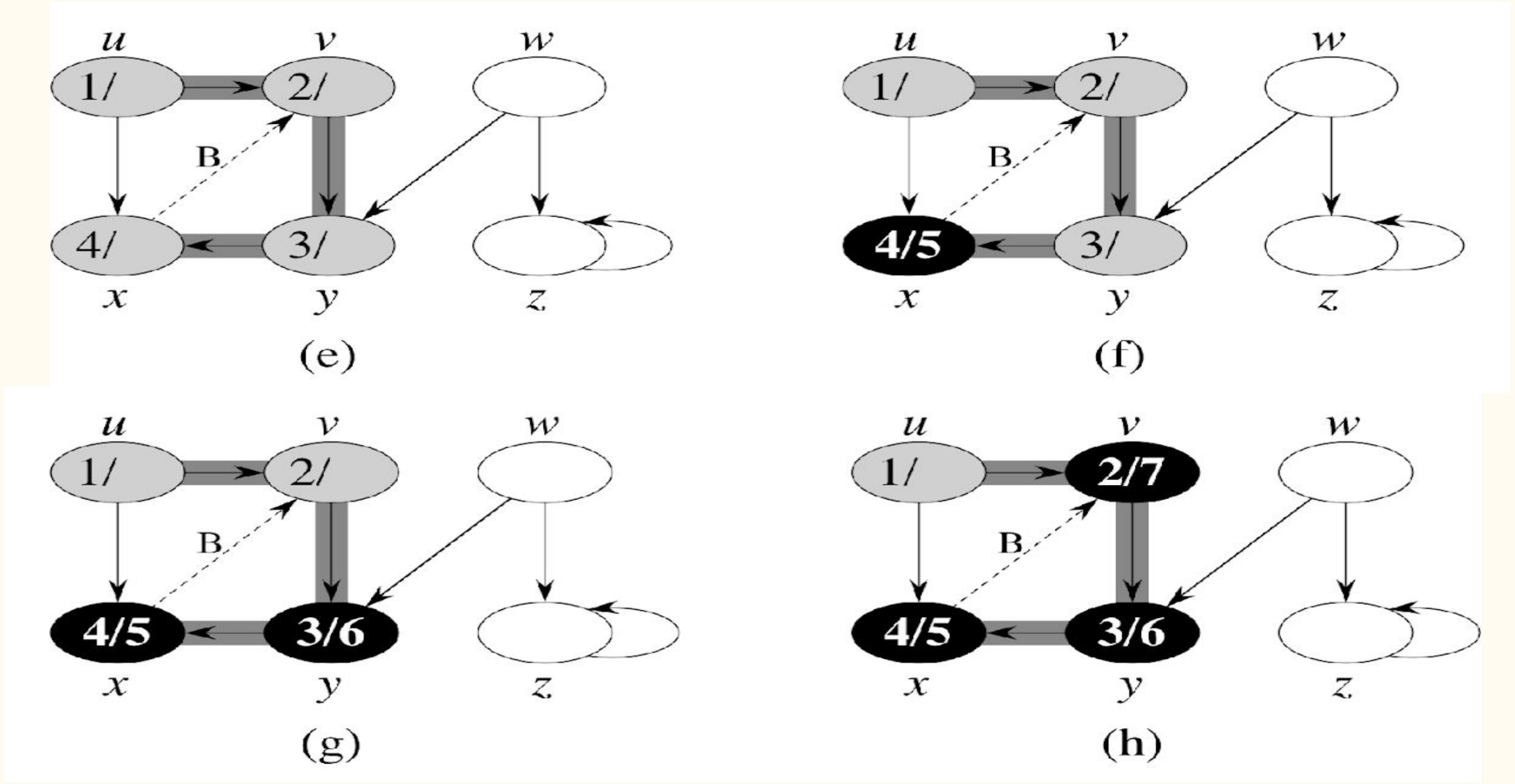
- Traverse deeper into the graph whenever possible
- We can use the same tagging method as the BFS
- We still need to guard against cycles
    - White: Undiscovered Nodes
    - Gray: Encountered, but not fully explored
    - Red: fully explored nodes
- DFS is useful for
    - When a tree is very wide, BFS can use too much memory
        - *such as solving a maze because in DFS you explore every possible path before backtracking.*

DFS: Example (1)

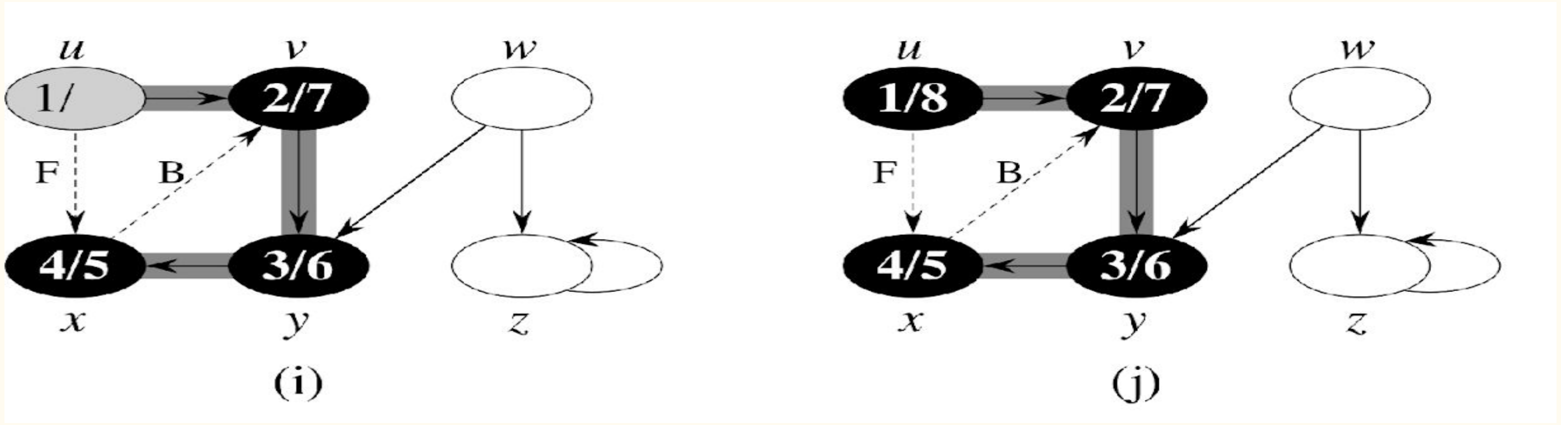




DFS: Example (2)



DFS: Example (3)



- Secondary Data Structure
    - Will this work with a queue? Why not?

Depth First Algorithm

- Basic Algorithm
  - Start a depth first search from node *s* by selecting 1 adjacent node.
  - The search proceeds from the most recently discovered node to the next adjacent node.
  - When you get to the last discovered node v, backtrack to the last node visited to discover v, and check adjacent.
    - *requires a **stack** rather than a **queue***
  - Eventually, the start node is fully explored.

DFS Implementation

```
DFS(G, v):
    for each u in G.V
        u.color = white
        u.pre = null
    DFSVisit(G, v)
DFSVisit(v)
    v.color = gray
    for each a in v.adjacent() do
        if a.color != gray
            a.distance = v.distance + v.edge(a)
            recursively call DFSVisit(a)
    v.color = black
```

Classwork

Traversing Graphs with DFS

- ACBFDE
- CBFEDA
- FBCAED
    - D can only be reached from F

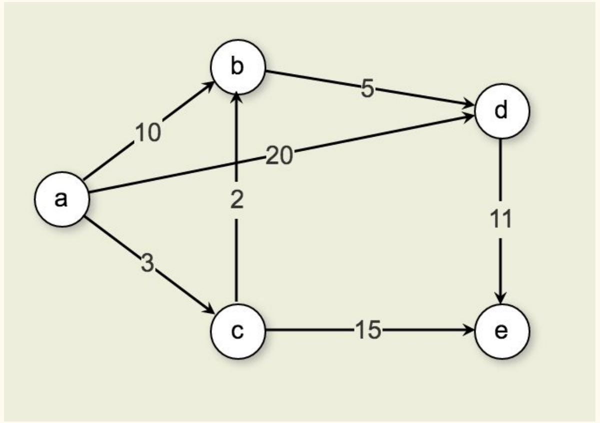
Shortest Path

Solutions to Shortest path

- BFS finds shortest path on unweighted tree
- Why would we want to find the shortest path from a single source within a graph?
    - GPS, networking, etc.
- Why is finding the shortest path between two vertices on a graph hard?
    - Shortest path may require traversal through intermediate points

Finding the Shortest Path

- Paths from A to D
  - The cost of the edge directly from A to D is 20.
  - The cost from A to C to B to D is 10.
    - *Thus, the shortest path from A to D is 10 (rather than along the edge connecting A to D).*
- On weighted graphs, the brute force solution is to find all possible paths, then select the shortest [O(n!)].
  - Goal: How can we find the shortest path without traversing every edge of the graph?

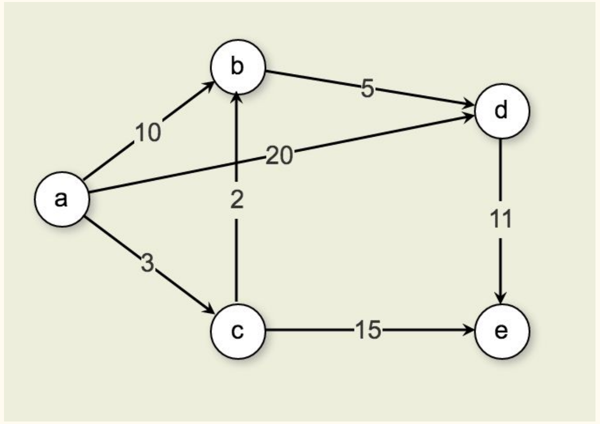


Single source shortest-paths

- Given
  - Weighted digraph, single source s.
- Goal:
  - Find the shortest path from s to every other vertex.
  - If just looking for a single destination, stop once the destination is found
- Note: Shortest paths form a tree
  - Meaning no cycles

Implementing a faster Shortest Path

- Observe: The shortest path from Start (S) to any Destination (D) is the shortest path resulting from the sum of the shortest path that goes from S to any Adjacent to D (A) + the path between A to D
  - Given the graph to the right, if I have the shortest path between a and d’s adjacents, then I only need to find the shortest path between adj and d to find the shortest path between a and d



Tracking our Path

- What variables do we need to keep track of?
  - G
    - *The graph which contains V and E*
  - dist[]
    - *vertex-indexed array containing distance from start*
  - pred[]
    - *Vertex index array containing the optimal previous node*

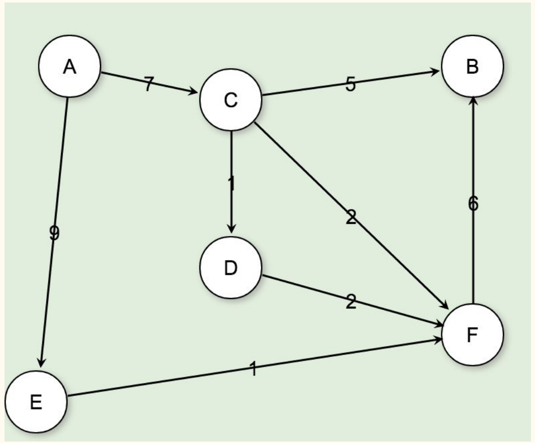
Edge Relaxation

- For all vertices, dist[v] is the length of some path from start (s) to some vertex (v)
  - Initially all are set to infinity
- Relaxation along edge e from v to w.
  - dist[v] is length of some path from s to v
  - dist[w] is length of some path from s to w
  - if v->w gives a shorter path from s to w through v, update dist[w] and pred[w]
    - *Relaxation sets dist[w] to the length of a shorter path from s to w (if v-w gives one)*

```
if (dist[w] > dist[v] + w.weight(v)){
        dist[w] = dist[v] + w.weight(v));
        pred[w] = v;
}
```

Dijkstra's Solution

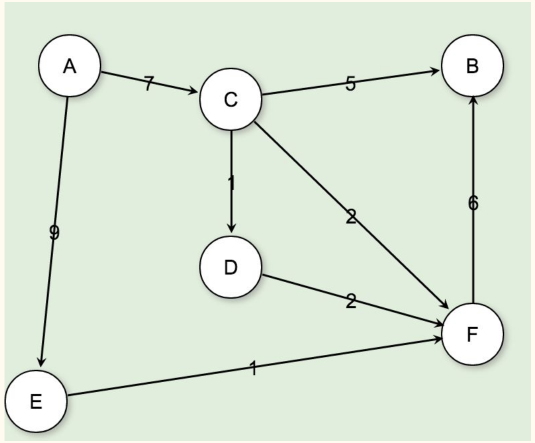
- Initialize an array of vertices
- Initialize all vertices dist[] to infinity
- Initialize all optimal previous to NULL



Vertices	Dist	OptimalPrev
A	∞	ø
B	∞	ø
C	∞	ø
D	∞	ø
E	∞	ø
F	∞	ø

Dijkstra’s Solution

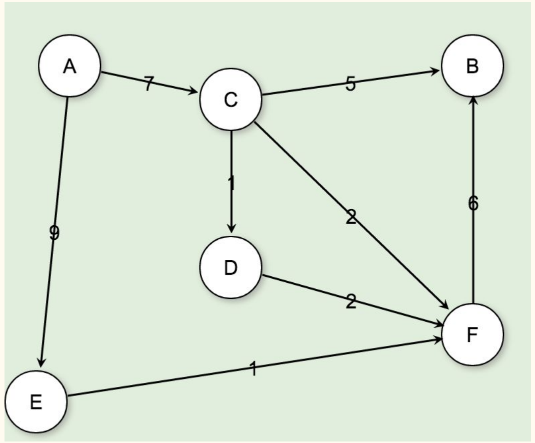
- Start at A, mark distance as 0
- Leave Optimal Previous Vertex at NULL
- Mark distance of each adjacent node to A and optimal previous node
- Mark A as a fully explored vertex



Vertices	Dist	OptimalPrev
A	0	ø
B	∞	ø
C	7	A
D	∞	ø
E	9	A
F	∞	ø

Dijkstra’s Solution

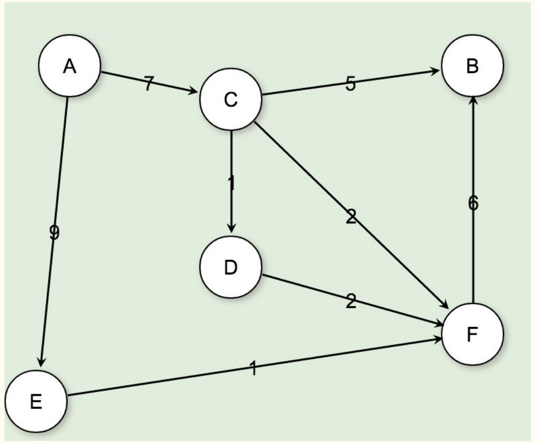
- Advance to the next unexplored Node with the least distance from A
  - This would be C
- Check all C’s unexplored adjacent nodes
  - if less than current distance to A, update distance and set optimal previous node
- Mark C as Explored



Vertices	Dist	OptimalPrev
A	0	ø
B	12	C
C	7	A
D	8	C
E	9	A
F	9	C

Dijkstra’s Solution

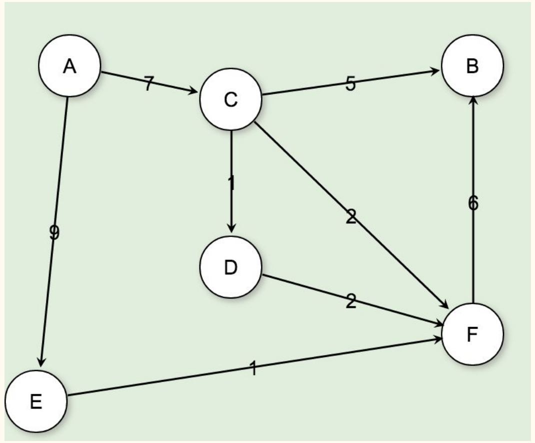
- Advance to the next unexplored Node with the least distance from A
    - This would be D
- Check its unexplored nodes to see if less distance
    - F would be 10, which is greater than 9, so ignore the path
- Mark D as Explored



Vertices	Dist	OptimalPrev
🟥 A	0	ø
B	12	C
🟥 C	7	A
🟥 D	8	C
E	9	A
F	9	C

Dijkstra's Solution

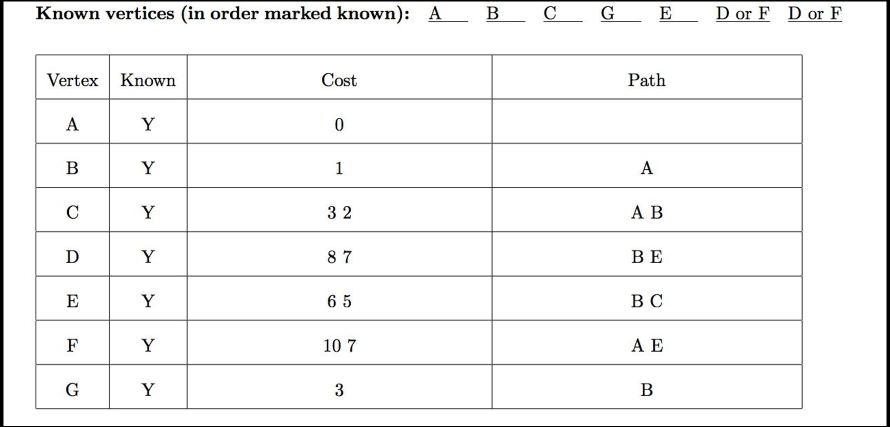
- Repeat until all nodes are fully explored
- We can now find the shortest path from any node back to a single source by following the optimal previous backward to the source



Vertices	Dist	OptimalPrev
A	0	ø
B	12	C
C	7	A
D	8	C
E	9	A
F	9	C

Classwork

Dijkstra's Shortest Path



Known vertices (in order marked known): A B C G E D or F D or F

Vertex	Known	Cost	Path
A	Y	0	
B	Y	1	A
C	Y	3 2	A B
D	Y	8 7	B E
E	Y	6 5	B C
F	Y	10 7	A E
G	Y	3	B

- No. For example, take the graph with three edges cost(u,v) = -2, cost(v,w) = 2, cost(u, w)=1. The shortest path to w is (u, v, w). But adding 2 to the cost of each edge and then running Dijkstra would give the shortest path as (u, w).

- Use a Priority Queue (heap) to get the minimum edge each time

Improving Dijkstra’s Algorithm

- We can improve our shortest path implementation with a priority queue
  - Using a heap, we can produce the minimum unvisited edge in logn time
- This gives Dijkstra’s shortest path algorithm an average case nlogn complexity
  - We explore n nodes once at most, O(n)
  - Adjacencies:
    - *If we had a dense graph we will go through at most n-1 adjacencies*
    - *If we have a sparse graph we will go through approximately O(1)*
  - Worst case for Dense graph is O(n^2), average case is O(nlogn)

Implementation

- Dijkstra’s shortest path algorithm must keep track of 3 things
    - explored nodes, distance, and optimal previous
        - *We have used array’s to keep track of these*
- Can we store this information in the object?
    - Sure, and it’s more organized too.
- Won’t we constantly need to update the object?
    - Yes, but is this worse than constantly updating an array?
        - *If you are caching previous results, maybe.*

Dijkstra's Algorithm

```
// Compute shortest path distances from s, store them in D
void Dijkstra(Graph G, int s, int[] D, int[] OP) {
    for (int i=0; i<G.nodeCount(); i++)    // Initialize
        D[i] = INFINITY;
        OP[i] = NULL
    D[s] = 0;
    for (int i=0; i<G.nodeCount(); i++) {  // Process the vertices
        int v = minVertex(G, D);     // find next-closest unexplored vertex
        G.setExplored(v); //set as fully explored
        if (D[v] == INFINITY) return; // Unreachable
        int[] adj = G.adjacents(v);
        for (int j=0; j<adj.length; j++) {
            int w = adj[j];
            if (D[w] > (D[v] + G.weight(v, w)))
                D[w] = D[v] + G.weight(v, w);
                OP[w] = v;
            }
        }
    }
}
```

Minimum Spanning Tree

Problem

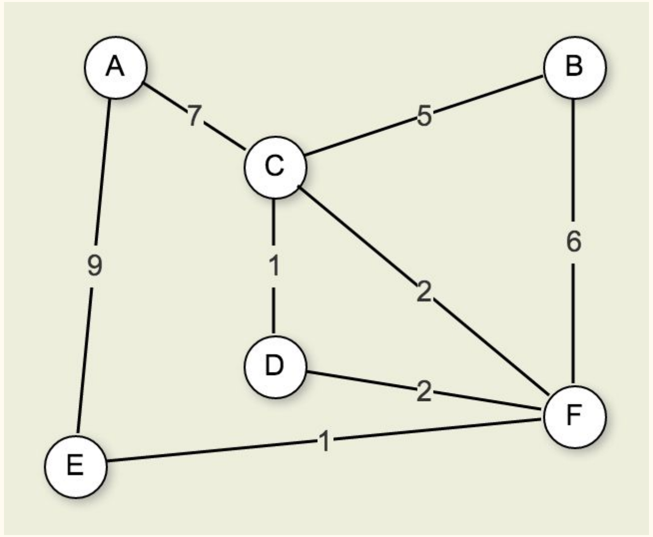
- You are asked to write software that gives the most efficient path layout to laying fiber optic cable between a set of cities.
- How is this a different problem than shortest path?
    - We are not concerned with a single source. We want to find the smallest connected subgraph in a given graph without regard to any single vertex.

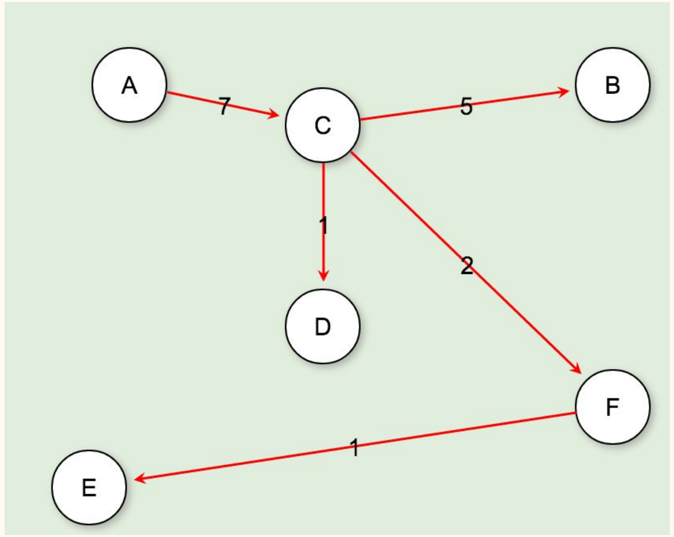
Reaching all the Vertices

- The minimal-cost spanning tree (MCST) problem takes as input a connected, undirected graph G, where each edge has a distance or weight measure attached.
- Minimum Spanning Tree
    - has minimum total cost as measured by summing the values for all of the edges in the subset, and
    - keeps the vertices connected.
    - contains no cycles

Minimum Cost Spanning Tree

- MCST is a free tree with |V|—1 edges.
    - Free tree has no root

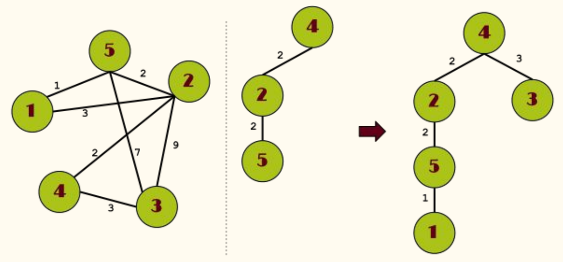




Prim's Algorithm

- Create an array of each vertices and a distance
  - initialize each distance to infinity
- Start with any vertex
  - For each adjacent vertex not yet in the tree, check the distance
  - Add the smallest, unvisited edge to the MCST
  - Review each node in the MCST for adjacent nodes with the smallest edge
  - repeat

PRIM'S APPROACH

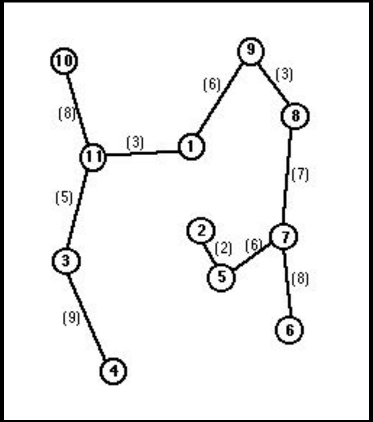


During the process of finding the MST the algorithm of Prim keeps a single tree!

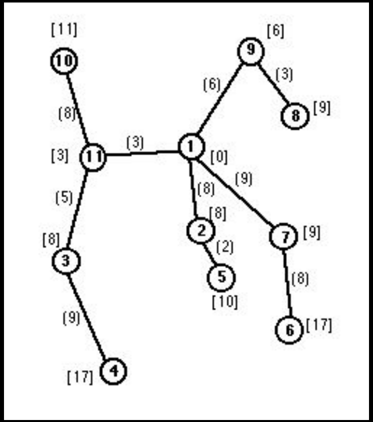
Classwork

MCST vs Shortest Path

MCST



SP



[11]
10
(8)
[3]
11
(3)
1
[0]
(6)
9
[6]
(3)
8
[9]
(9)
7
[9]
(8)
6
[17]
(8)
2
[8]
(2)
5
[10]
(5)
3
[8]
(9)
4
[17]