DS221 | 19 Sep – 19 Oct, 2017

Data Structures, Algorithms & Data Science Platforms

Yogesh Simmhan

simmhan@cds.iisc.ac.in

CDS
Department of Computational and Data Sciences

L4: Graphs

Graph ADT, Algorithms

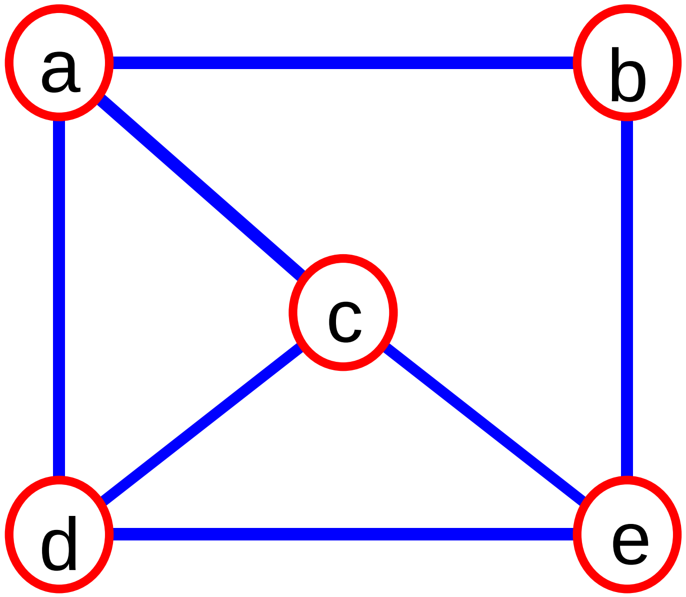
Slides courtesy:
Venkatesh Babu, CDS, IISc

What is a Graph?

- A graph **G = (V,E)** is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V
- An edge **e = (u,v)** is a pair of vertices
- Example:

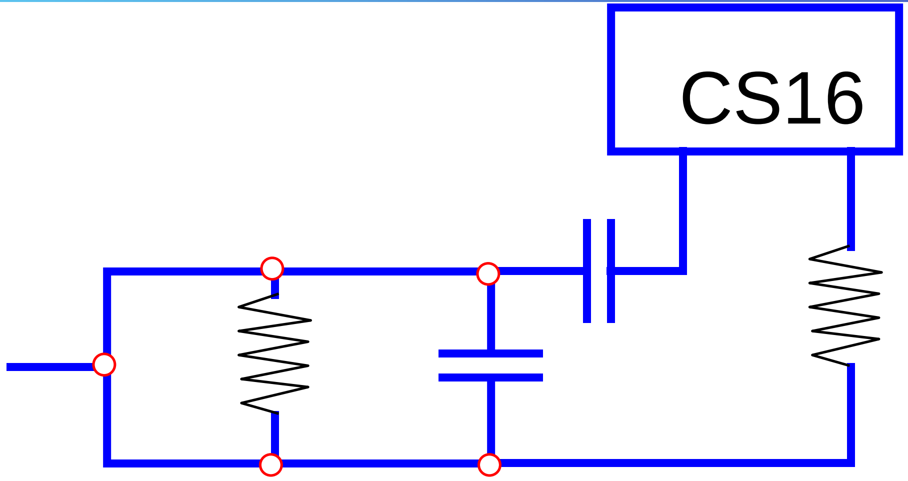


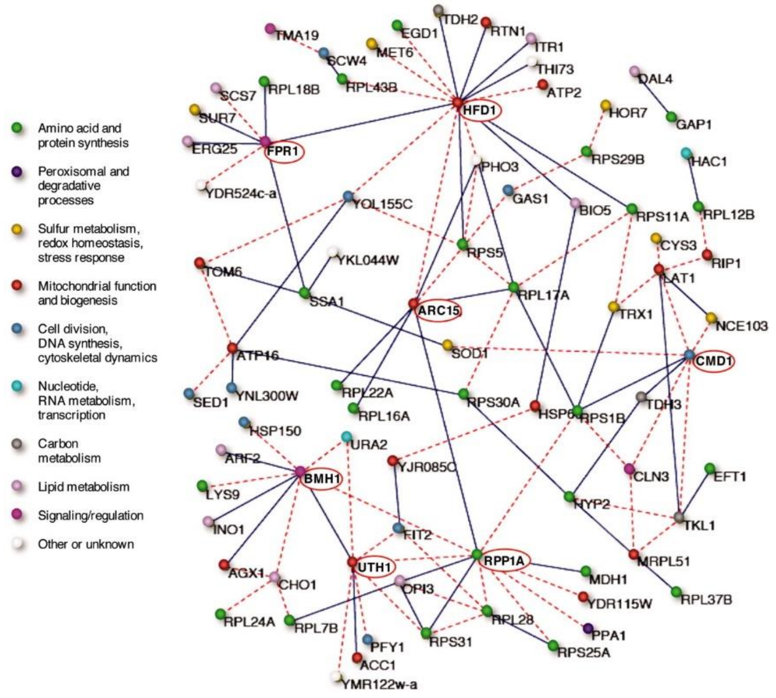
V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Applications

- Electronic circuit design
- Transport networks
- Biological Networks




Amino acid and protein synthesis
Peroxisomal and degradative processes
Sulfur metabolism, redox homeostasis, stress response
Mitochondrial function and biogenesis
Cell division, DNA synthesis, cytoskeletal dynamics
Nucleotide, RNA metabolism, transcription
Carbon metabolism
Lipid metabolism
Signaling/regulation
Other or unknown
HFD1
FPR1
ARC15
CMD1
BMH1
UTH1
RPP1A

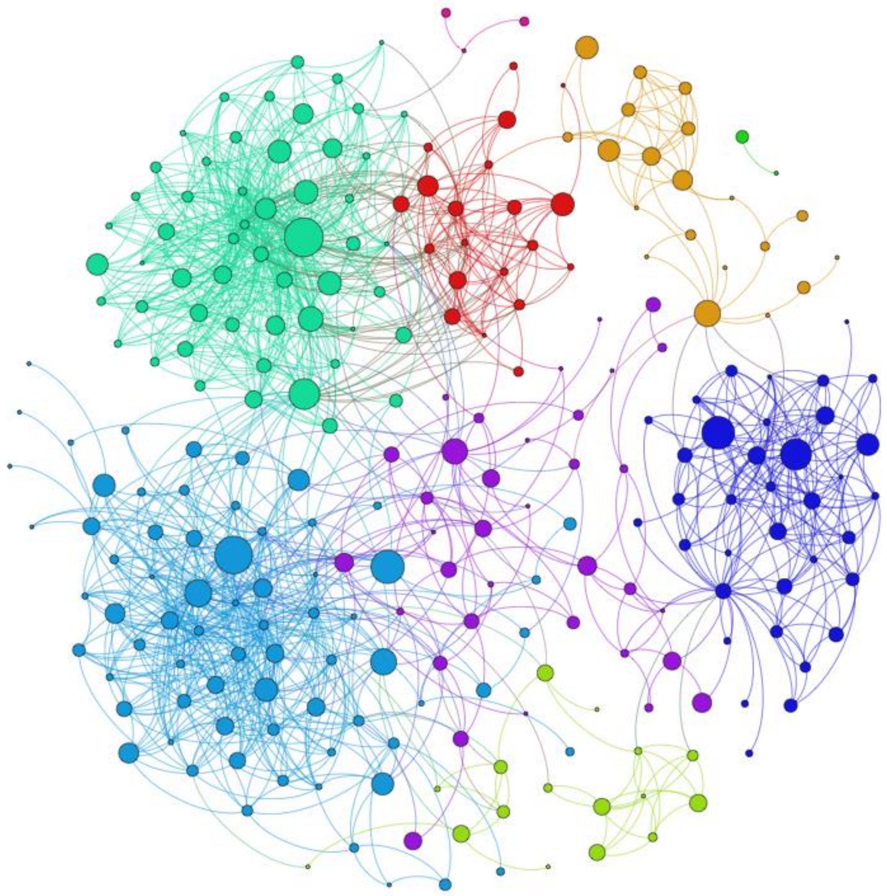
http://www.pnas.org/content/103/50/19033/F3.expansion.html

IndiGo
Delhi
Jaipur
Guwahati
Agartala
Imphal
Vadodara
Kolkata
Nagpur
Mumbai
Pune
Hyderabad
Goa
Bangalore
Chennai
IndiGo Route Map

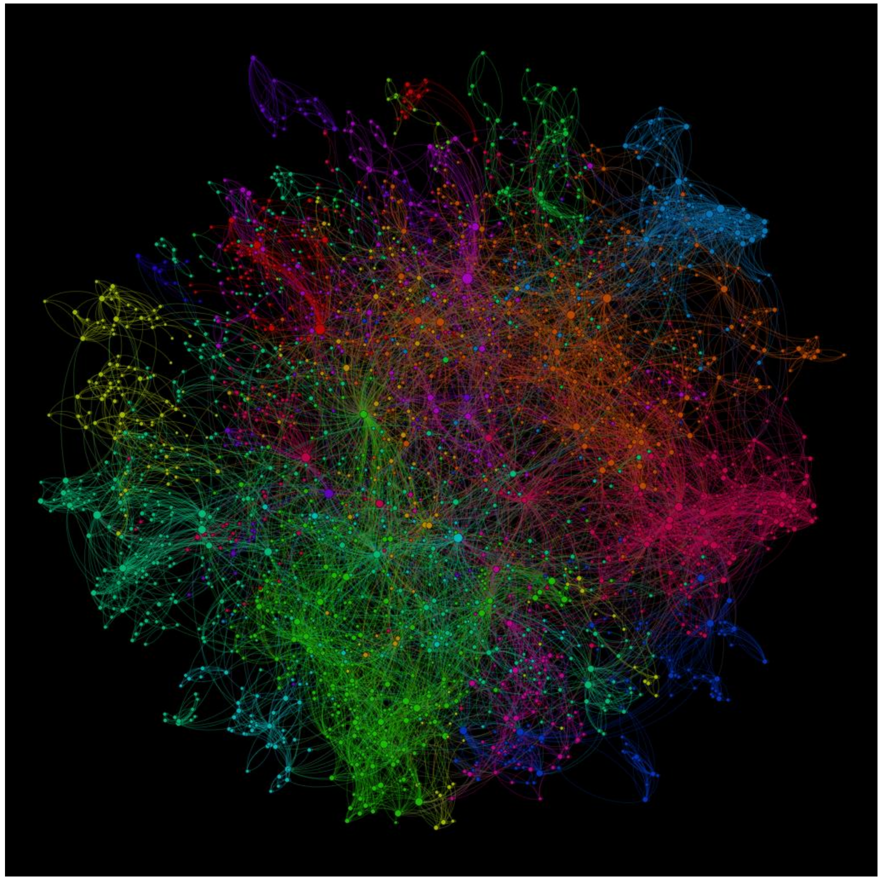
http://images.slideplayer.com/18/5684225/slides/slide_28.jpg

Applications

LinkedIn Social Network Graph



Java Call Graph for Neo4J



http://allthingsgraphed.com/2014/10/16/your-linkedin-network/

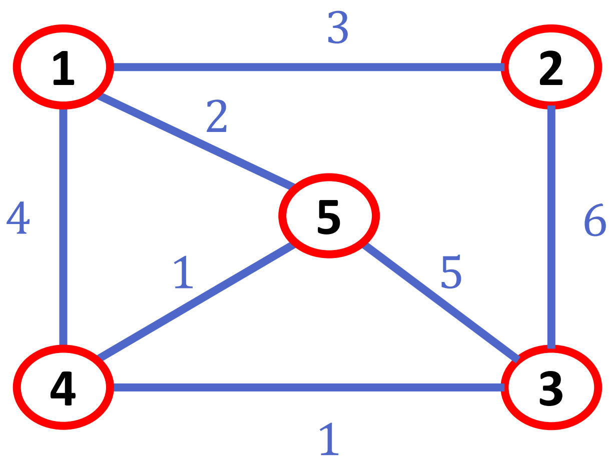
http://allthingsgraphed.com/2014/11/12/code-graphs-5-top-open-source-data-projects/

Terminology

- If (v_0, v_1) is an edge in an **undirected** graph,
  - v_0 and v_1 are adjacent, or are neighbors
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a **directed** graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1
  - v_0 is the source vertex and v_1 is the sink vertex

Terminology

- Vertices & edges can have **labels** that uniquely identify them
    - Edge label can be formed from the pair of vertex labels it is incident upon…*assuming only one edge can exist between a pair of vertices*
- Edge **weights** indicate some measure of distance or cost to pass through that edge



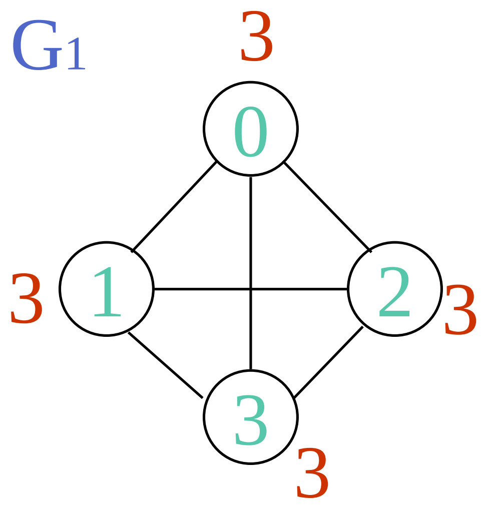
Terminology

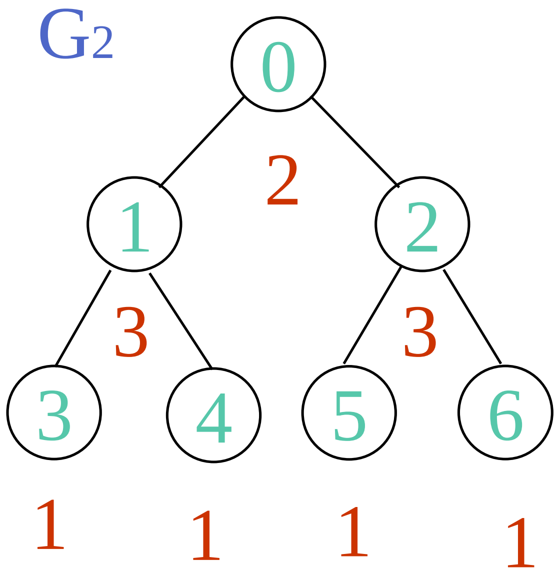
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the sink vertex
  - the out-degree of a vertex *v* is the number of edges that have *v* as the source vertex
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

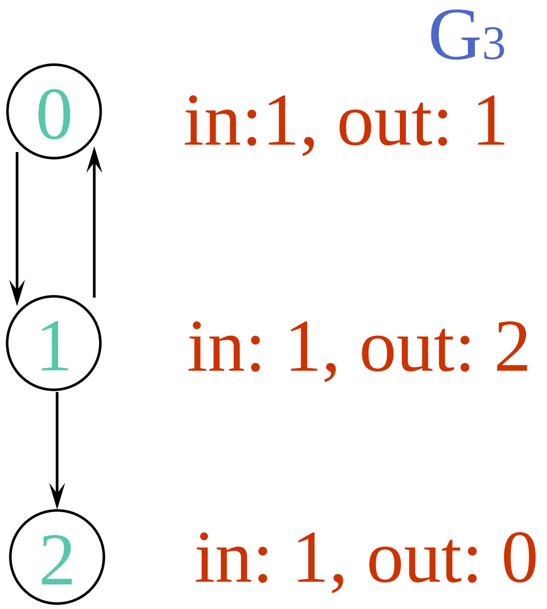
$$e = (\sum_{0}^{n-1} d_i) / 2$$

Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples





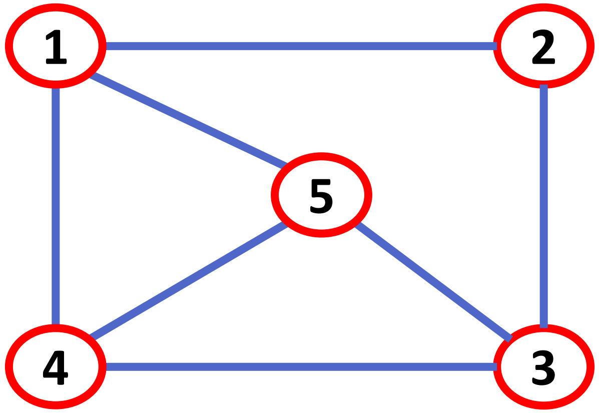


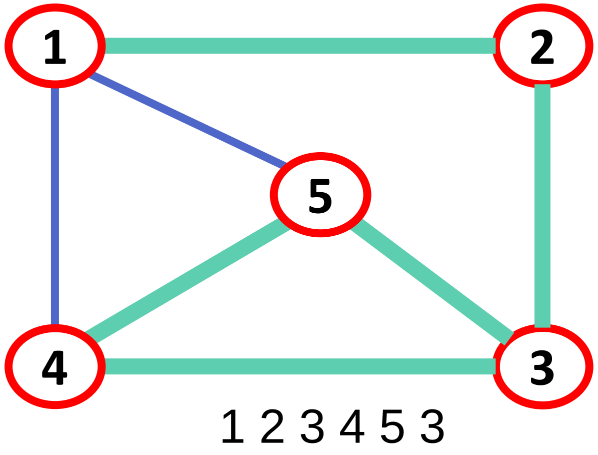
undirected graphs

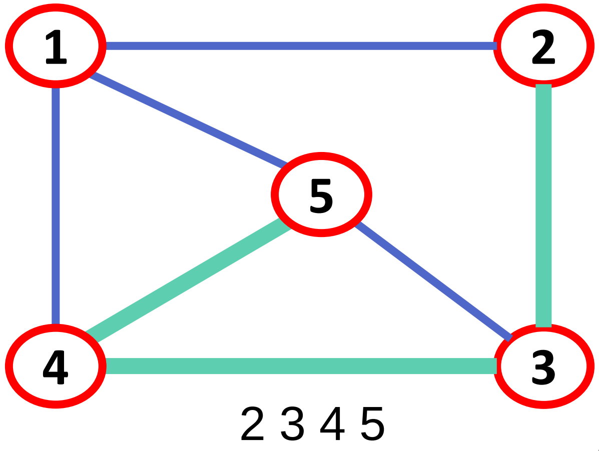
directed graph

Terminology

- path is a sequence of vertices $<v_1, v_2, ... v_k>$ such that consecutive vertices v_i and v_{i+1} are adjacent

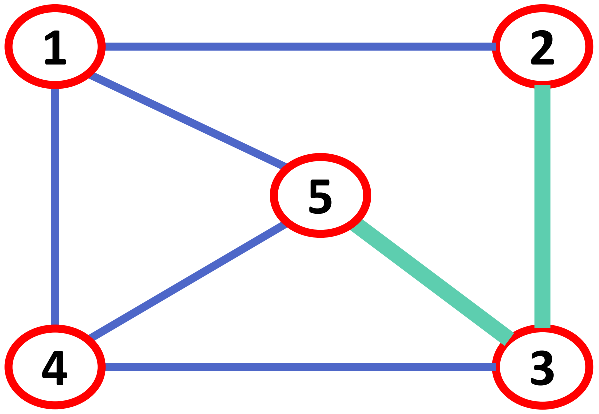






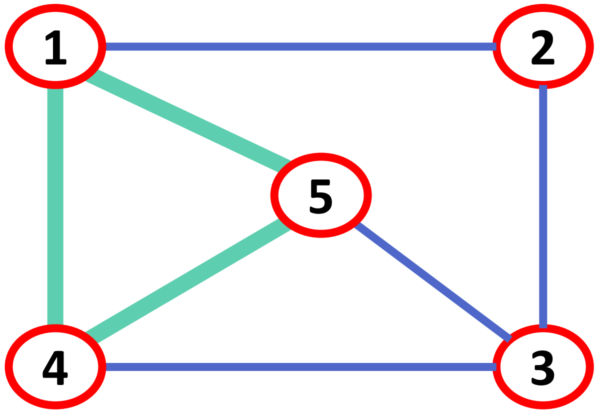
Terminology

- simple path: no repeated vertices



2 3 5

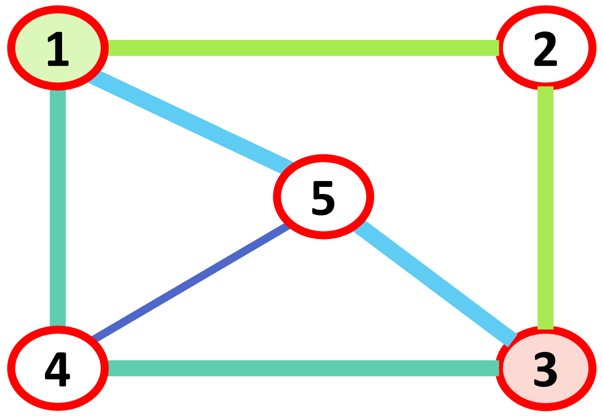
- cycle: simple path, except that the last vertex is the same as the first vertex



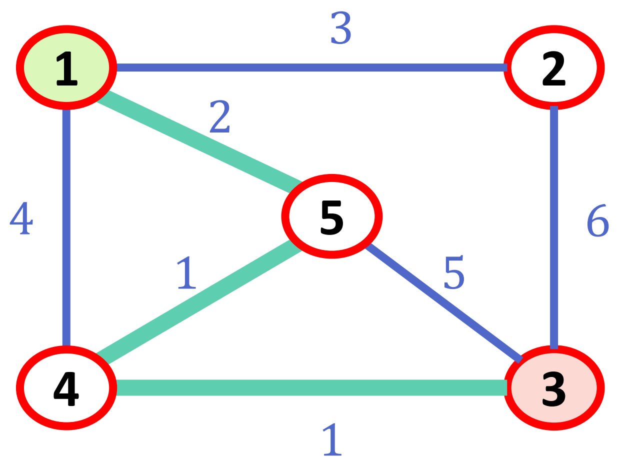
1 5 4 1

Terminology

- Shortest Path: Path between two vertices where the sum of the edge weights is the smallest
  - Has to be a simple path (why?)
  - Assume "unit weight" for edges if not specified



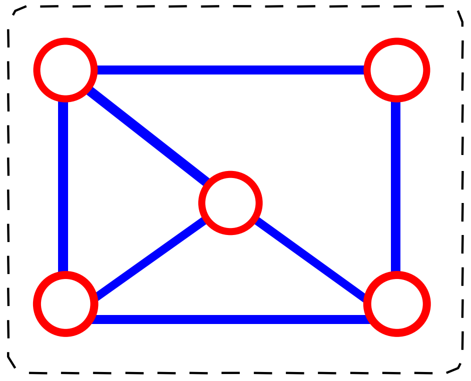
1 2 3
1 5 3
1 4 3

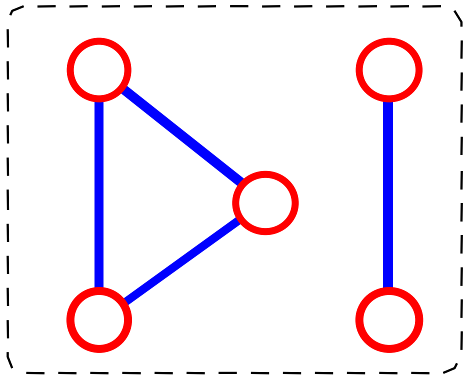


1 5 4 3

Connected Graph

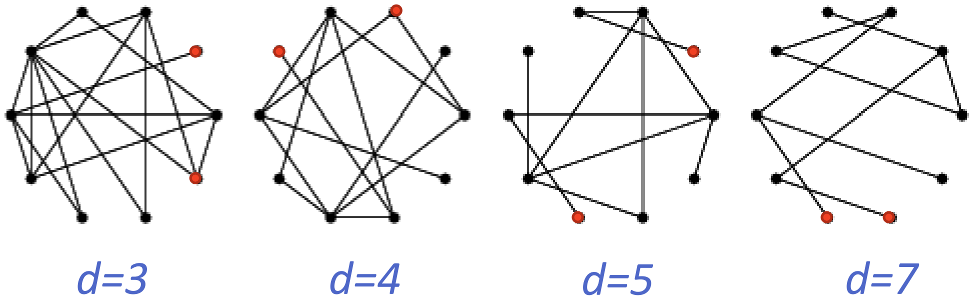
- connected graph: any two vertices are connected by some path


connected


not connected

Graph Diameter

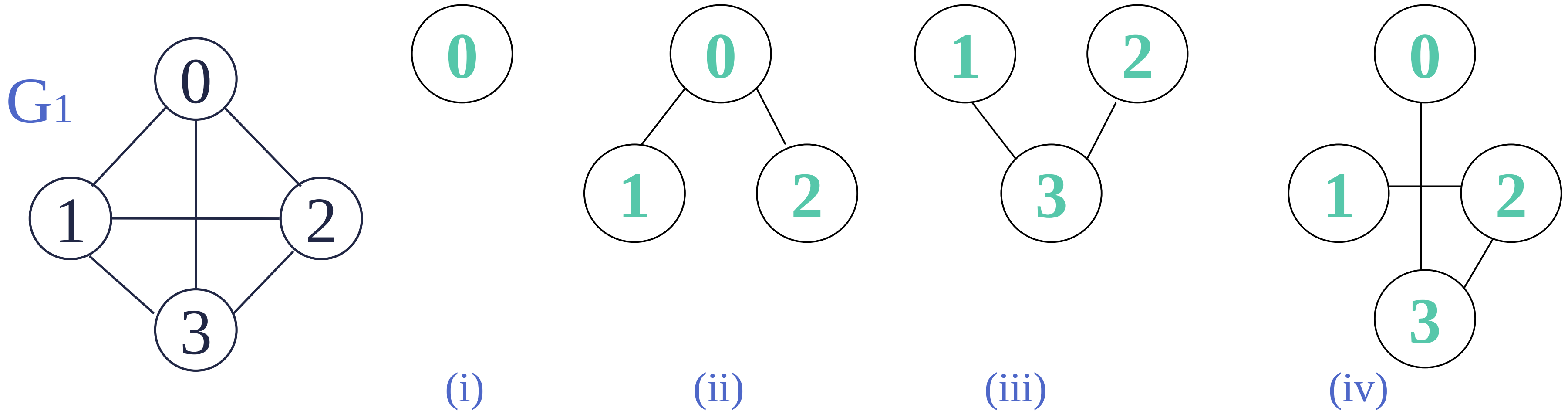
- A graph's dimeter is the distance of its *longest shortest path*
- if d(u,v) is the distance of the shortest path between vertices u and v, then:
- *diameter = Max(d(u,v))*, for all u, v in V
- A disconnected graph has an infinite diameter



http://mathworld.wolfram.com/GraphDiameter.html

Subgraph

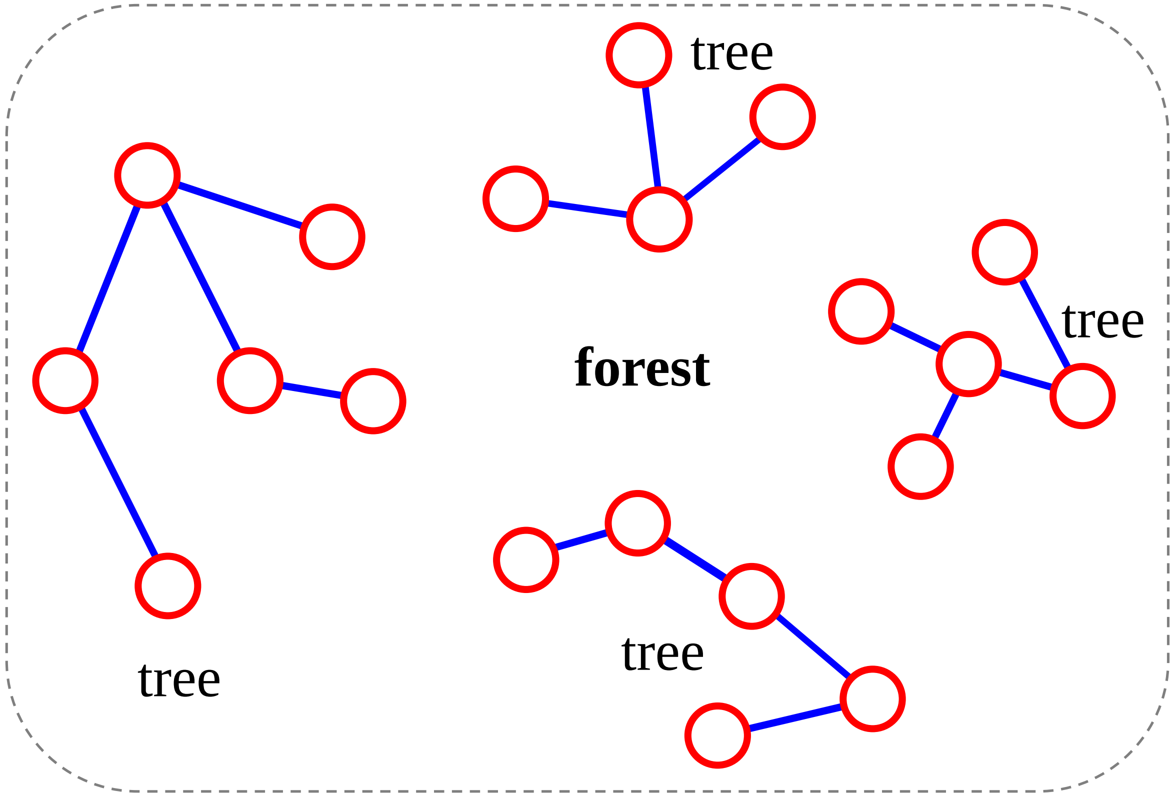
- subgraph: subset of vertices and edges forming a graph



(a) Some of the subgraph of G_1

Trees & Forests

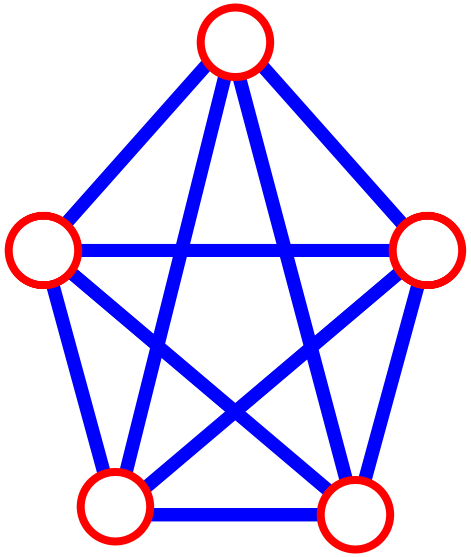
- tree - connected graph without cycles
- forest - collection of trees



Fully Connected Graph

- Let **n** = #vertices, and **m** = #edges
- **Complete graph (or) Fully connected graph**: One in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** -1)/2.

If a graph is not complete:
m < **n**(**n** -1)/2



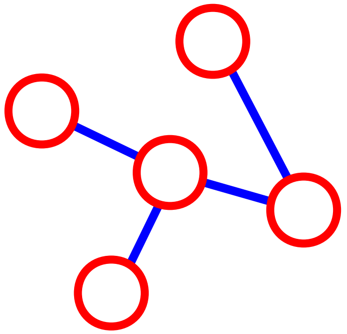
n = 5
m = (5*4)/2 = 10

More Connectivity

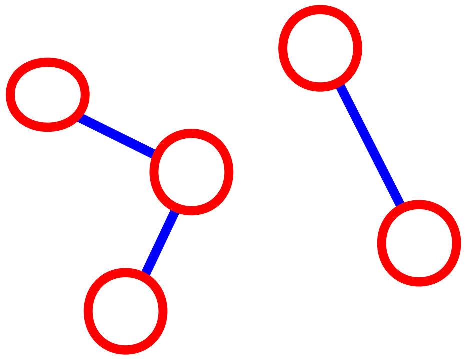
n = #vertices

m = #edges

- For a tree **m** = **n** - 1



$\mathbf{n} = 5$
$\mathbf{m} = 4$

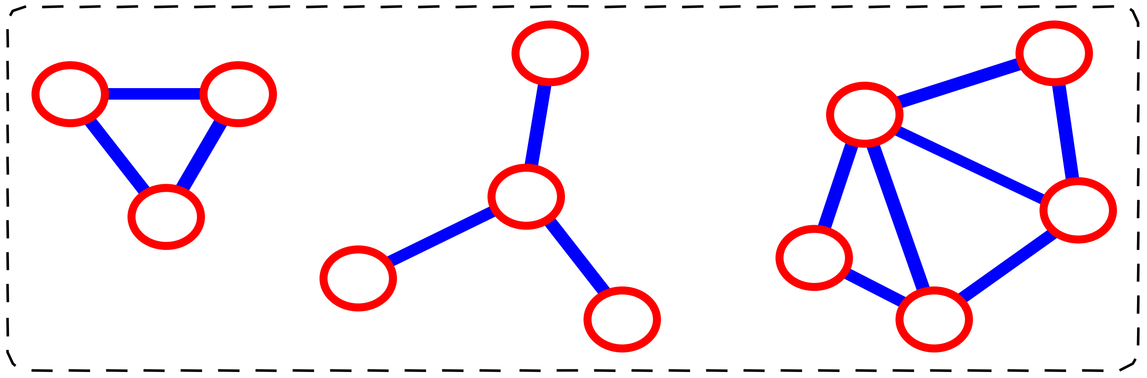


$\mathbf{n} = 5$
$\mathbf{m} = 3$

If **m** < **n** - 1, G is not connected

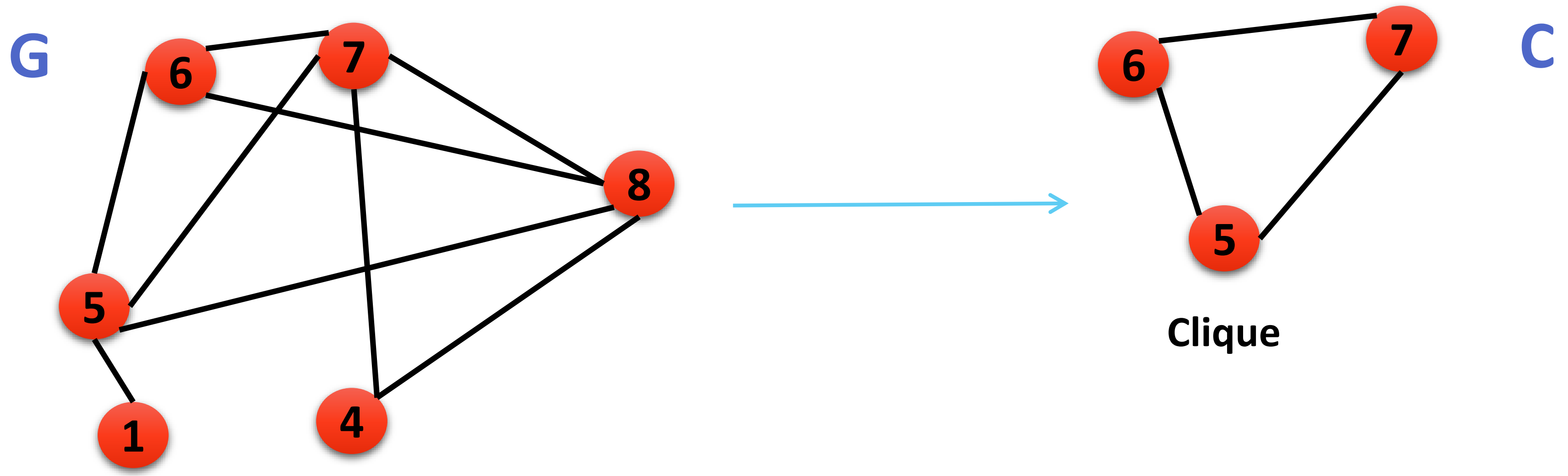
Connected Component

- A connected component is a maximal subgraph that is connected.
  - Cannot add vertices and edges from original graph and retain connectedness.
- A connected graph has exactly 1 component.



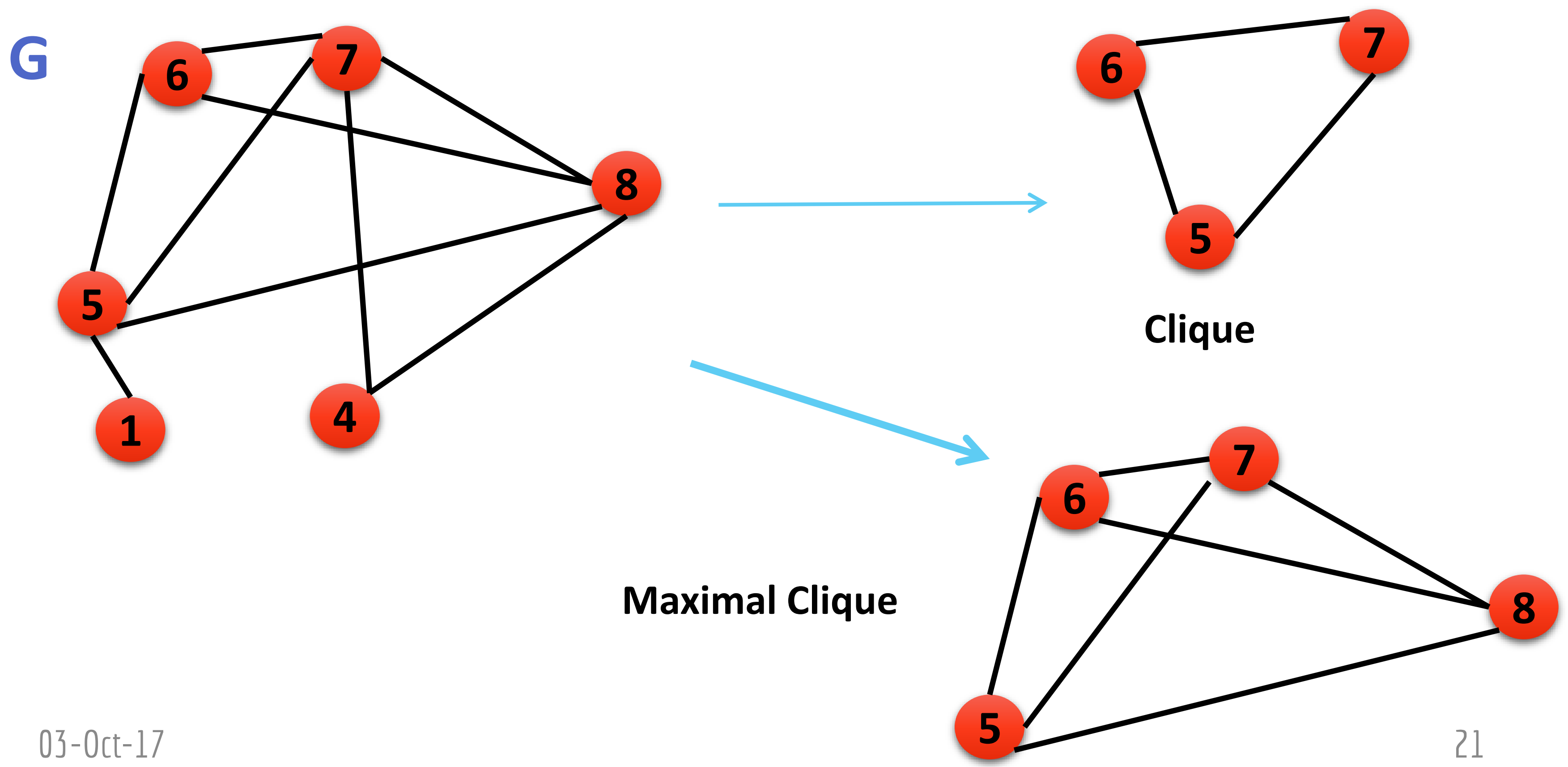
Clique

- A subgraph C of a graph G with *edges between all pairs of vertices*



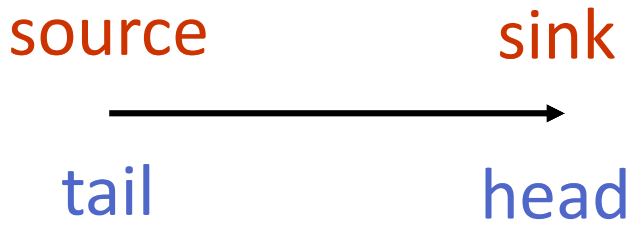
Maximal Clique

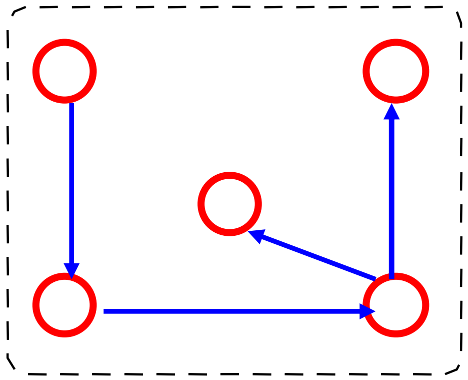
- A maximal clique is a clique that is not part of a larger clique



Directed vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph (or **Digraph**) is one in which each edge is a directed pair of vertices, $<v_0, v_1> \, != \, <v_1, v_0>$



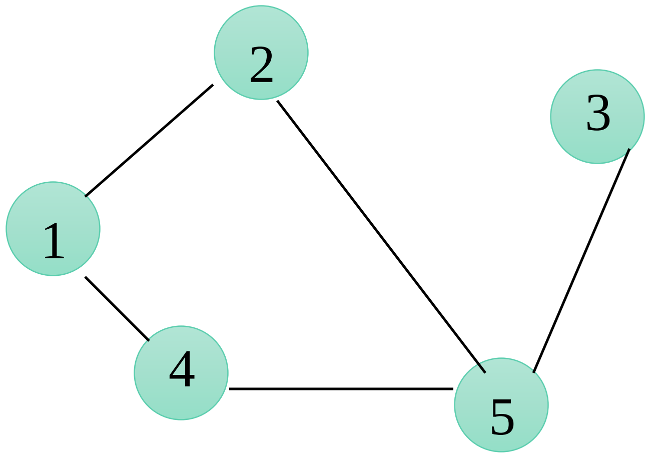


Graph Representation

- Adjacency Matrix
- Adjacency Lists
  - Linked Adjacency Lists
  - Array Adjacency Lists

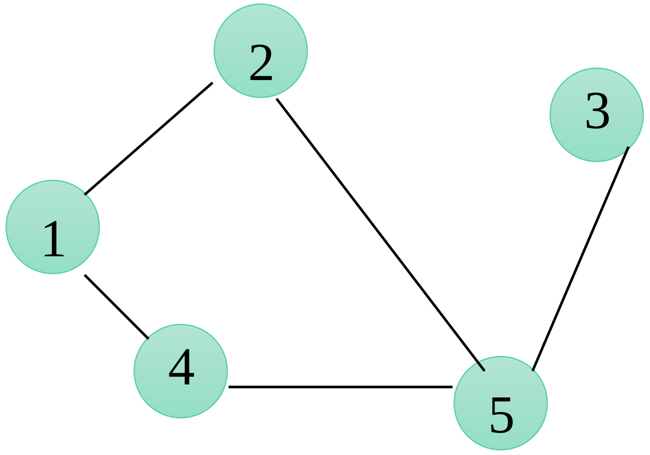
Adjacency Matrix

- 0/1 n x n matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \end{array}$$

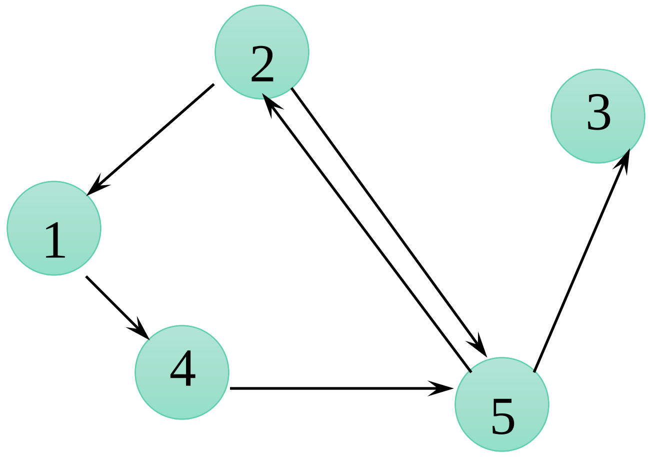
Adjacency Matrix Properties



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \end{array}$$

- Diagonal entries are zero.
- Adjacency matrix of an *undirected graph* is *symmetric*.
  - A(i,j) = A(j,i) for all i and j.

Adjacency Matrix (Digraph)



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 1 & 1 & 0 & 0 \end{array}$$

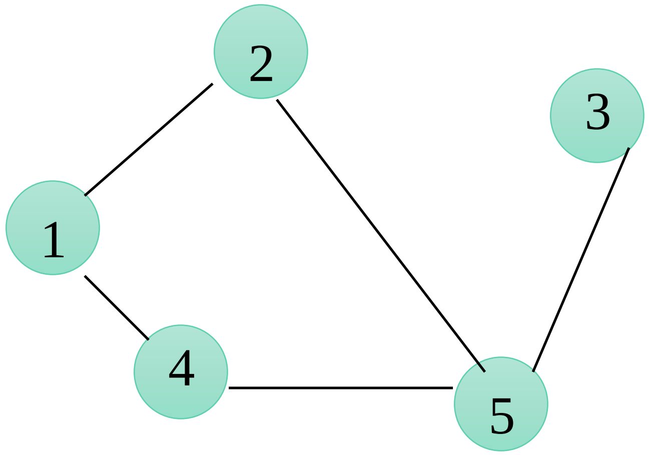
- Diagonal entries are zero.
- Adjacency matrix of a directed graph need not be symmetric.

Adjacency Matrix

- n^2 bits of space
- For an *undirected graph*, may store only lower or upper triangle (exclude diagonal)
  - $(n^2 - n)/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex.

Adjacency Lists

- Adjacency list for vertex *i* is a linear list of vertices adjacent from vertex *i*.
- An array of *n* adjacency lists.



aList[1] = (2,4)

aList[2] = (1,5)

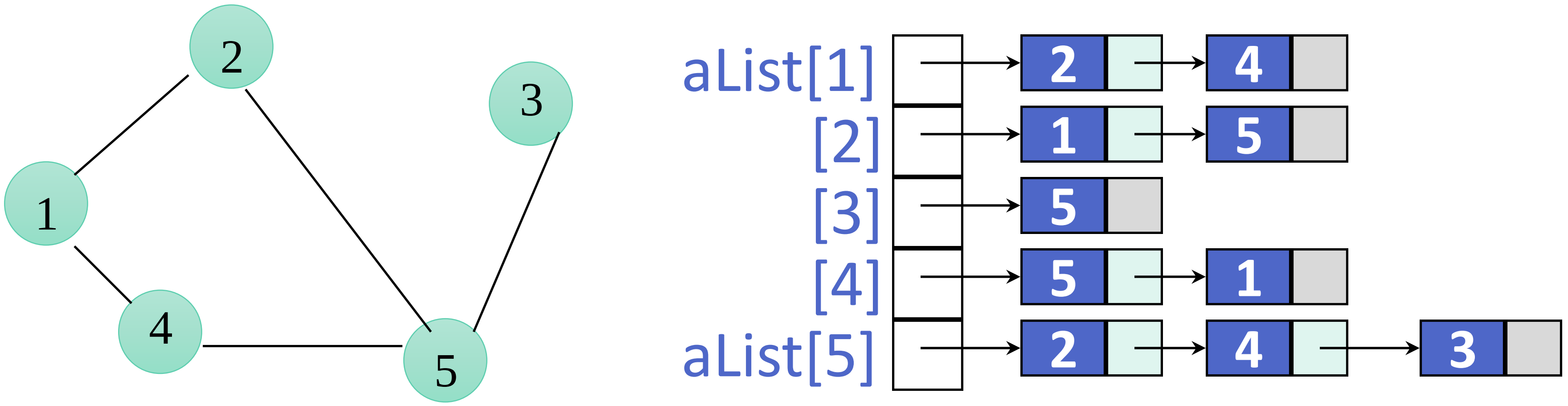
aList[3] = (5)

aList[4] = (5,1)

aList[5] = (2,4,3)

Linked Adjacency Lists

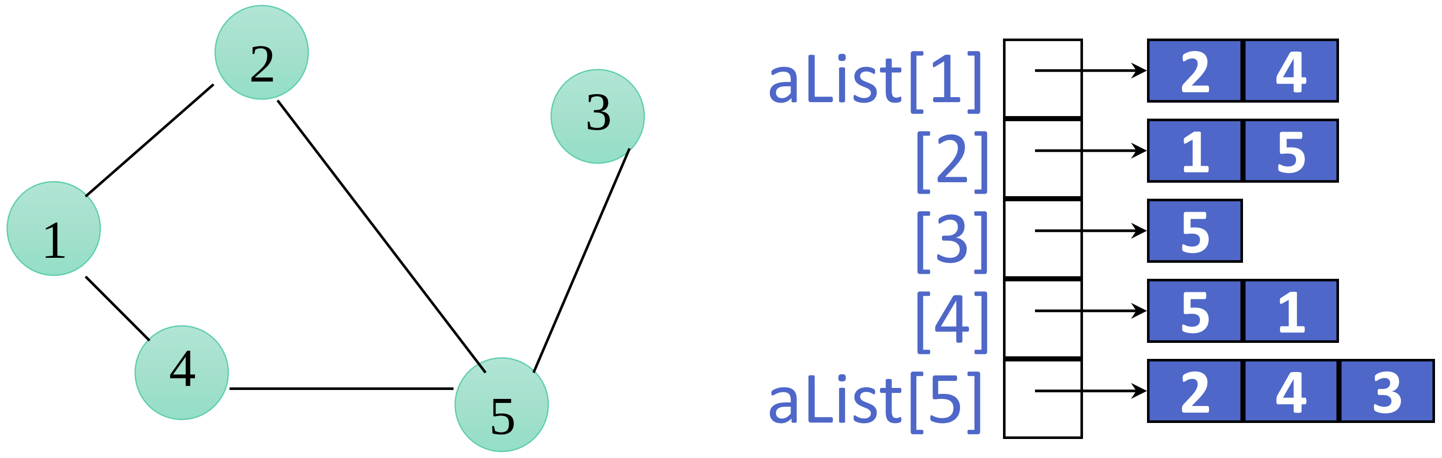
- Each adjacency list is a chain.



- Array Length = n
- # of chain nodes = 2e (undirected graph)
- # of chain nodes = e (digraph)

Array Adjacency Lists

- Each adjacency list is an array list.



- Array Length = n
- # of list elements = 2e (undirected graph)
- # of list elements = e (digraph)

Storing Weighted Graphs

- Cost adjacency matrix
  - C(i,j) = cost of edge (i,j) instead of 0/1
- Adjacency lists
  - Each list element is a pair (adjacent vertex, edge weight)

ADT for Graph

```
class Vertex<V,E> {
   int id;
   V value;
   int GetId();
   V GetValue();
   List<Edge<V,E>> Neighbors();
}
class Edge<V,E> {
   int id;
   E value;
   int GetId();
   E GetValue();
   Vertex<V,E> GetSource();
   Vertex<V,E> GetSink();
}
```

ADT for Graph

```
class Graph<V,E>{
   List<Vertex<V,E>> vertices;
   List<Edge<V,E>> edges;

   void InsertVertex(Vertex<V,E> v);
   void InsertEdge(Edge<V,E> e);

   bool DeleteVertex(int vid);
   bool DeleteEdge(int eid);

   List<Vertex<V,E>> GetVertices();
   List<Edge<V,E>> GetEdges();

   bool IsEmpty(graph);
}
```

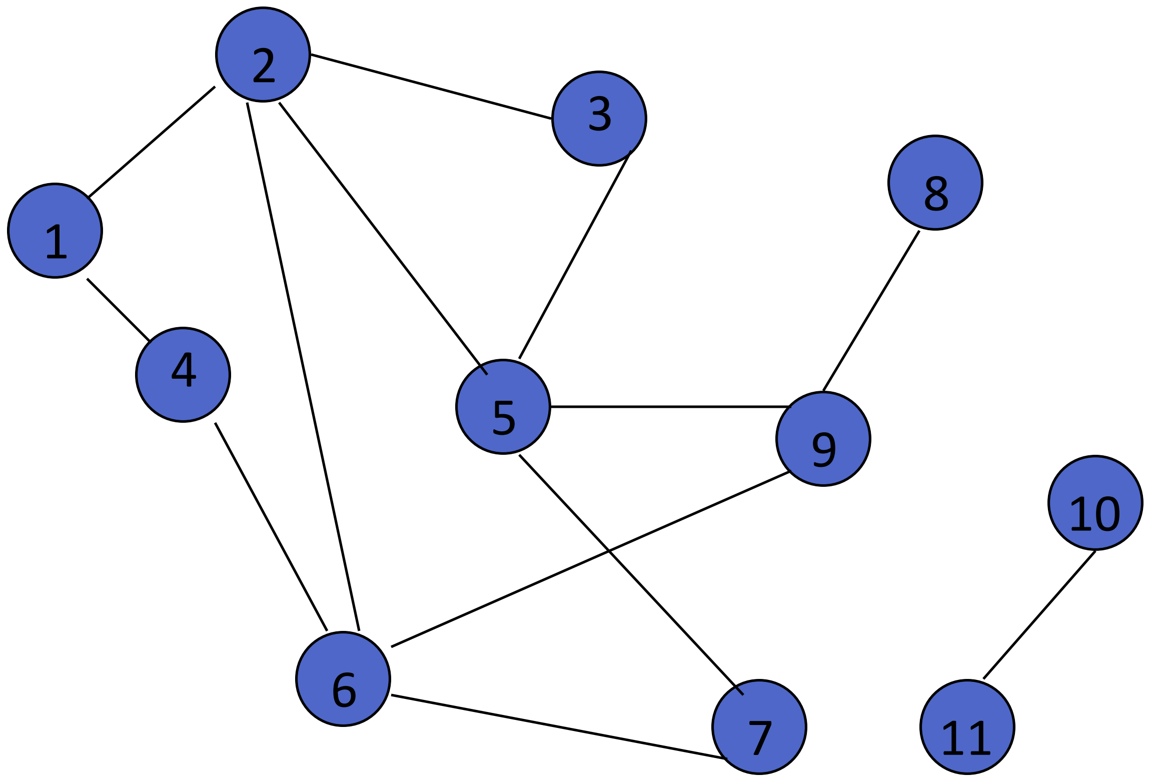
Sample Graph Problems

- Graph traversal
  - Searching
  - Shortest Paths
  - Connectedness
  - Spanning tree
- Graph centrality
  - PageRank
  - Betweenness centrality
- Graph clustering

Graph Search & Traversal

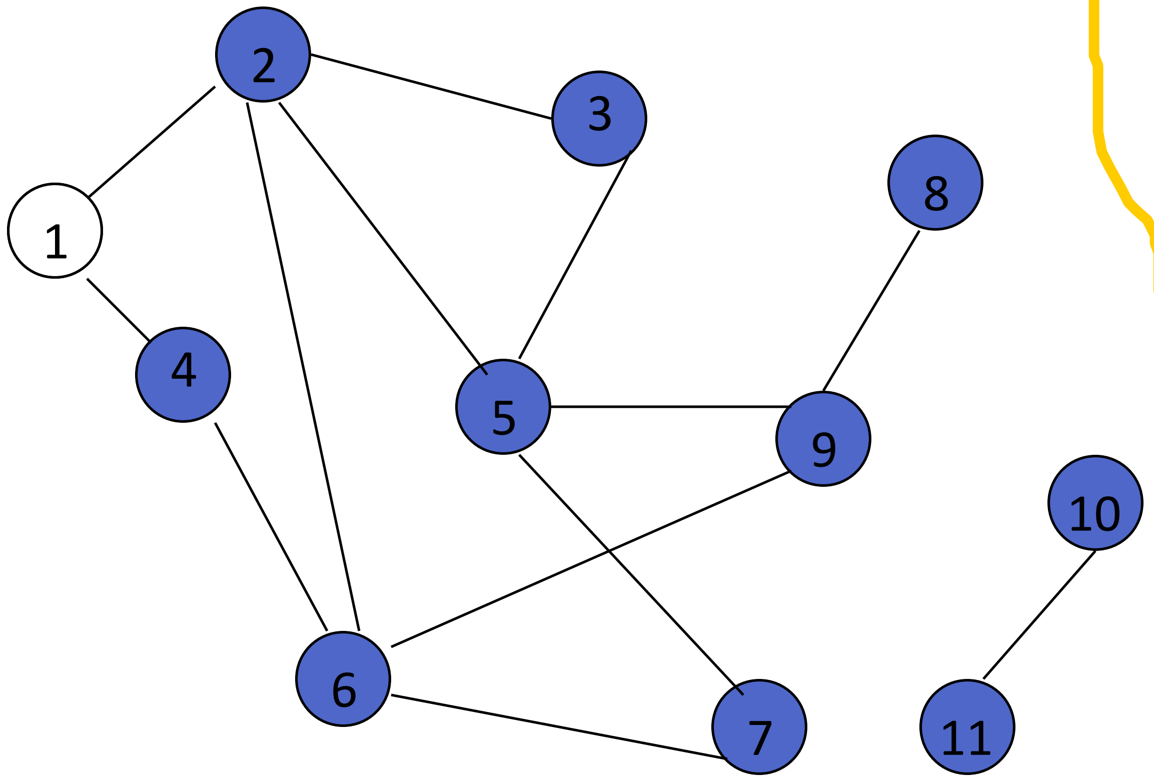
- Find a vertex (or edge) with a given ID or value
  - If list of vertices/edges is available, linear scan!
  - BUT, goal here is to traverse the neighbors of the graph, not scan the list
- Traverse through the graph to list all vertices in a particular order
  - Finding the item can be side-effect of traversal

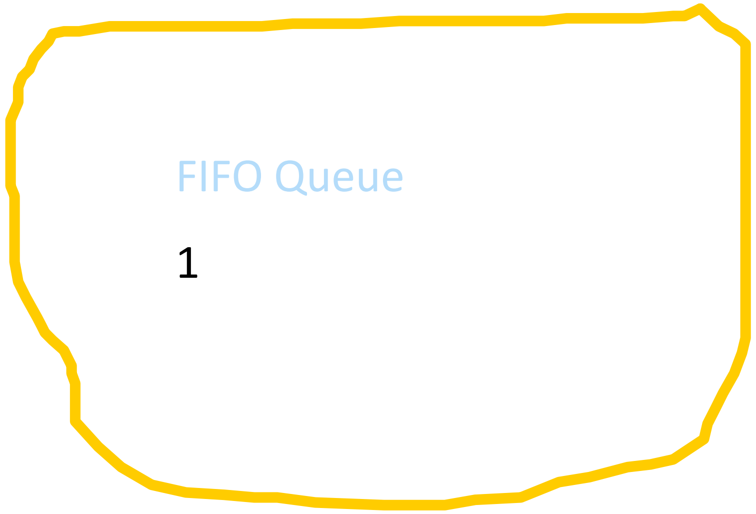
Breadth-First Search Example



Start search at vertex 1.

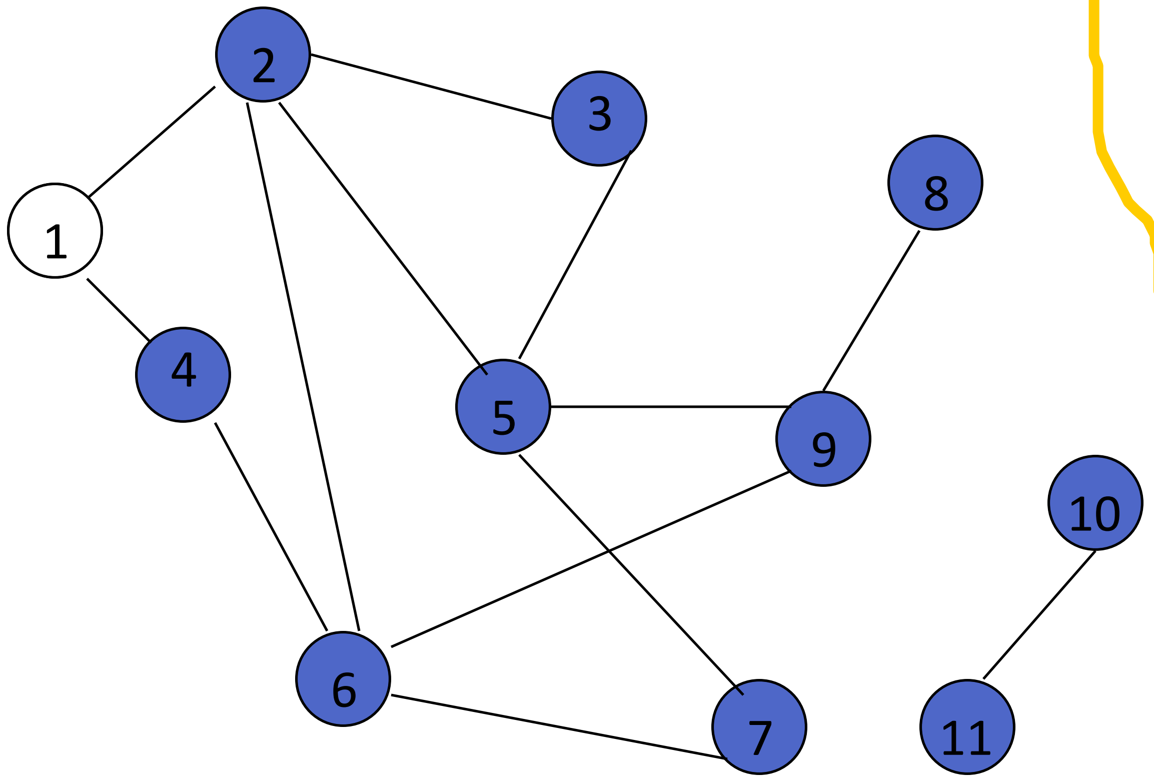
Breadth-First Search Example

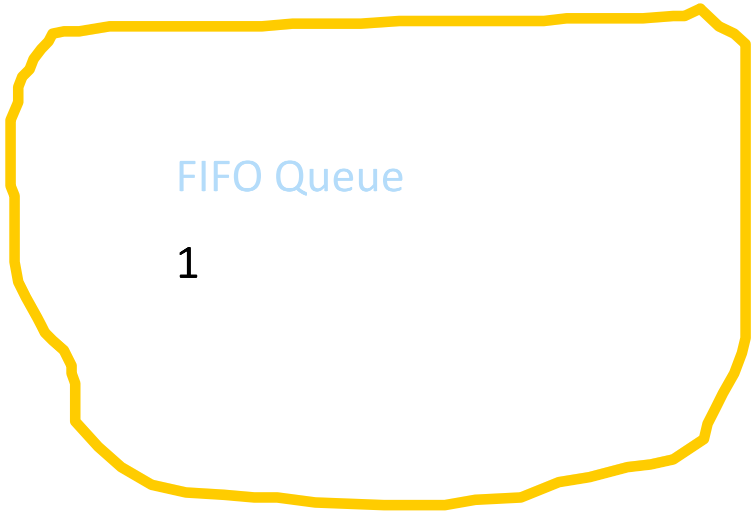




Visit/mark/label start vertex and put in a FIFO queue.

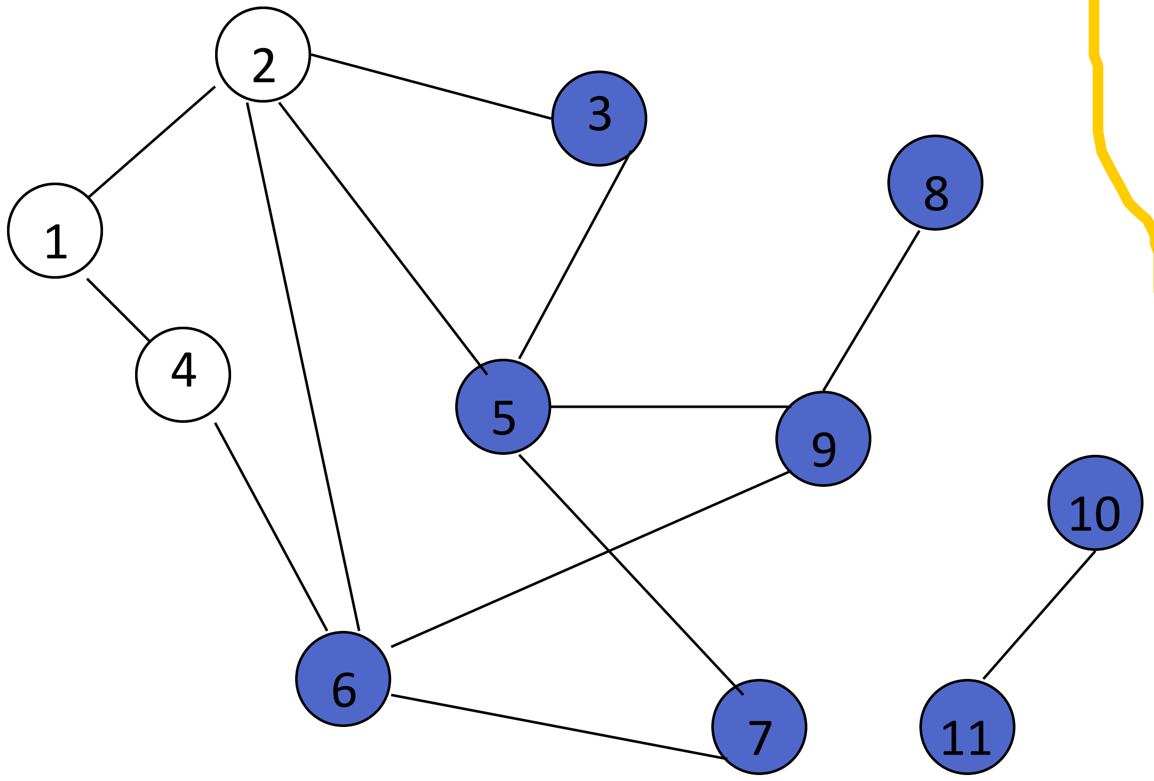
Breadth-First Search Example

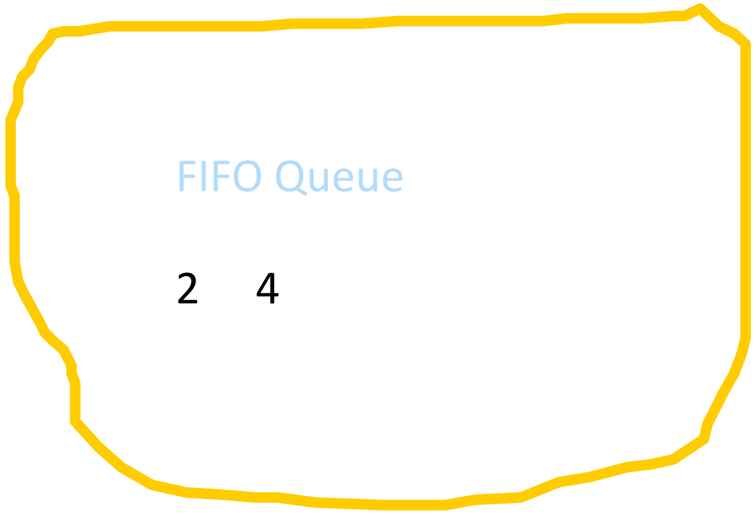




Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

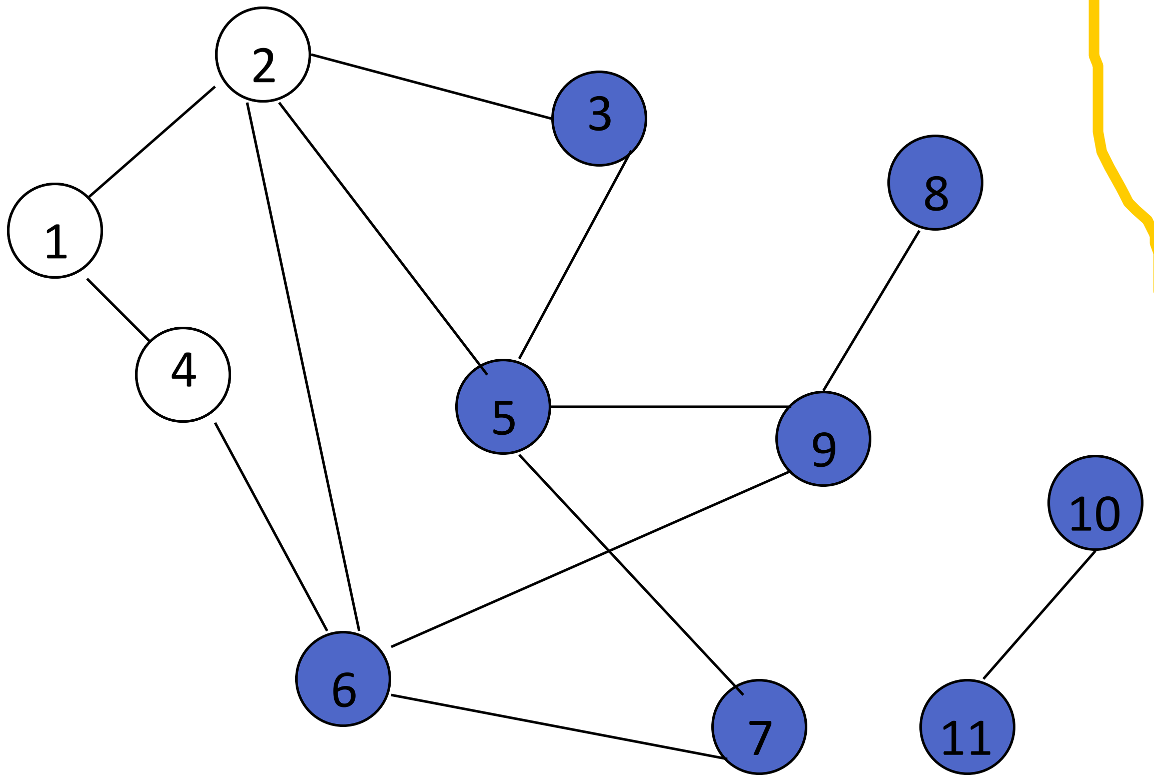
Breadth-First Search Example

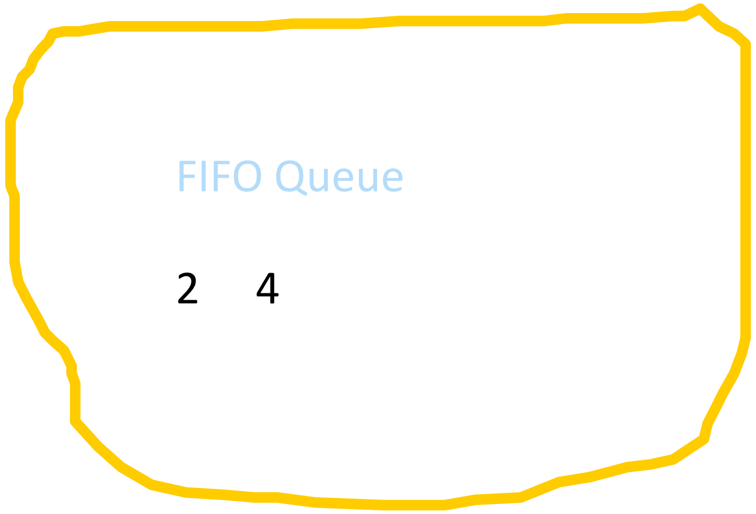




Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

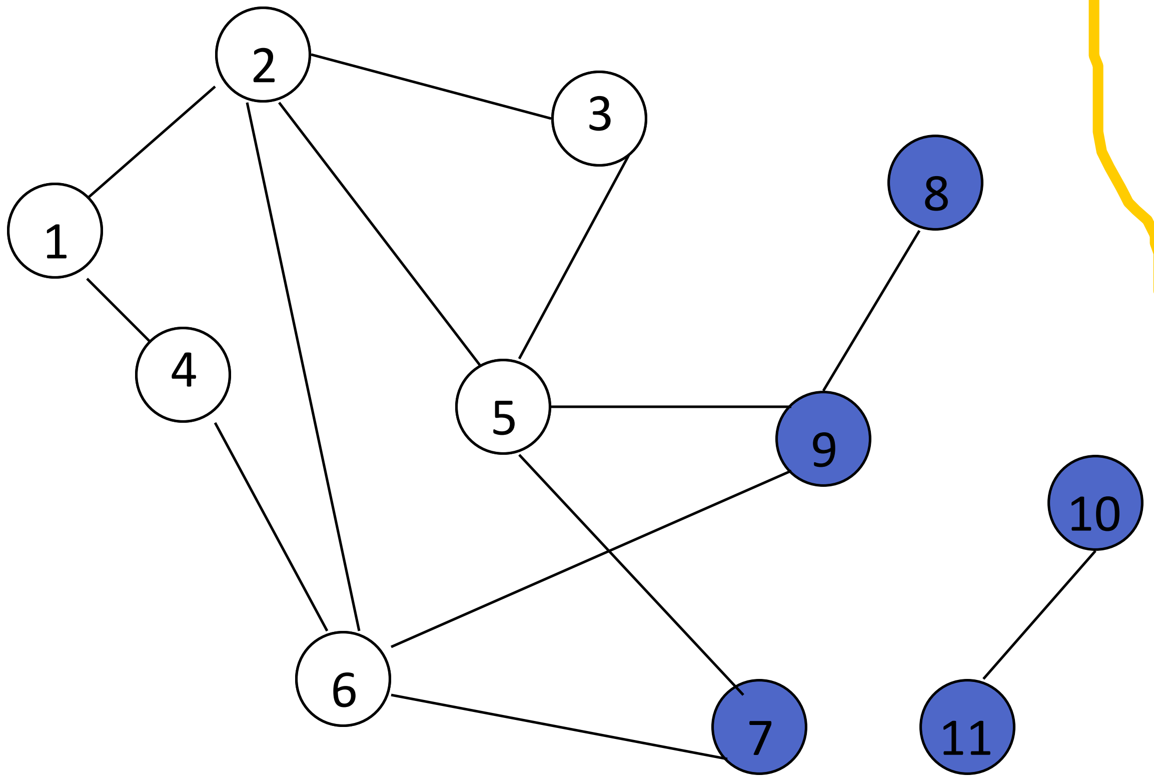
Breadth-First Search Example

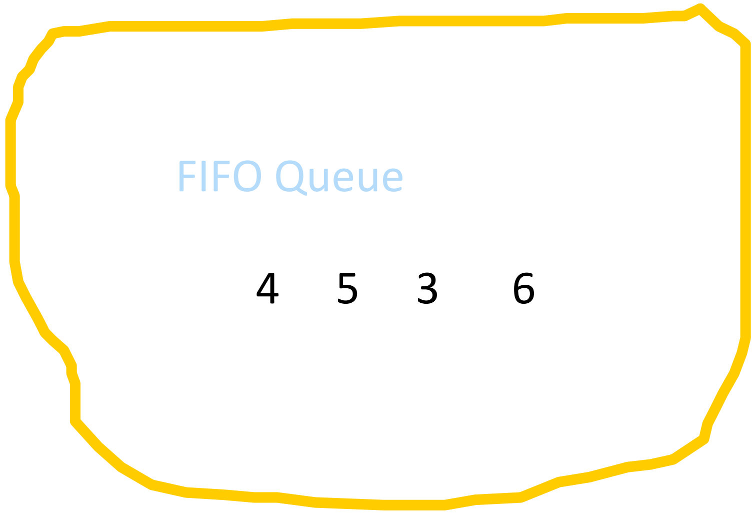




Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

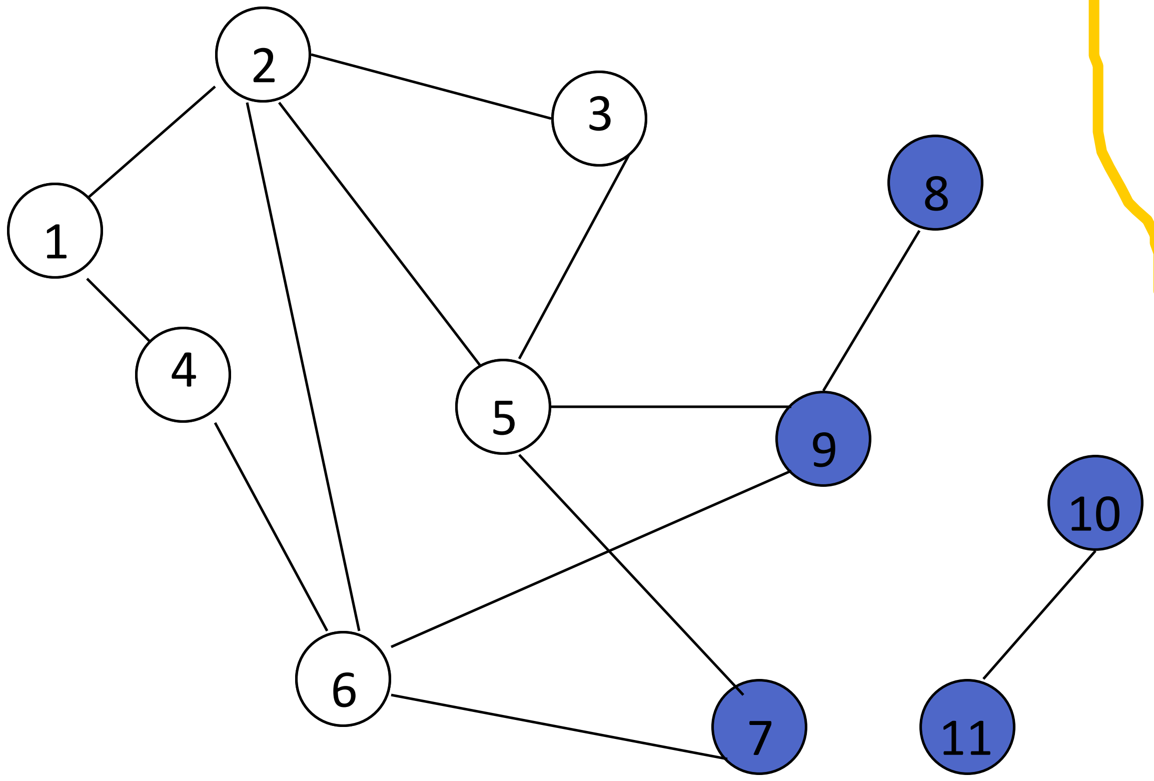
Breadth-First Search Example

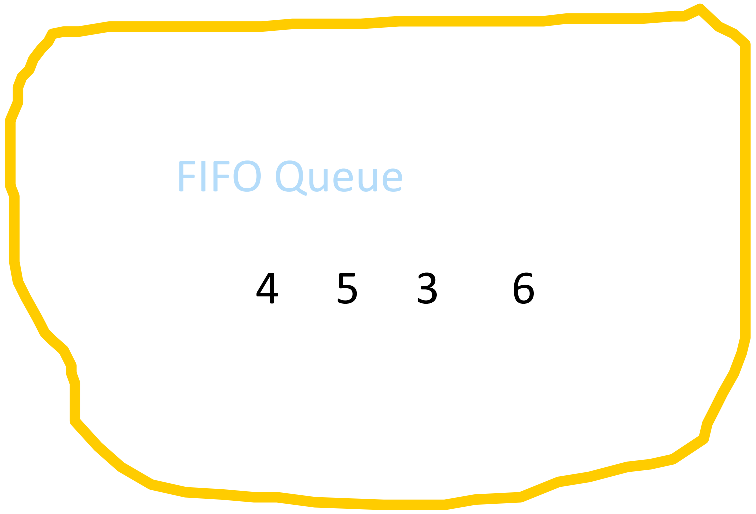




Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

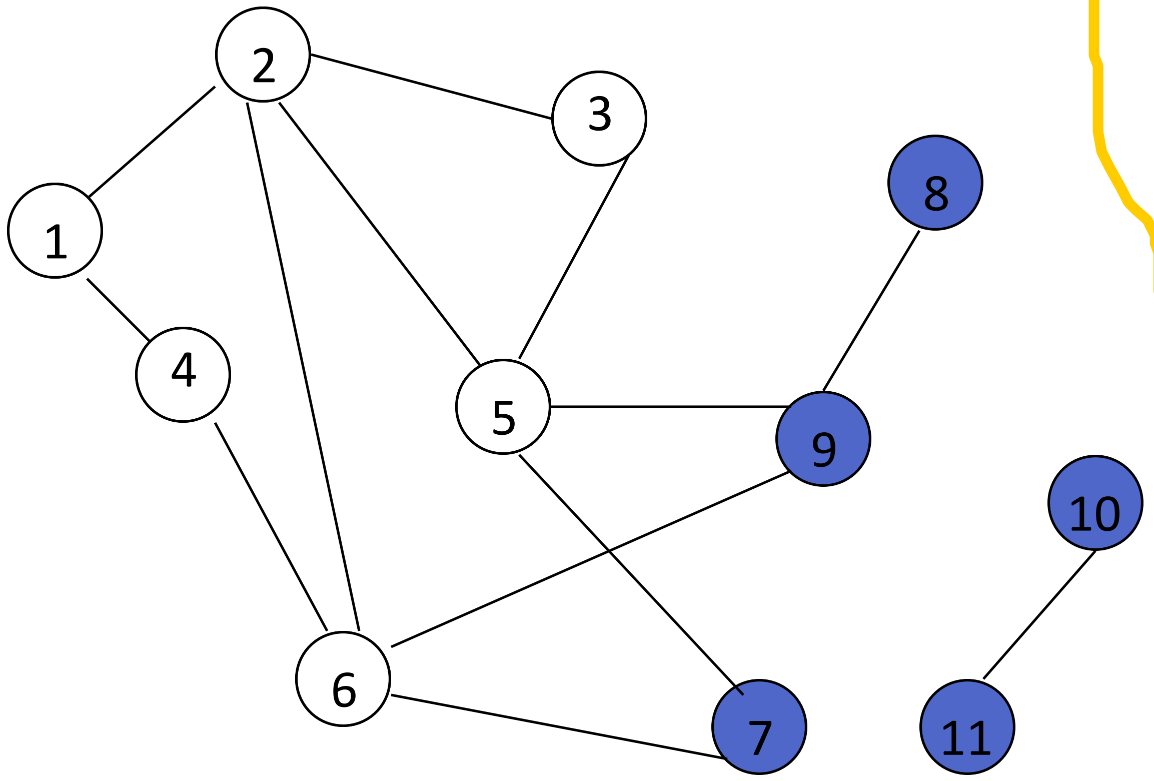
Breadth-First Search Example

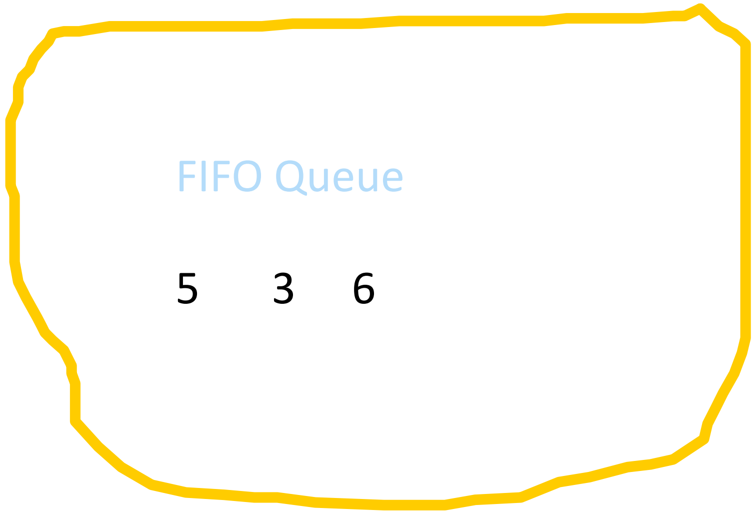




Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

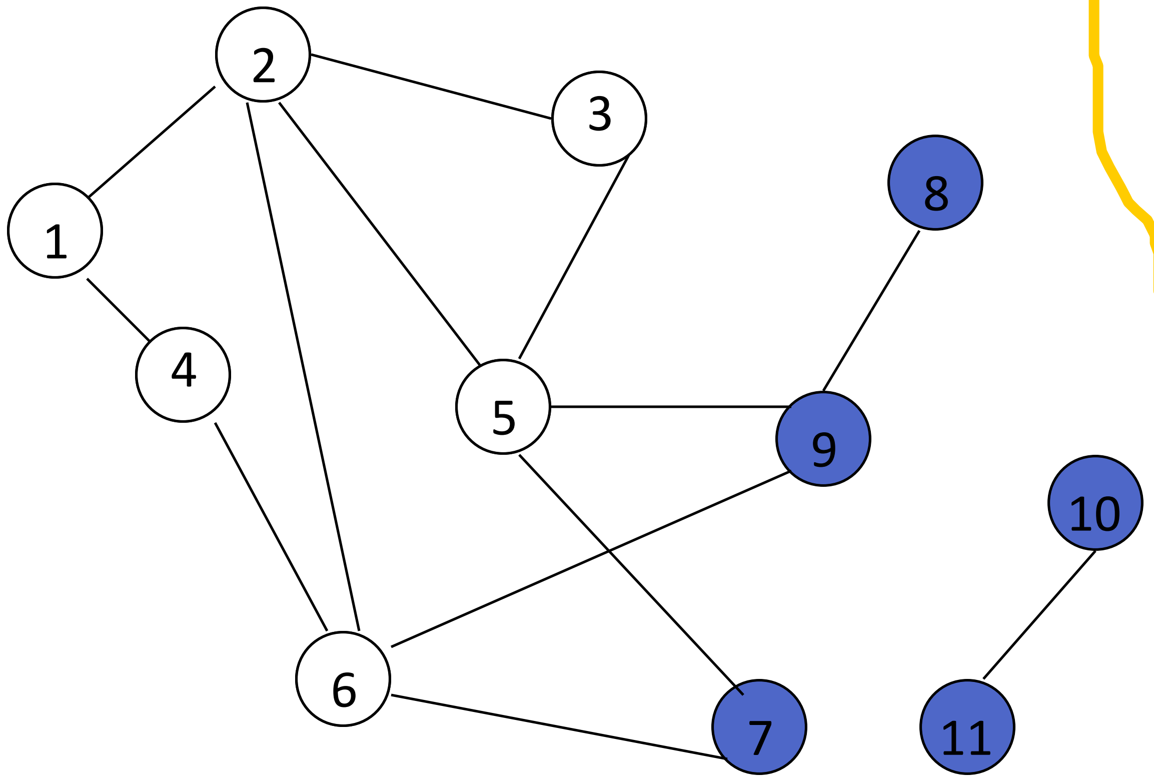
Breadth-First Search Example

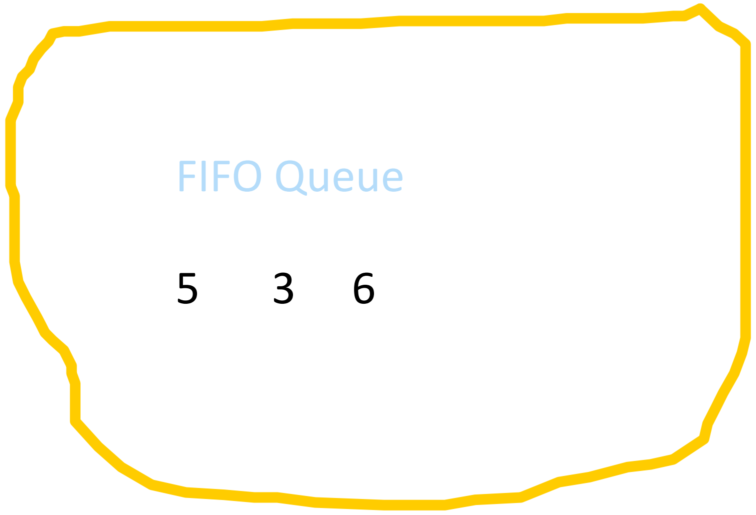




Remove 4 from Q; visit adjacent unvisited vertices;
put in Q.

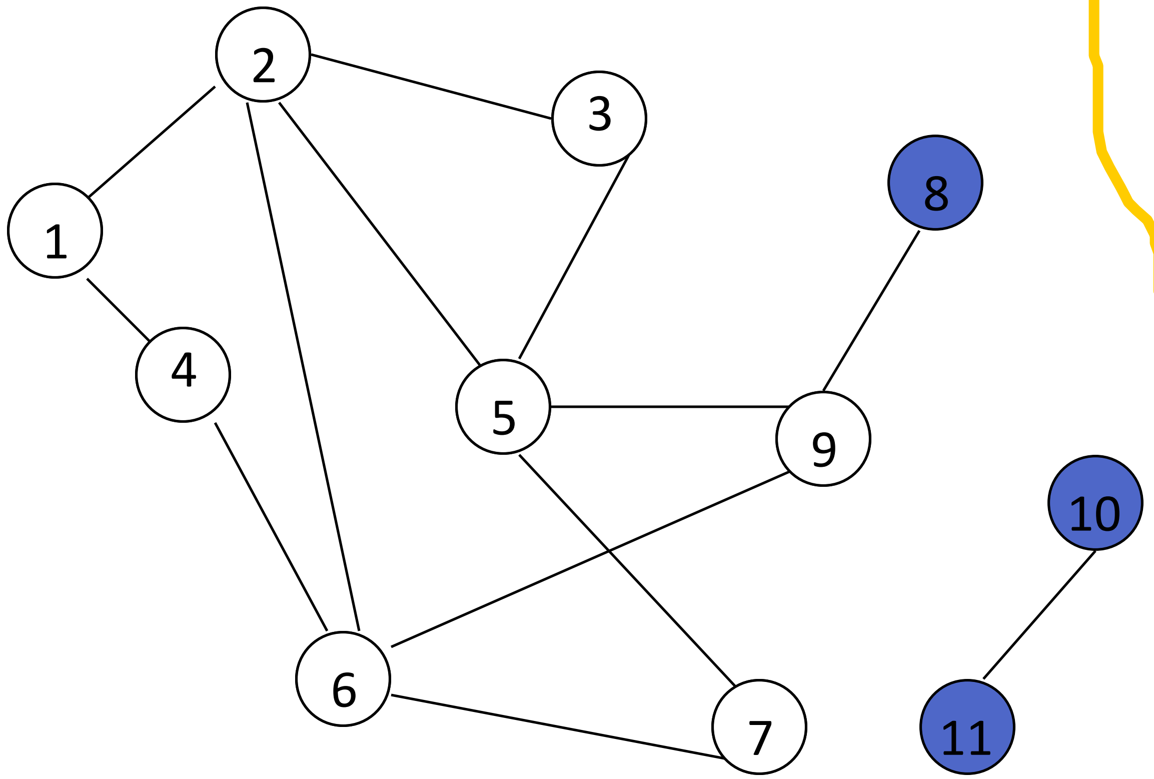
Breadth-First Search Example

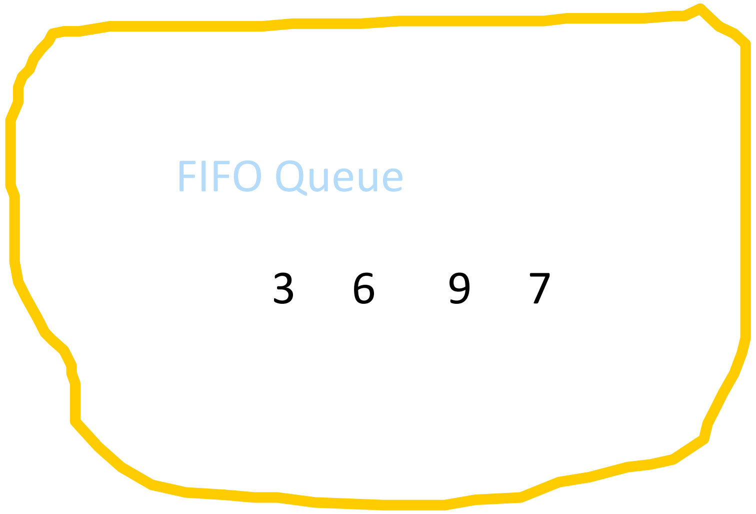




Remove 5 from Q; visit adjacent unvisited vertices;
put in Q.

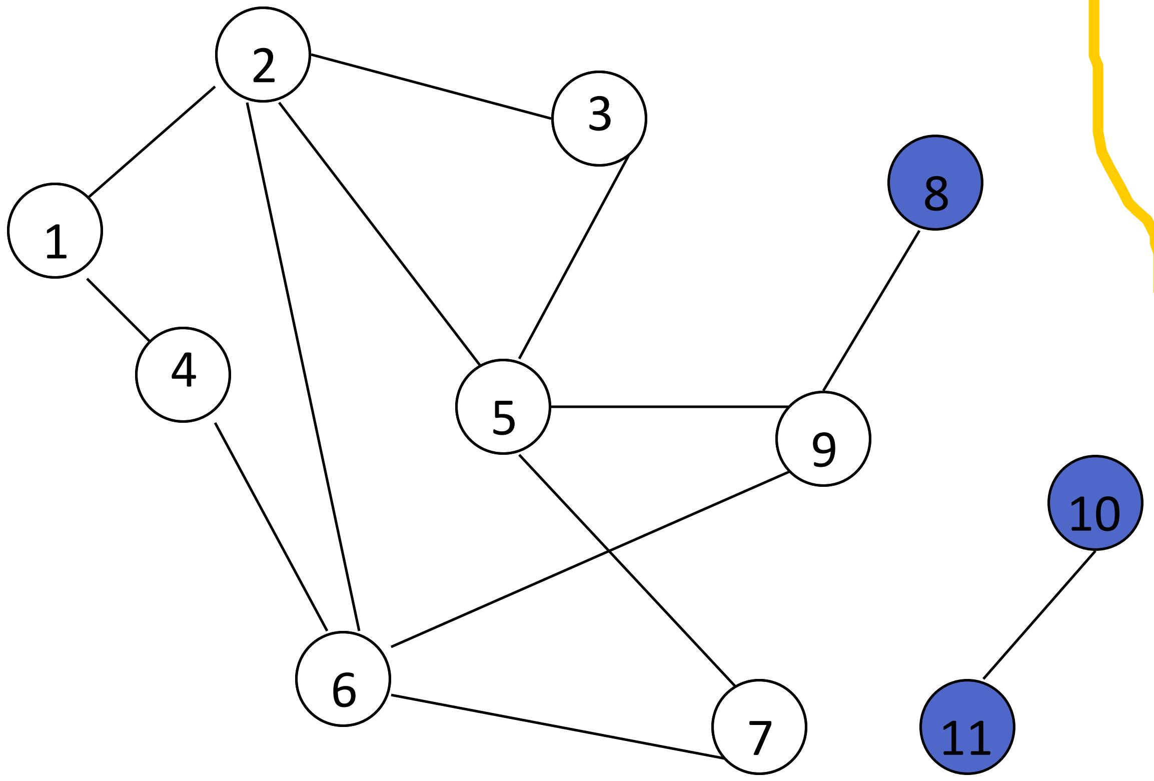
Breadth-First Search Example

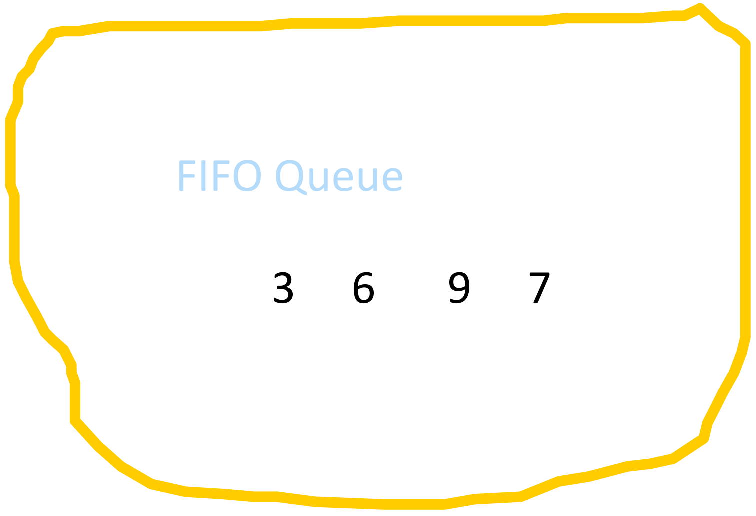




Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

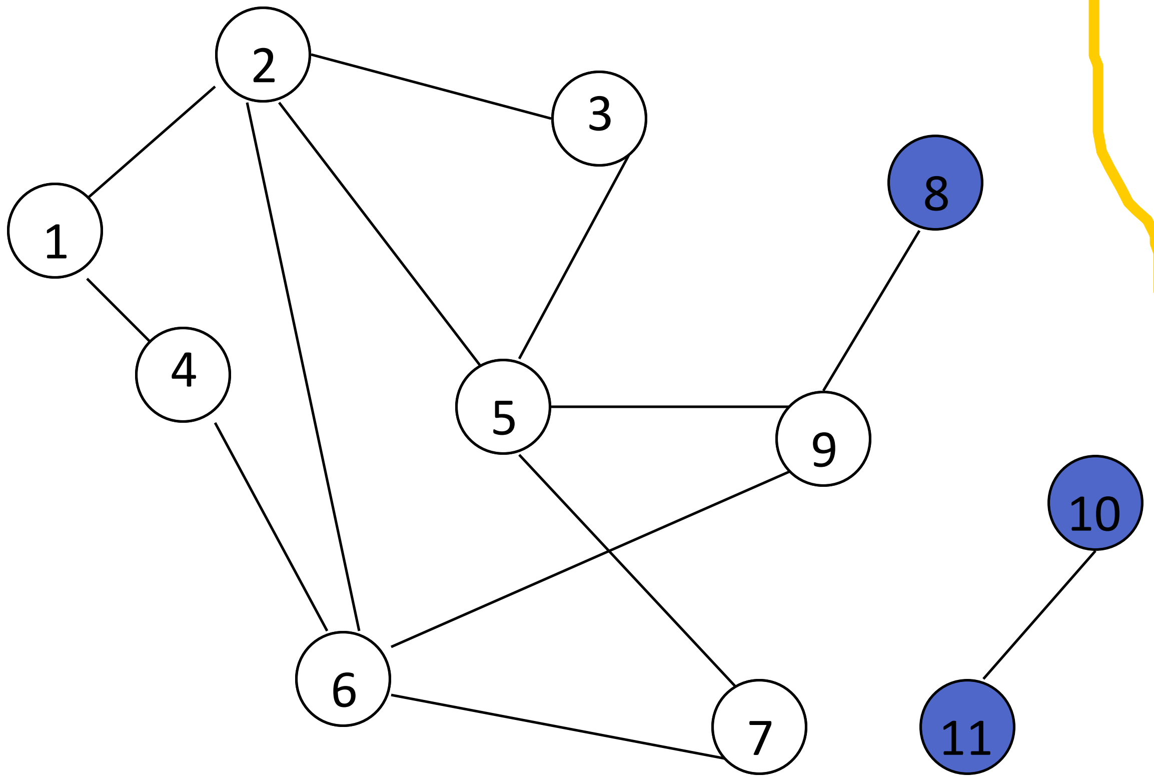
Breadth-First Search Example

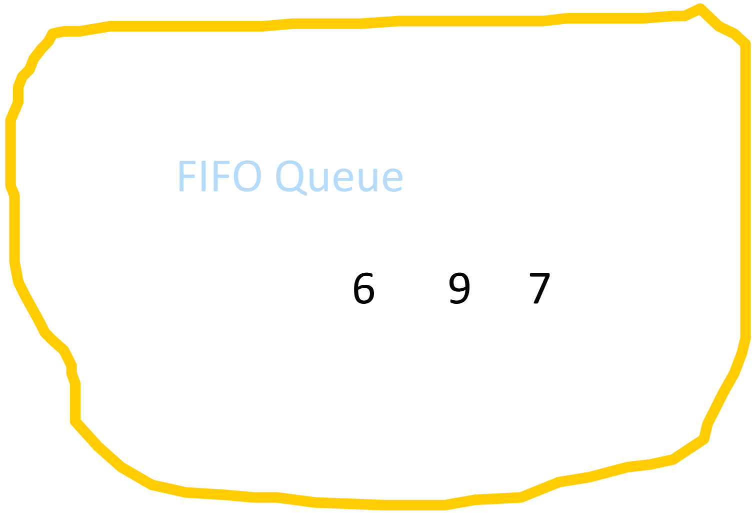




Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

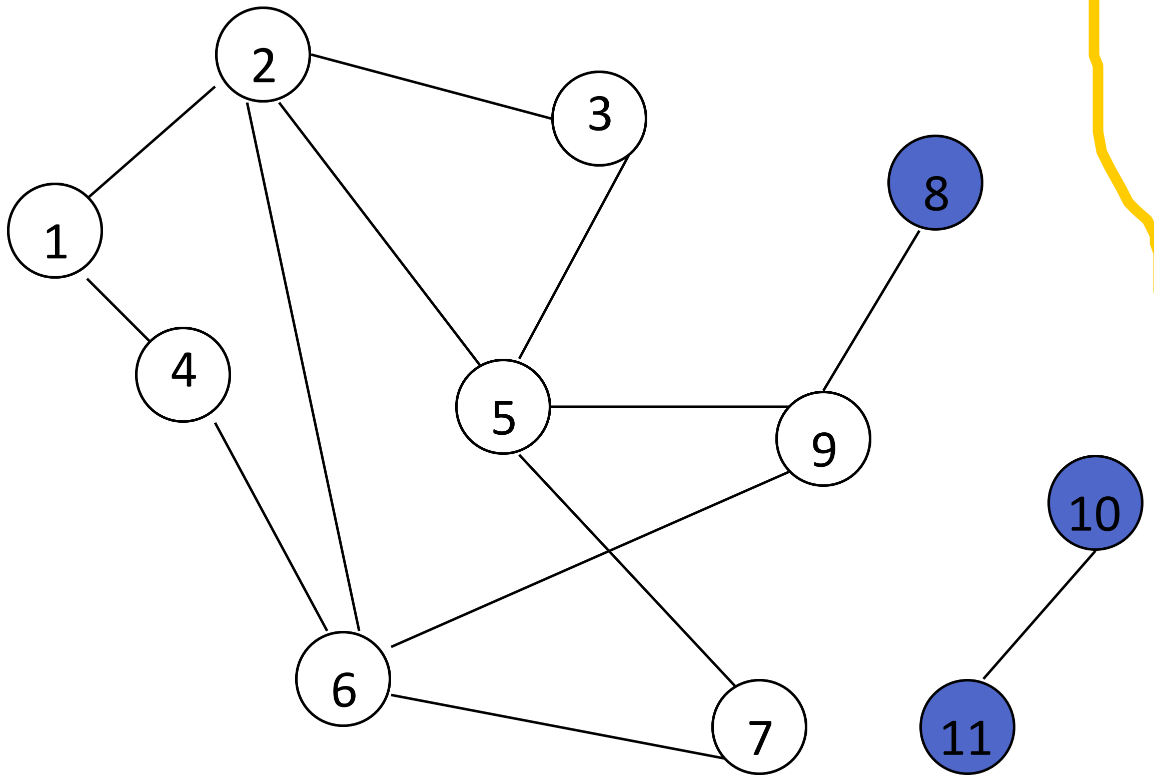
Breadth-First Search Example

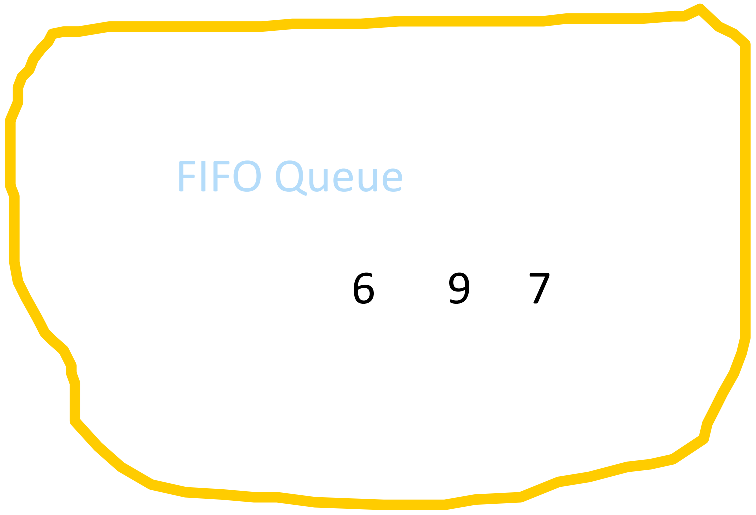




Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

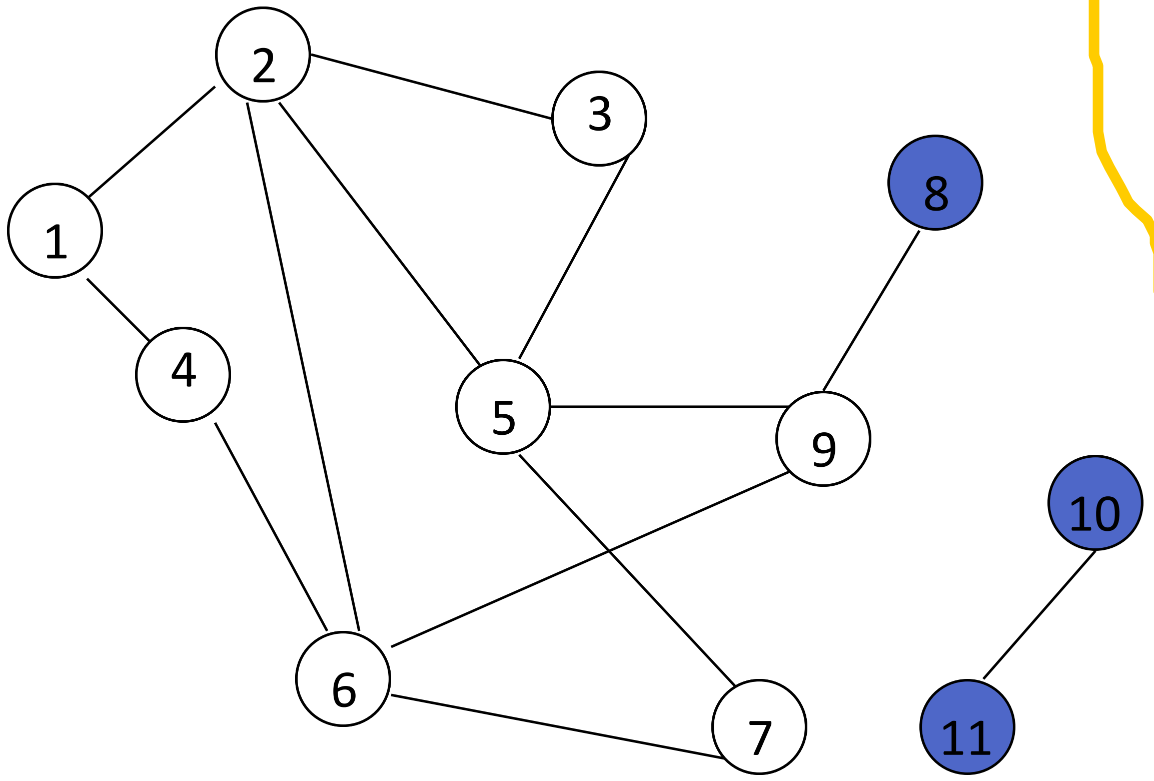
Breadth-First Search Example

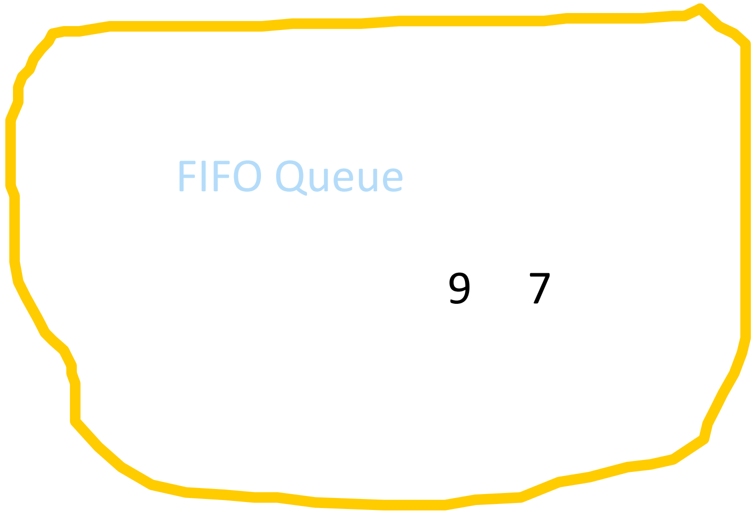




Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

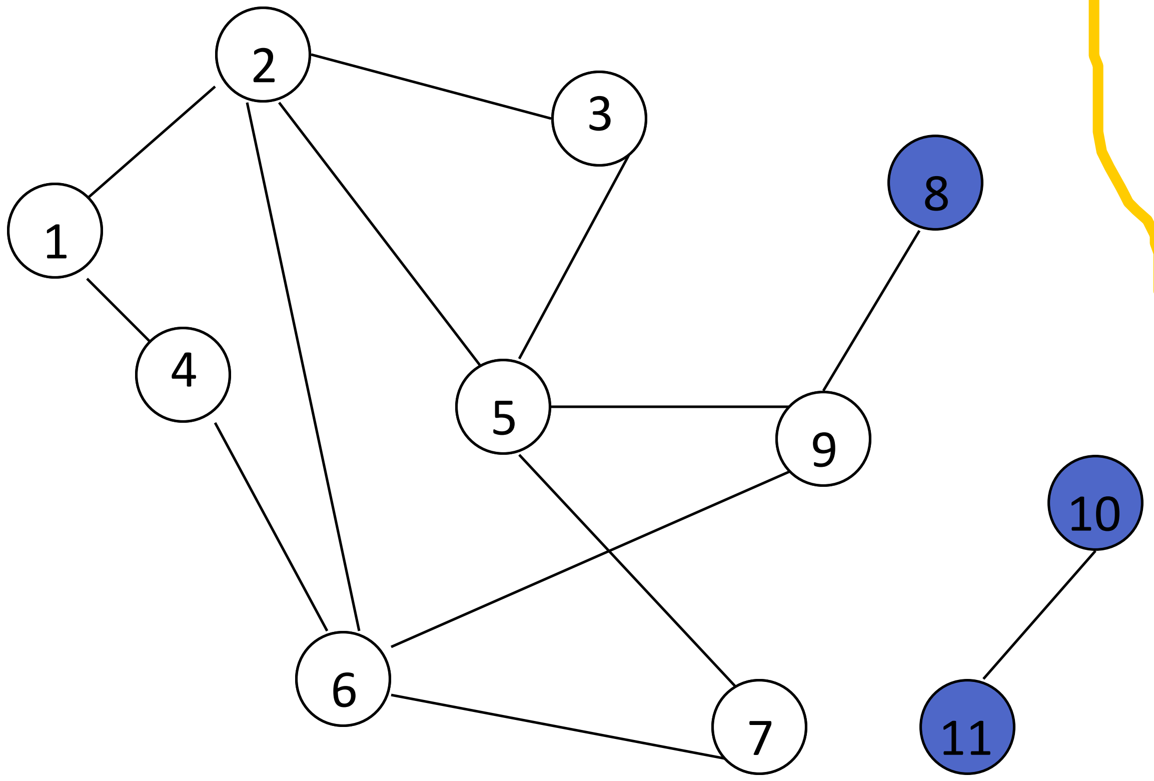
Breadth-First Search Example

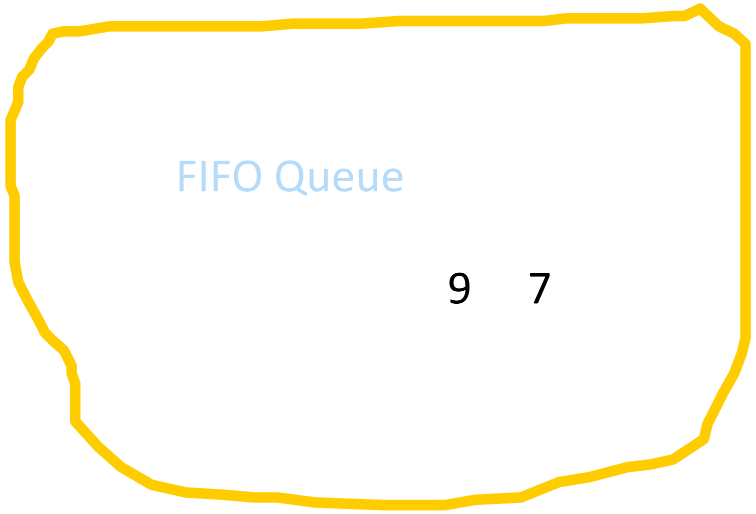




Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

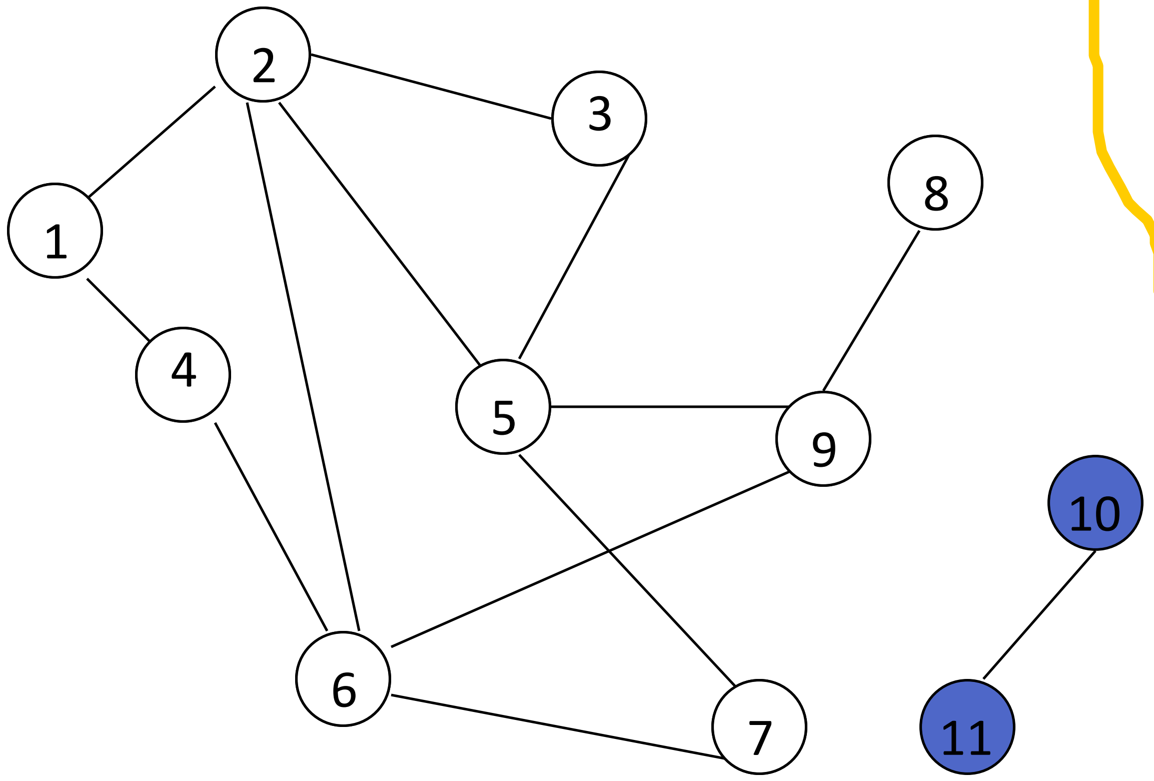
Breadth-First Search Example

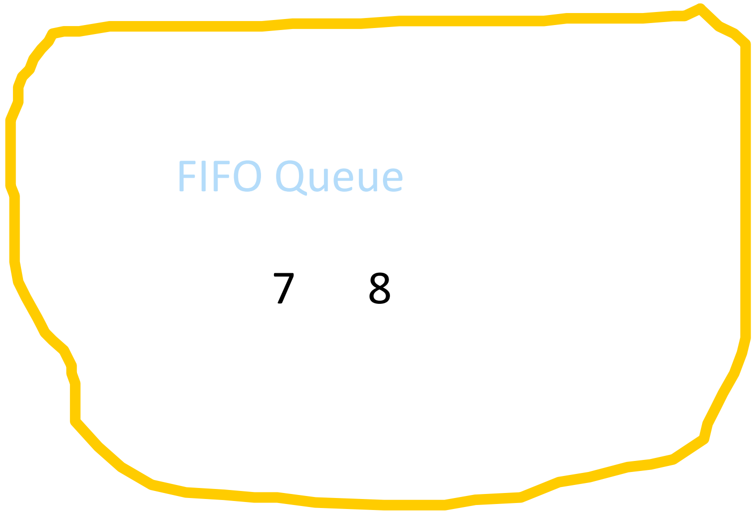




Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

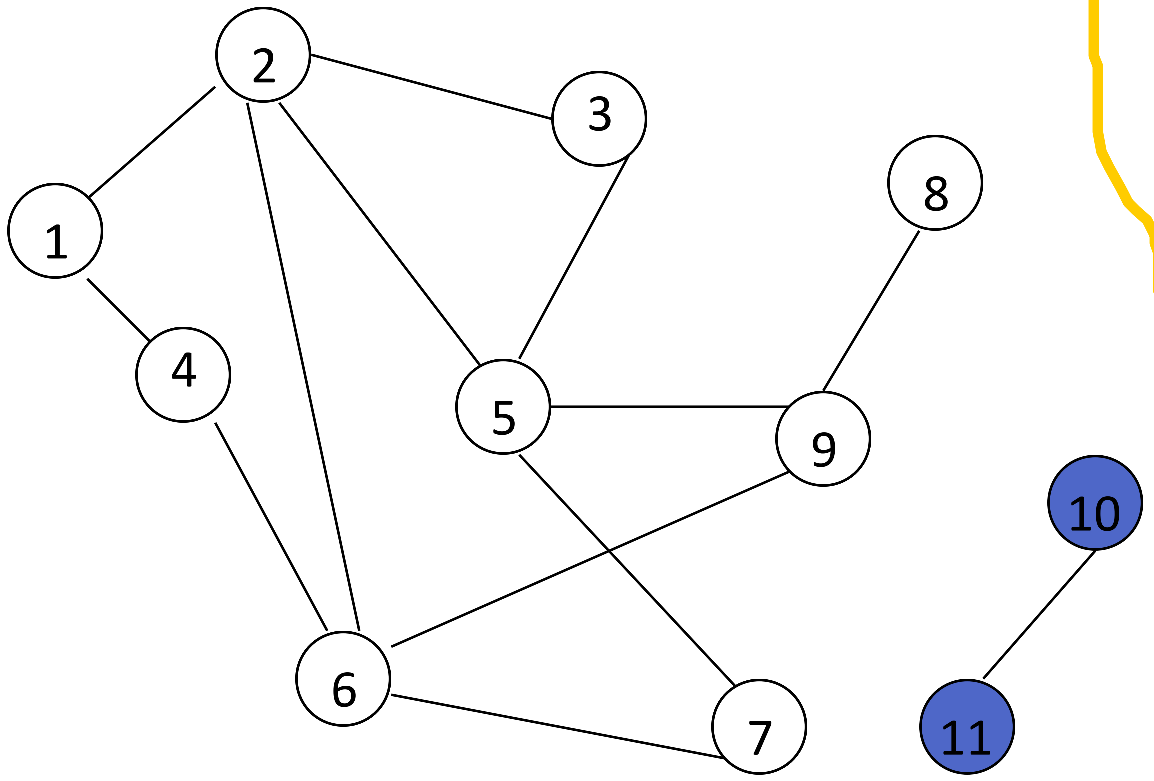
Breadth-First Search Example

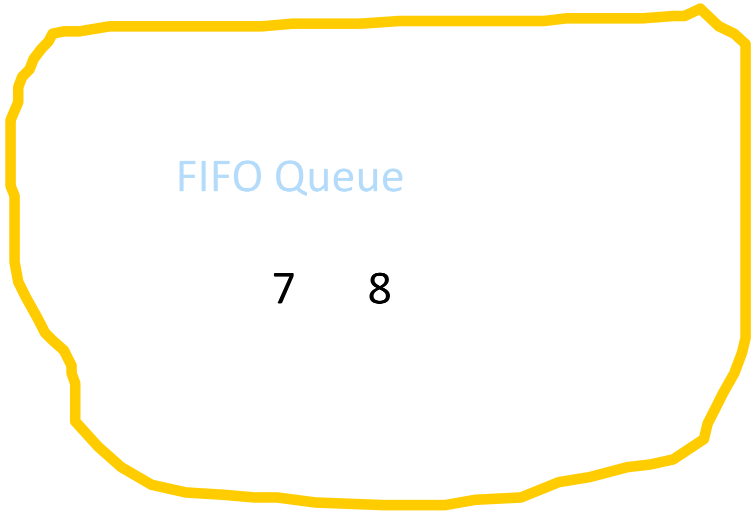




Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

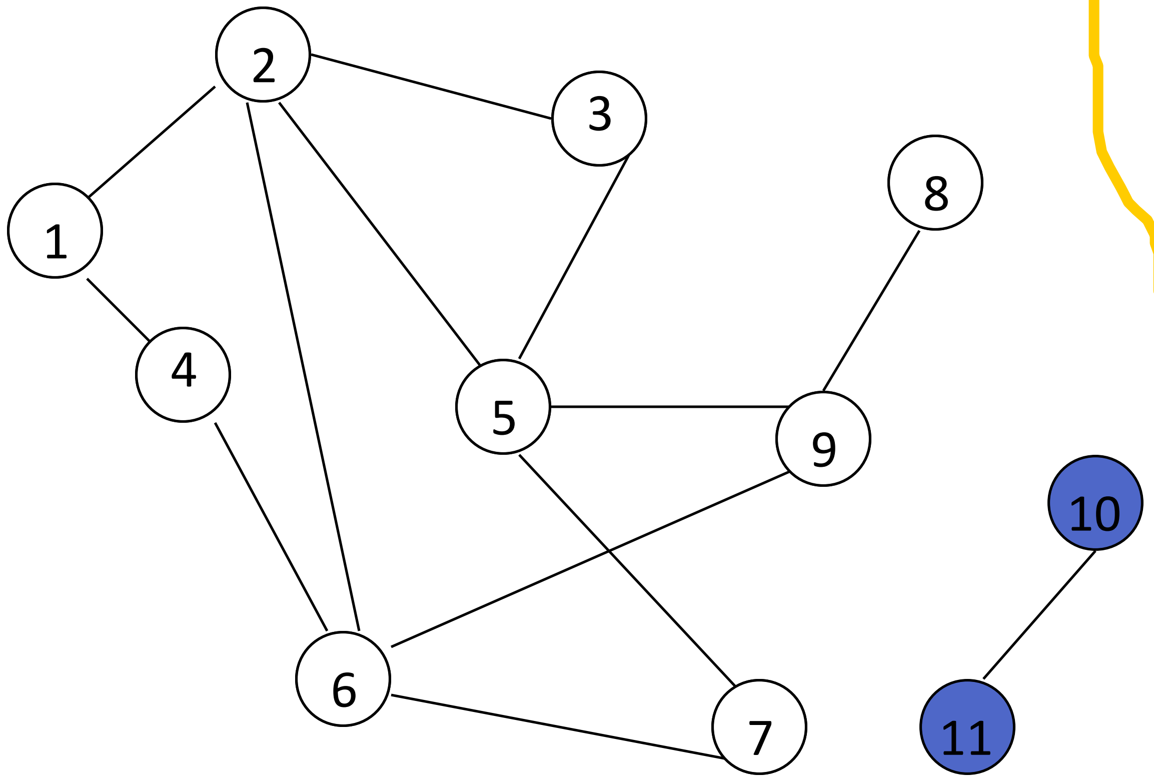
Breadth-First Search Example

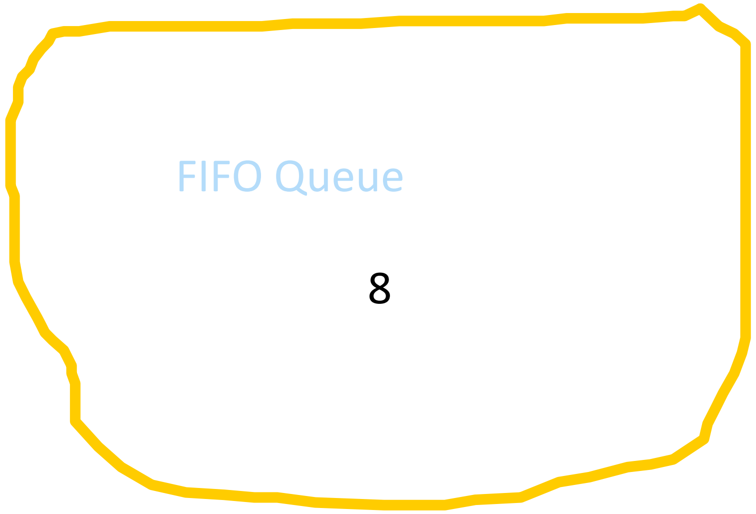




Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

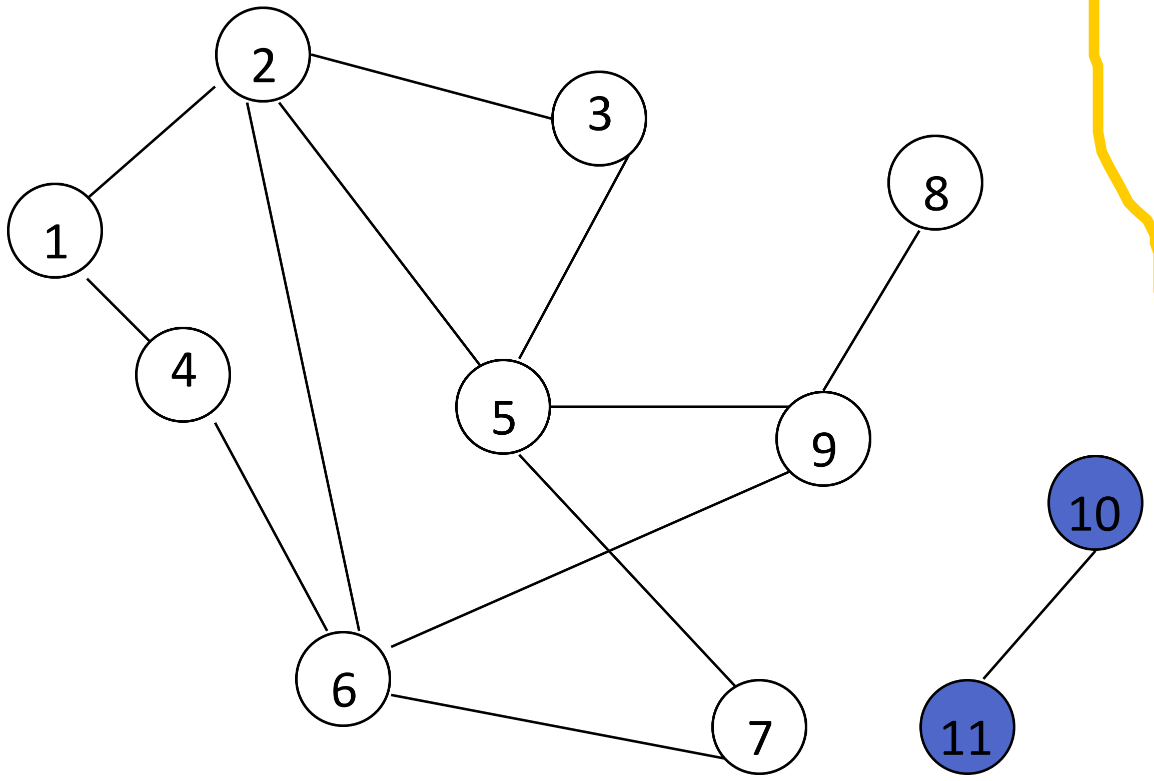
Breadth-First Search Example

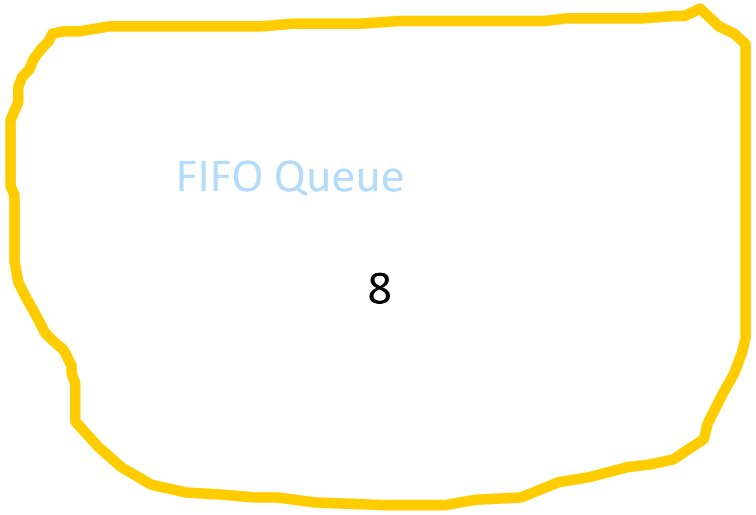




Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

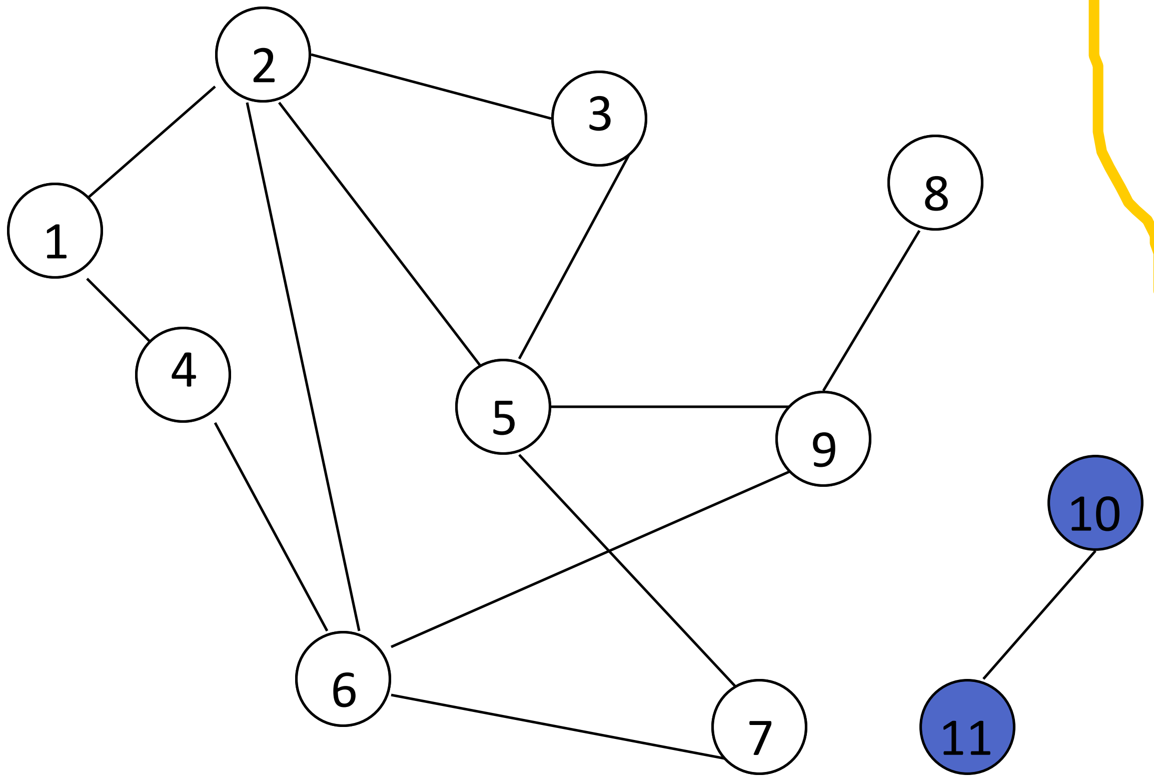
Breadth-First Search Example





Remove 8 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



Queue is empty. Search terminates.

Breadth-First Search Property

- All vertices reachable from the start vertex (including the start vertex) are visited.

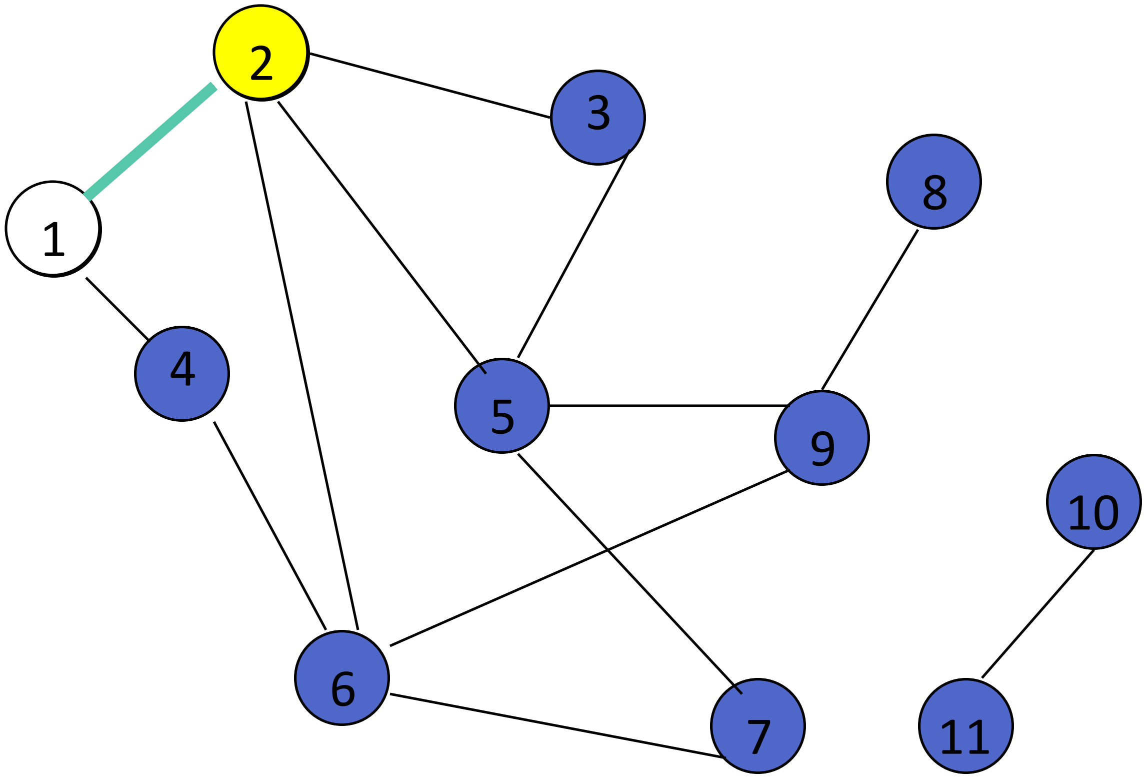
Time Complexity

- Each visited vertex is added to (and so removed from) the queue exactly once
- When a vertex is removed from the queue, we examine its adjacent vertices
  - O(v) if adjacency matrix is used, where v is number of vertices in whole graph
  - O(d) if adjacency list is used, where d is *edge degree*
- Total time
  - Adjacency matrix: O(w.v), where w is number of vertices in the *connected component* that is searched
  - Adjacency list: O(w+f), where f is number of edges in the *connected component* that is searched

Depth-First Search

```
depthFirstSearch(v) {
  Label vertex v as reached;
  for(each unreached vertex u
  adjacent to v)
      depthFirstSearch(u);
}
```

Depth-First Search

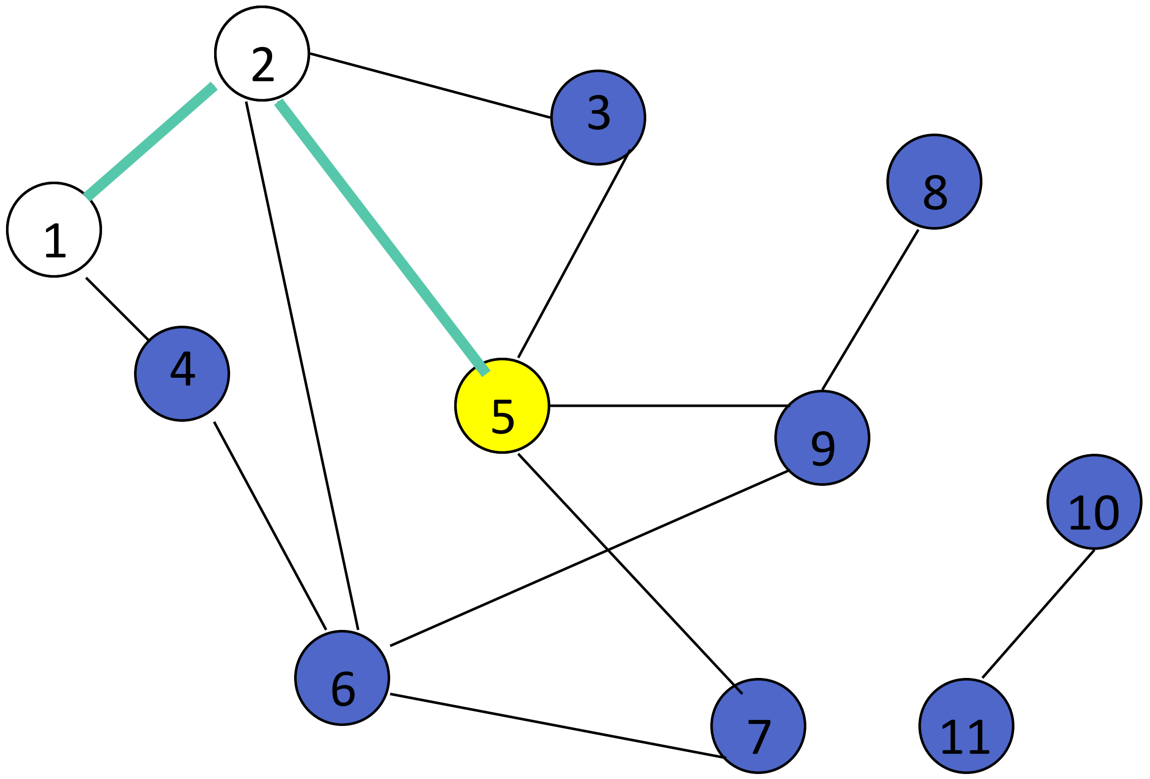


Start search at vertex 1.

Label vertex 1 and do a depth first search from either 2 or 4.

Suppose that vertex 2 is selected.

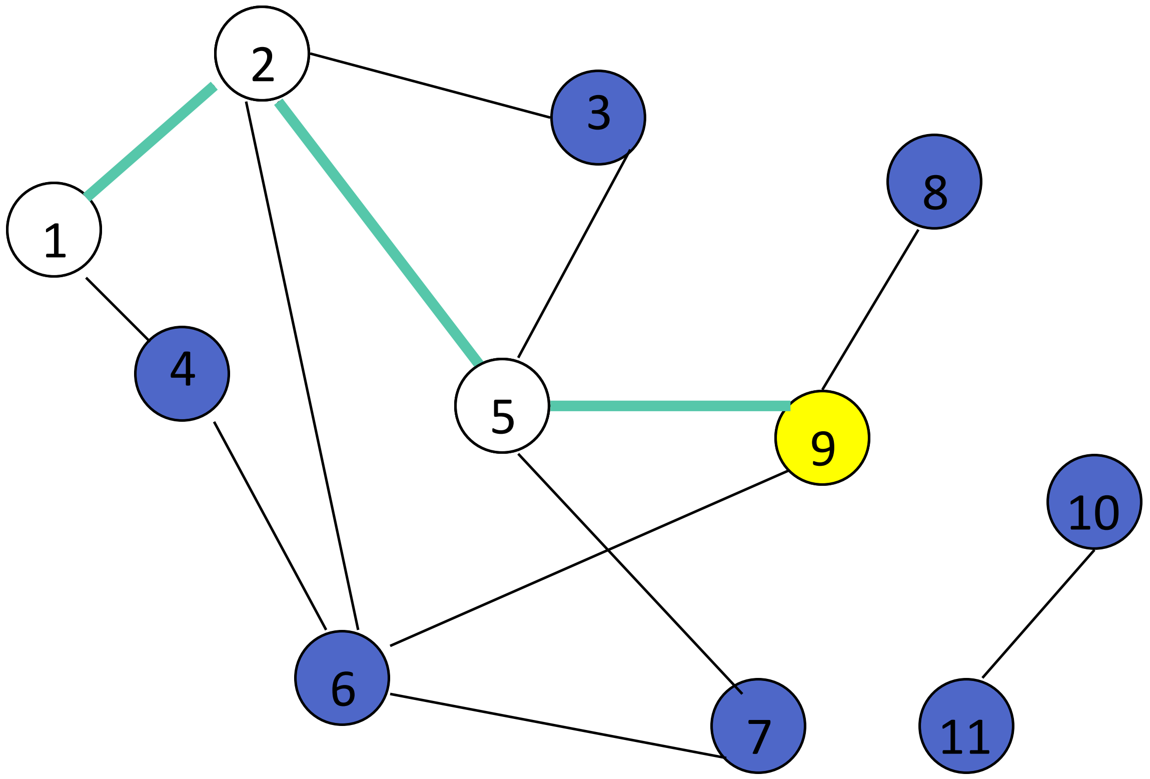
Depth-First Search Example



Label vertex 2 and do a depth first search from either 3, 5, or 6.

Suppose that vertex 5 is selected.

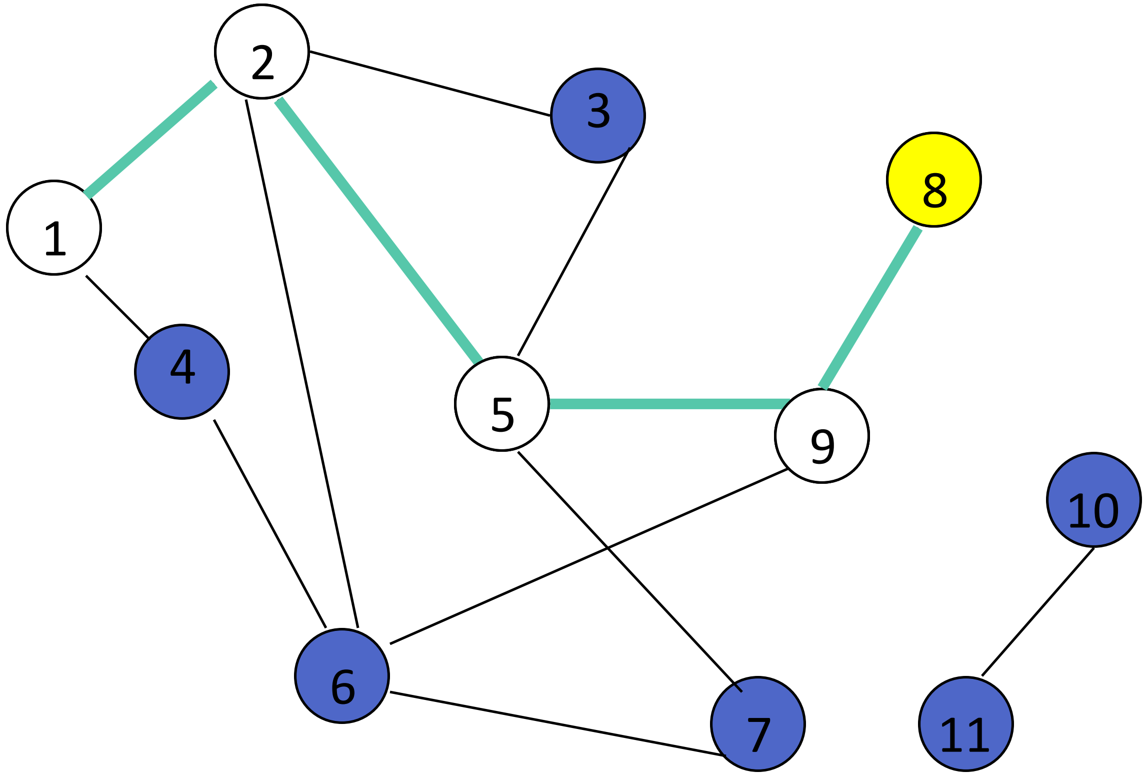
Depth-First Search



Label vertex 5 and do a depth first search from either 3, 7, or 9.

Suppose that vertex 9 is selected.

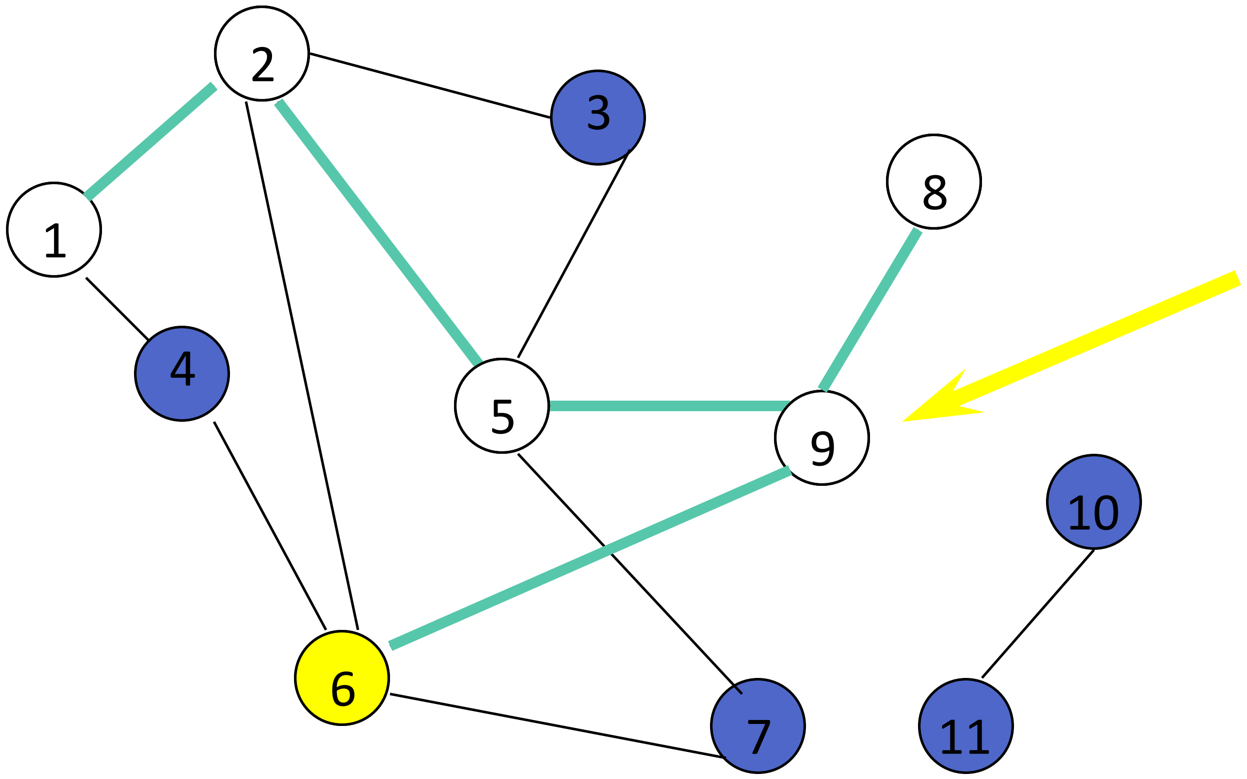
Depth-First Search



Label vertex 9 and do a depth first search from either 6 or 8.

Suppose that vertex 8 is selected.

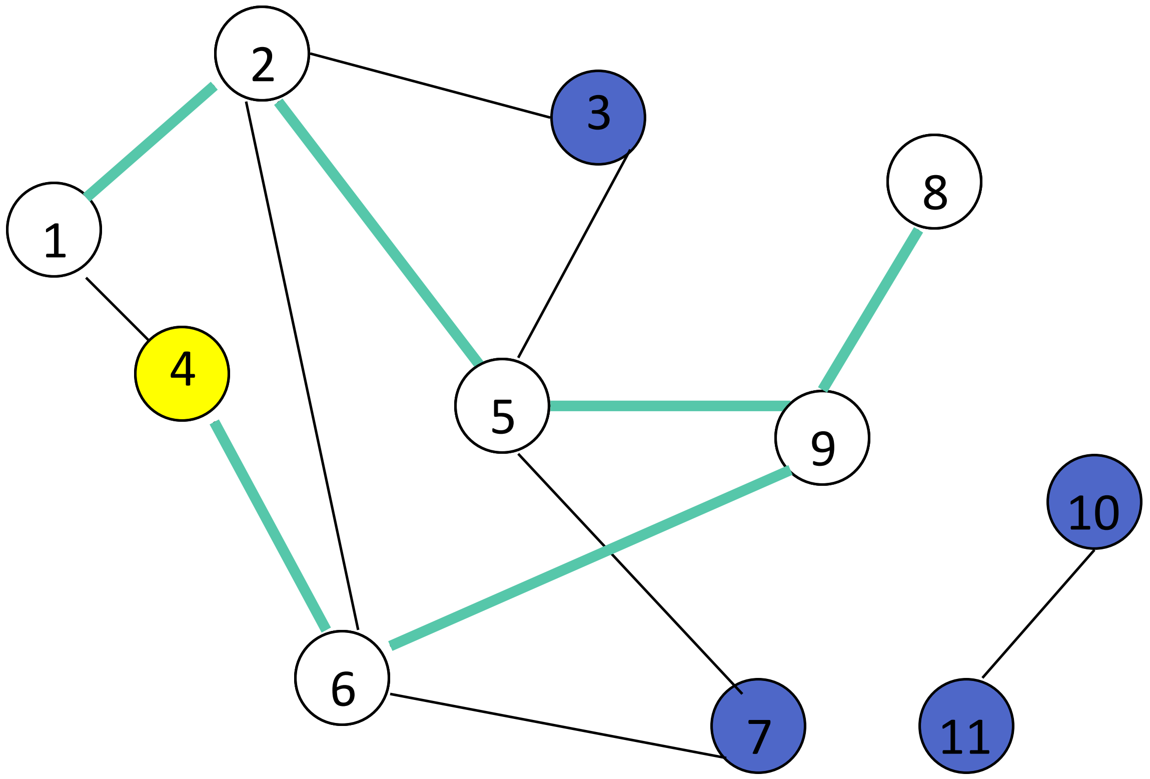
Depth-First Search



Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6)

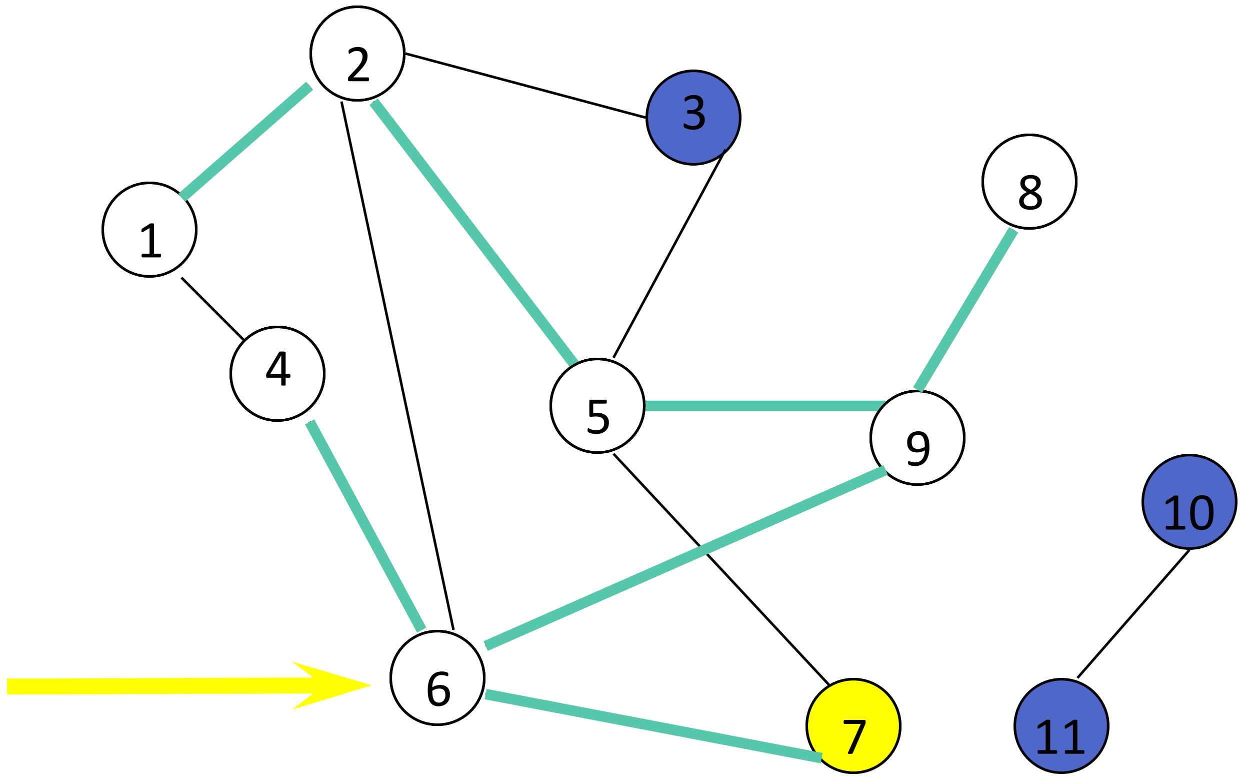
Depth-First Search



Label vertex 6 and do a depth first search from either 4 or 7.

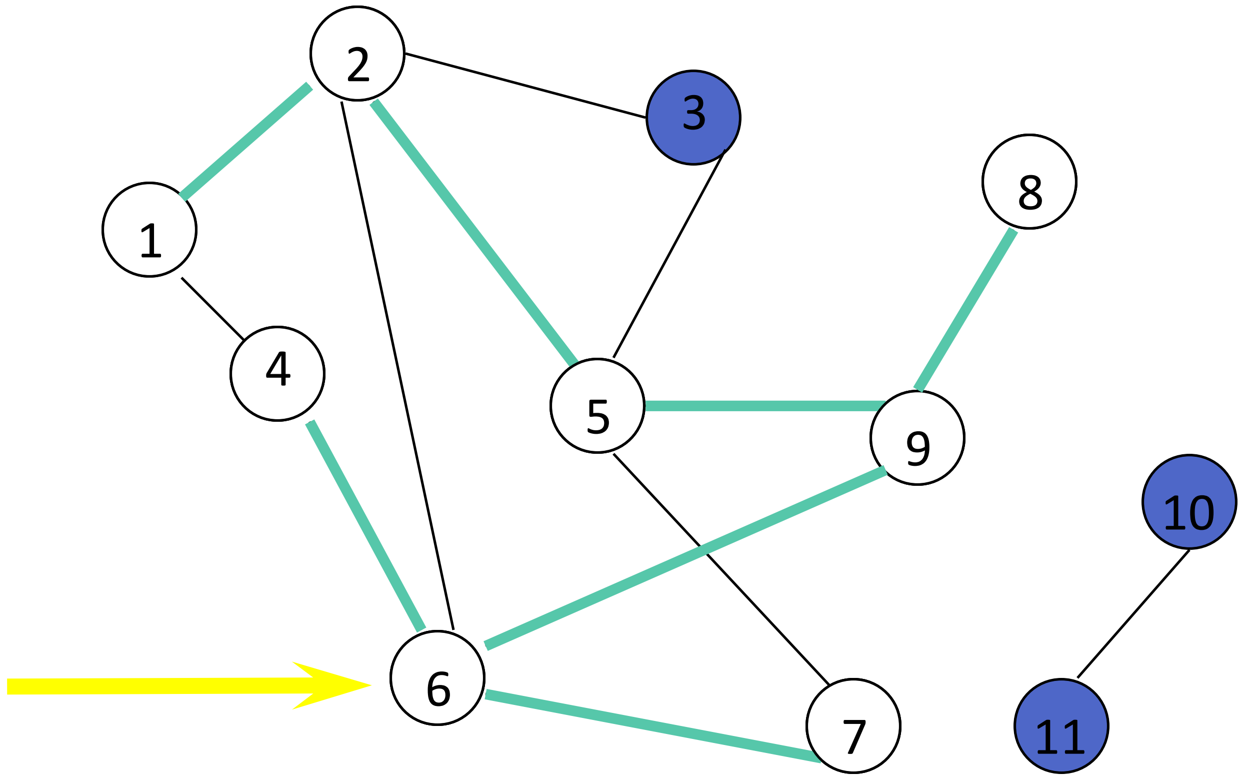
Suppose that vertex 4 is selected.

Depth-First Search



Label vertex 4 and return to 6.
From vertex 6 do a dfs(7).

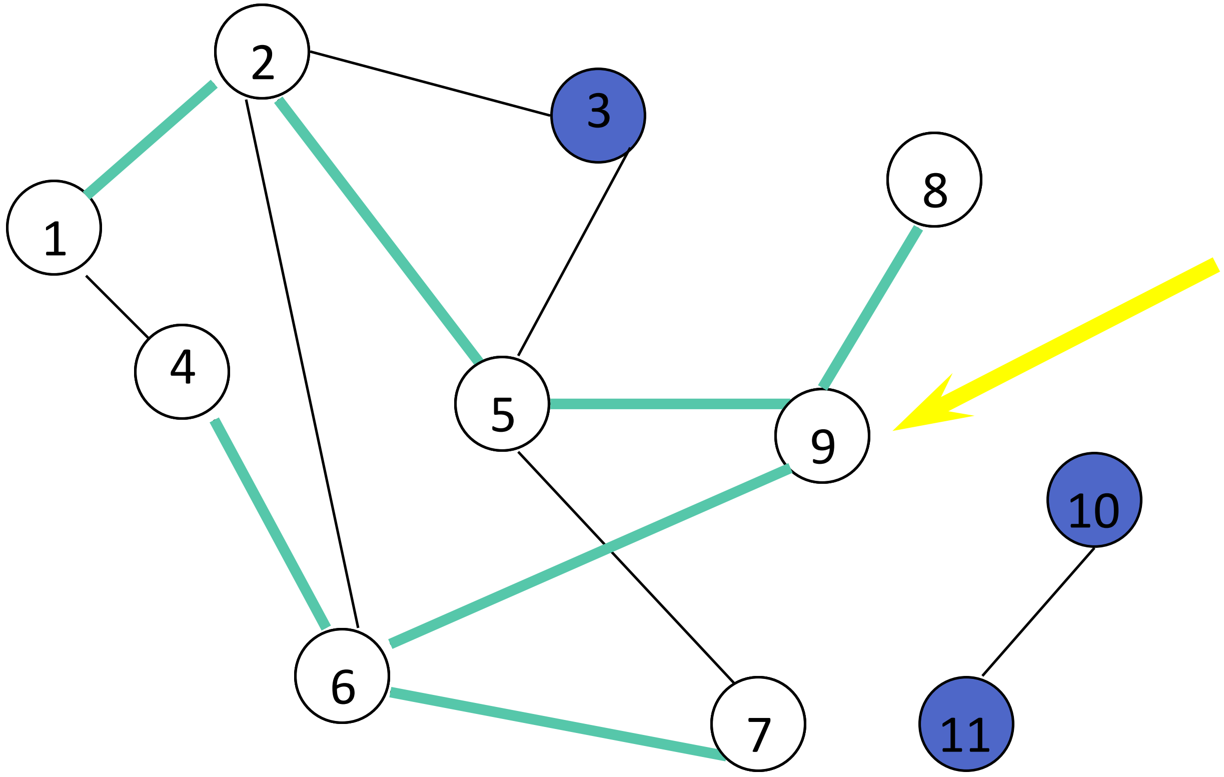
Depth-First Search



Label vertex 7 and return to 6.

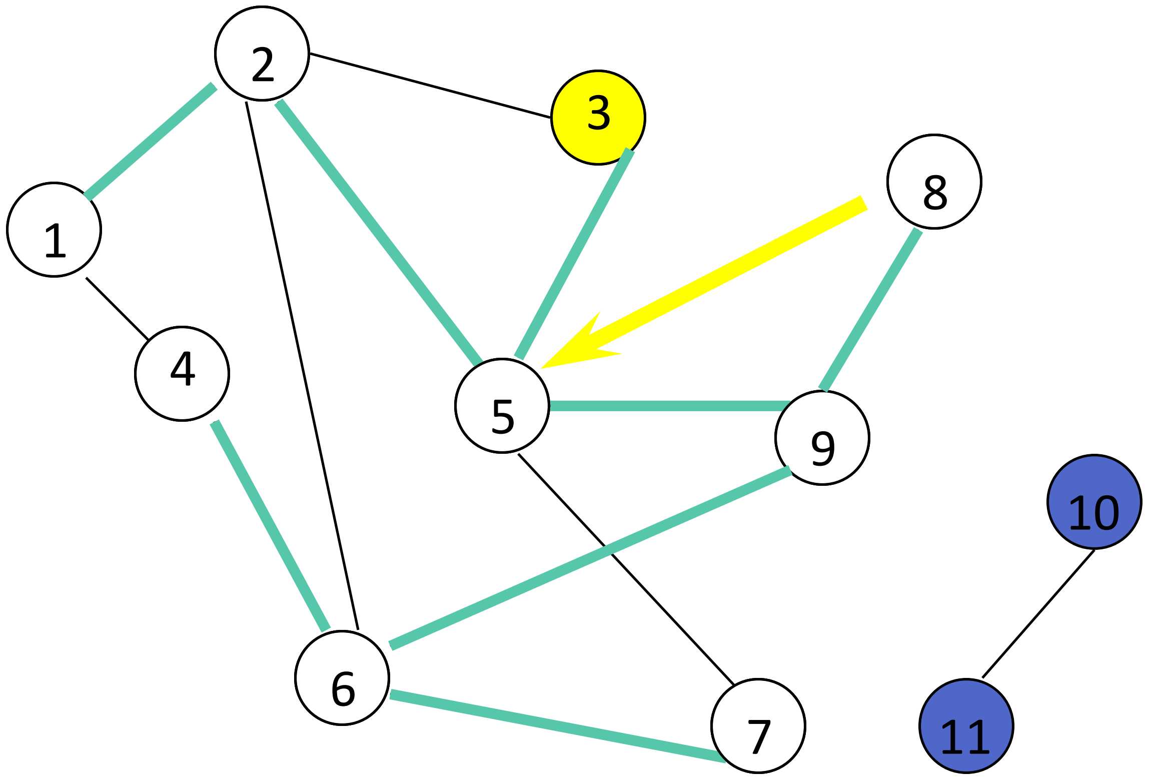
Return to 9.

Depth-First Search



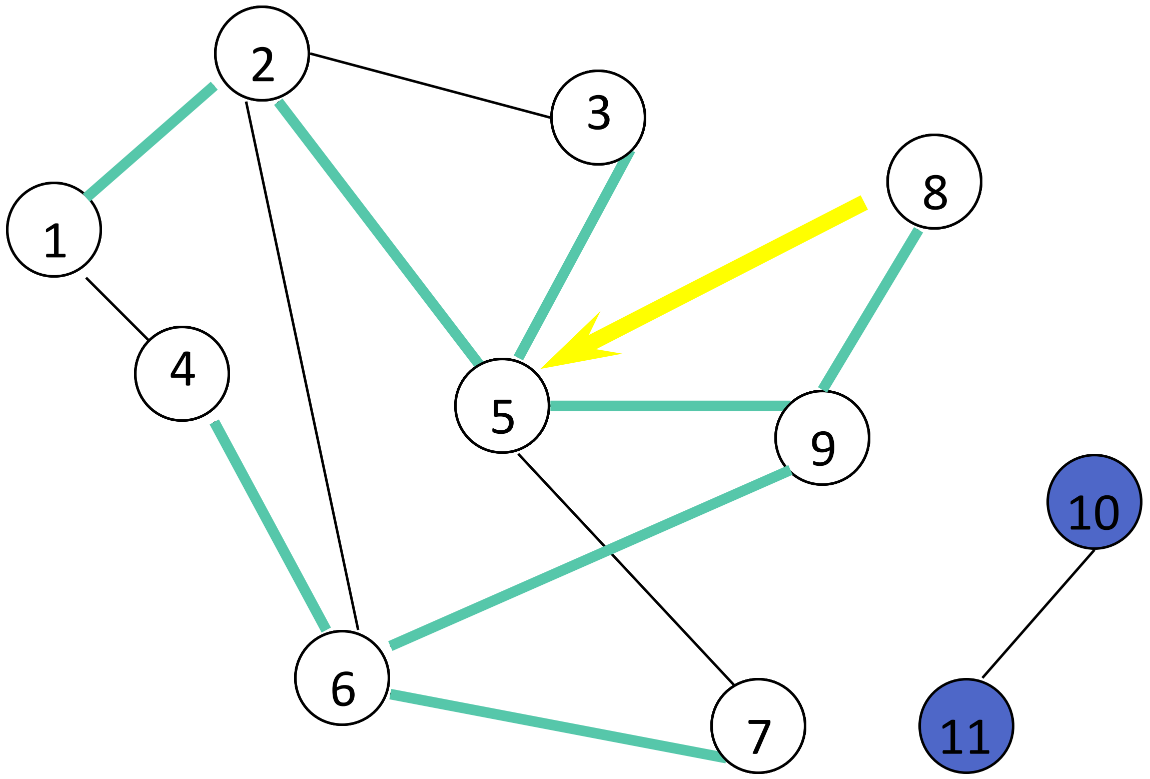
Return to 5.

Depth-First Search



Do a dfs(3).

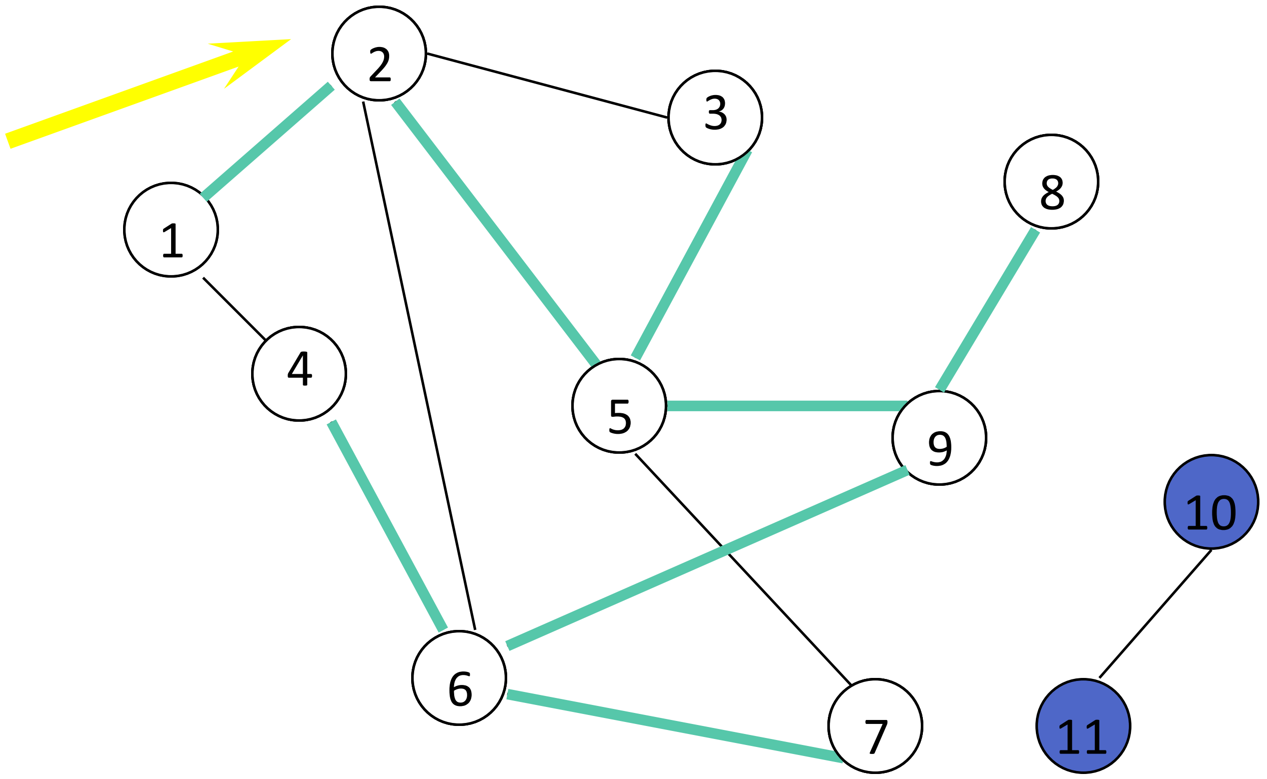
Depth-First Search



Label 3 and return to 5.

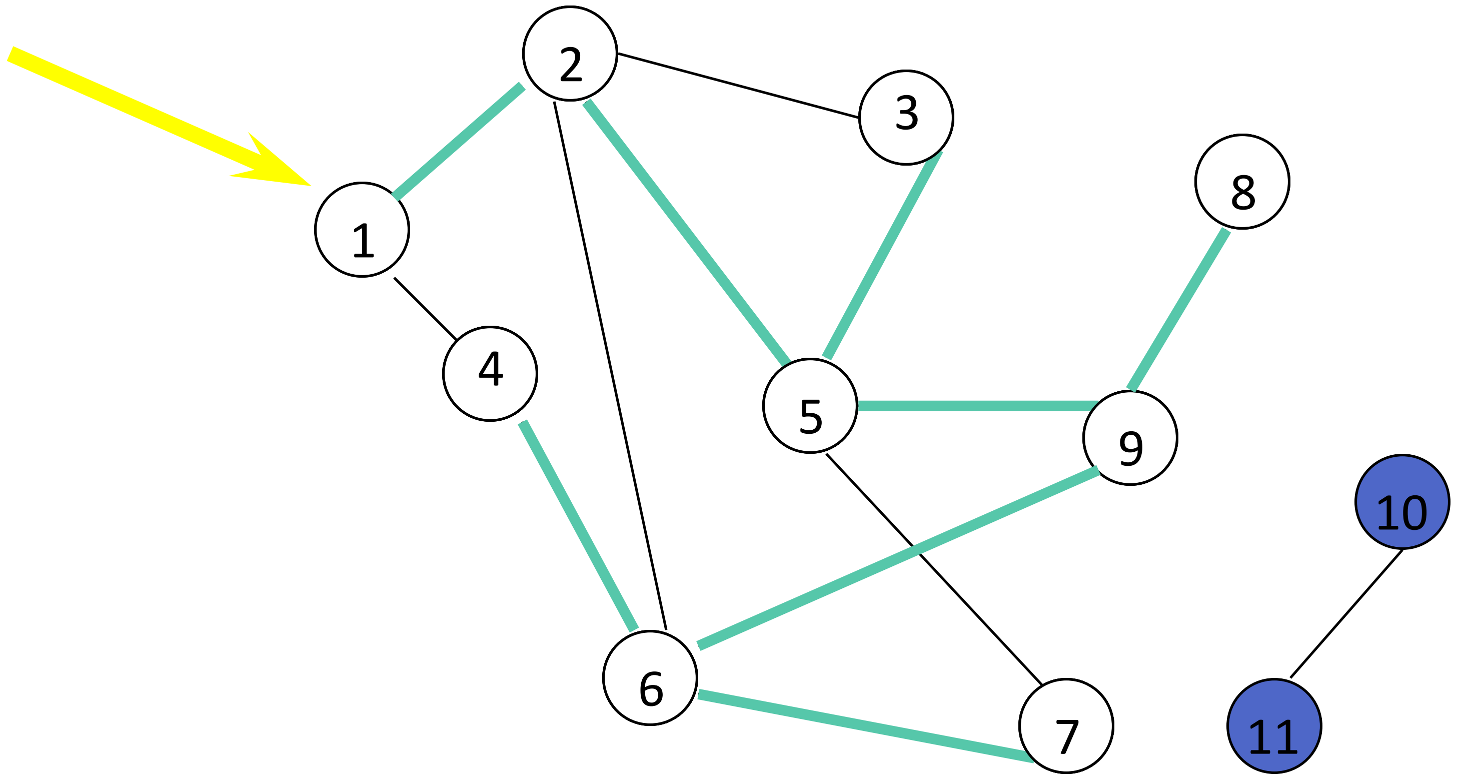
Return to 2.

Depth-First Search



Return to 1.

Depth-First Search



Return to invoking method.

DFS Properties

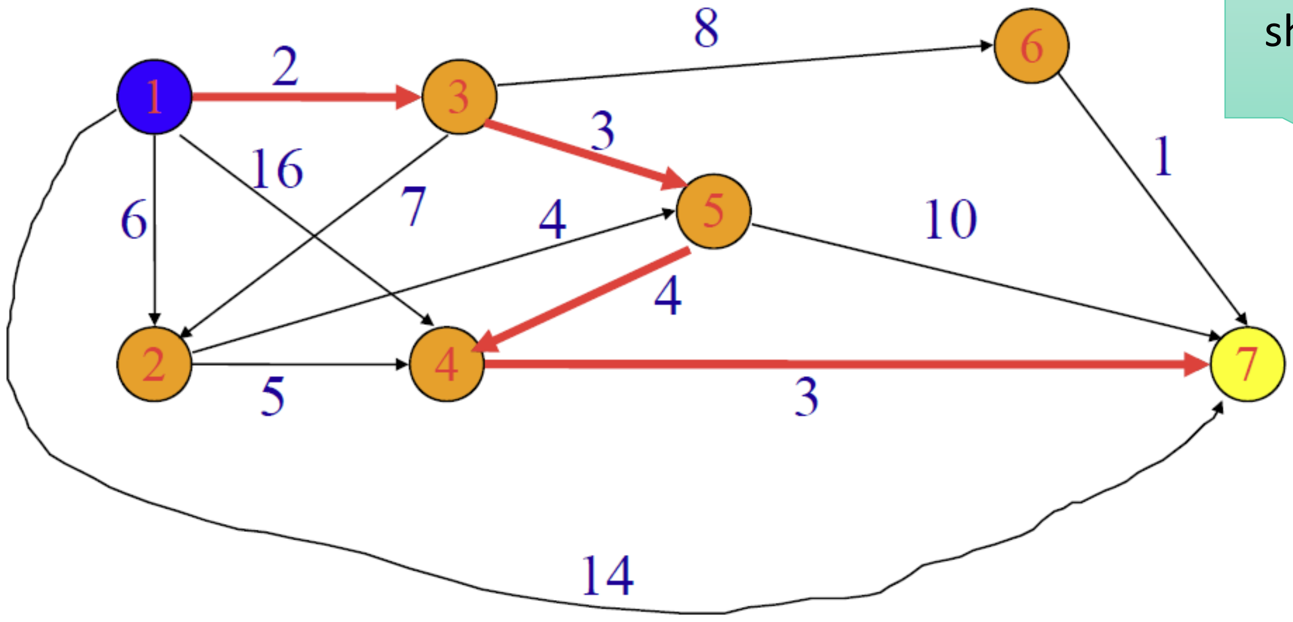
- DFS has same time complexity as BFS
- DFS requires O(h) memory for recursive function stack calls while BFS requires O(w) queue capacity
- Same properties with respect to path finding, connected components, and spanning trees.
  - Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
- One is better than the other for some problems, e.g.
  - When searching, if the item is far from source (leaves), then DFS may locate it first, and vice versa for BFS
  - BFS traverses vertices at same distance (level) from source
  - DFS can be used to detect cycles (revisits of vertices in current stack)

Shortest Path: Single source, single destination

- Possible greedy algorithm
  - Leave source vertex using *shortest edge*
  - Leave new vertex using cheapest edge, to reach an *unvisited vertex*
  - Continue until destination is reached

DFS with shortest edge selected!

Greedy Path from 1 To 7



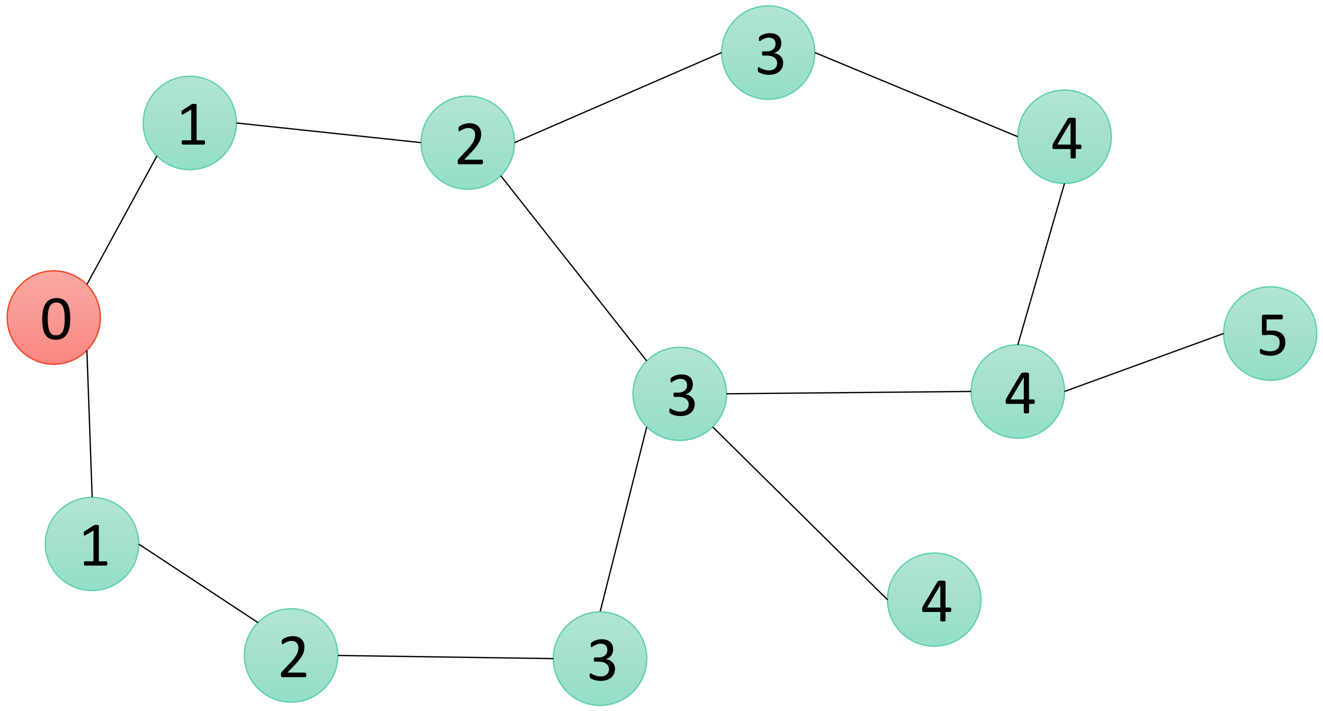
Length of 12 is not shortest path!

Single Source Shortest Path

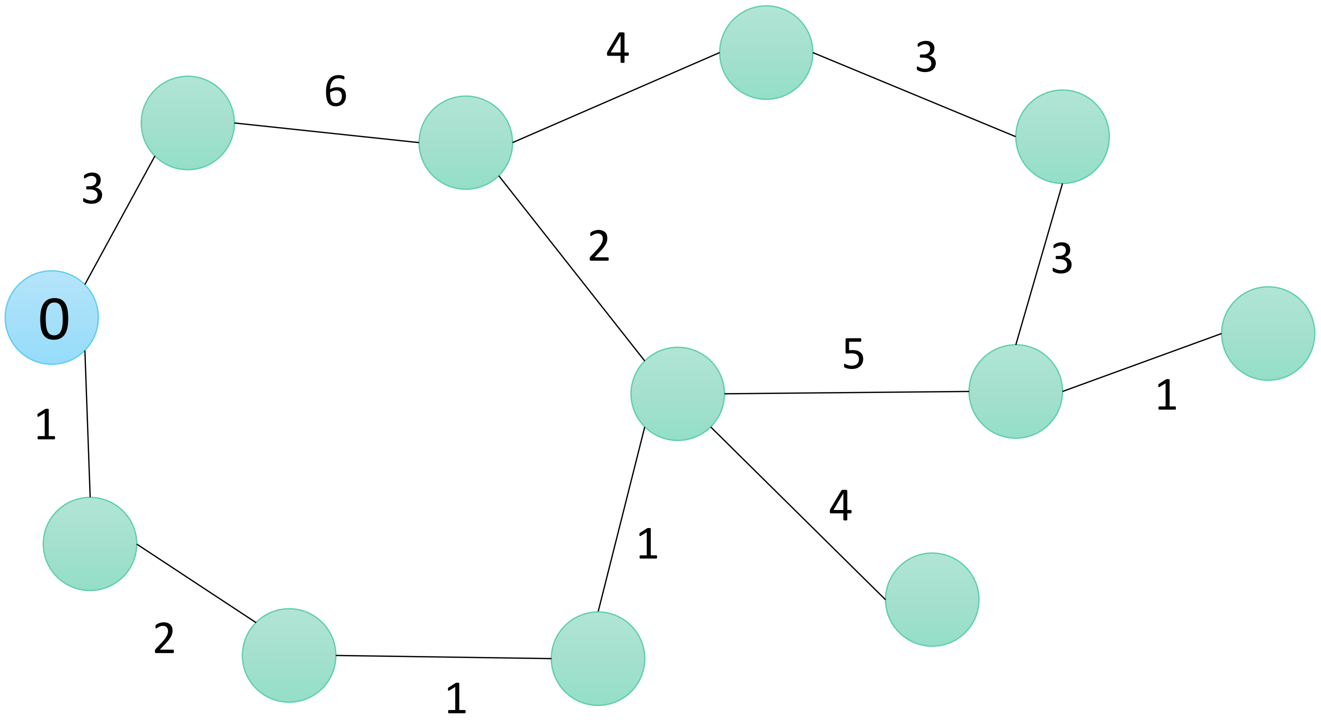
- Shortest distance from one source vertex to all destination vertices
- Is there a simple way to solve this?
- …Say if you had an unit-weighted graph?

- Just do Breadth First Search (BFS)! ☺

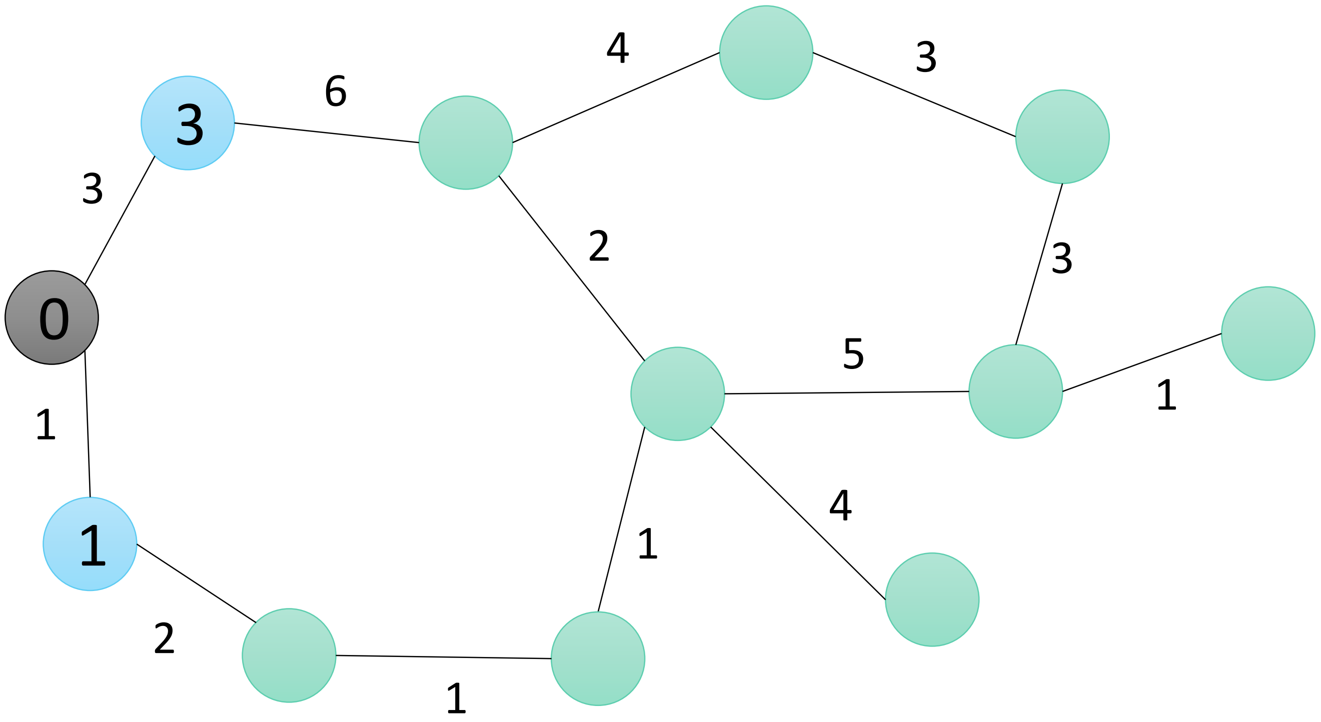
SSSP: BFS on Unweighted Graphs



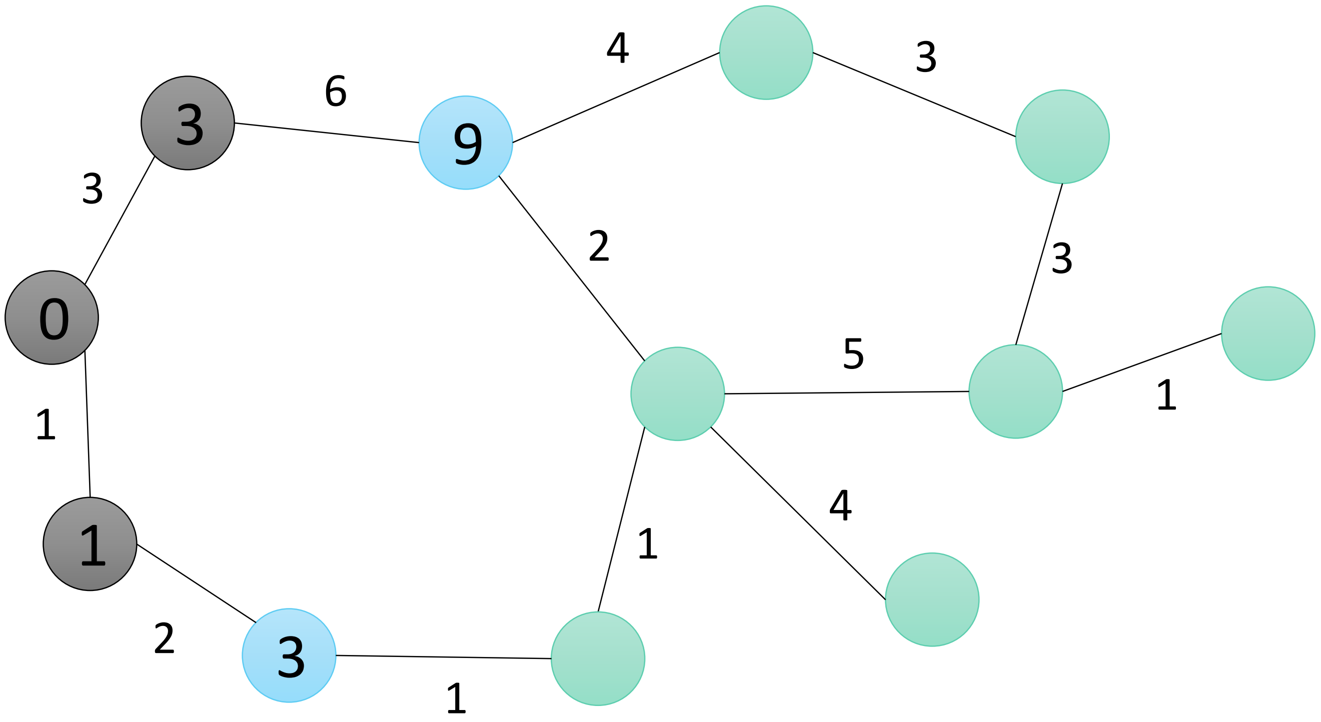
SSSP: *BFS on Weighted Graphs?*



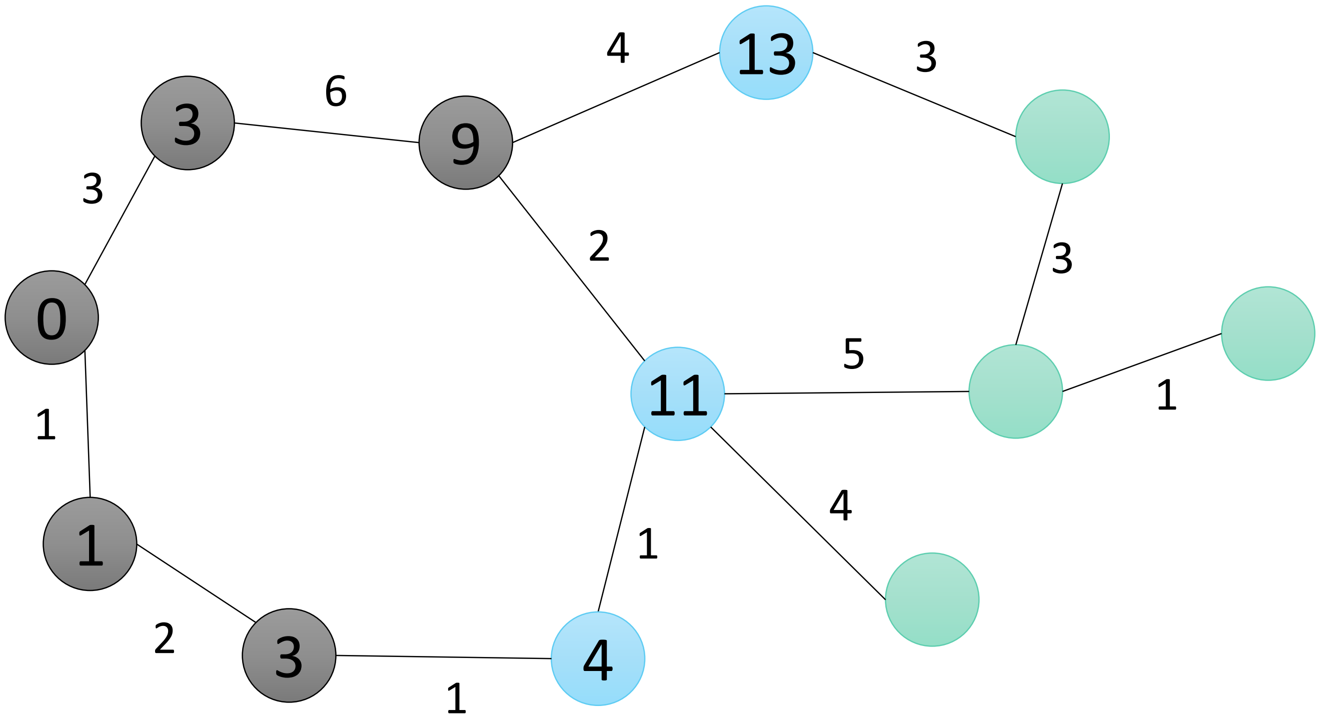
SSSP: BFS on Weighted Graphs?



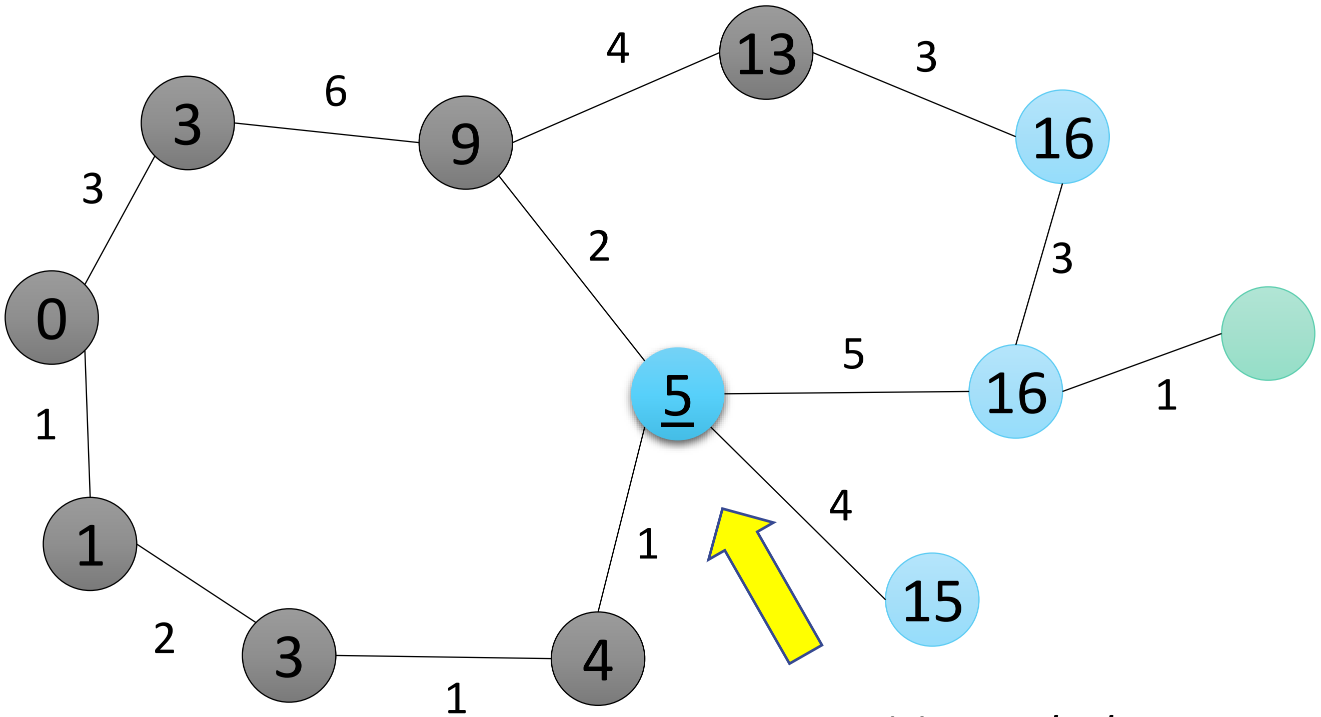
SSSP: BFS on Weighted Graphs?



SSSP: BFS on Weighted Graphs?

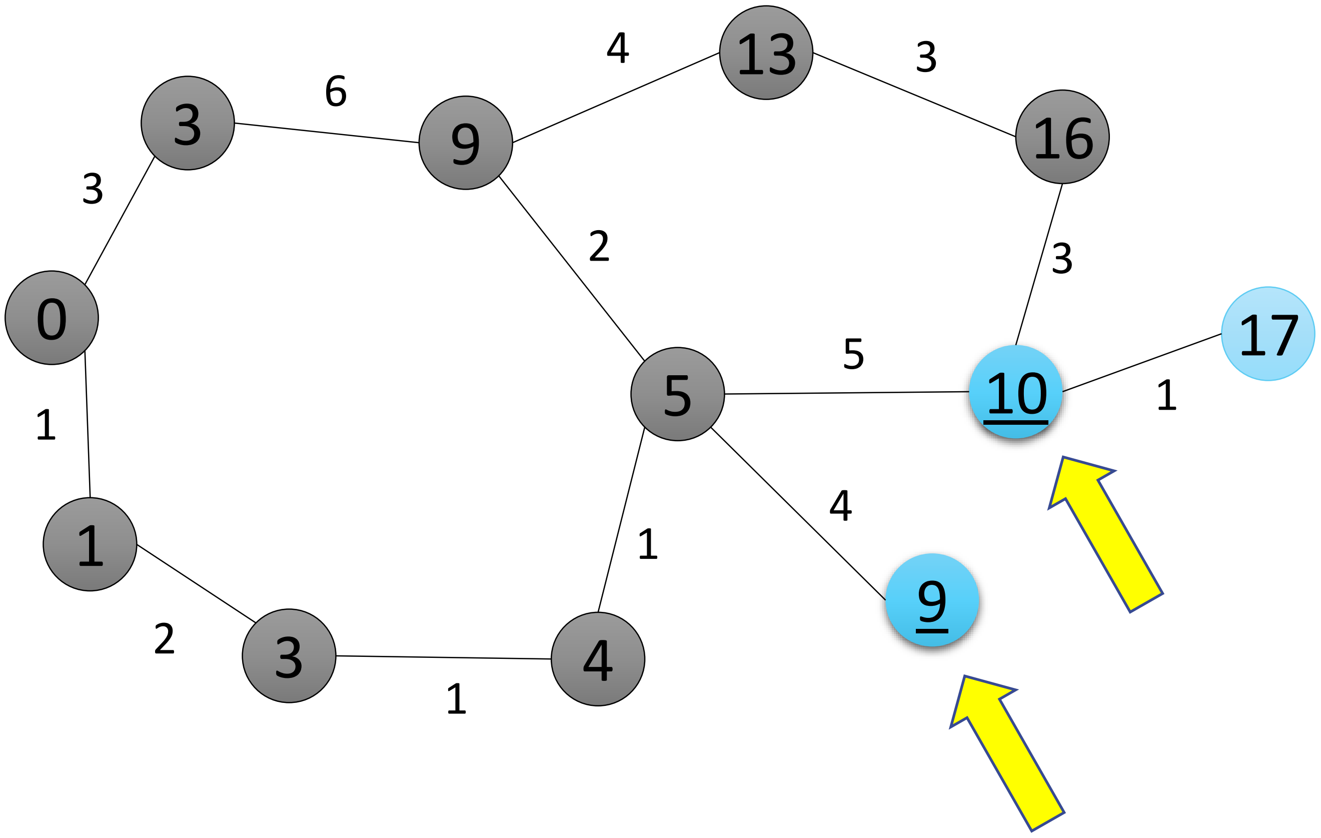


SSSP: BFS on Weighted Graphs?

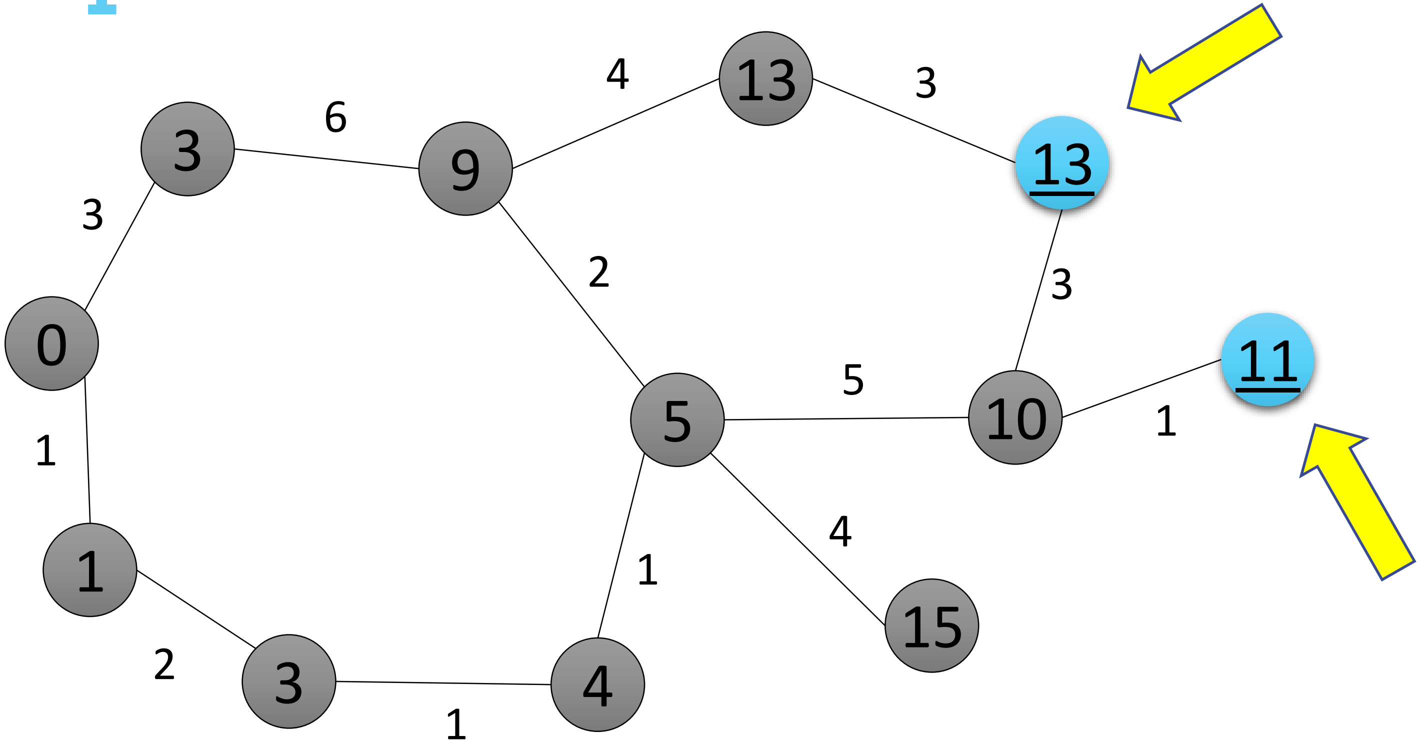


Revisit, recalculate, re-propagate…
cascading effect

SSSP: BFS on Weighted Graphs?



SSSP: BFS on Weighted Graphs?



BFS with revisits is not efficient. Can we be smart about order of visits?

Dijkstra's Single Source Shortest Path (SSSP)

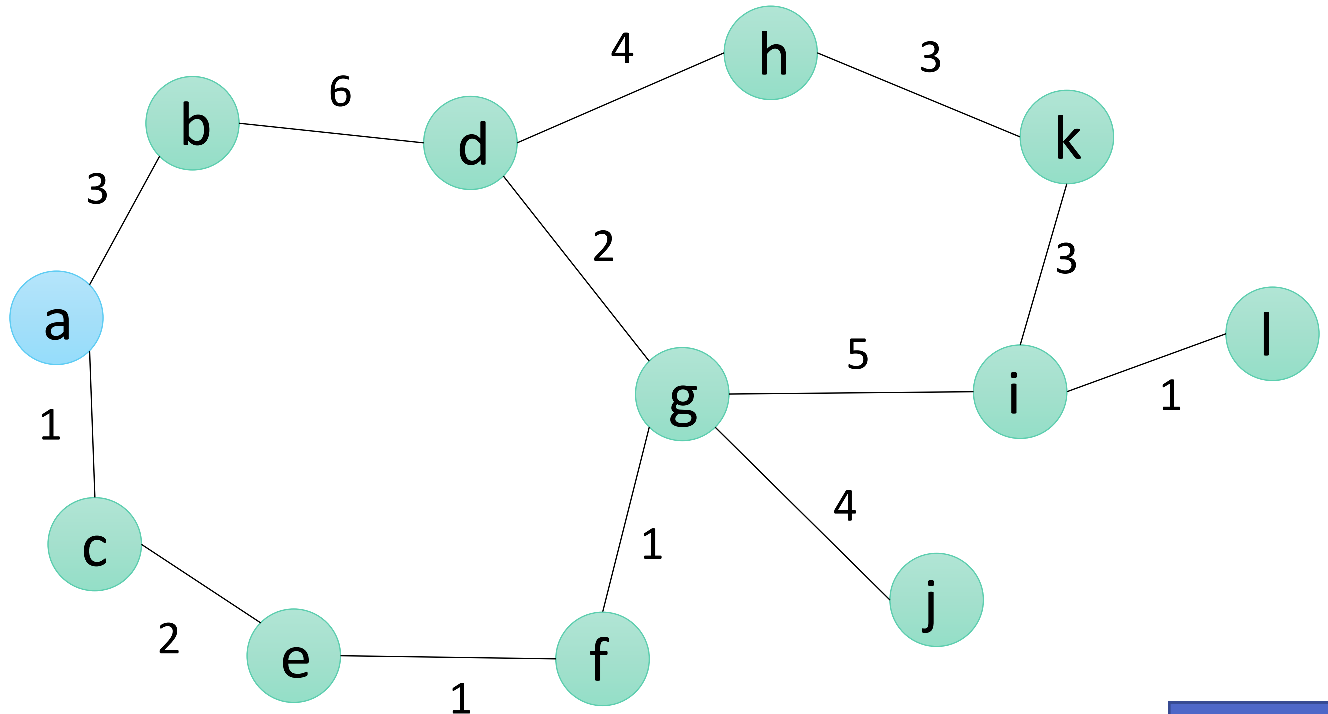
- Prioritize the vertices to visit next
  - Pick "unvisited" vertex with *shortest distance* from source
- Do not visit vertices that have already been visited
  - Avoids false propagation of distances

Dijkstra's Single Source Shortest Path (SSSP)

- Let w[u,v] be array with weight of edge from u to v
- Initialize distance vector d[ ] for all vertices to infinity, except for source which is set to 0
- Add all vertices to queue Q
- while(Q is not empty)
  - Remove **u** from **Q** such that **d[u] is the smallest in Q**
  - Add u to visited set
  - for each **v** adjacent to **u** that is not visited
    - d' = d[u] + w[u,v]
    - if(d' < d[v]) set d[v] = d' & add v to Q

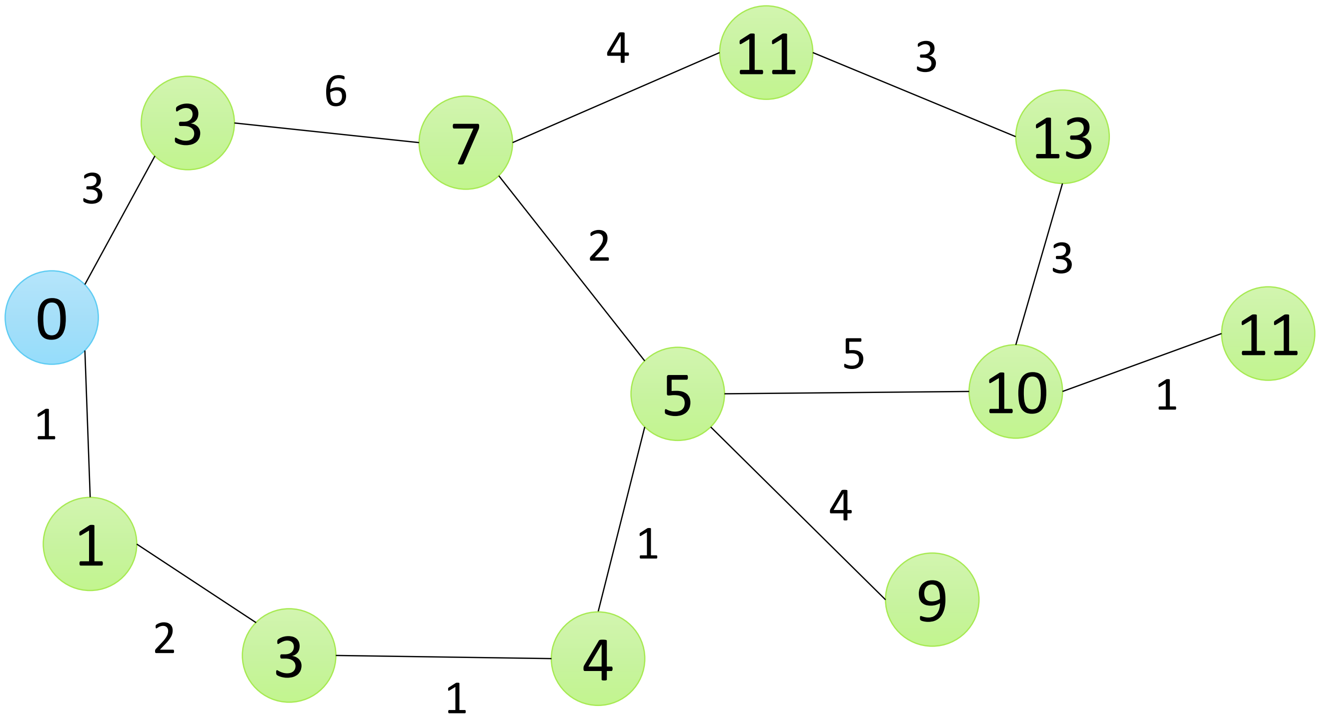
Only change relative to BFS!

SSSP on Weighted Graphs



Work out!

SSSP on Weighted Graphs

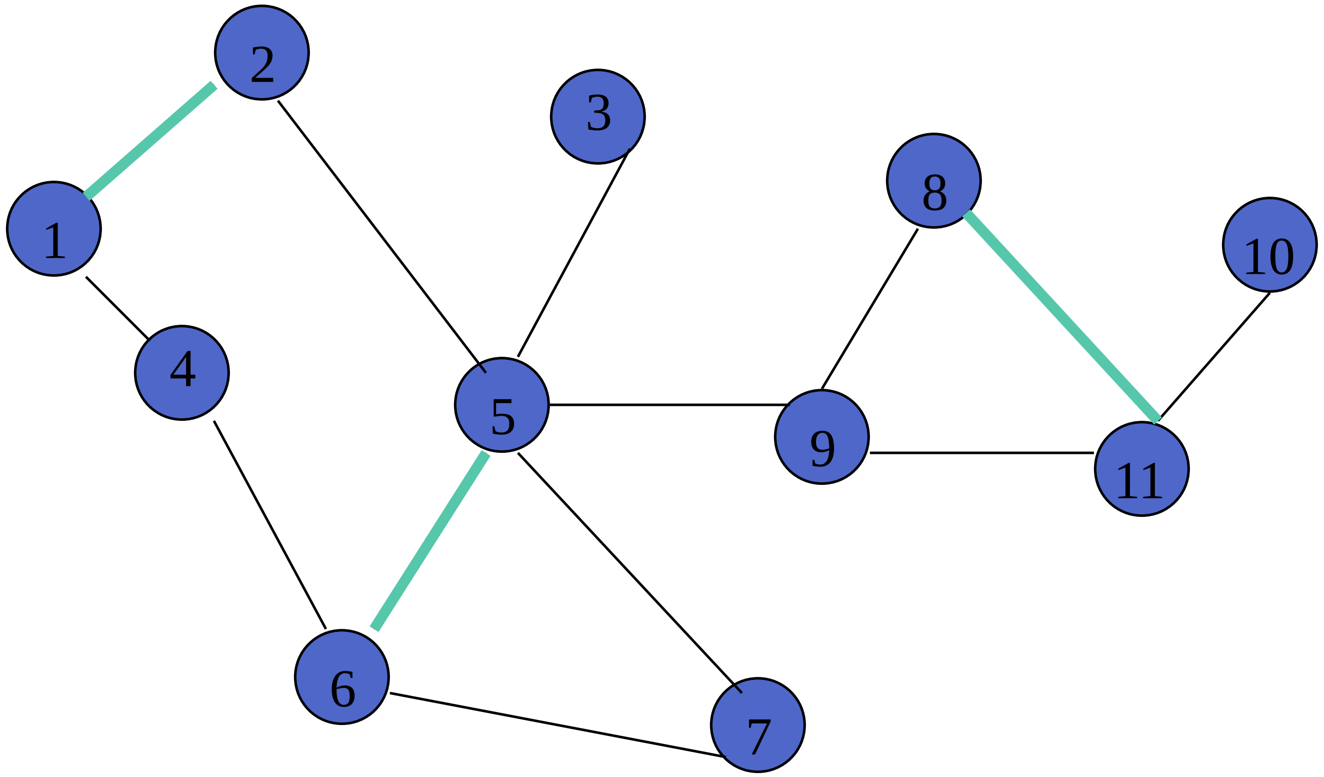


Complexity

- Using a **linked list** for queue, it takes **$O(v^2 + e)$**
- For each vertex,
  - we linearly search the linked list for smallest: O(v)
  - we check and update for each incident edge once: O(d)
- When a **min heap** (priority queue) with distance as priority key, total time is **O(e + v log v)**
  - O(log v) to insert or remove from priority queue
  - O(v) *remove min* operations
  - O(e) *change d[ ] value* operations (insert/update)
- When e is $O(v^2)$ *[highly connected, small diameter]*, using a min heap is worse than using a linear list
- When a **Fibonacci heap** is used, the total time is O(e + v log v)

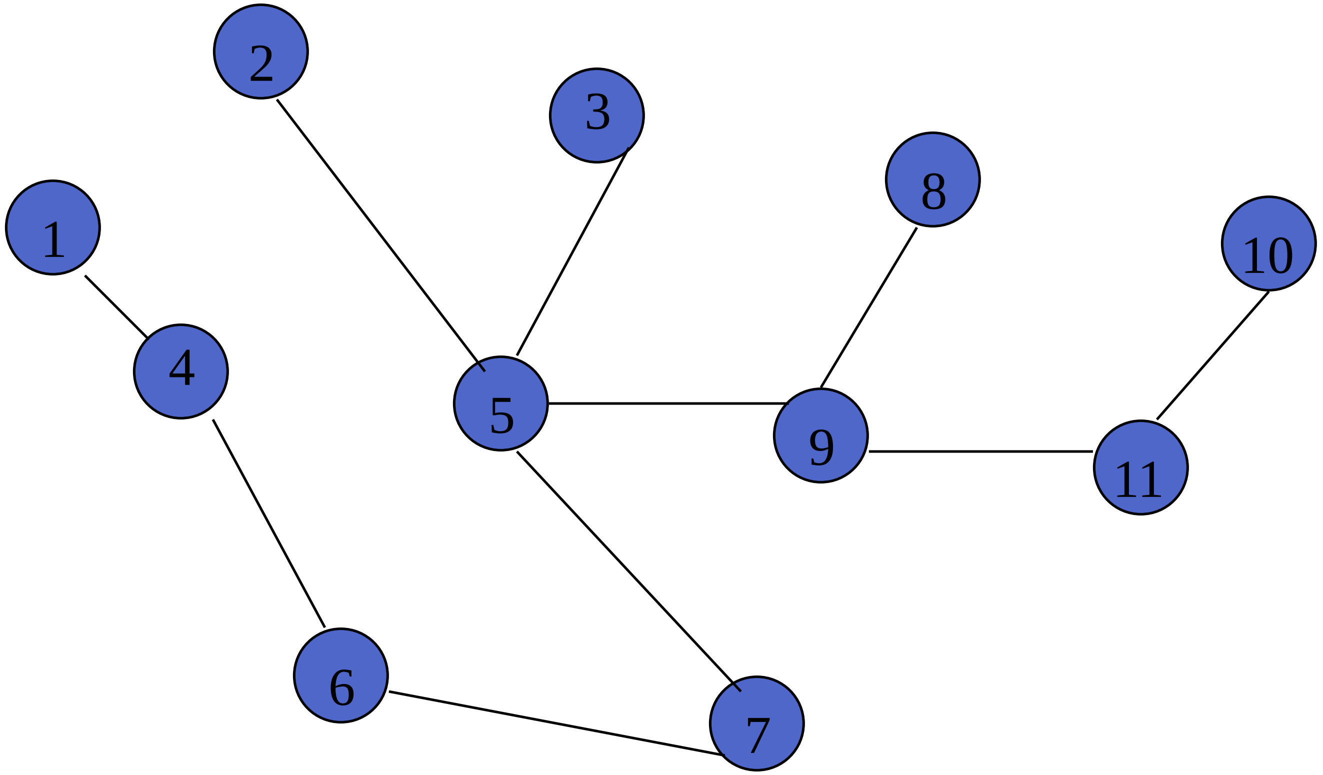
Cycles And Connectedness

Removal of an edge that is on a cycle does not affect connectedness.



Cycles And Connectedness

Connected subgraph with all vertices and minimum number of edges has no cycles.



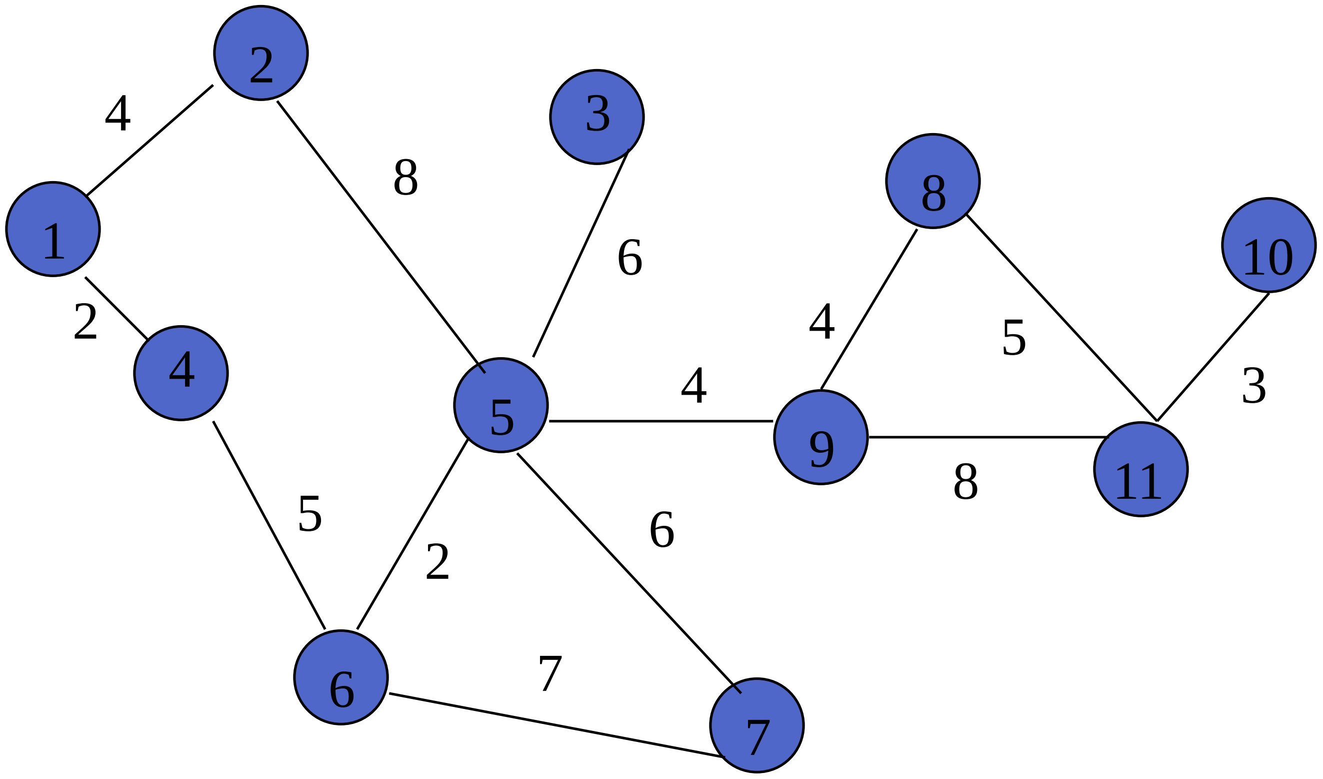
Spanning Tree

- Communication Network Problems
  - Is the network connected?
  - Can we communicate between every pair of cities?
  - Find the components.
  - Want to construct smallest number of feasible links so that resulting network is connected.

- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

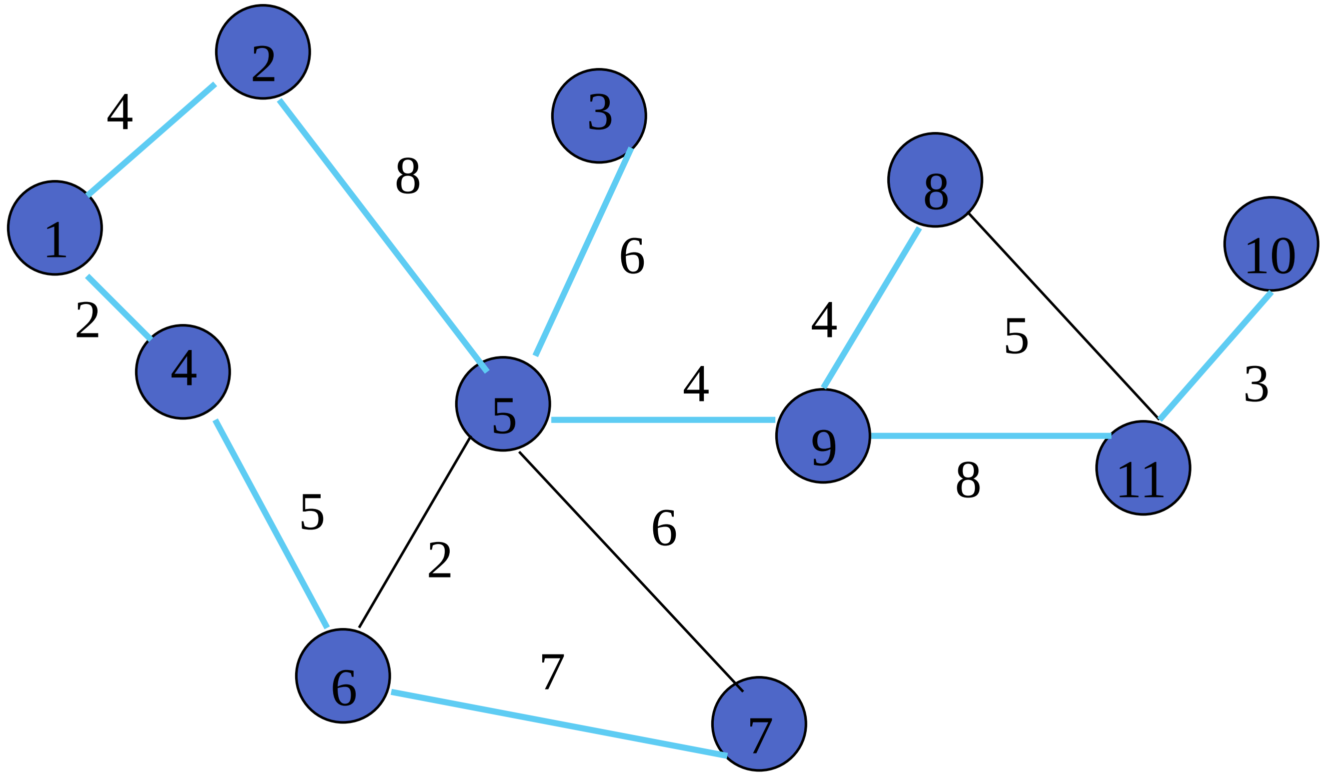
Minimum Cost Spanning Tree

- Tree cost is sum of edge weights/costs.



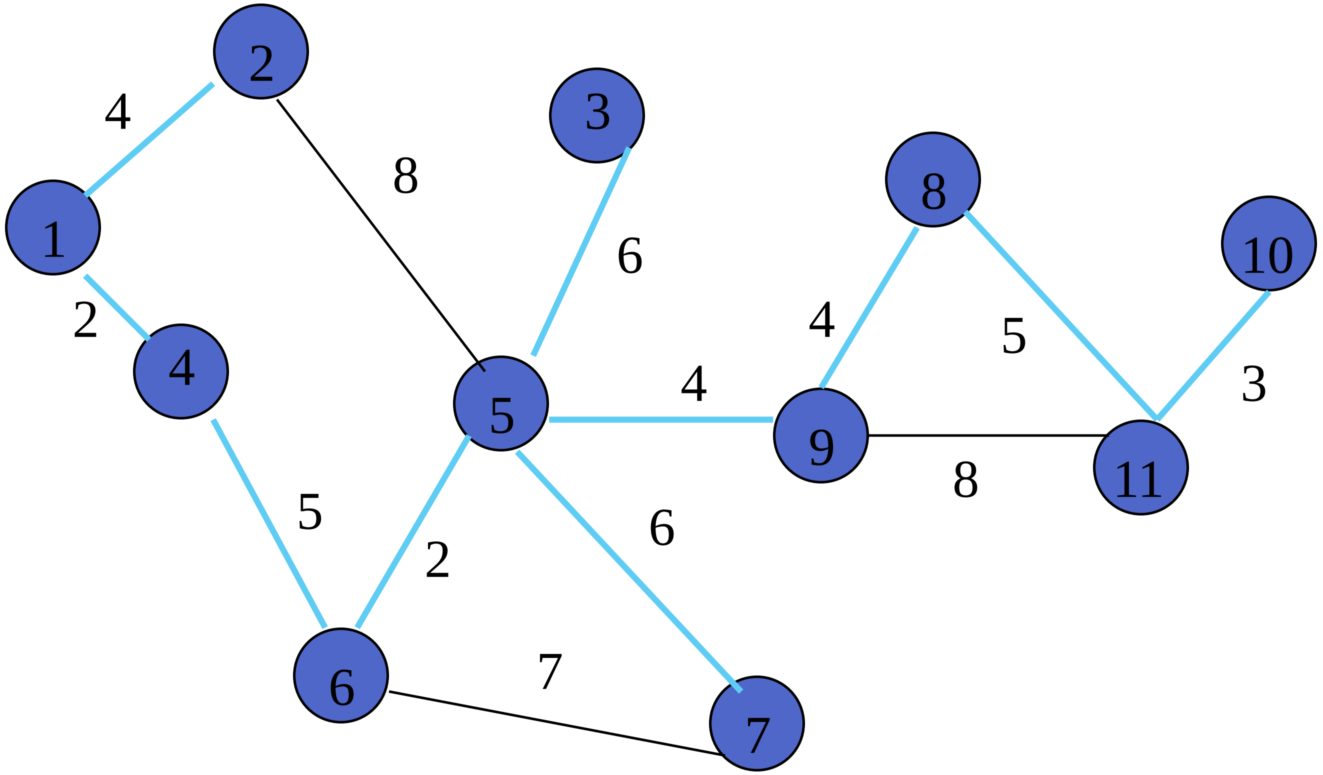
A Spanning Tree

A Spanning tree, cost = 51.



Minimum Cost Spanning Tree

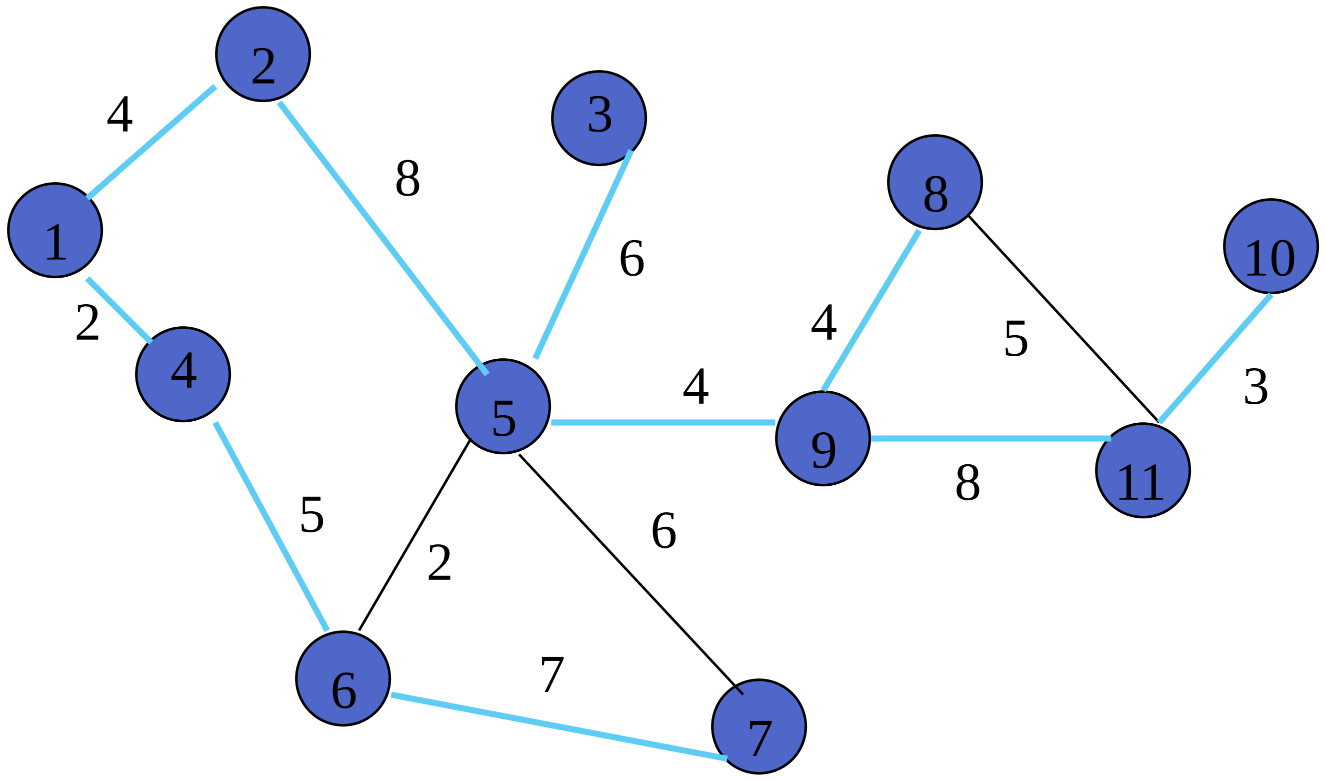
Minimum Spanning tree, cost = 41.



A Wireless Broadcast Tree

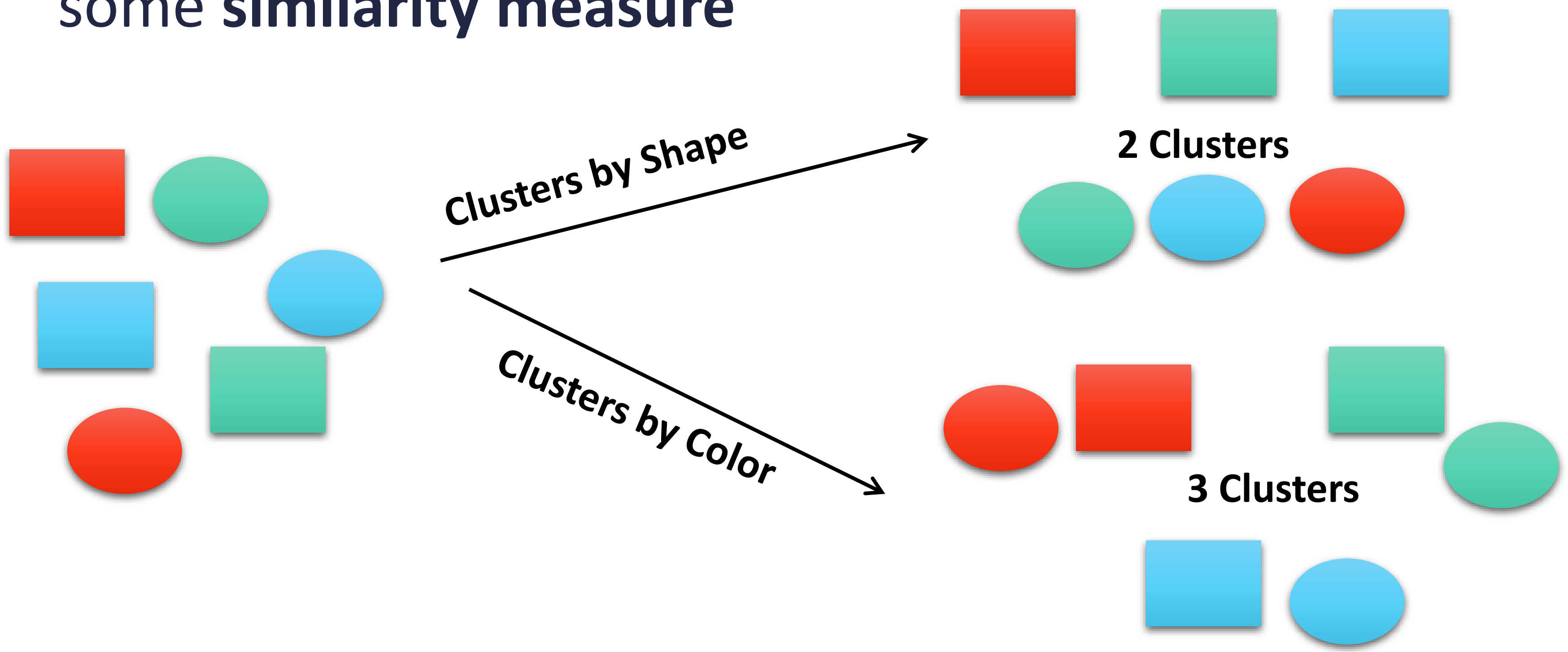
Source = 1, weights = needed power.

Cost = 4 + 8 + 5 + 6 + 7 + 8 + 3 = 41.



Graph Clustering

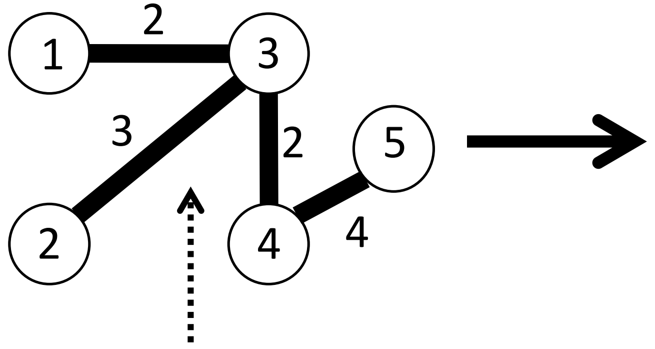
- **<u>Clustering</u>**: The process of dividing a set of input data into possibly overlapping, subsets, where elements in each subset are considered related by some **similarity measure**



Graph Clustering

- Between-graph
  - Clustering a set of graphs
  - E.g. structural similarity between chemical compounds
- Within-graph
  - Clustering the nodes/edges of a single graph
  - E.g., In a social networking graph, these clusters could represent people with same/similar hobbies

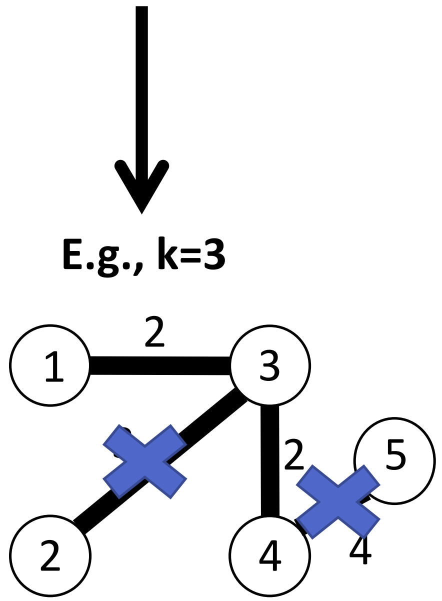
Graph Clustering: k-spanning Tree

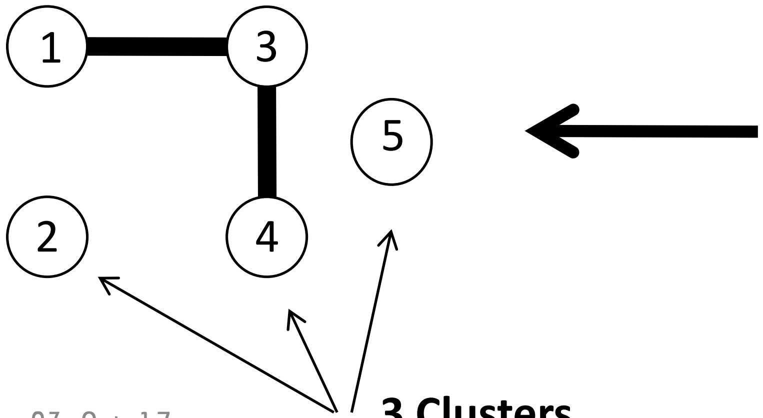


Minimum Spanning Tree

Remove k-1 edges with highest weight

Note: *k – is the number of clusters*





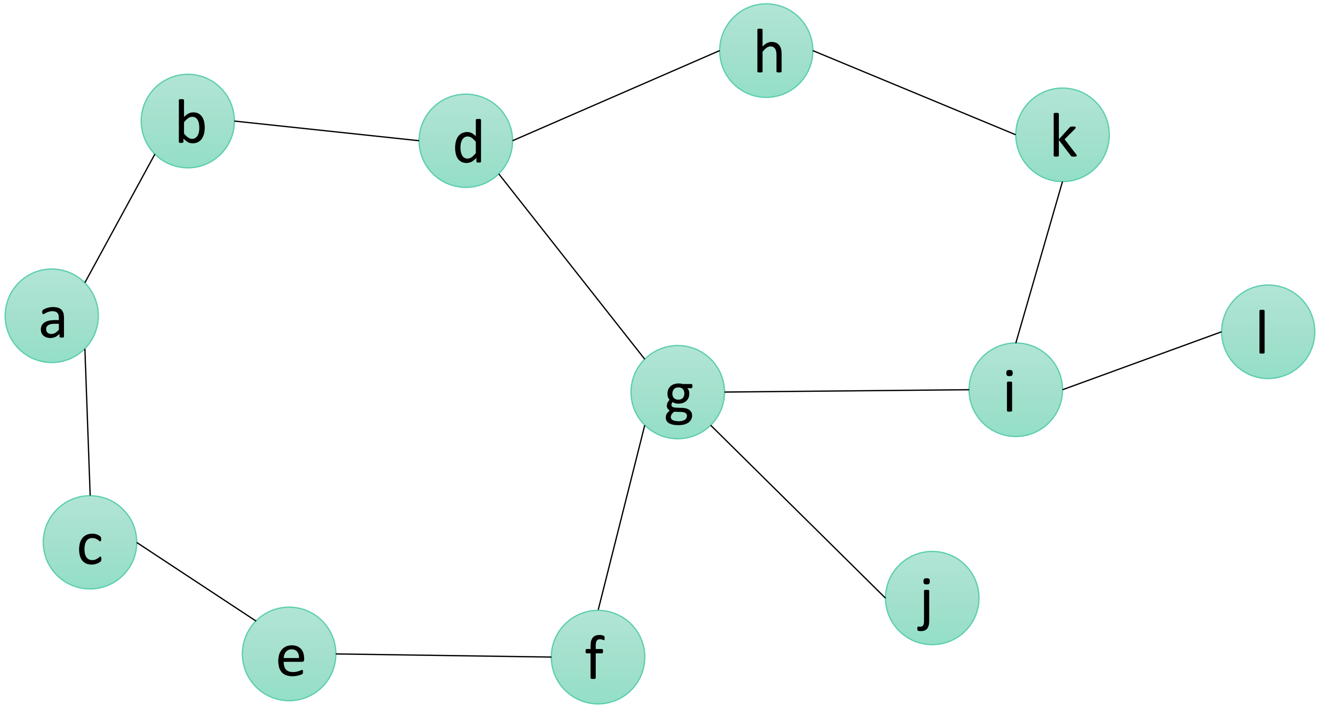
3 Clusters

Graph Clustering: k-means Clustering

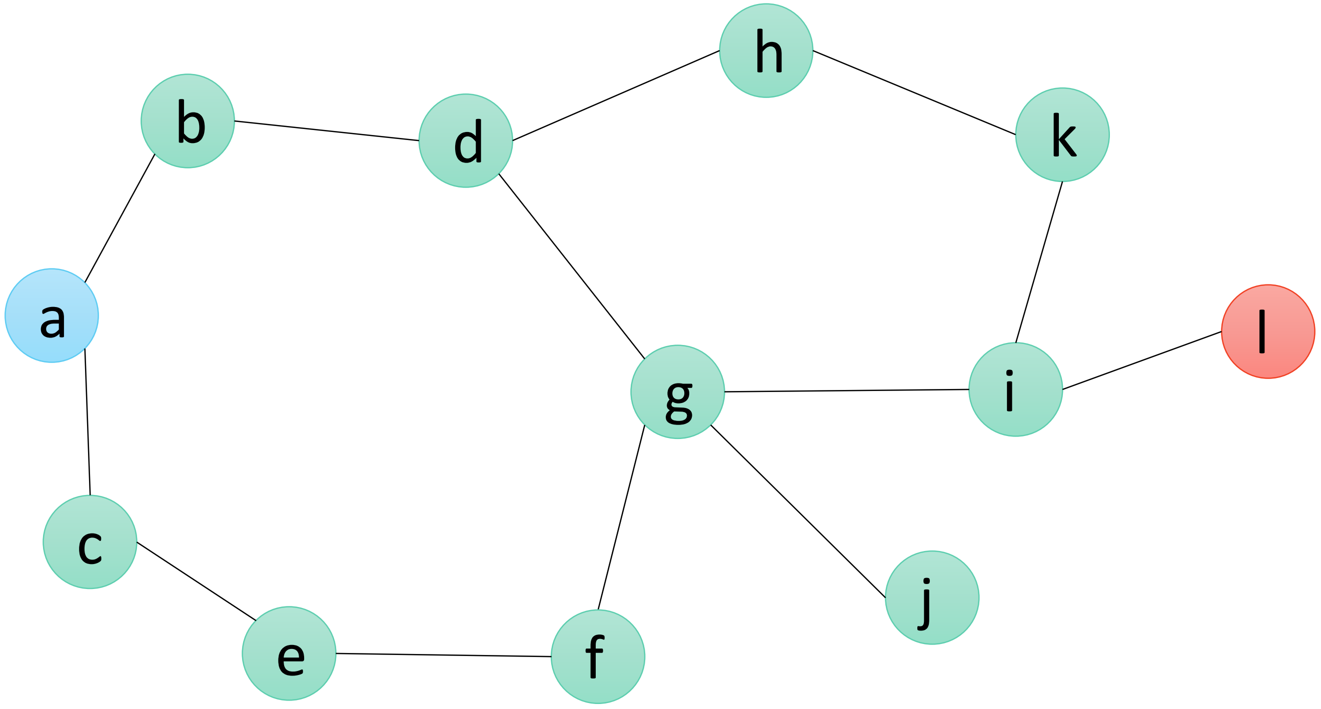
1. Identify k random vertices as centers, label them with unique colors
2. Start BFS traversal from each center, one level at a time
3. Label the vertices reached from each BFS center with its colors
4. If multiple centers reach the same vertex at same level, pick one of the colors
5. Continue propagation till all vertices colored
6. Calculate edge-cuts between vertices of different colors
7. If cut less than threshold, stop. Else repeat and pick k new centers

K-Means Clustering

k=2, maxcut = 2

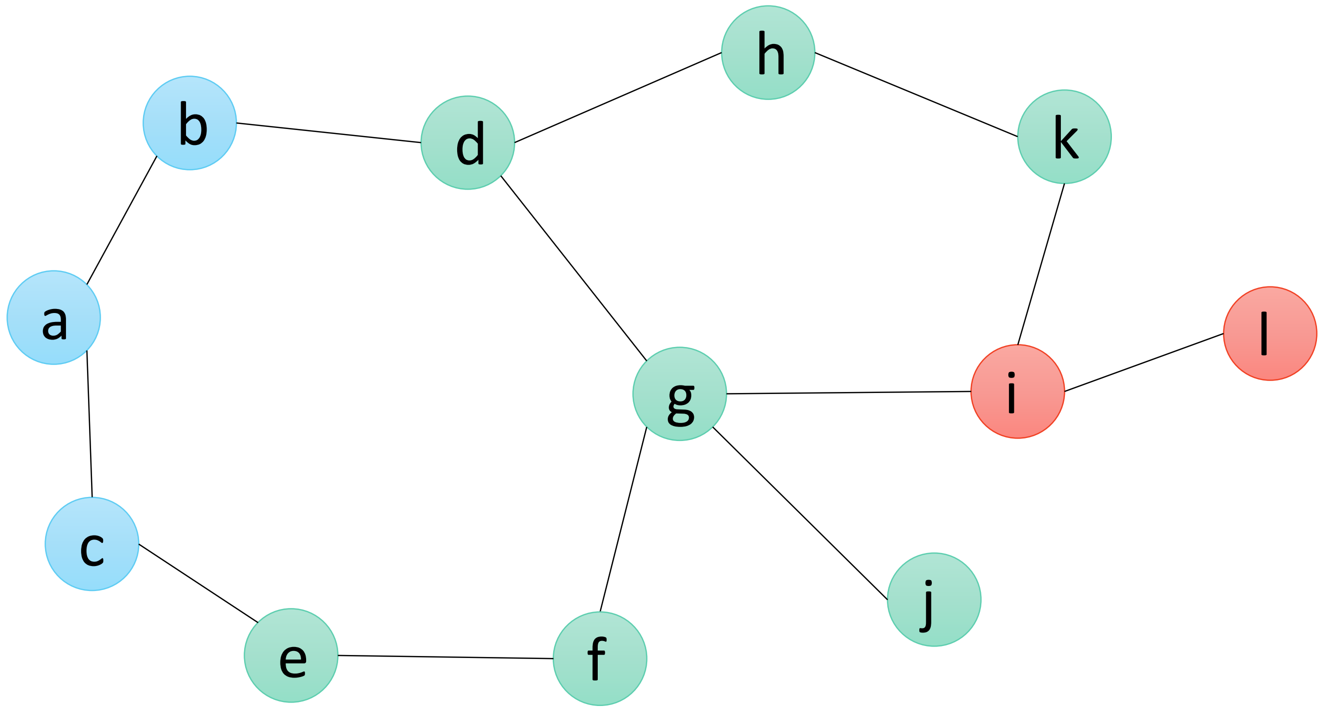


K-Means Clustering



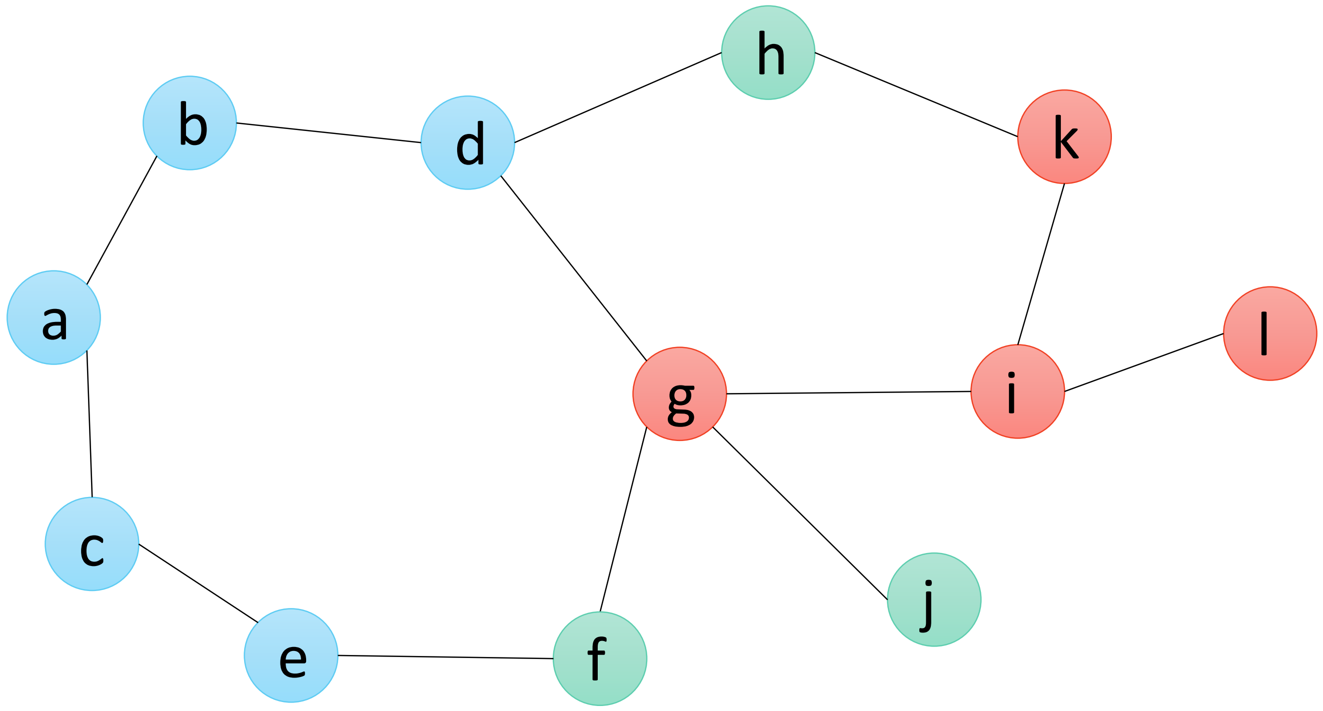
Pick *k* random vertices

K-Means Clustering



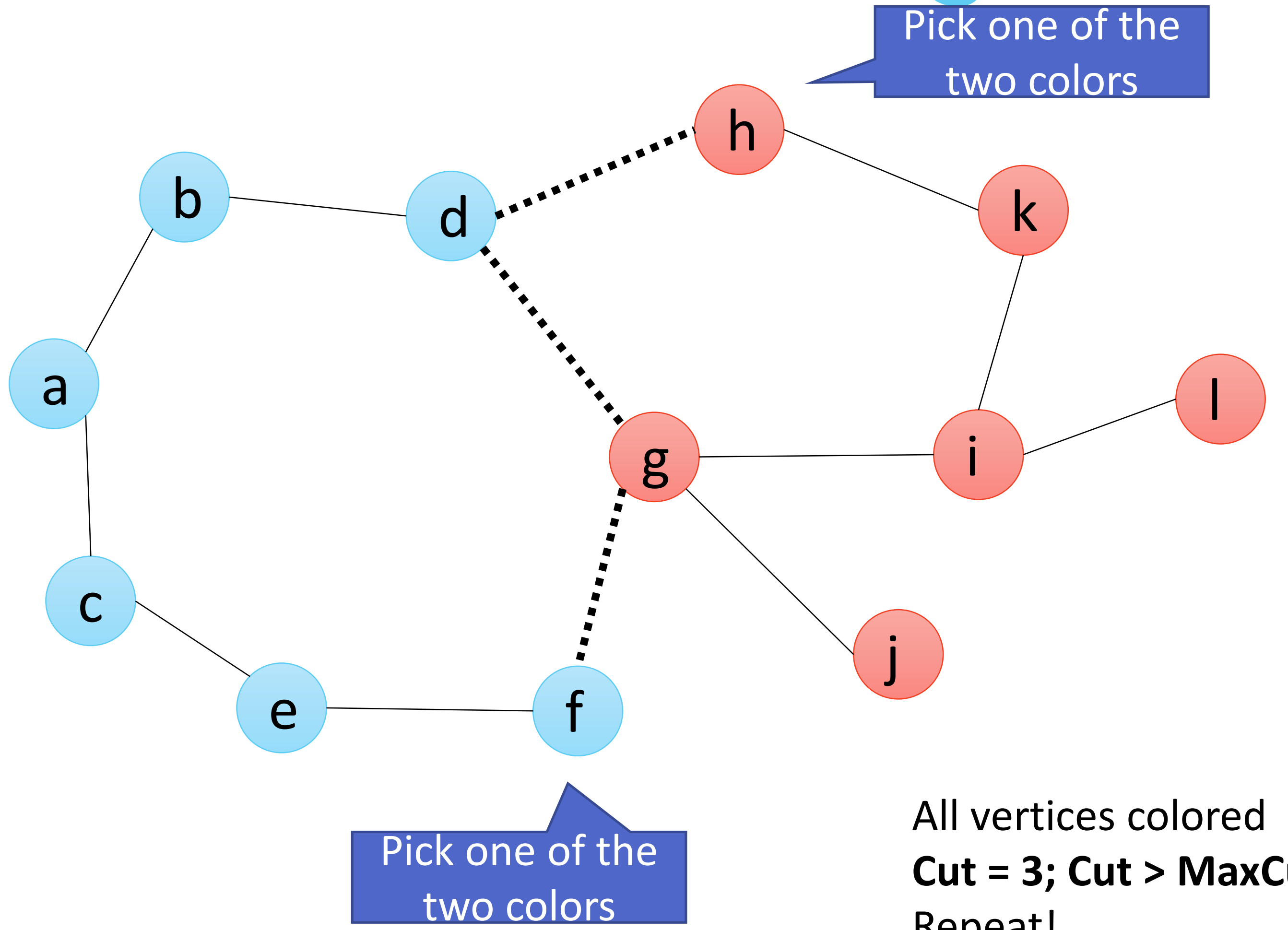
Perform k BFS simultaneously

K-Means Clustering



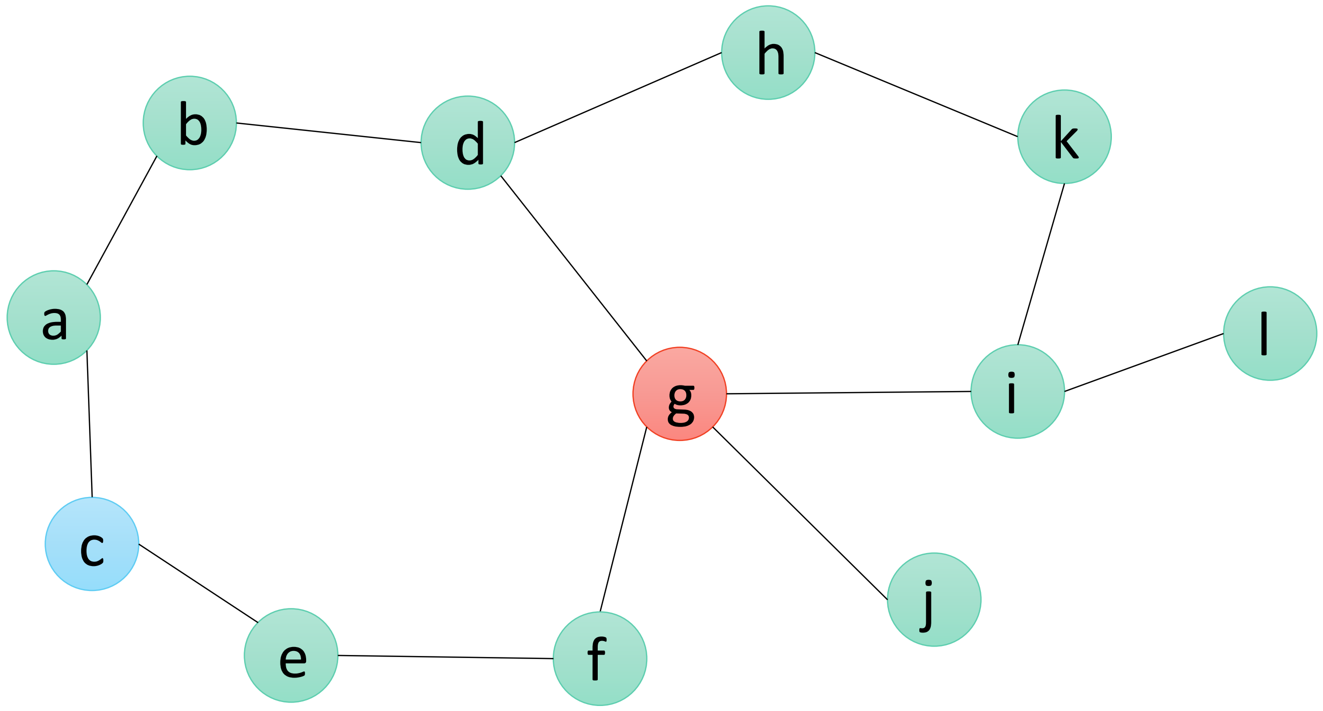
Perform k BFS simultaneously

K-Means Clustering



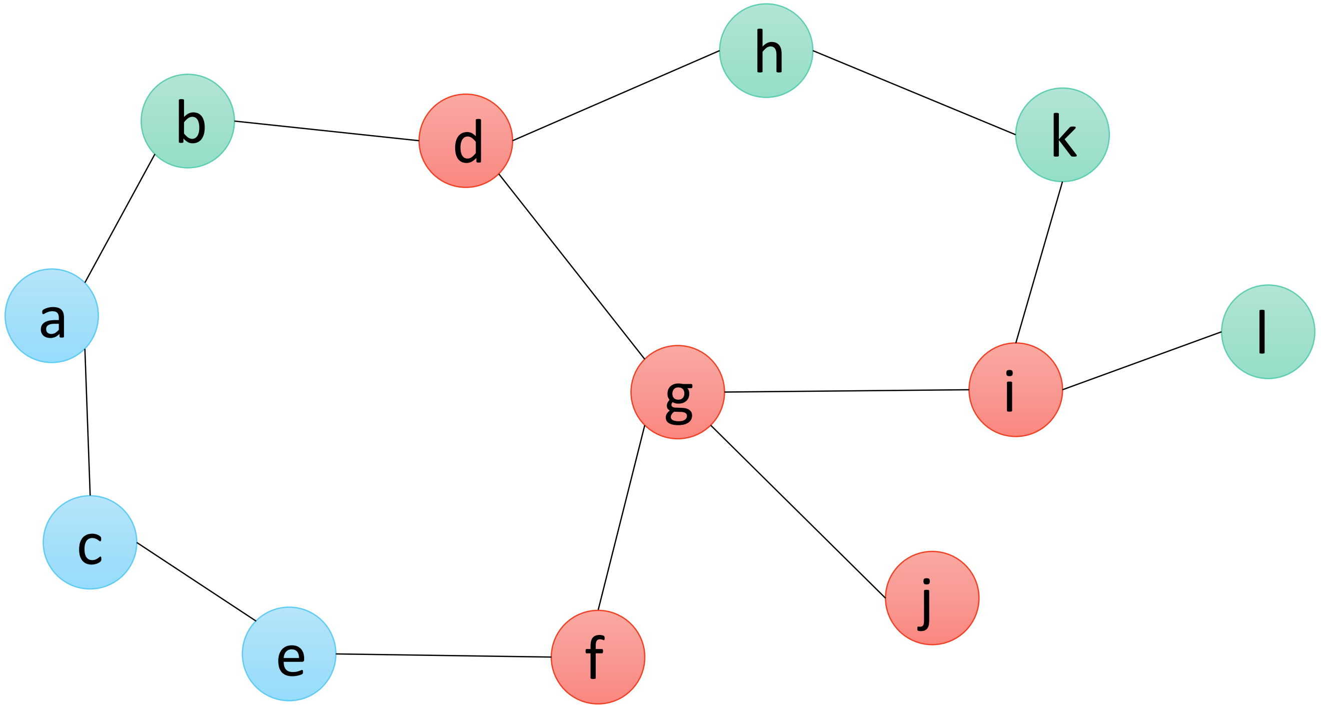
All vertices colored
Cut = 3; Cut > MaxCut
Repeat!

K-Means Clustering



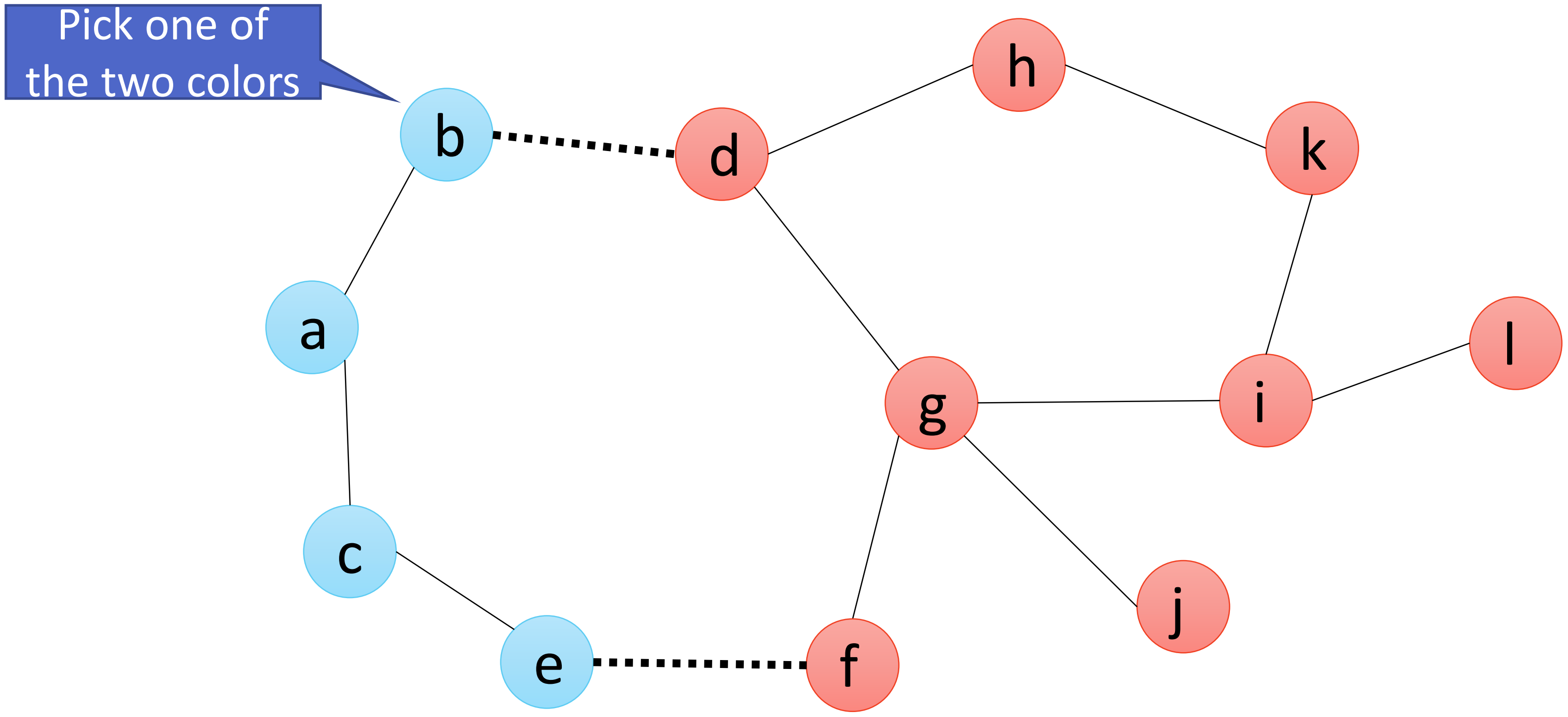
Pick *k* random vertices

K-Means Clustering



Perform k BFS simultaneously

K-Means Clustering



All vertices colored

Cut = 2; Cut <= MaxCut

Done!

PageRank

- Centrality measure of web page quality based on the web structure
  - How important is this vertex in the graph?
- Random walk
  - Web surfer visits a page, randomly clicks a link on that page, and does this repeatedly.
  - How frequently would each page appear in this surfing?
- Intuition
  - Expect high-quality pages to contain "endorsements" from many other pages thru hyperlinks
  - Expect if a high-quality page links to another page, then the second page is likely to be high quality too

PageRank, recursively

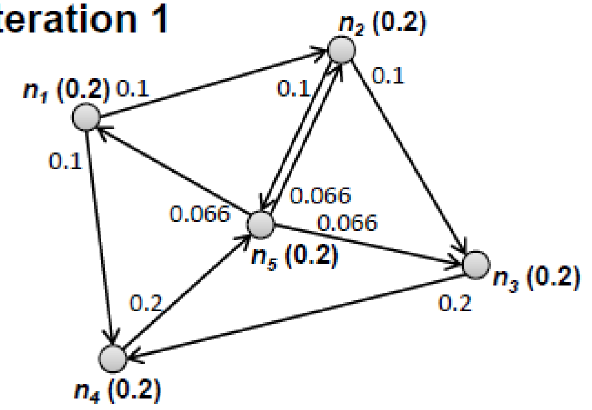
$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

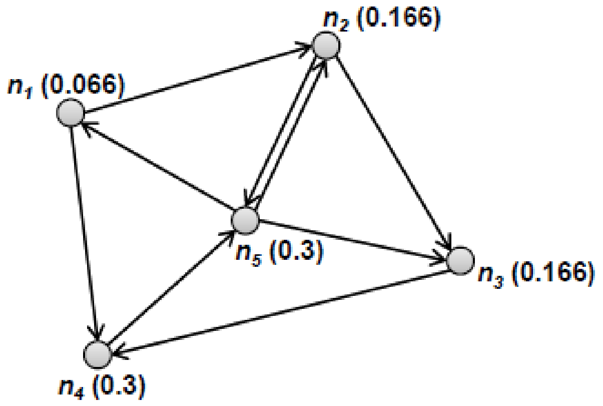
- P(n) is PageRank for webpage/URL ‘n’
  - Probability that you’re in vertex ‘n’
- |G| is number of URLs (vertices) in graph
- α is probability of random jump
- L(n) is set of vertices that link to ‘n’
- C(m) is out-degree of ‘m’
- Initial P(n) = 1/|G|

PageRank Iterations

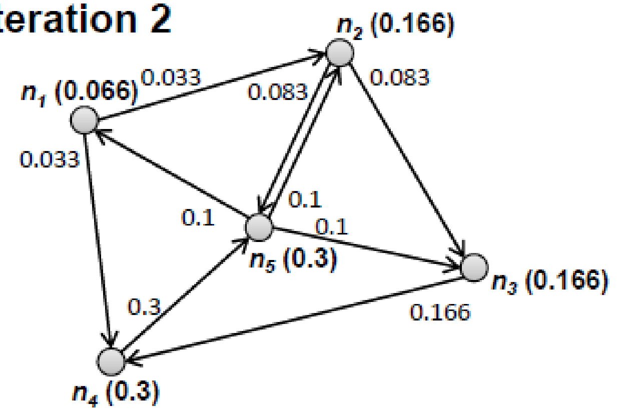
α=0
Initialize P(n)=1/|G|

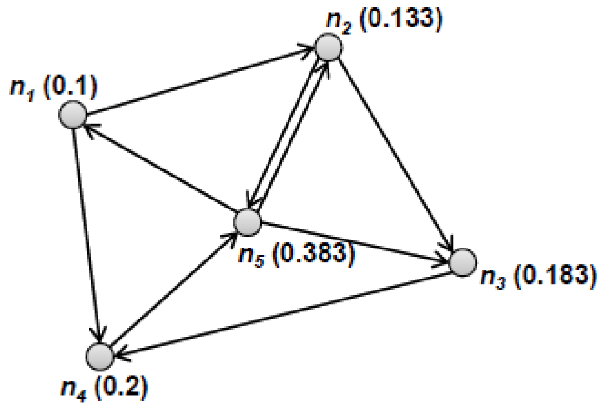
Iteration 1





Iteration 2





Lin, Fig 5.7

Tasks

- Self study
  - **Read**: Graphs and graph algorithms (online sources)
- Attend tutorial on C++, turing cluster at 5pm today
- Finish Assignment 2 by Fri Oct 13 *(10% points)*
  - **Posted online today**

Questions?