DS221

Data Structures, Algorithms & Data Science Platforms

Instructor: Chirag Jain
(slides from Prof. Simmhan)

L5: Graphs

Graph ADT, Algorithms

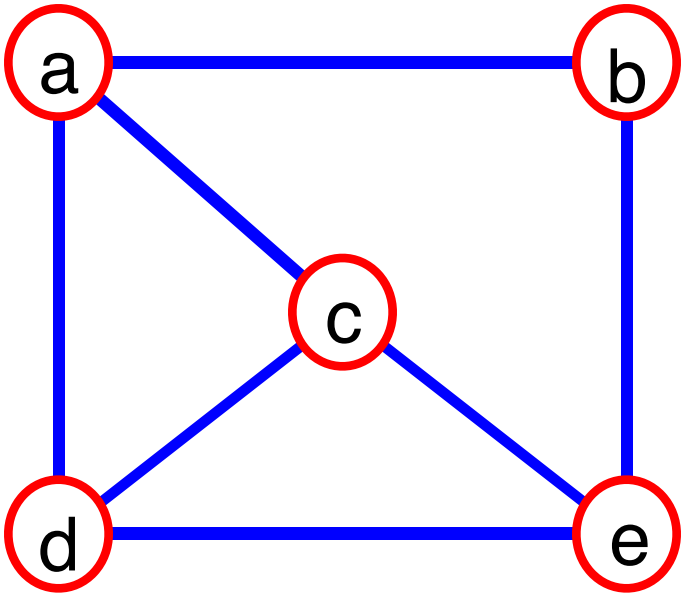
What is a Graph?

- 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:

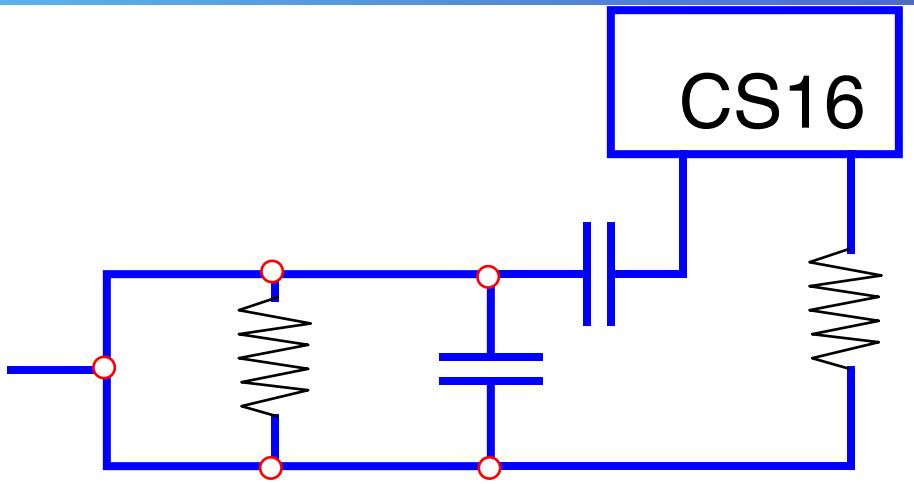


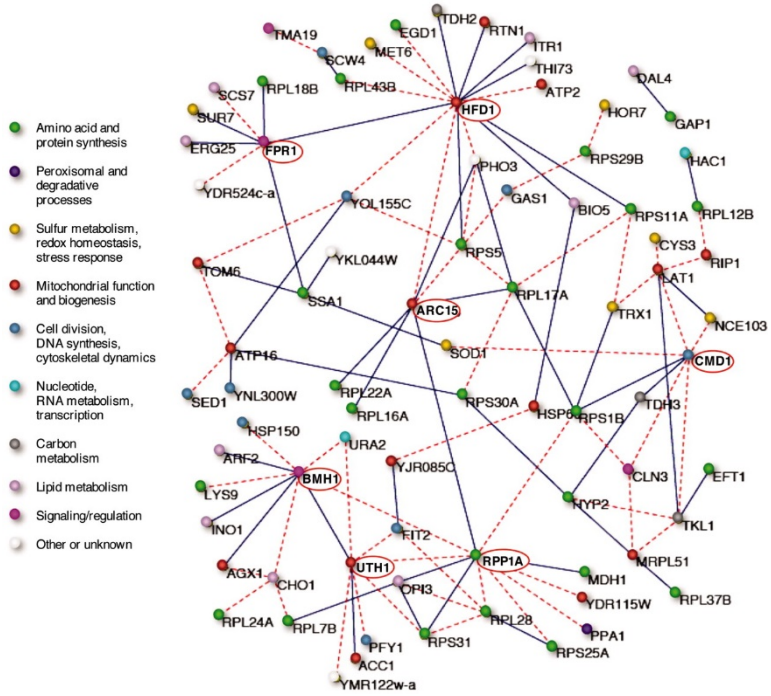
V= {a,b,c,d,e}

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

Applications

- Electronic circuit design
- Transport networks
- Biological Networks



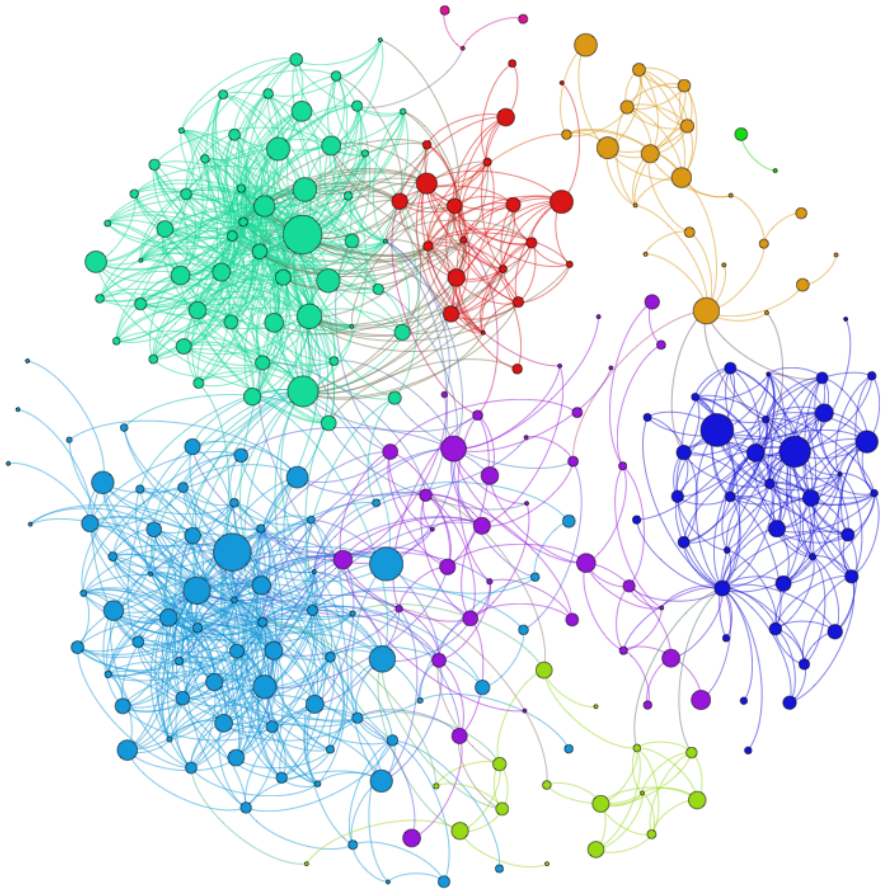


http://www.pnas.org/content/103/50/19033/F3.expansion.html

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

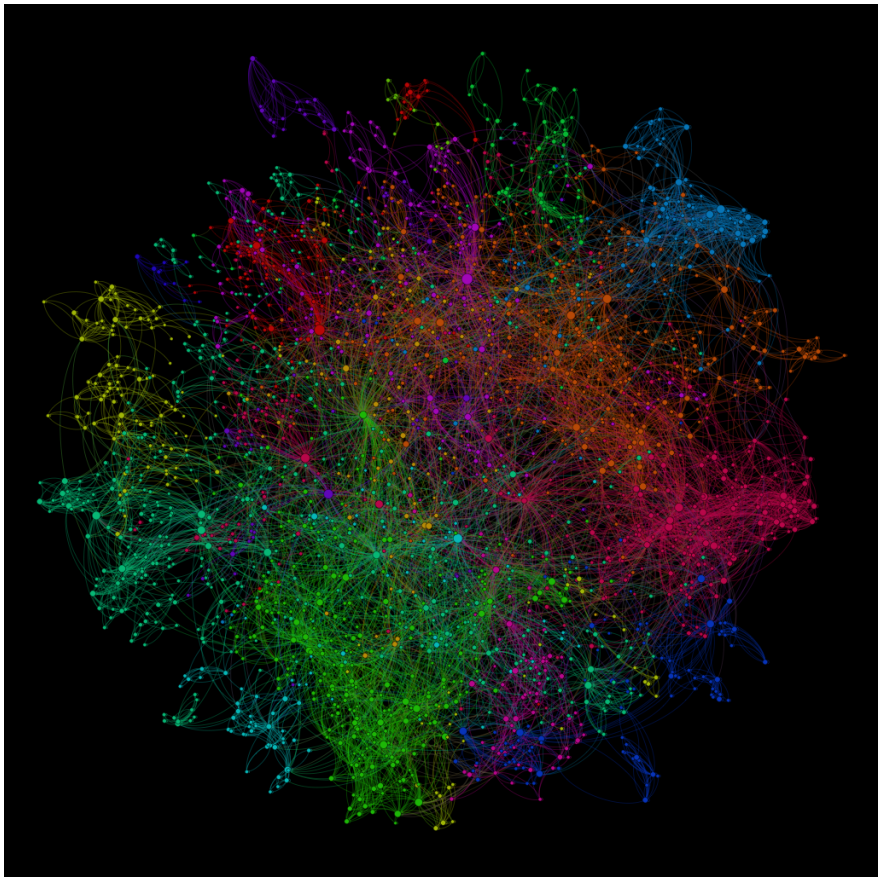
Applications

LinkedIn Social Network Graph



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

Java Call Graph for Neo4J



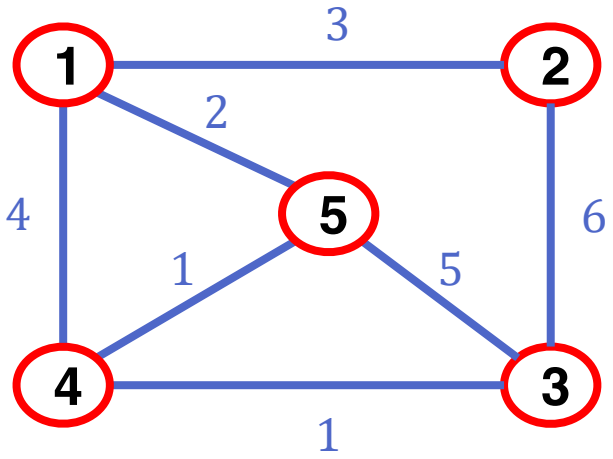
http://allthingsgraphed.com/2014/11/12/code-graphs-5-top-open-source-data-projects

Terminology

- If (v_0, v_1) is an edge in an **undirected** graph,
  - v_0 and v_1 are adjacent, or are neighbours
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $\langle v_0, v_1 \rangle$ is an edge in a **directed** graph
  - v_0 is the source vertex and v_1 is the target 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, also
  - the in-degree of a vertex *v* is the number of edges that have *v* as the target 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

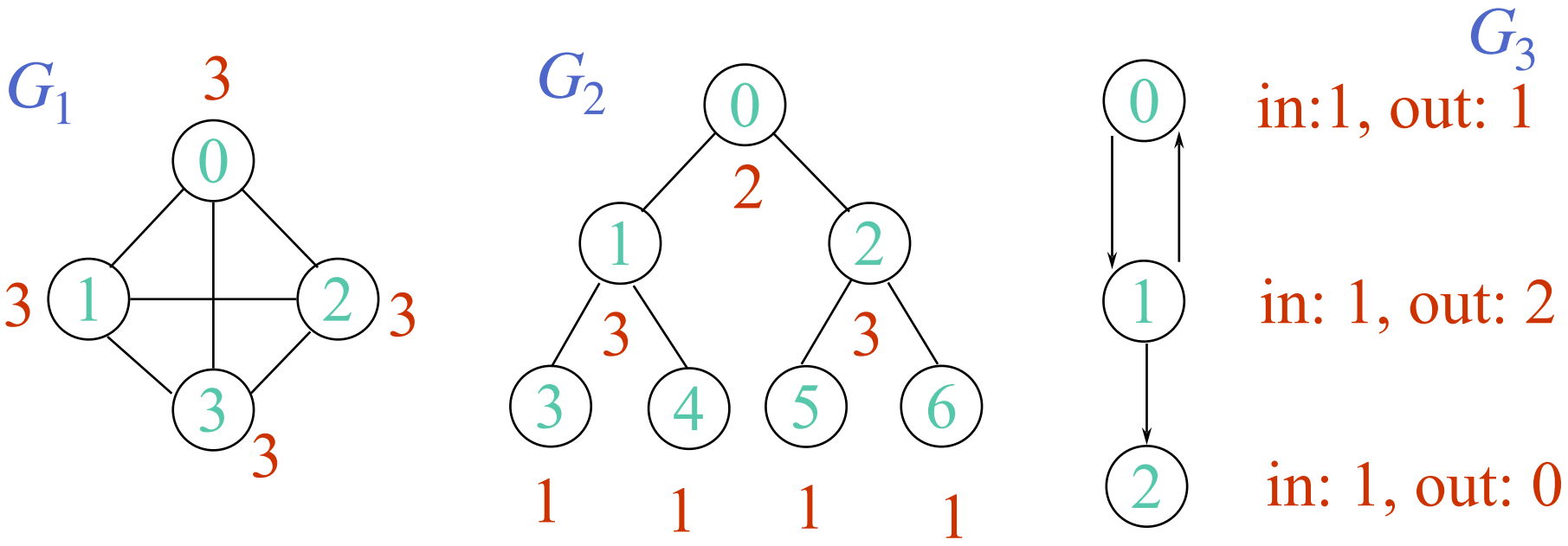
Terminology

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph, also
  - the in-degree of a vertex *v* is the number of edges that have *v* as the target 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 two vertices count an edge, so it will be counted twice

Examples

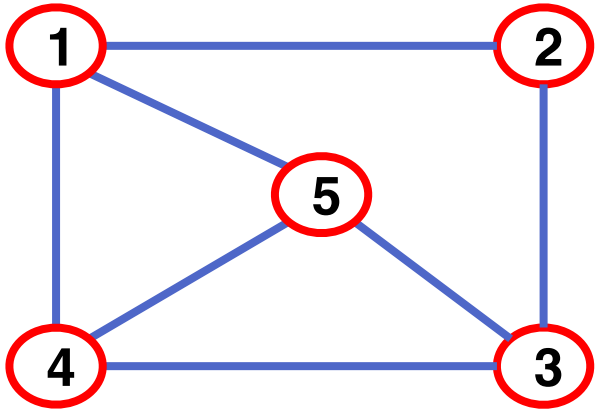


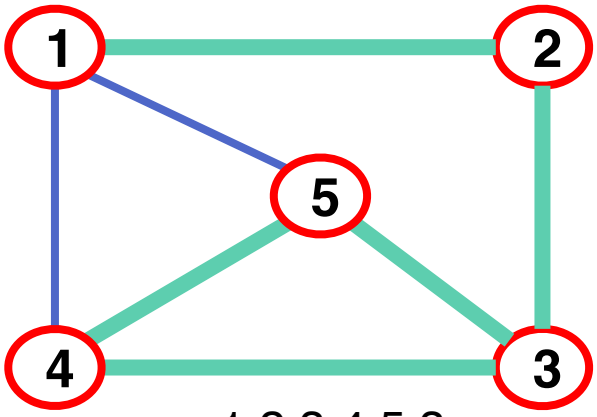
undirected graphs

directed graph

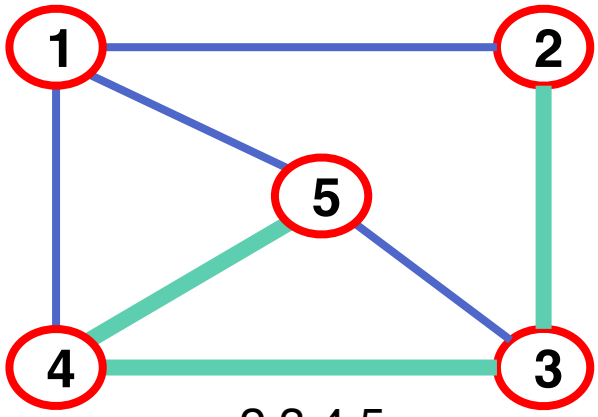
Terminology

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





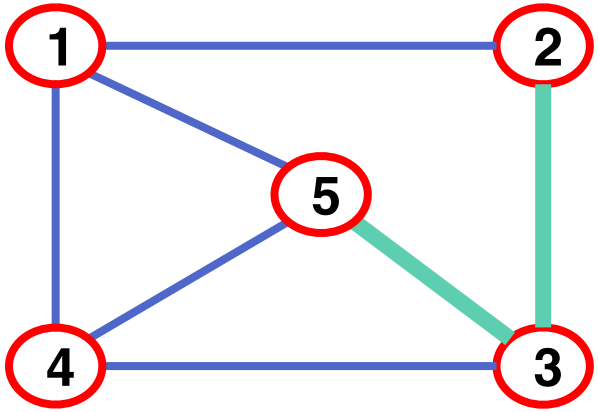
1 2 3 4 5 3



2 3 4 5

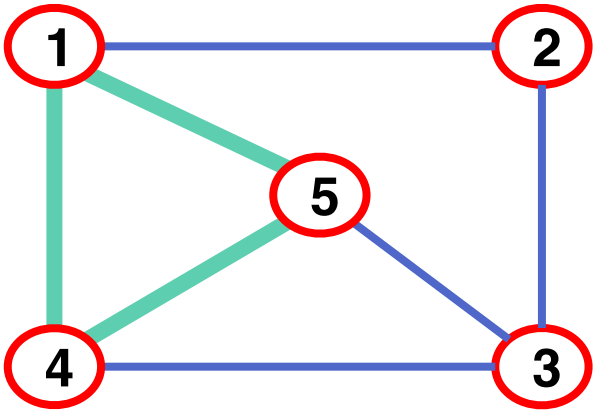
Terminology

- simple path: no repeated vertices



2 3 5

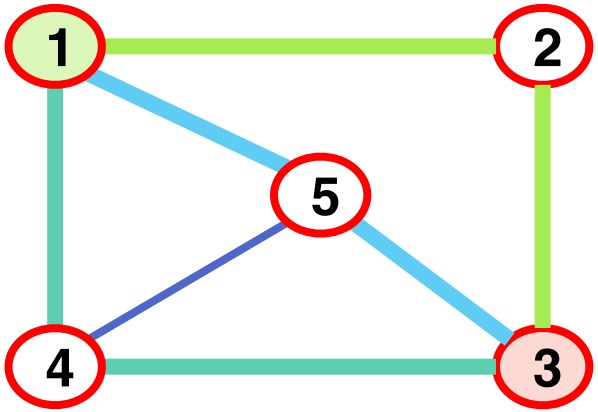
- cycle: simple path, except that the last vertex is the same as the first vertex



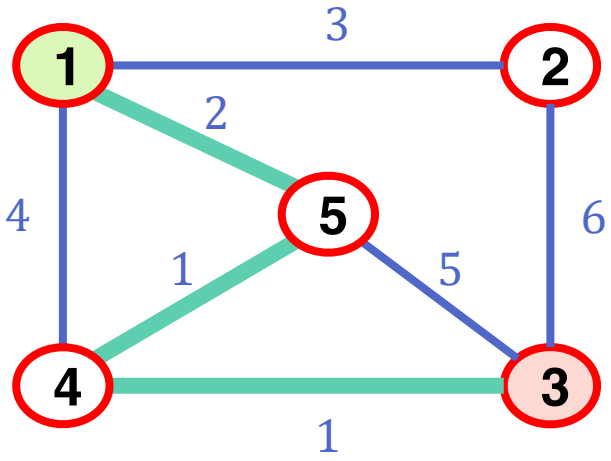
1 5 4 1

Terminology

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



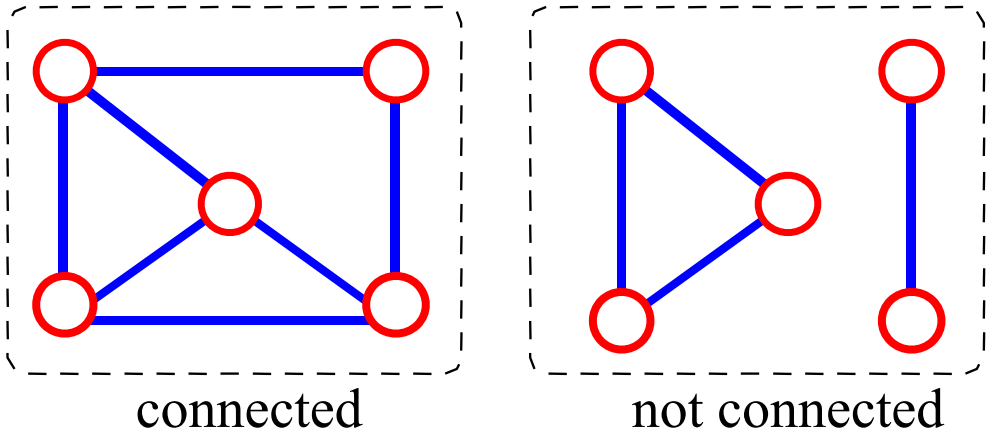
1 2 3
1 5 3
1 4 3



1 5 4 3

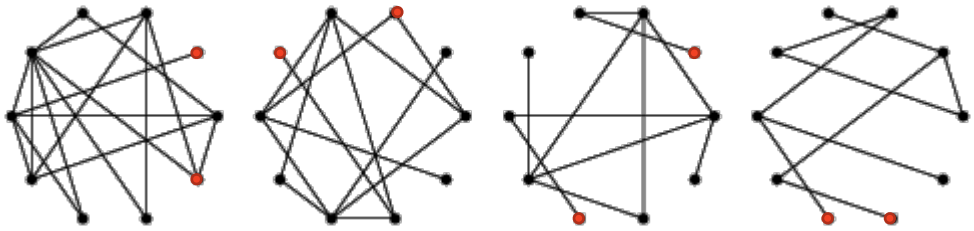
Connected Graph

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



Graph Diameter

- A graph's diameter 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

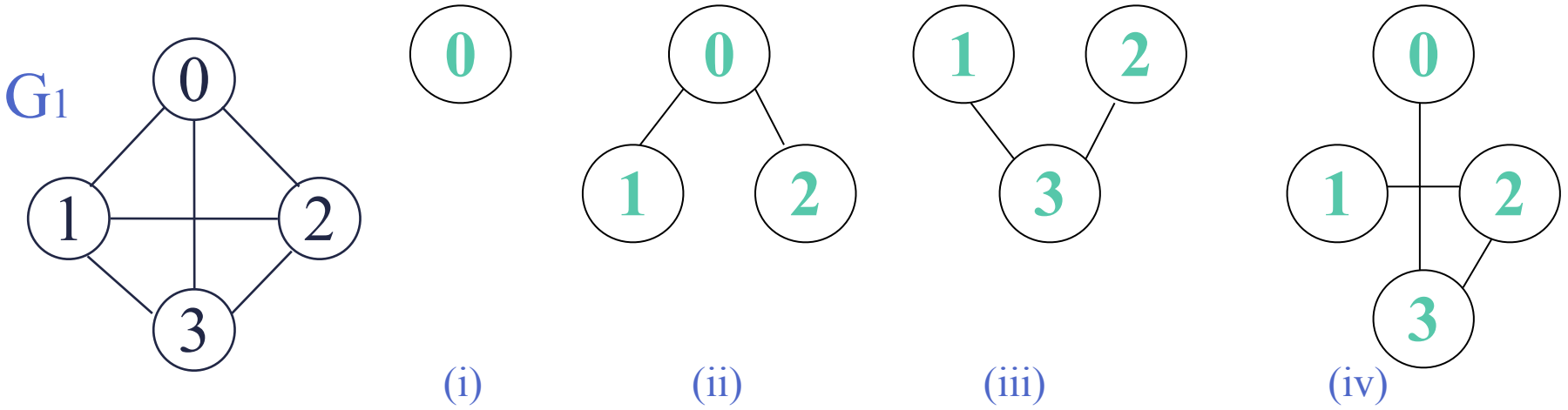


d=3 *d=4* *d=5* *d=7*

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

Subgraph

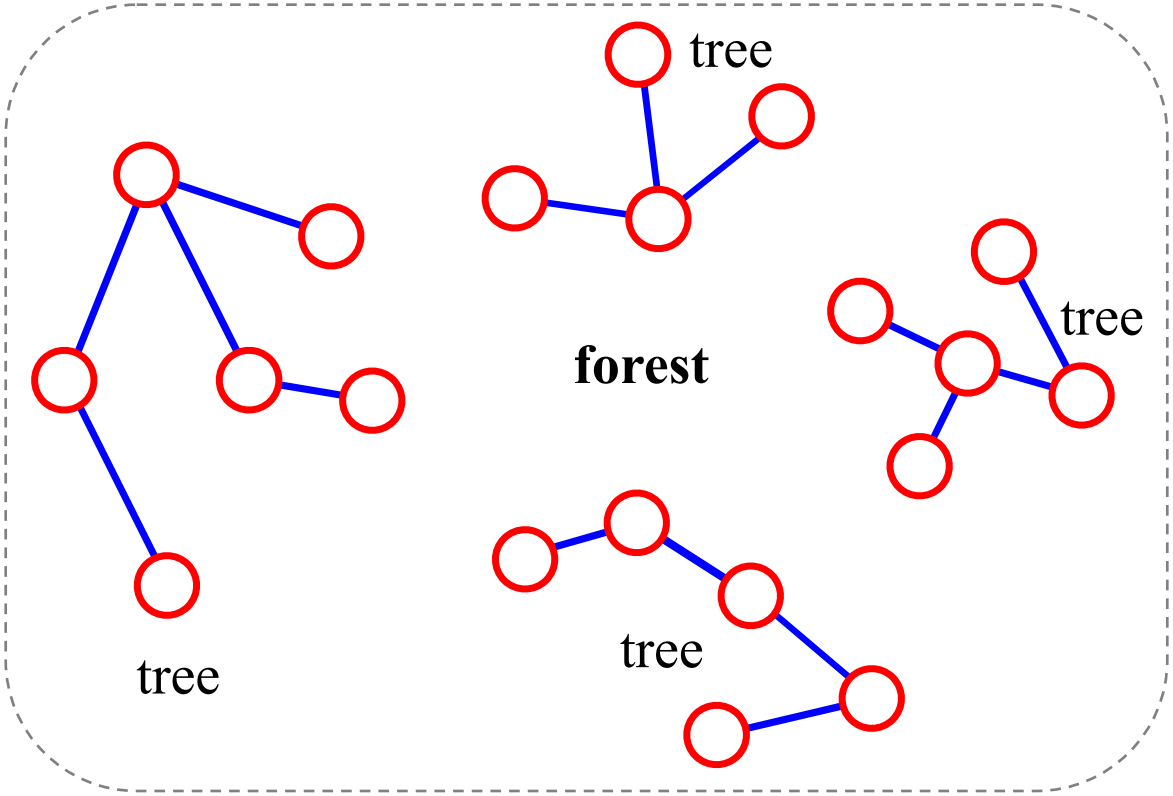
- subgraph: subset of vertices and edges forming a graph



(a) Some of the subgraphs of G_1

Trees & Forests

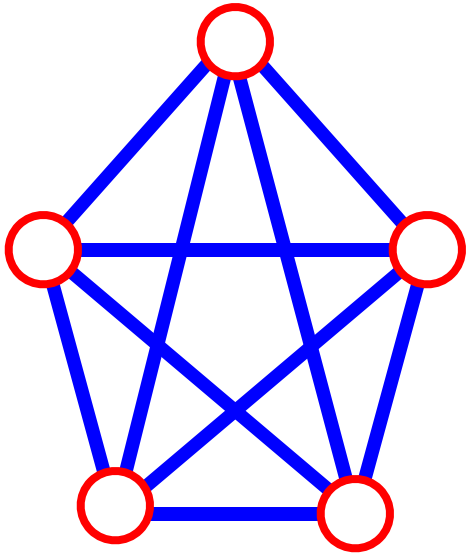
- tree - connected graph without cycles
- forest - collection of trees



Fully Connected Graph

- Let **n** = #vertices, and **m** = #edges
- **Complete graph (or) Fully connected graph**: One in which all pairs of vertices are adjacent
- *Count of 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, m = **n**(**n** -1)/2.

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



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

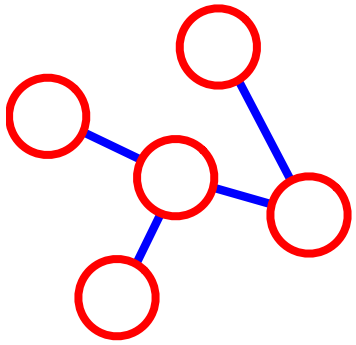
More Connectivity

n = #vertices

m = #edges

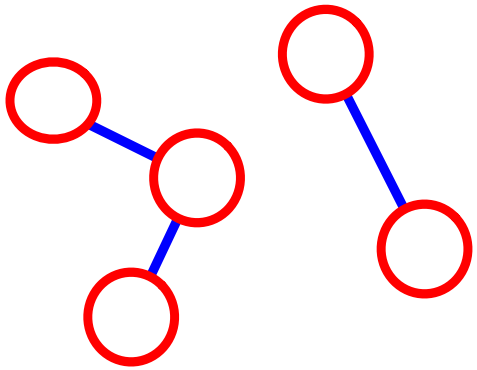
- For a tree **m** = **n** - 1

(Try to prove using induction)



n = 5
m = 4

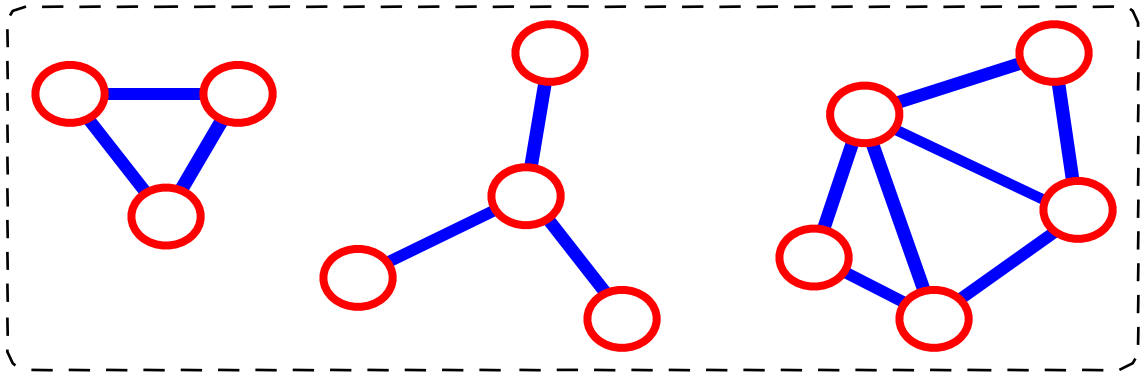
If **m** < **n** - 1, G is not connected



n = 5
m = 3

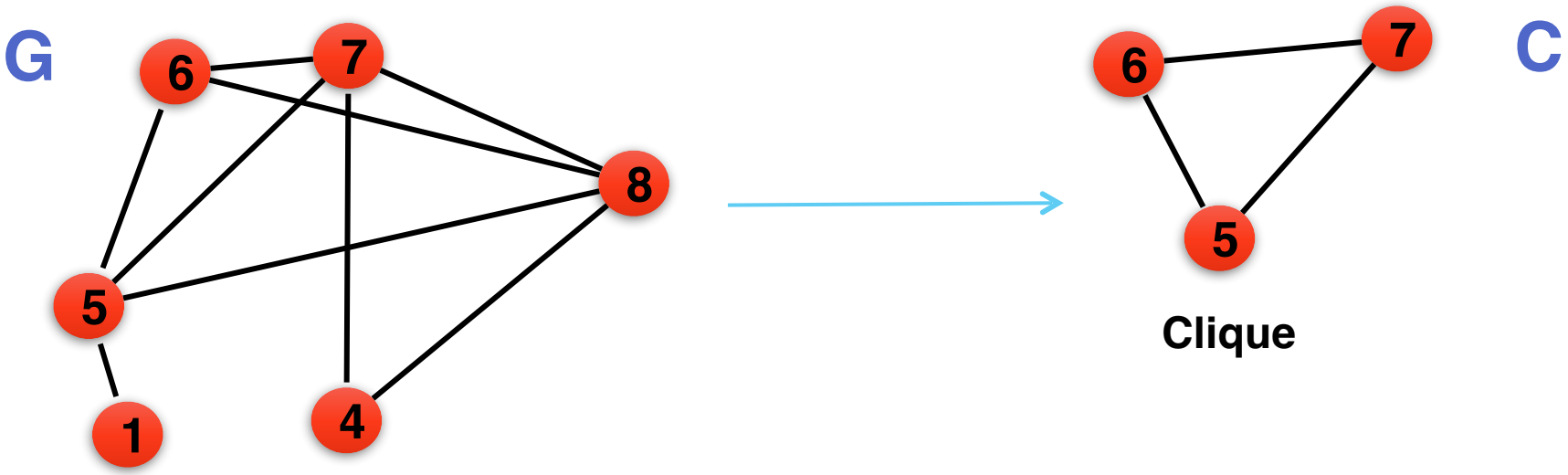
Connected Component

- A connected component is a maximal set of nodes such that each pair of nodes is connected by a path
- A connected graph has exactly one connected component.



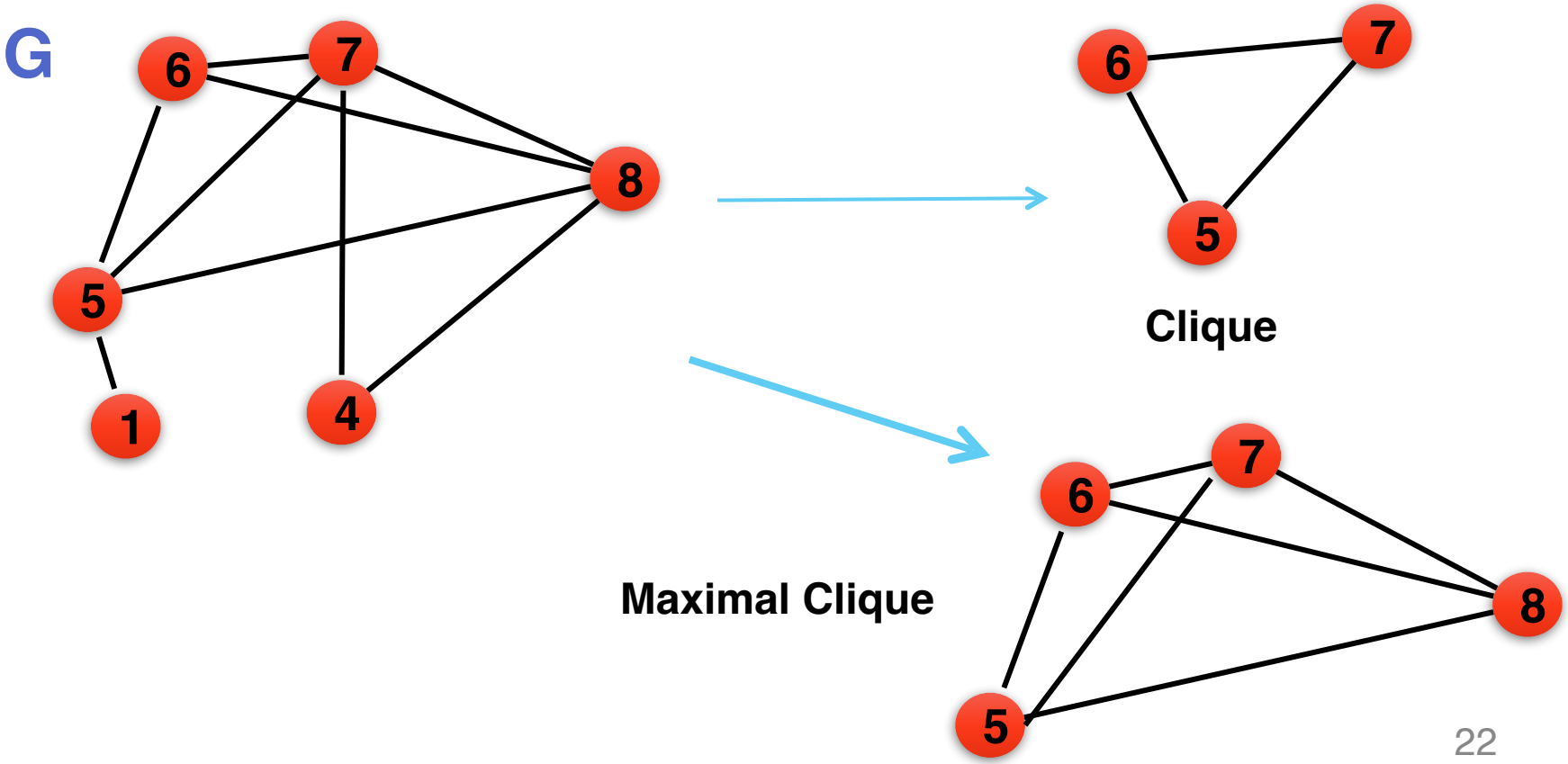
Clique

- 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

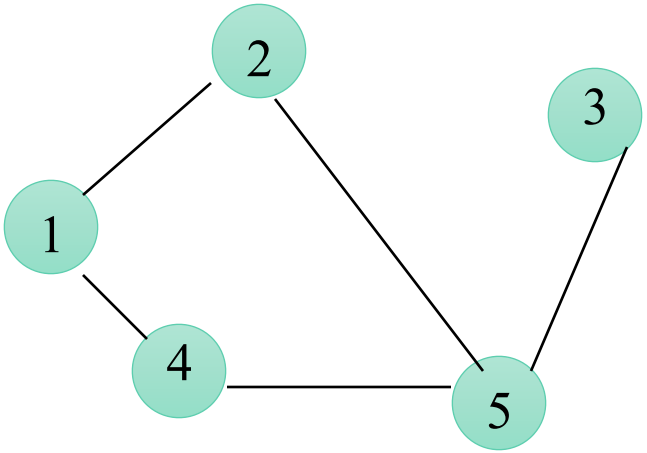


Graph Representation

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

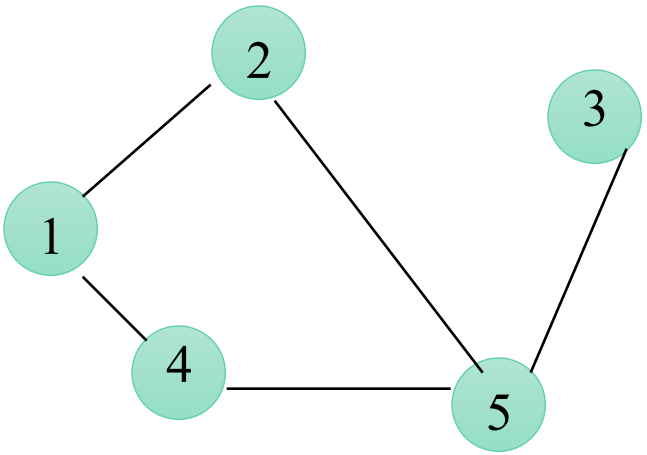
Adjacency Matrix

- Boolean $n \times n$ matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



	1	2	3	4	5
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

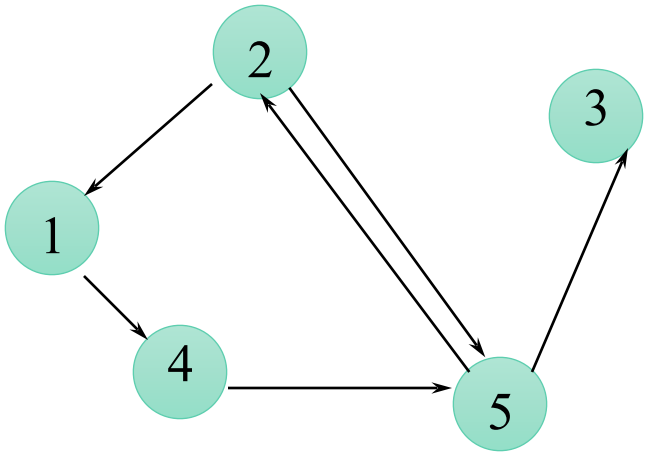
Adjacency Matrix Properties



	1	2	3	4	5
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

- 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)



	1	2	3	4	5
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

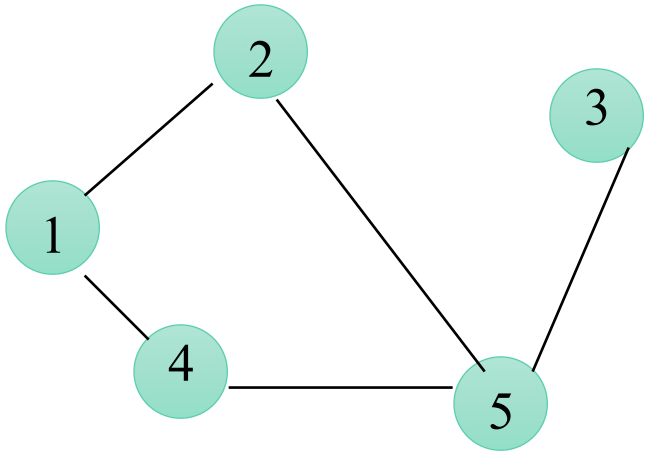
- Diagonal entries are zero.
- Adjacency matrix of a directed graph need not be symmetric.

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)

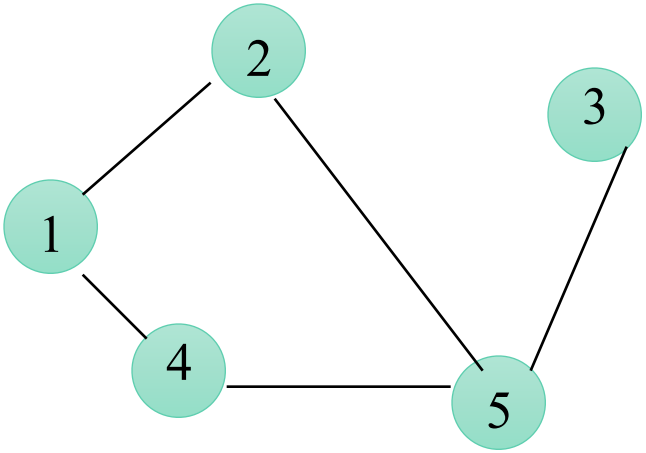
aList[3] = (5)

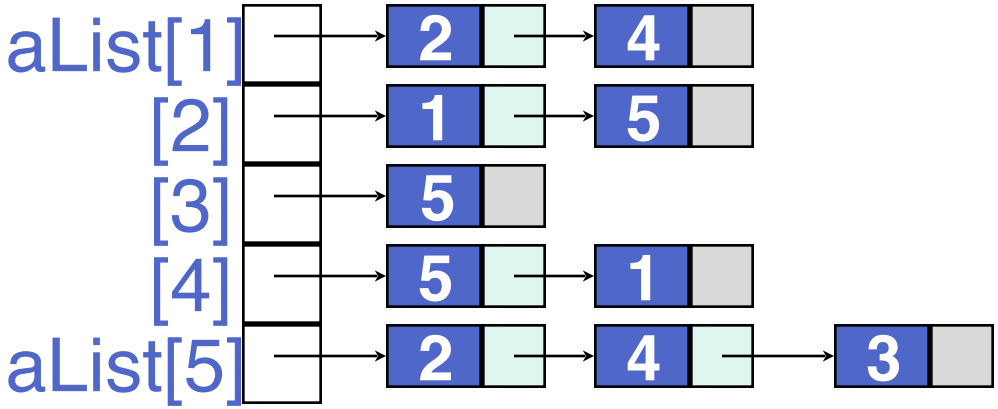
aList[4] = (5,1)

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)

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> GetTarget();
}
```

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);
}
```

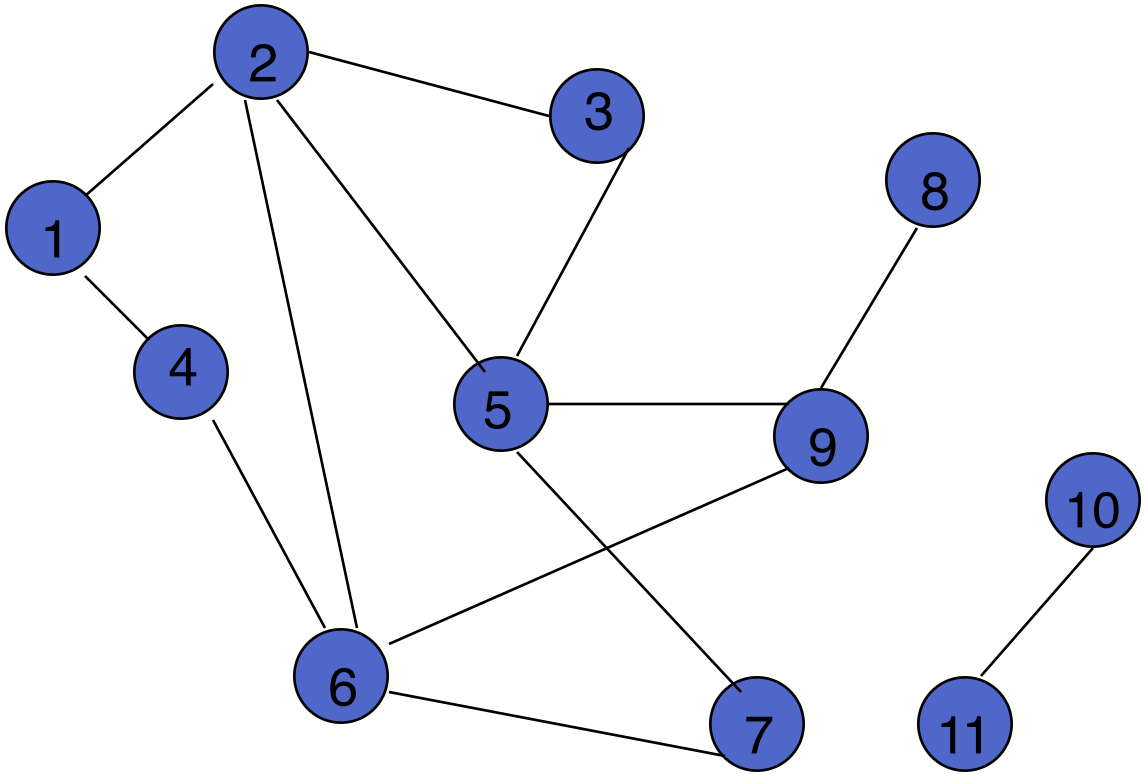
Sample Graph Problems

- Graph traversal
  - Searching
  - Shortest Paths
  - Connectedness
- Spanning Tree
- Graph centrality
  - PageRank
  - Betweenness centrality
- Graph clustering
  - K-means clustering

Graph Search & Traversal

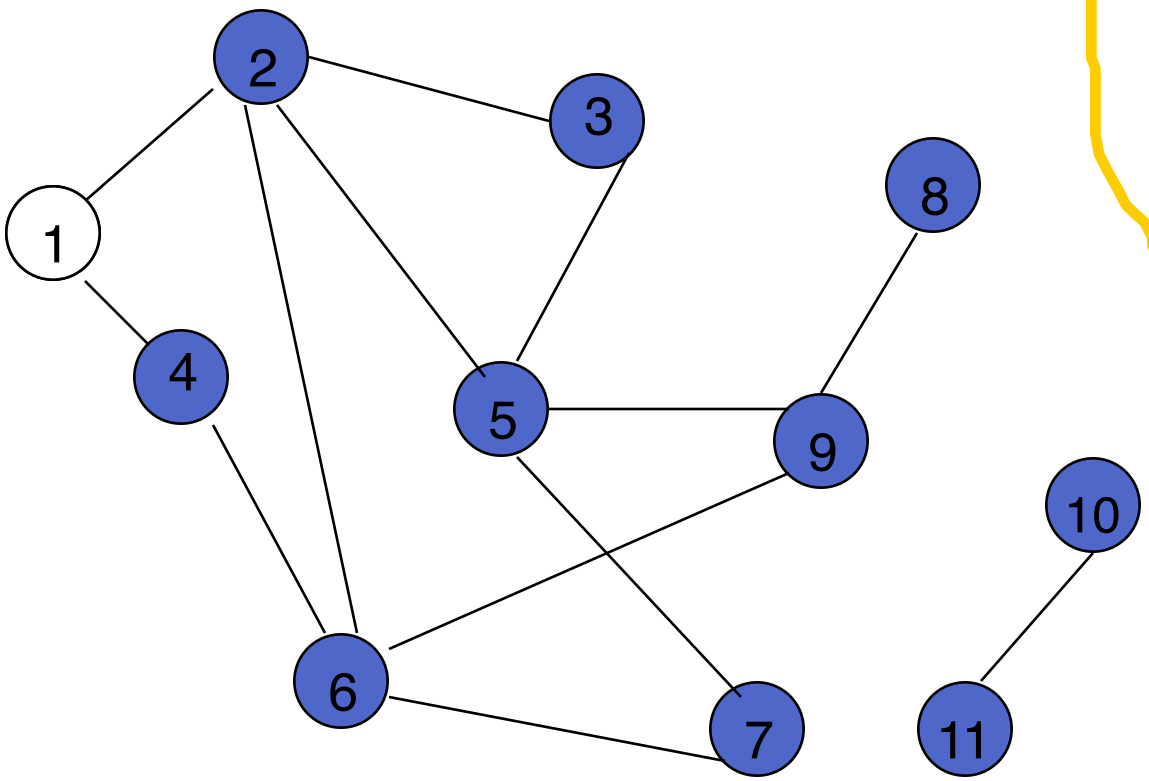
- Find a vertex (or edge) with a given ID or value
- Traverse through the graph to list all vertices in a particular order
  - Finding the item can be side-effect of traversal

Breadth-First Search Example



Start search at vertex 1.

Breadth-First Search Example

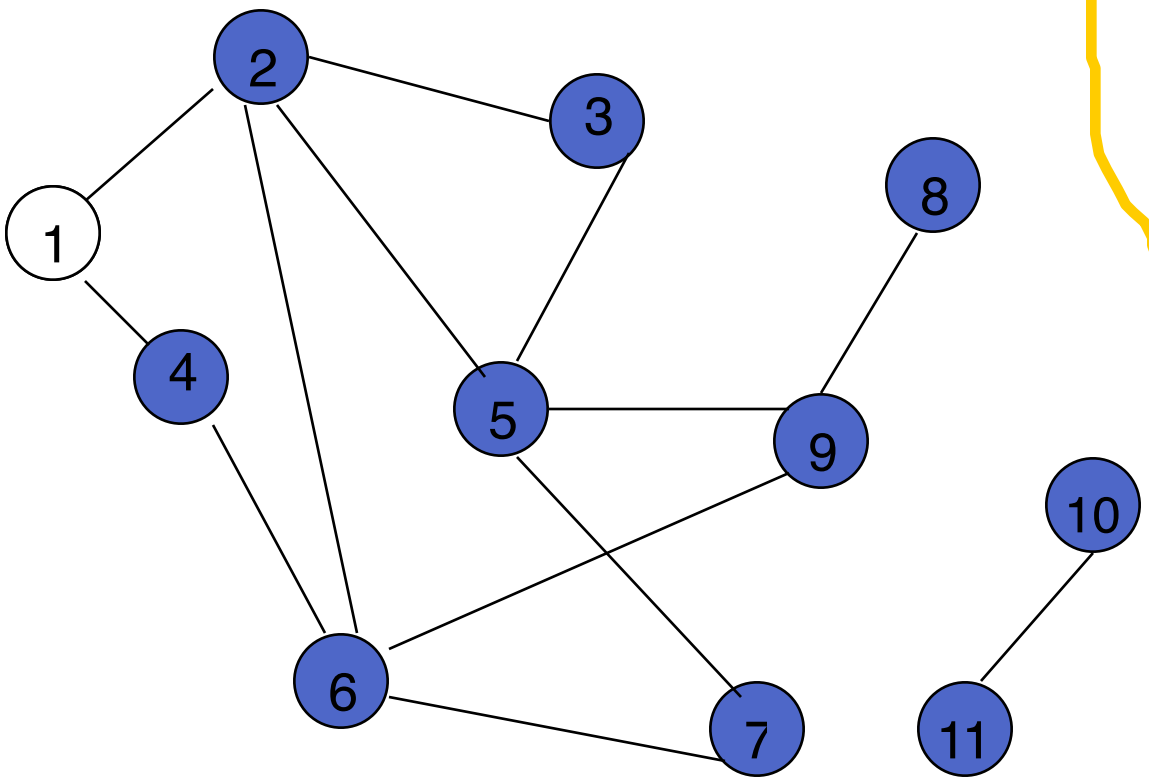


FIFO Queue

1

Visit the start vertex and put in a FIFO queue.

Breadth-First Search Example

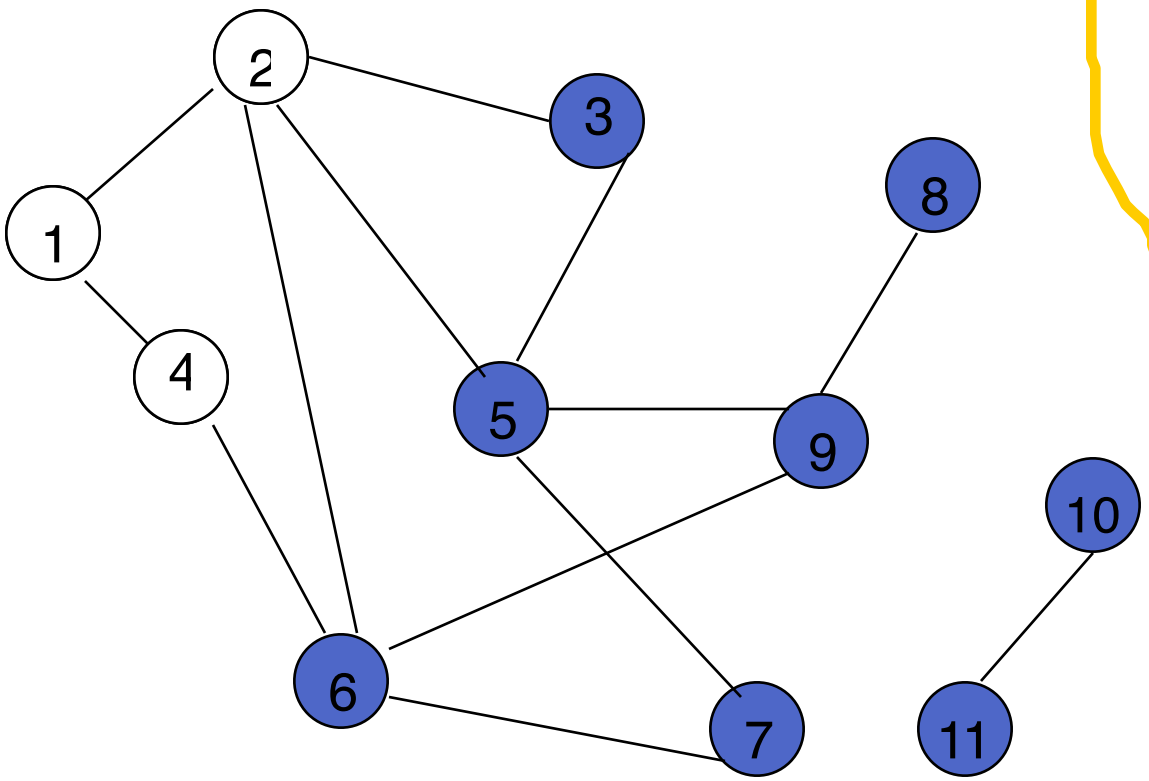


FIFO Queue

1

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

Breadth-First Search Example

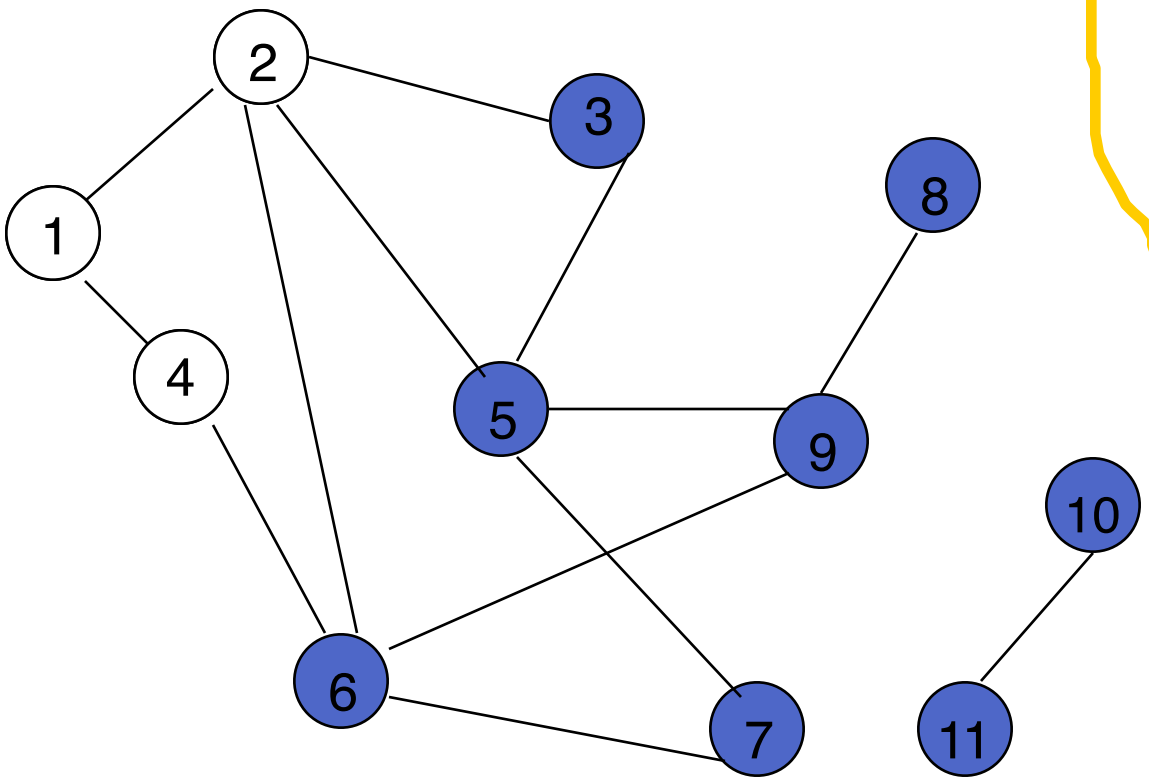


FIFO Queue

2 4

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

Breadth-First Search Example

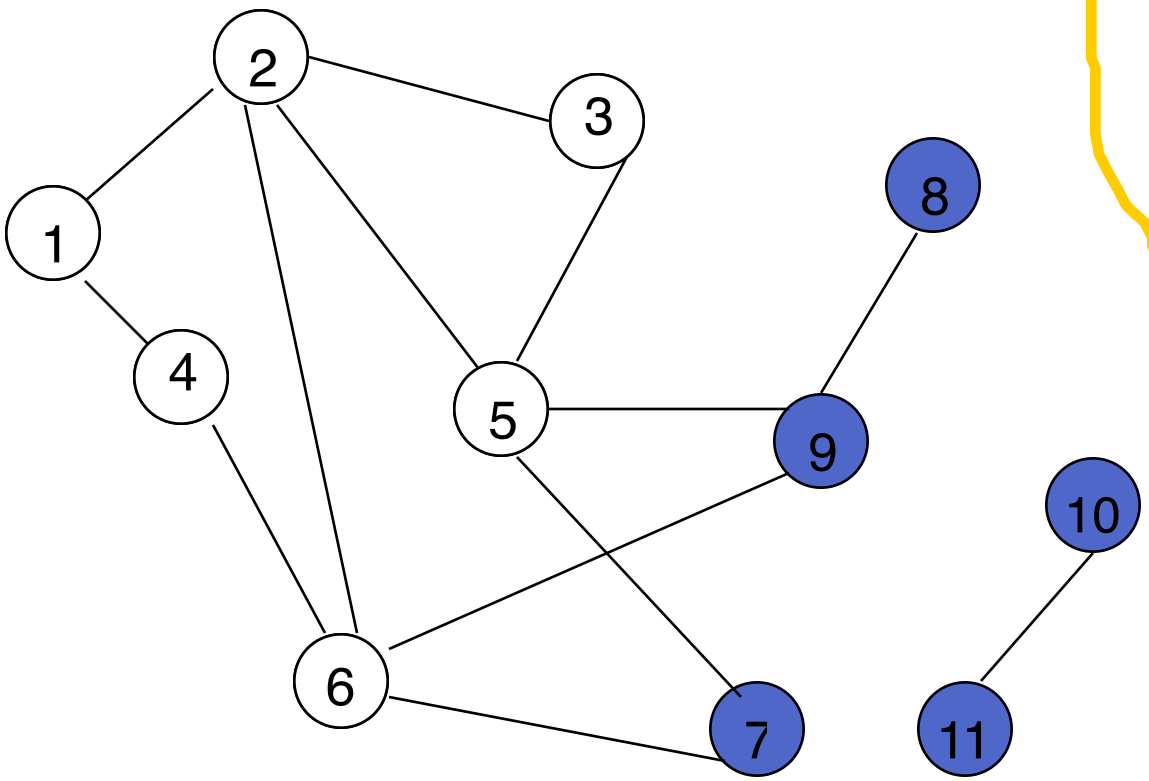


FIFO Queue

2 4

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

Breadth-First Search Example

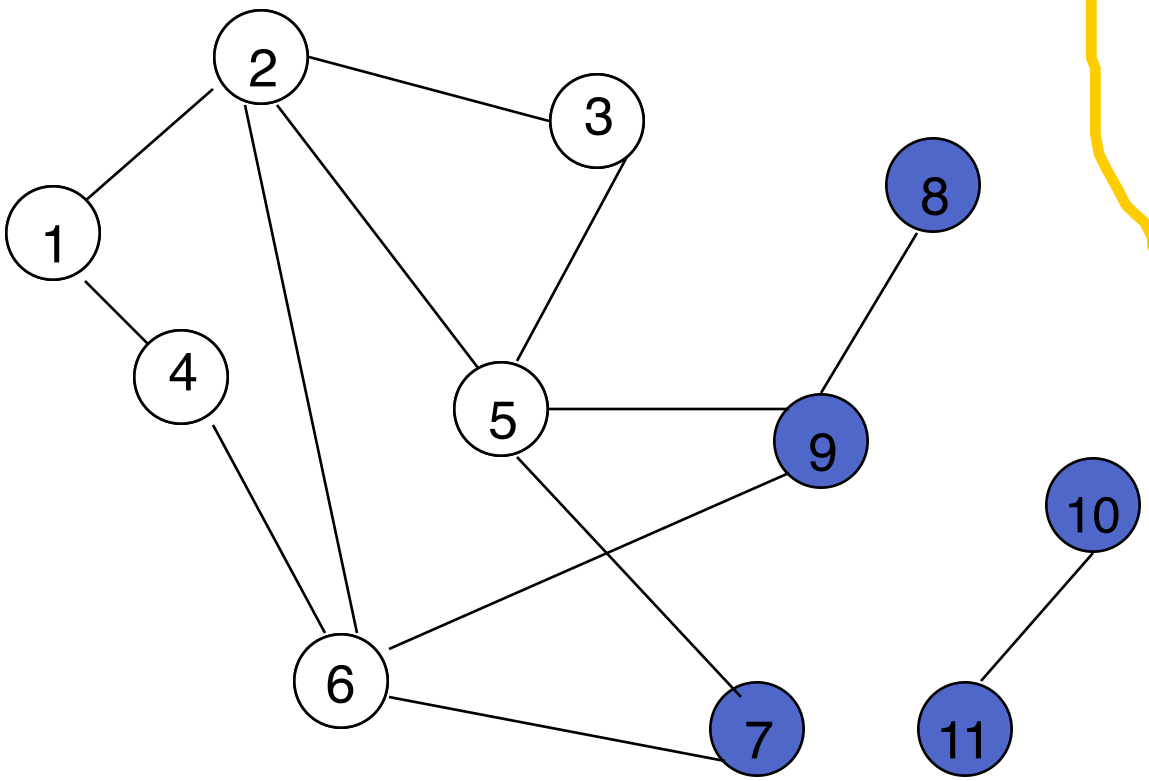


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

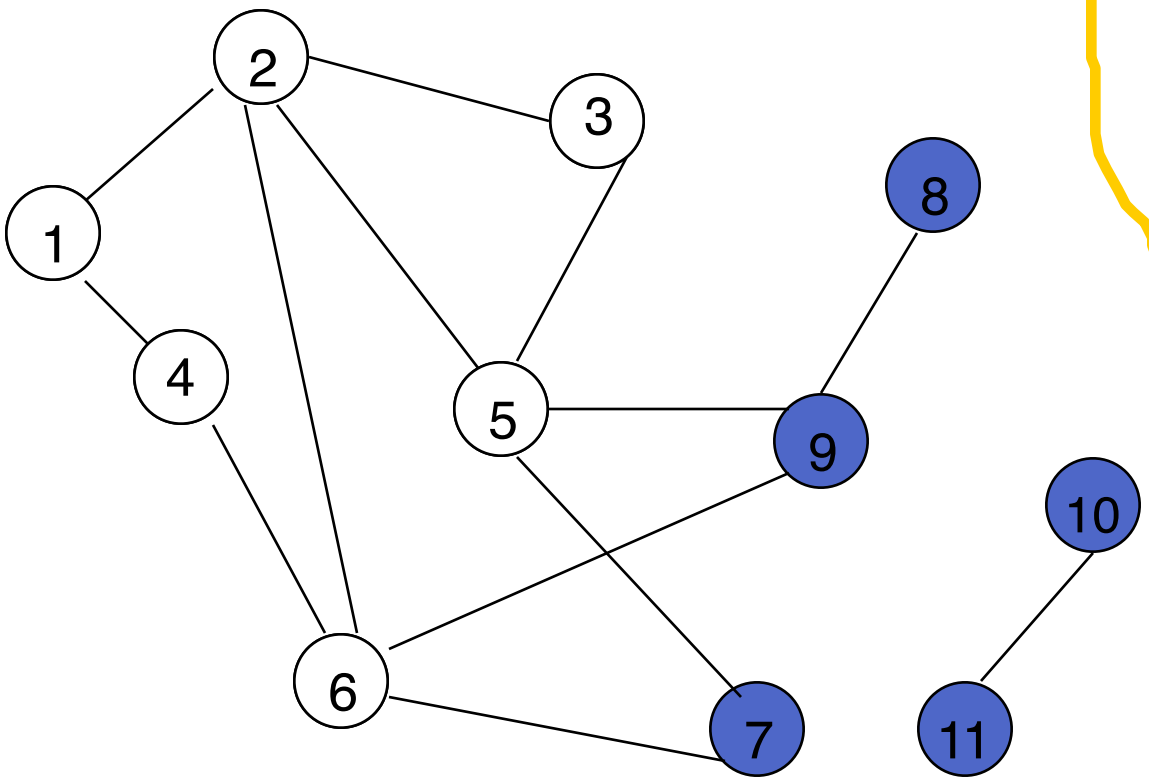


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

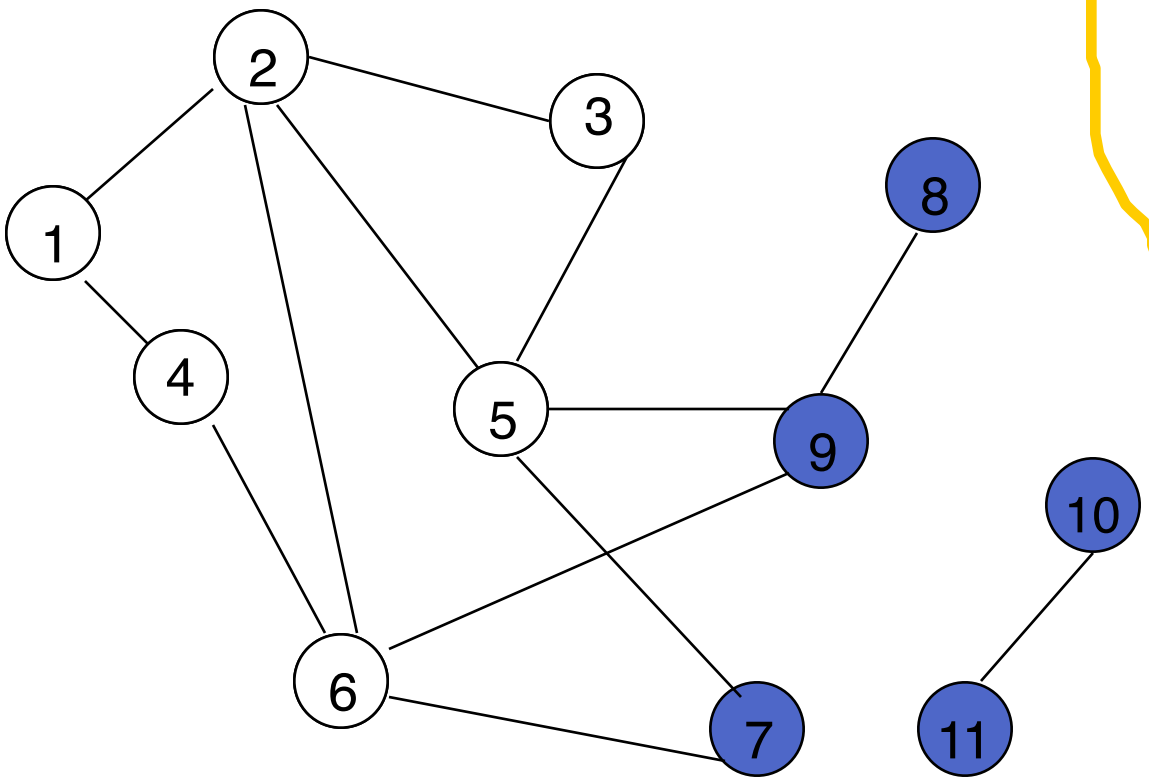


FIFO Queue

5 3 6

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

Breadth-First Search Example

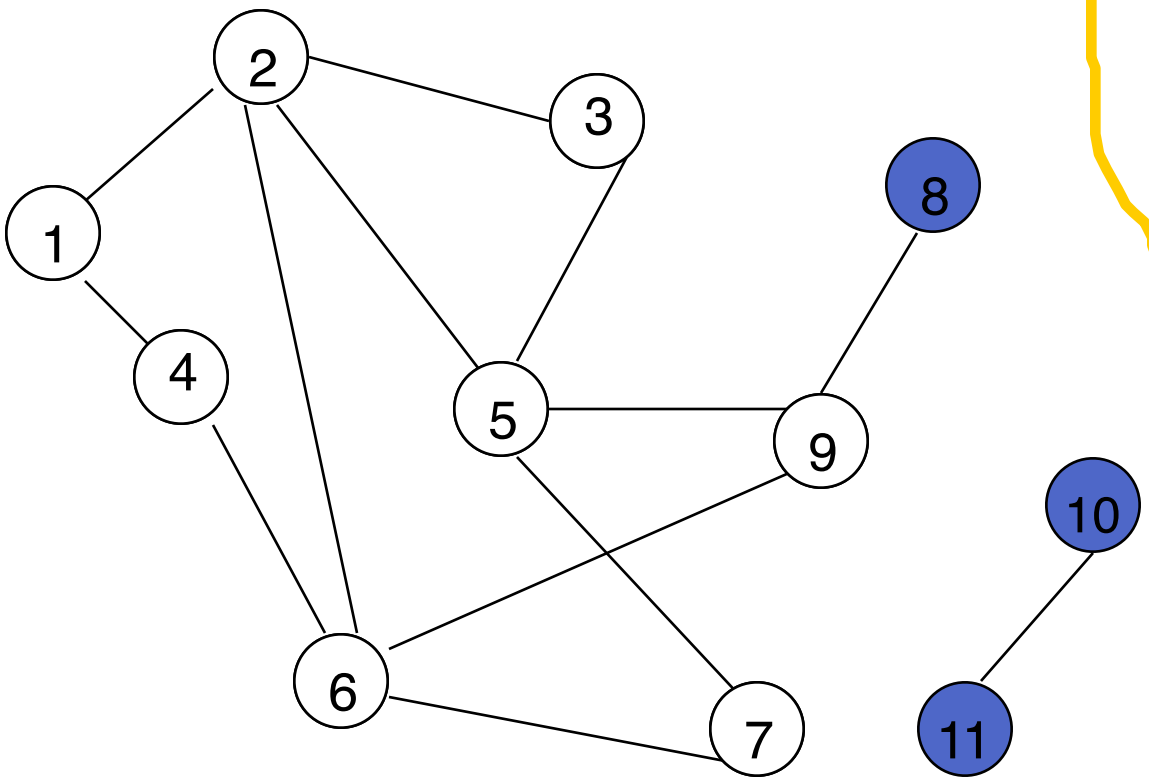


FIFO Queue

5 3 6

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

Breadth-First Search Example

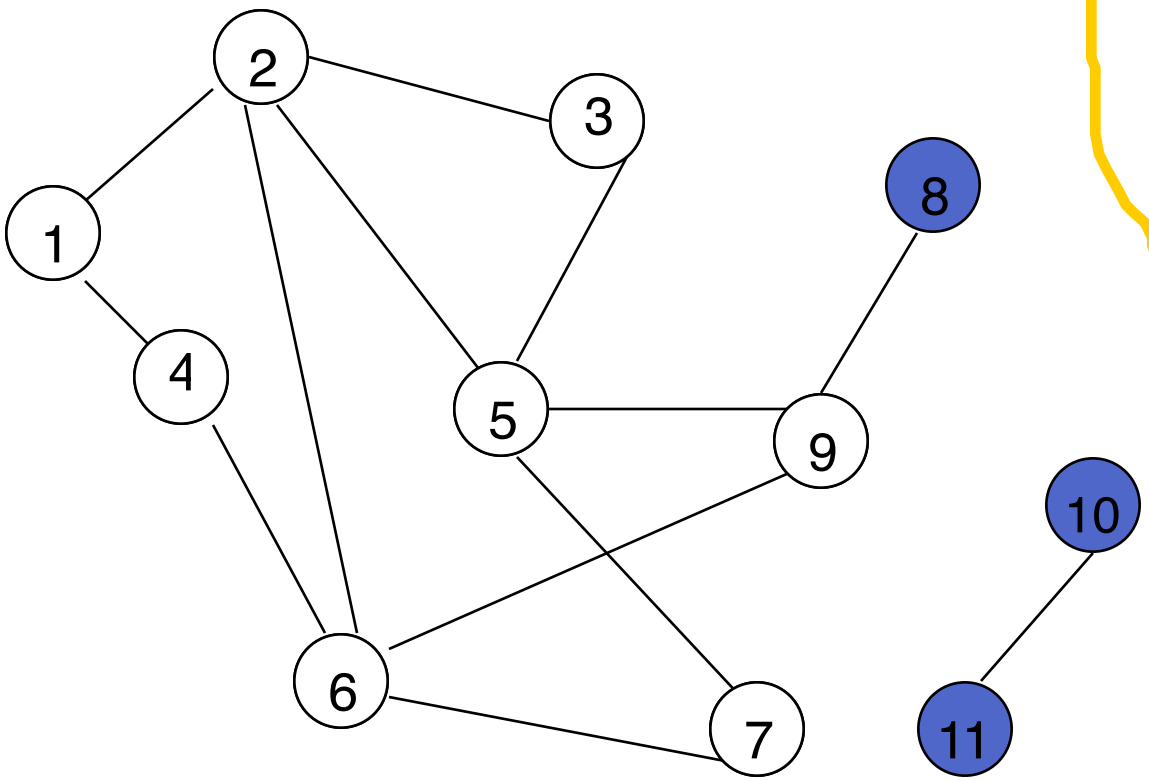


FIFO Queue

3 6 9 7

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

Breadth-First Search Example

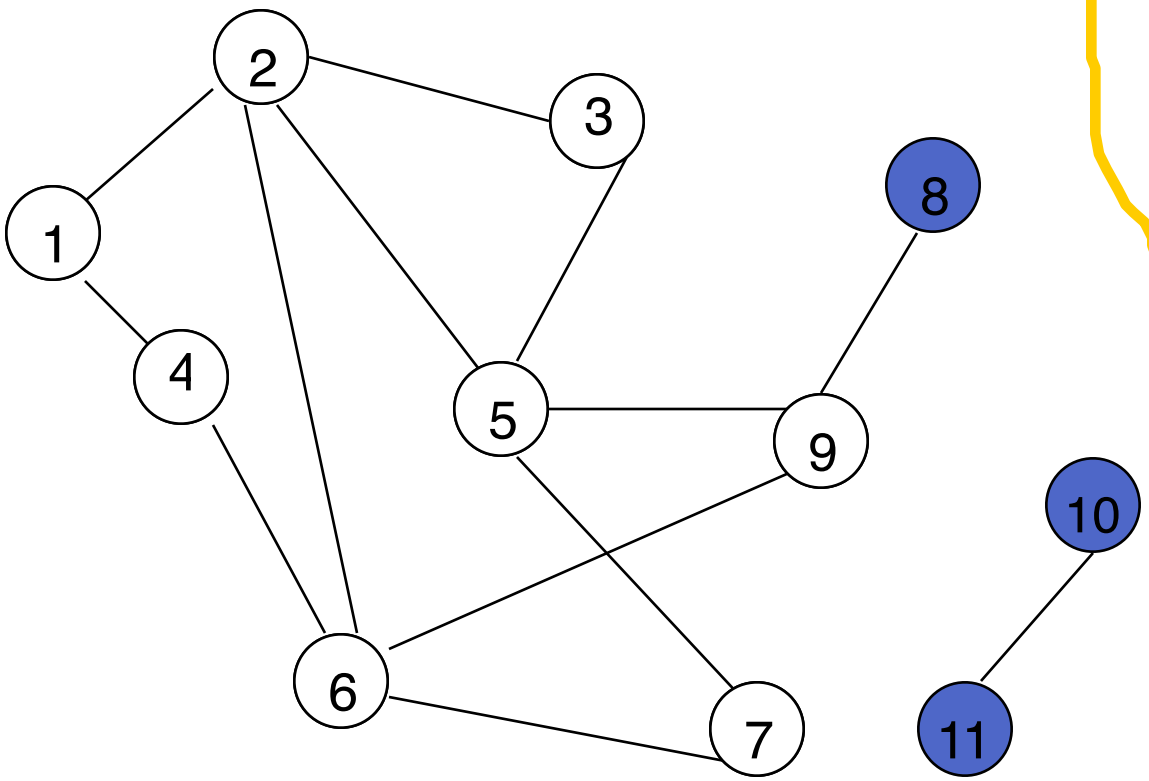


FIFO Queue

3 6 9 7

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

Breadth-First Search Example

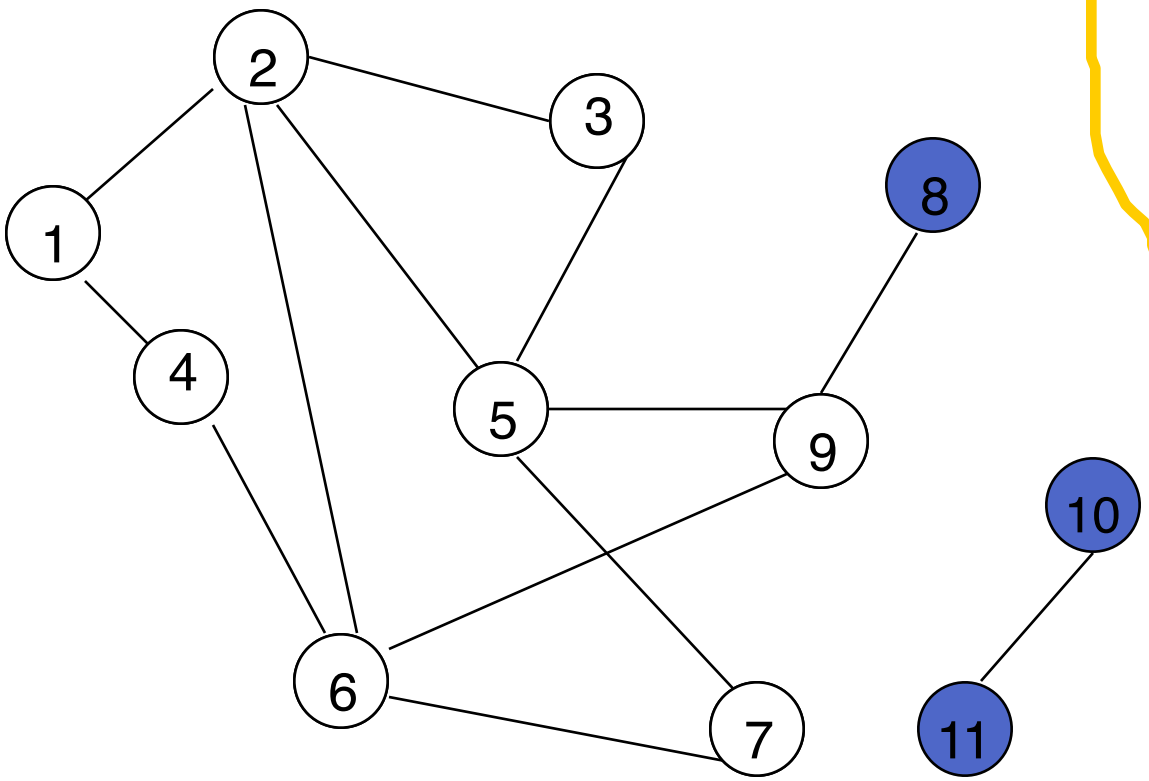


FIFO Queue

6 9 7

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

Breadth-First Search Example

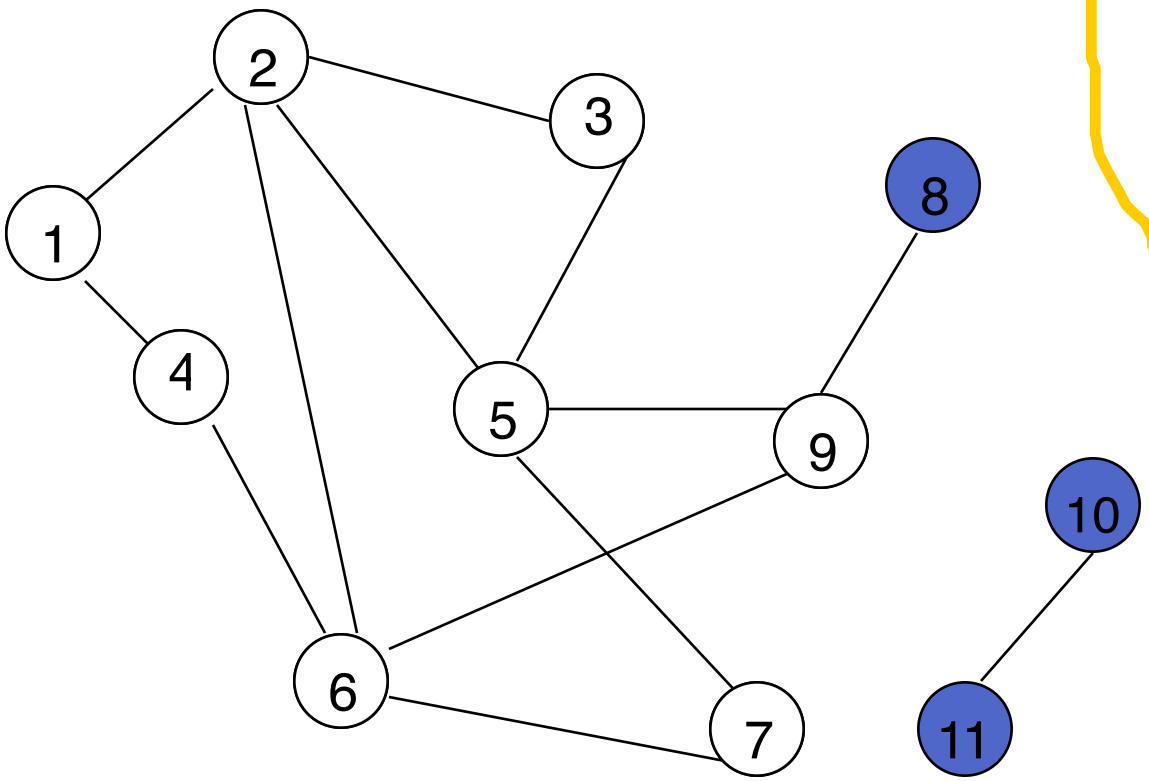


FIFO Queue

6 9 7

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

Breadth-First Search Example

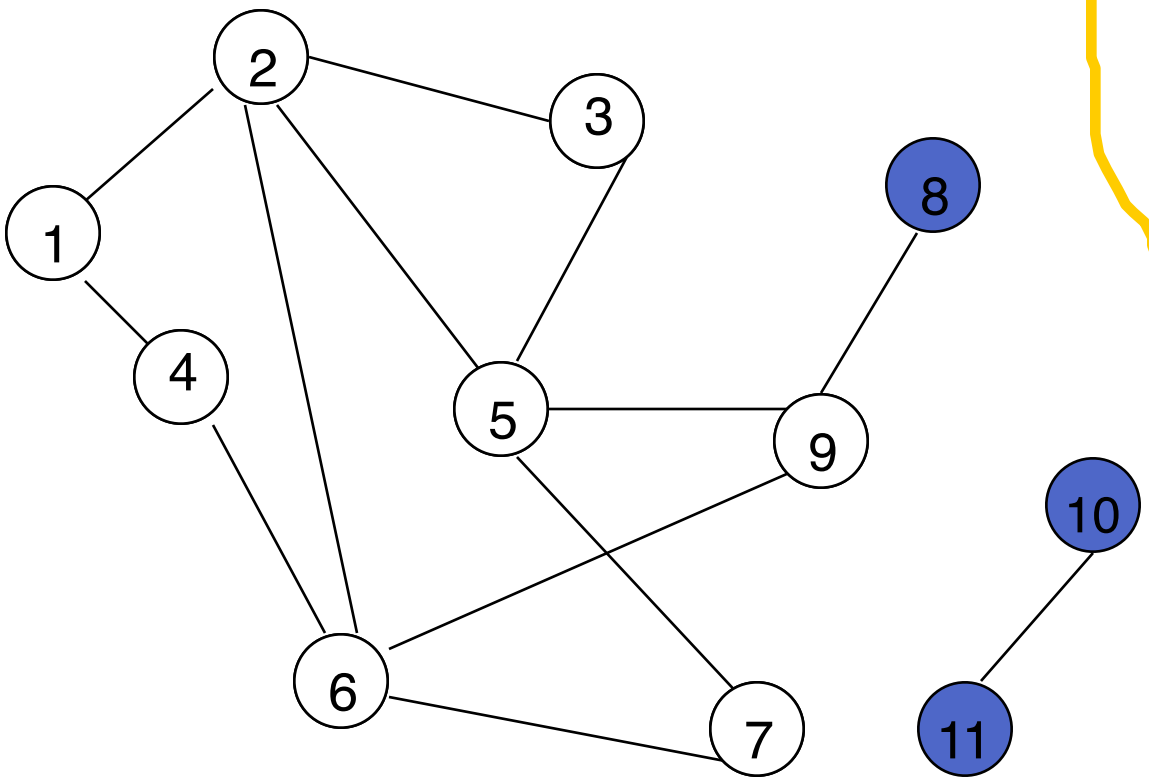


FIFO Queue

9 7

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

Breadth-First Search Example

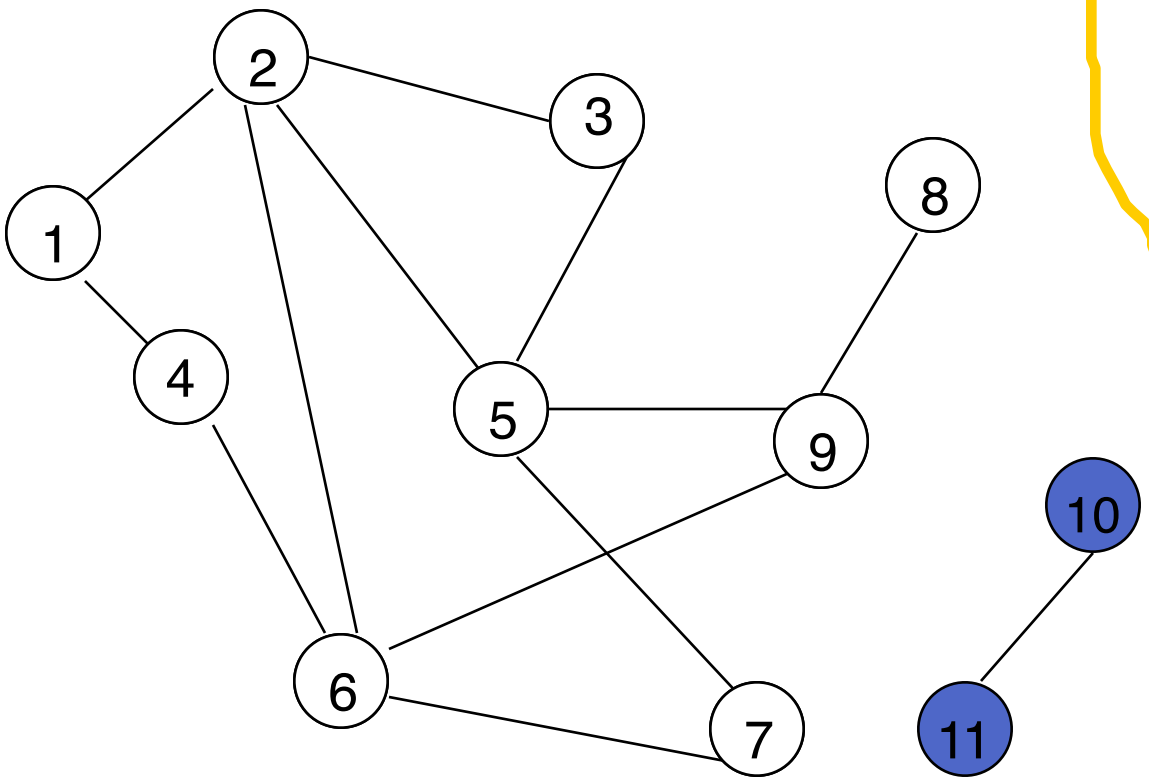


FIFO Queue

9 7

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

Breadth-First Search Example

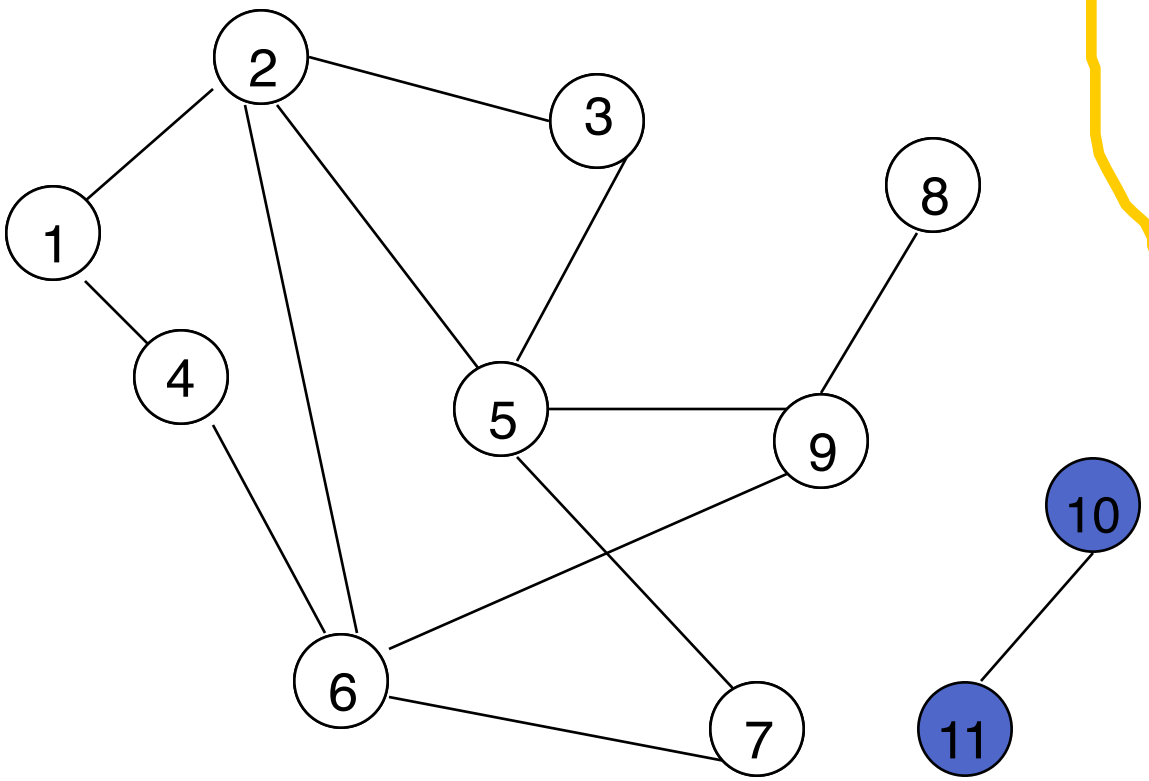


FIFO Queue

7 8

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

Breadth-First Search Example

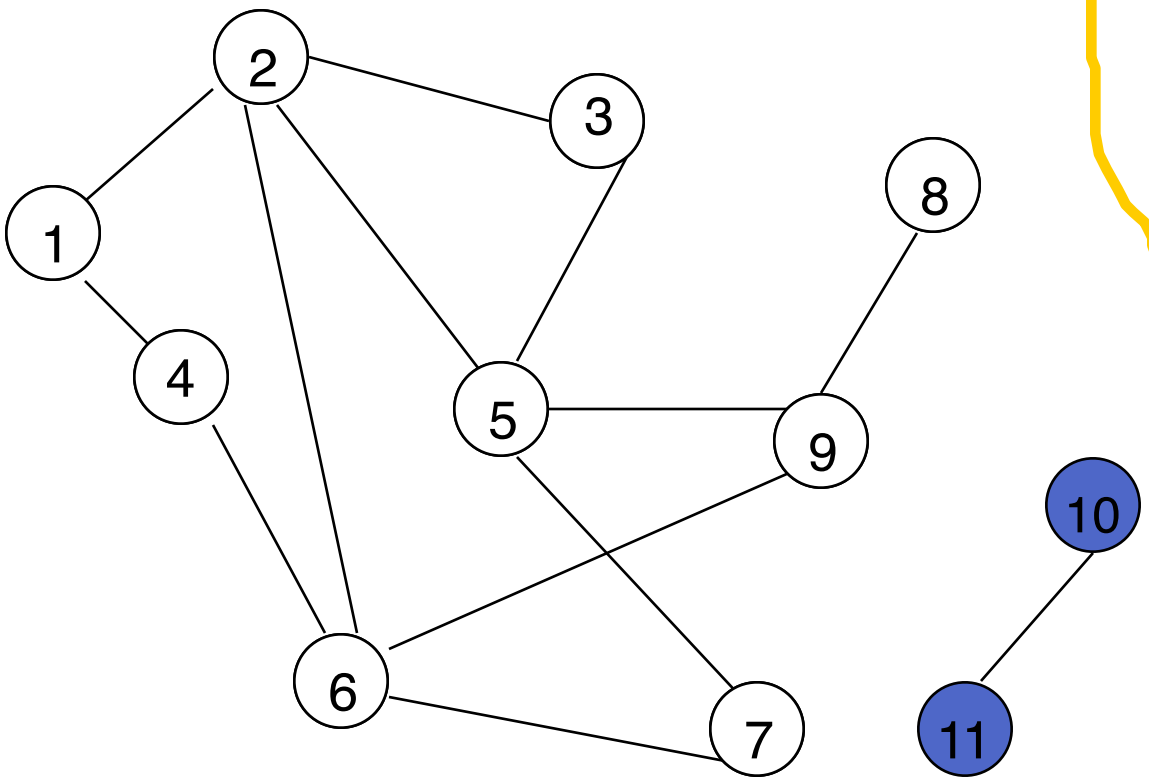


FIFO Queue

7 8

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

Breadth-First Search Example

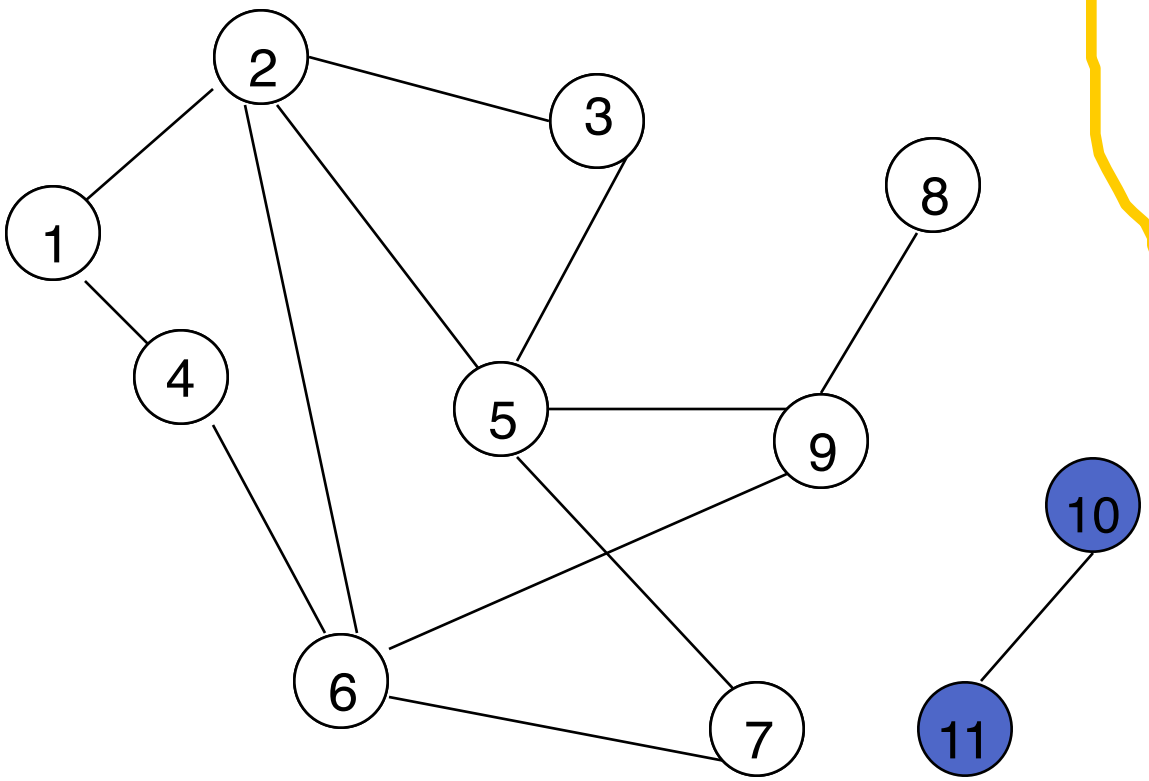


FIFO Queue

8

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

Breadth-First Search Example

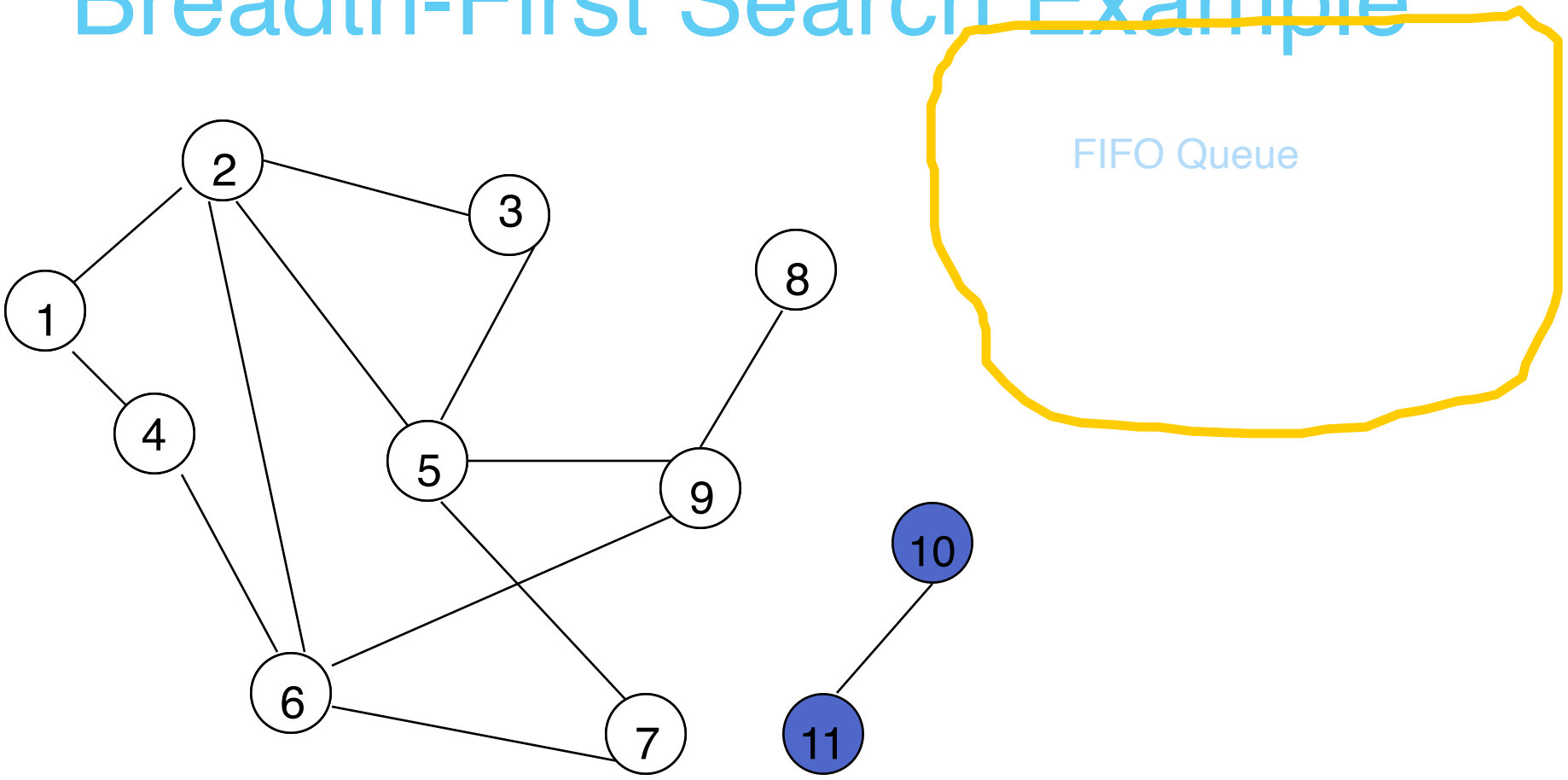


FIFO Queue

8

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.

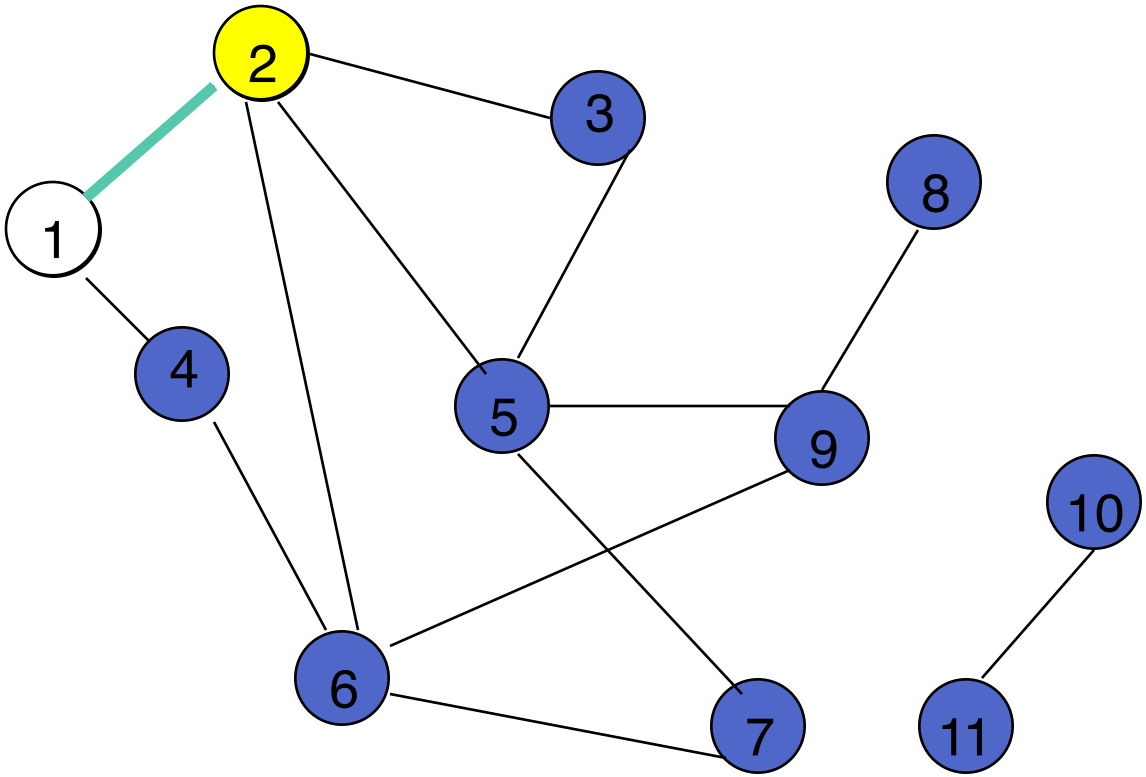
Time Complexity

- 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

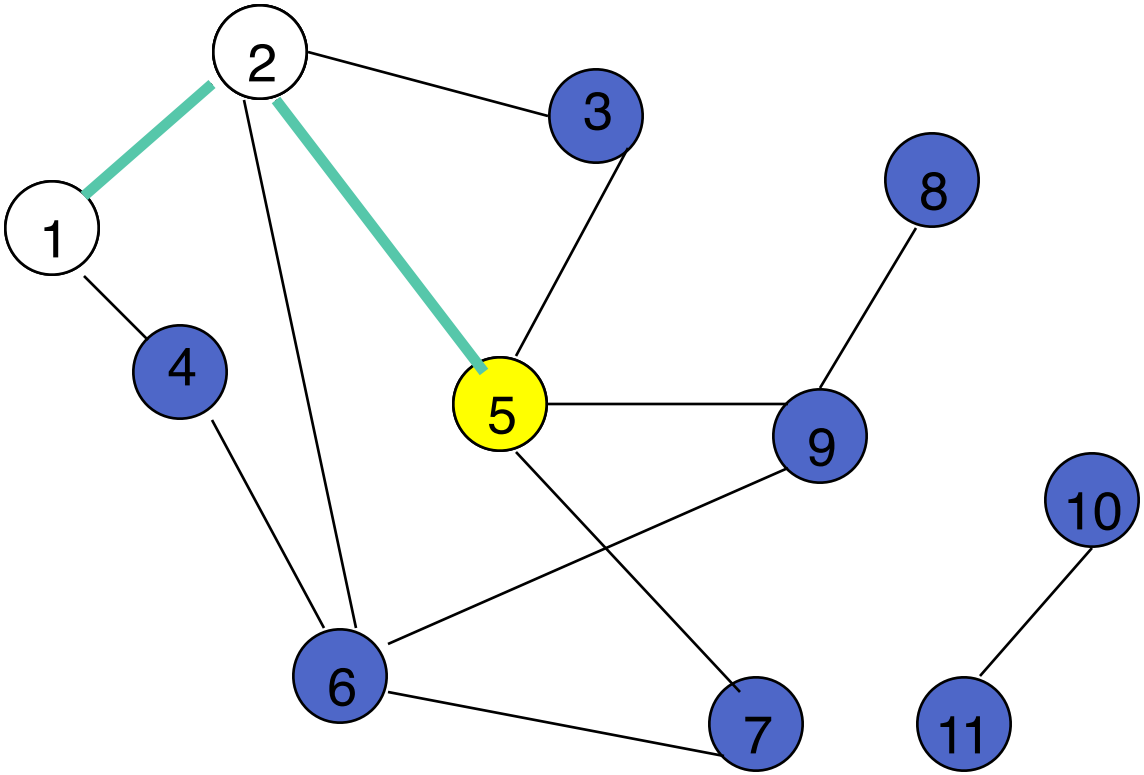
```
depthFirstSearch(v) {
  Label vertex v as reached;
  for(each unreached vertex u adjacent to v)
      depthFirstSearch(u);
}
```

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.

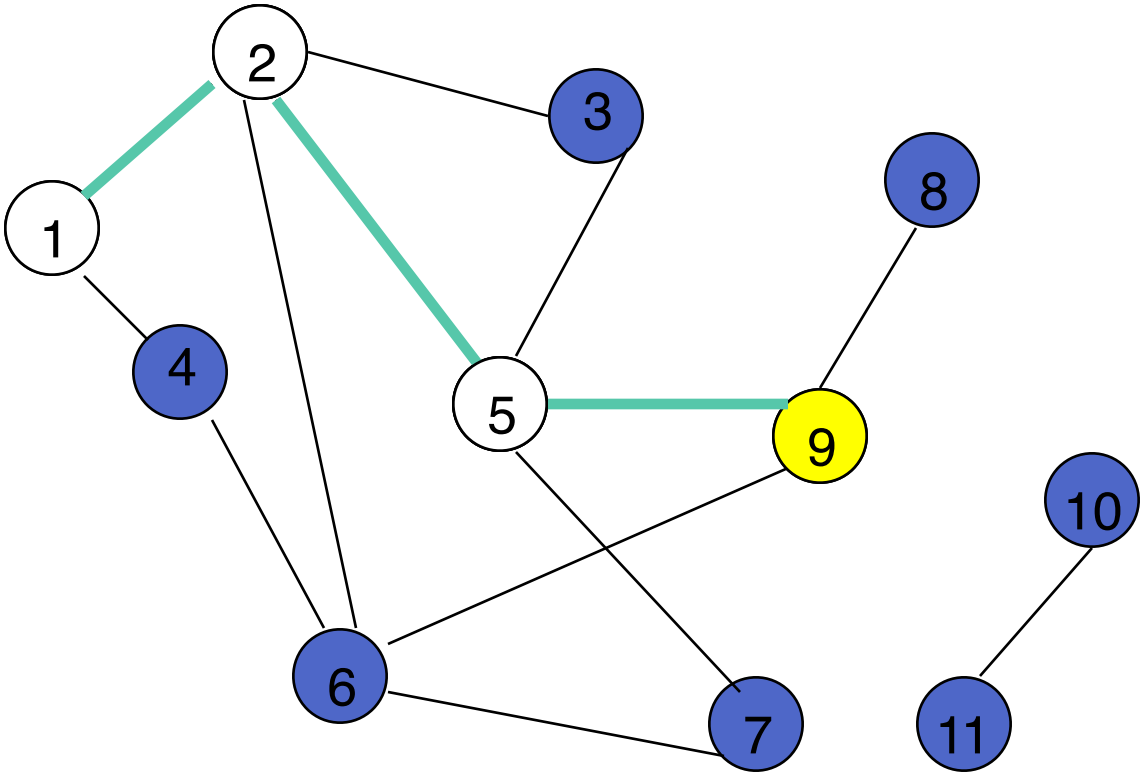
Depth-First Search



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

Suppose that vertex 5 is selected.

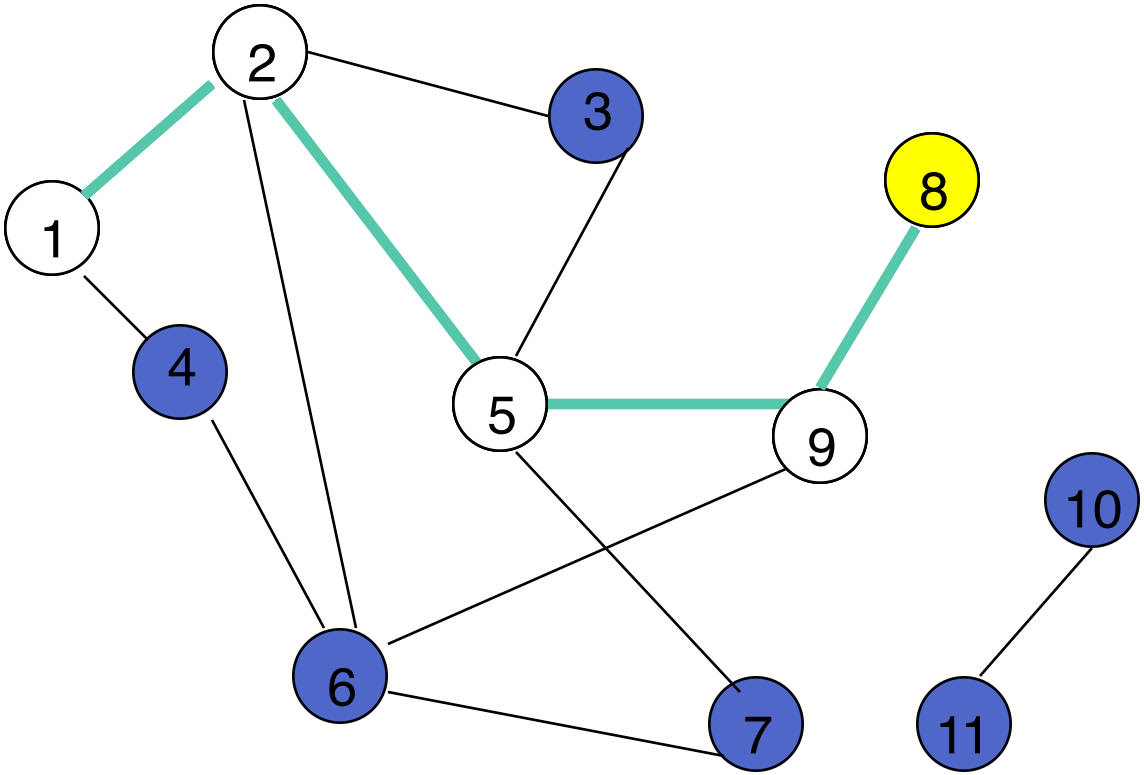
Depth-First Search



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

Suppose that vertex 9 is selected.

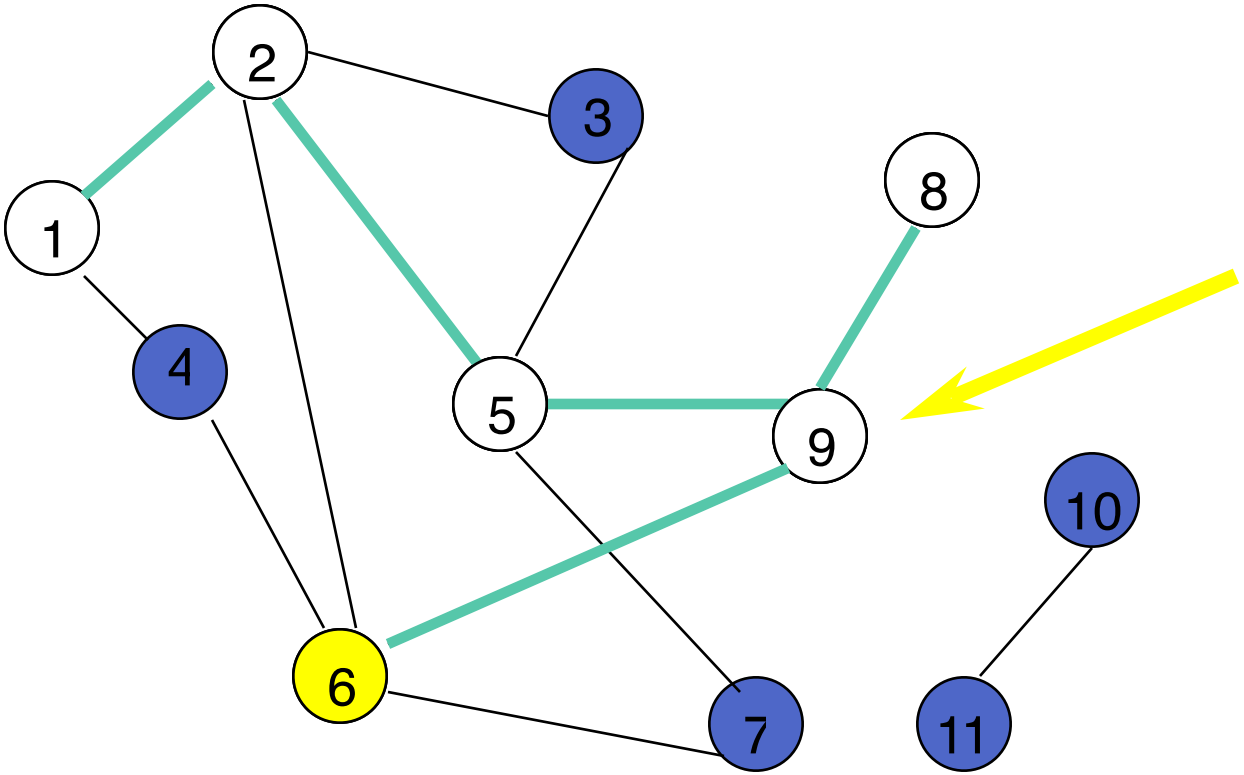
Depth-First Search



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

Suppose that vertex 8 is selected.

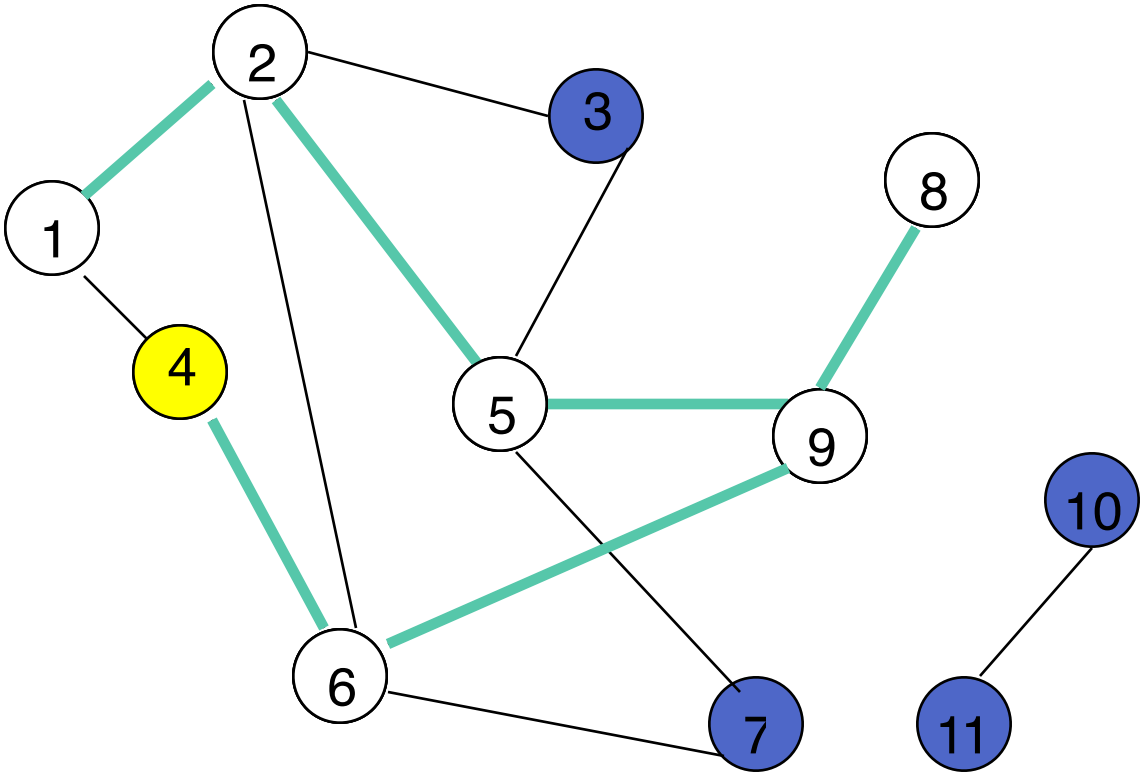
Depth-First Search



Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6)

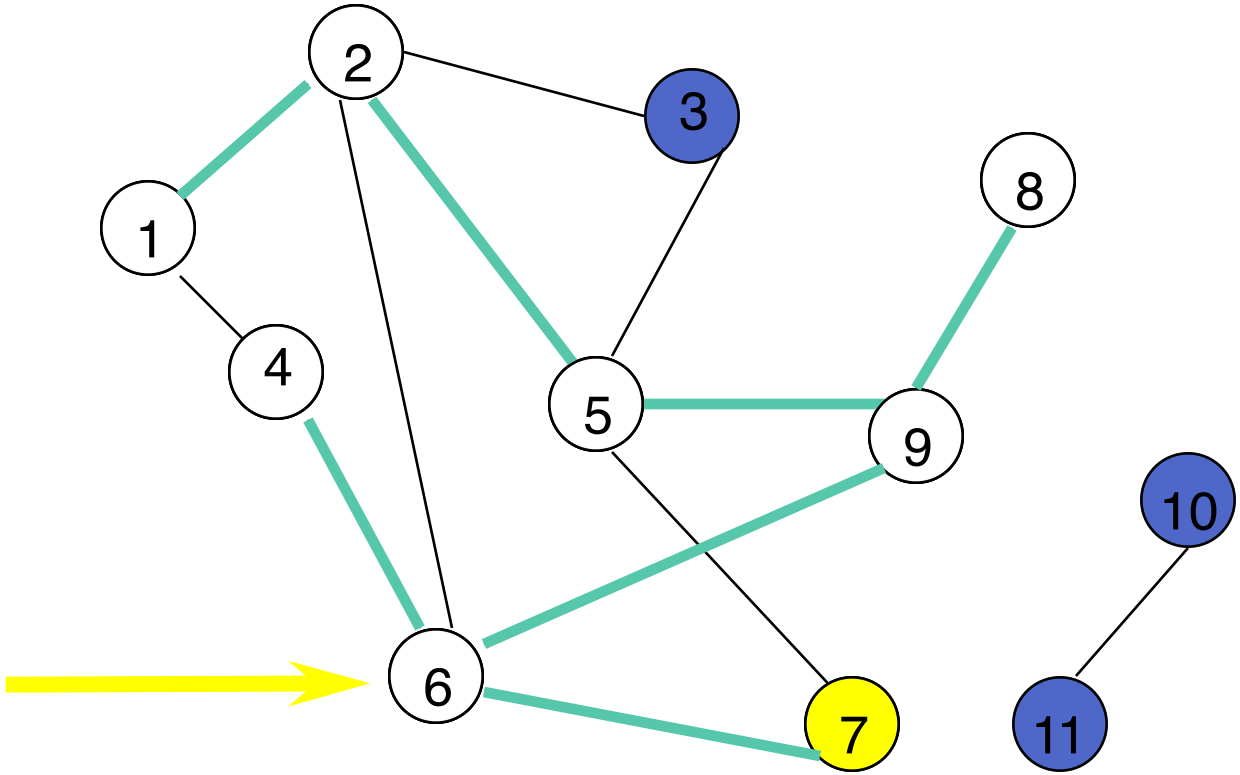
Depth-First Search



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

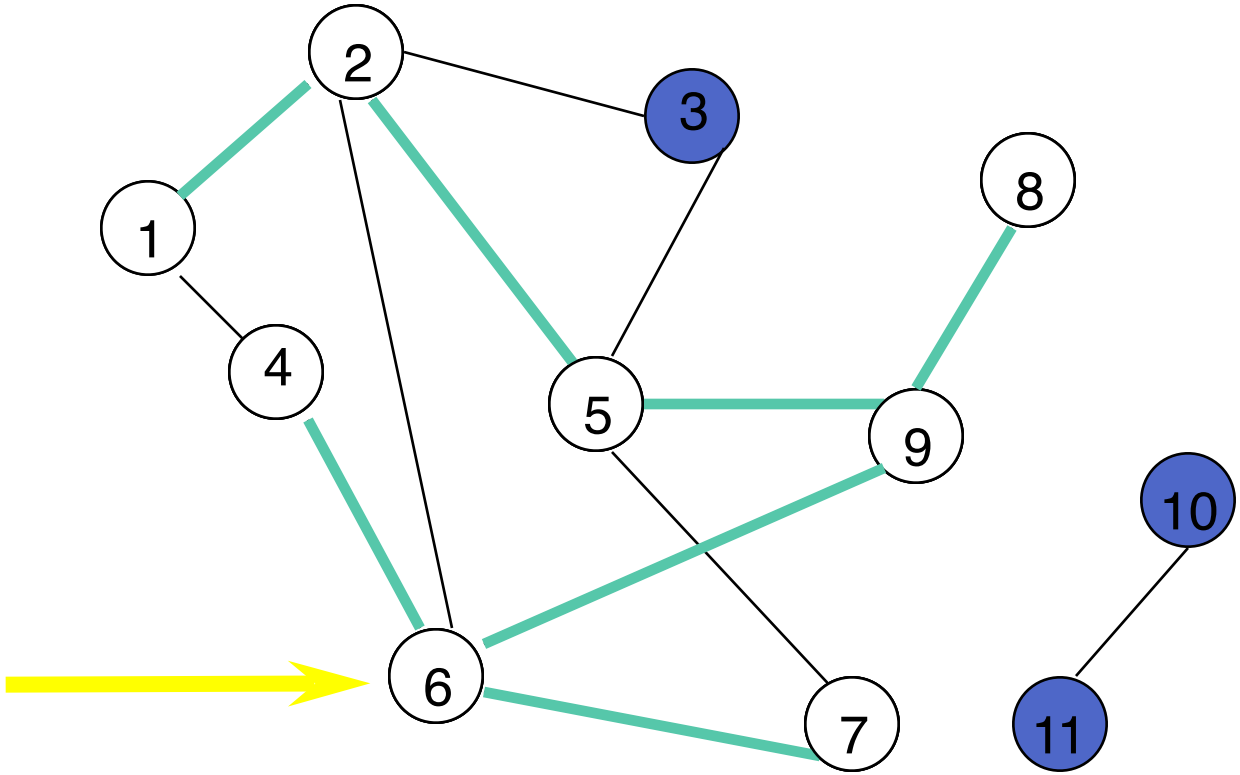
Suppose that vertex 4 is selected.

Depth-First Search



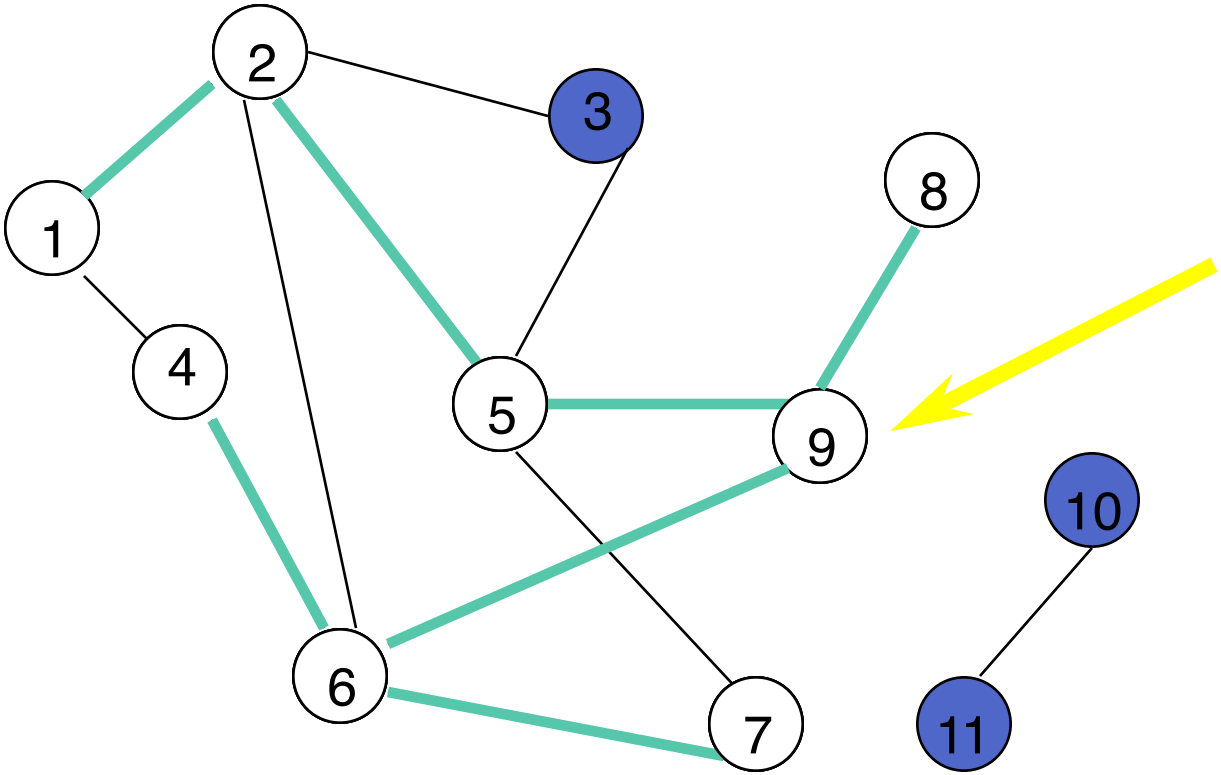
Label vertex 4 and return to 6.
From vertex 6 do a dfs(7).

Depth-First Search



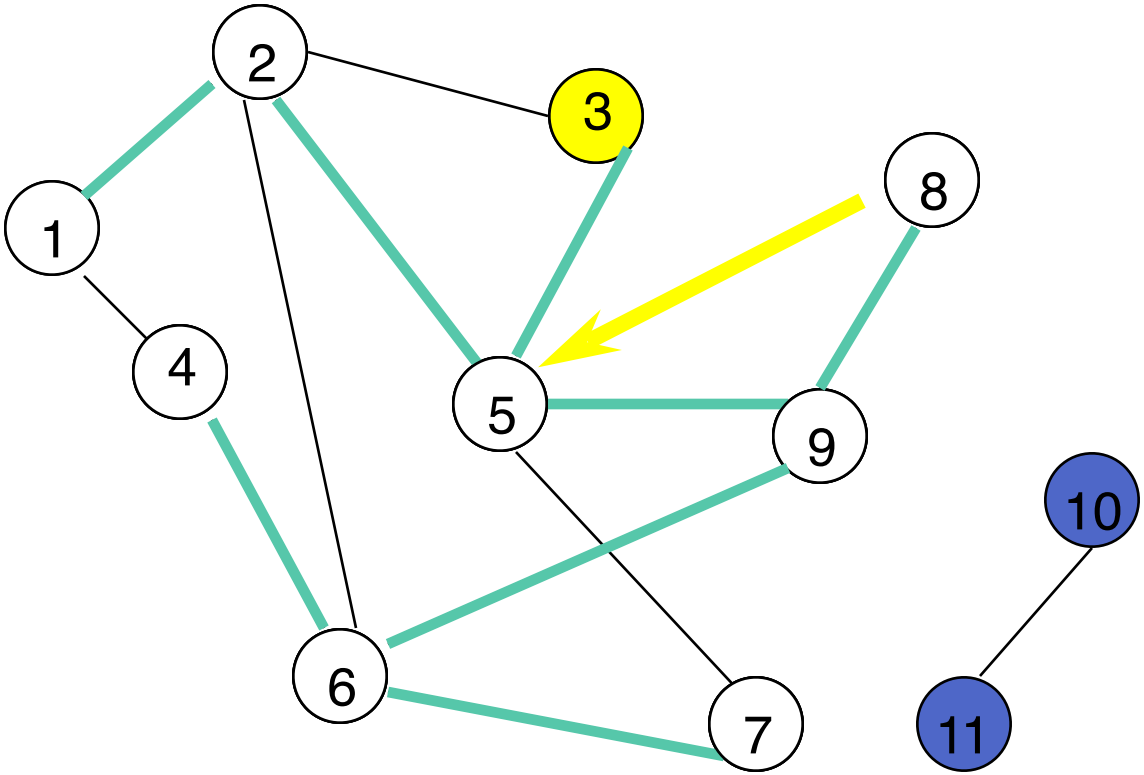
Label vertex 7 and return to 6.
Return to 9.

Depth-First Search



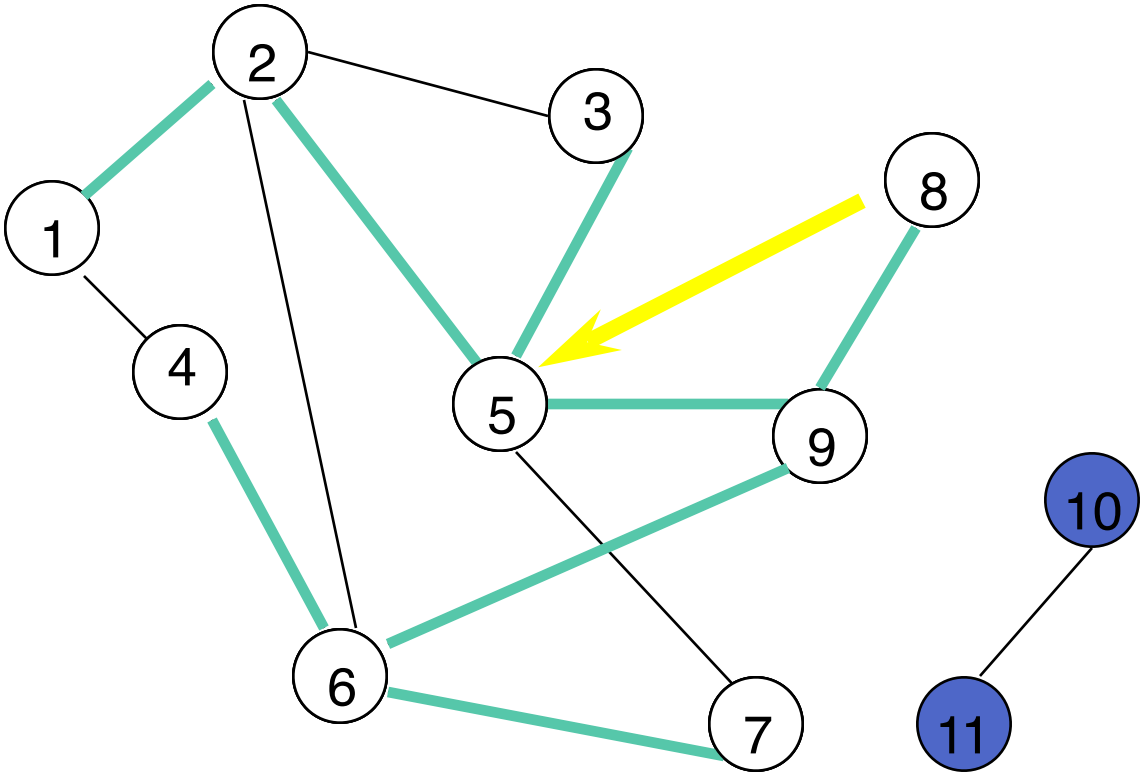
Return to 5.

Depth-First Search



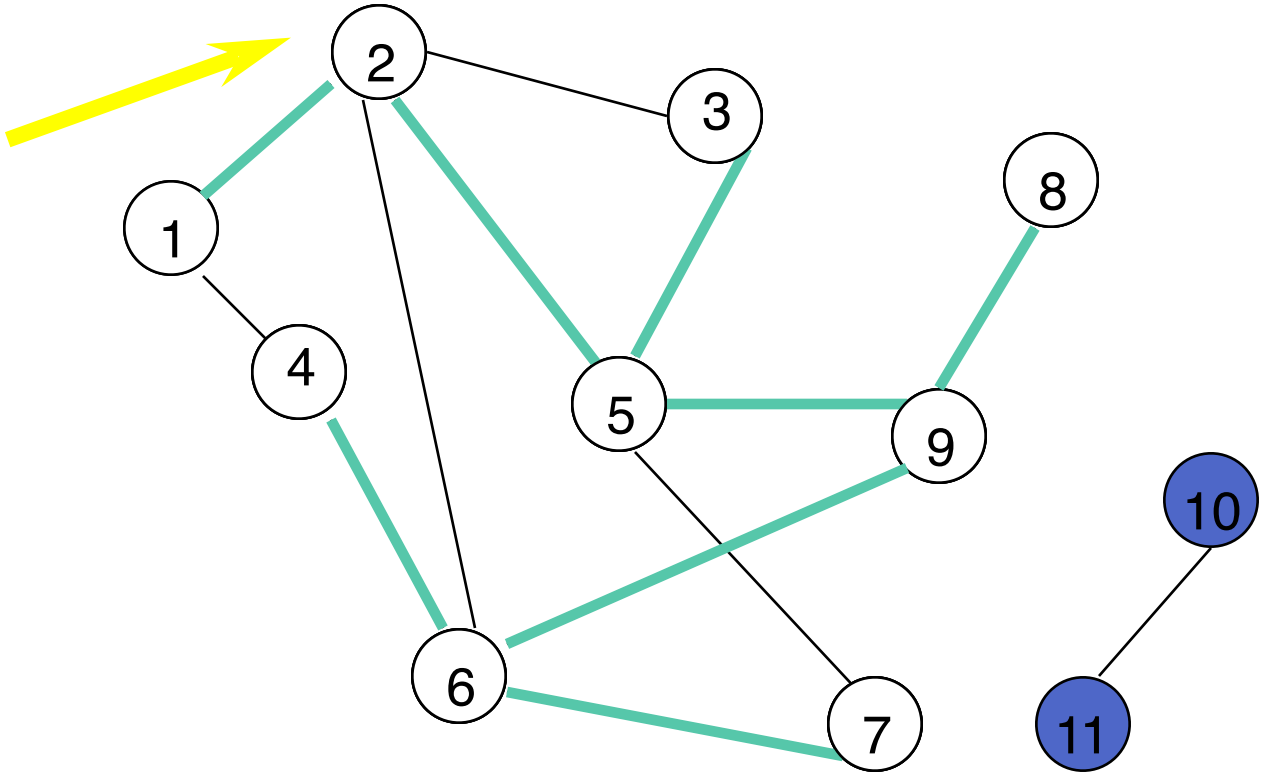
Do a dfs(3).

Depth-First Search



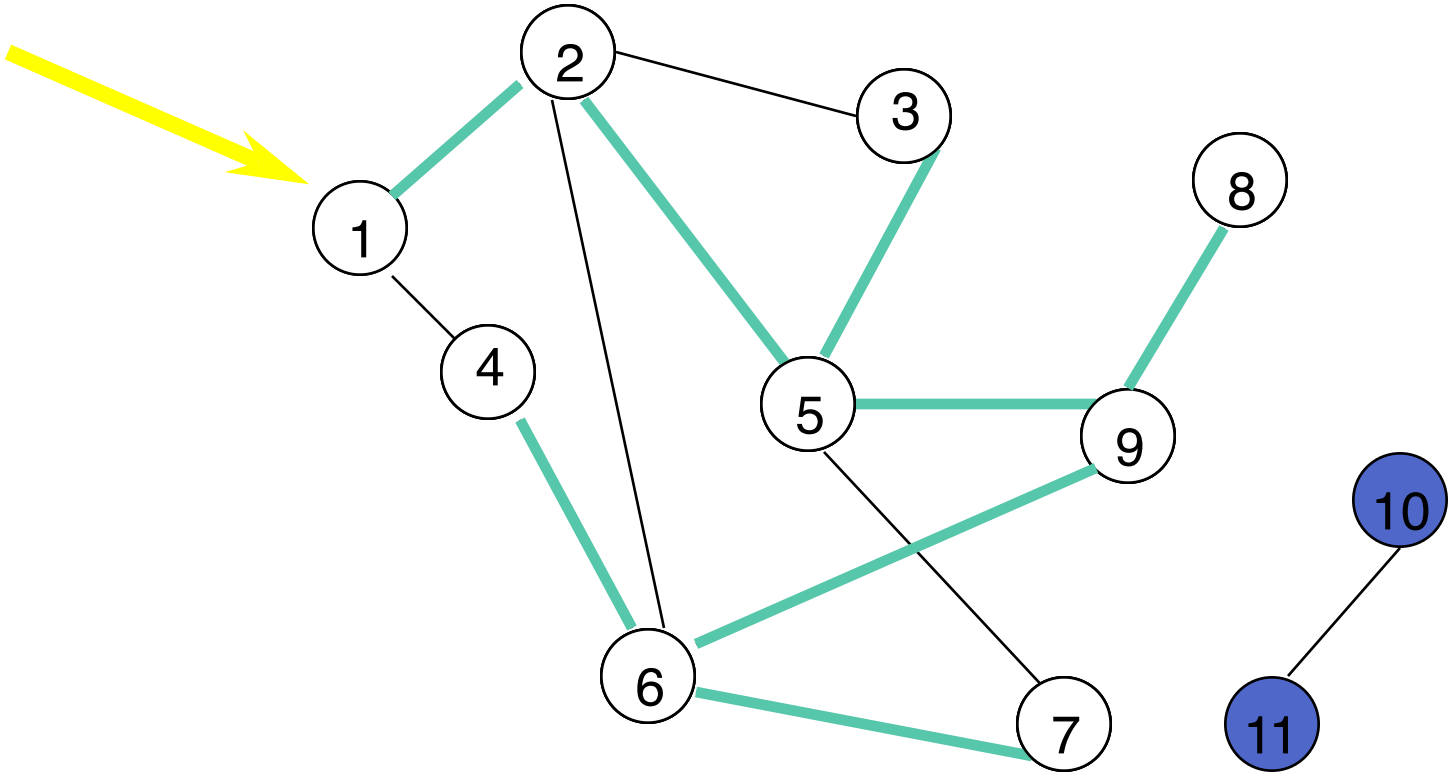
Label 3 and return to 5.
Return to 2.

Depth-First Search



Return to 1.

Depth-First Search



Return to invoking method.

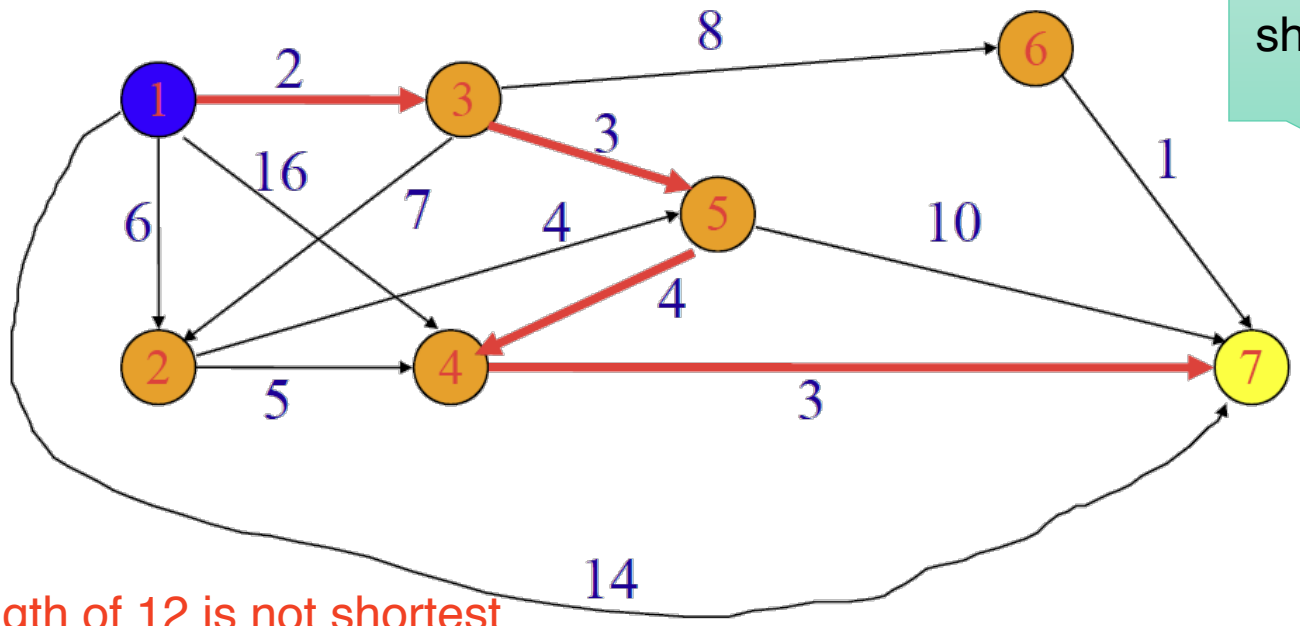
DFS Properties

- DFS has same time complexity as BFS
- DFS requires O(v) memory for recursive function stack while BFS requires O(v) queue capacity, v=#vertices
- Same properties with respect to path finding, connected components, and spanning trees.
  - Edges used to reach unlabelled 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

Greedy Path from 1 To 7



DFS with shortest edge selected!

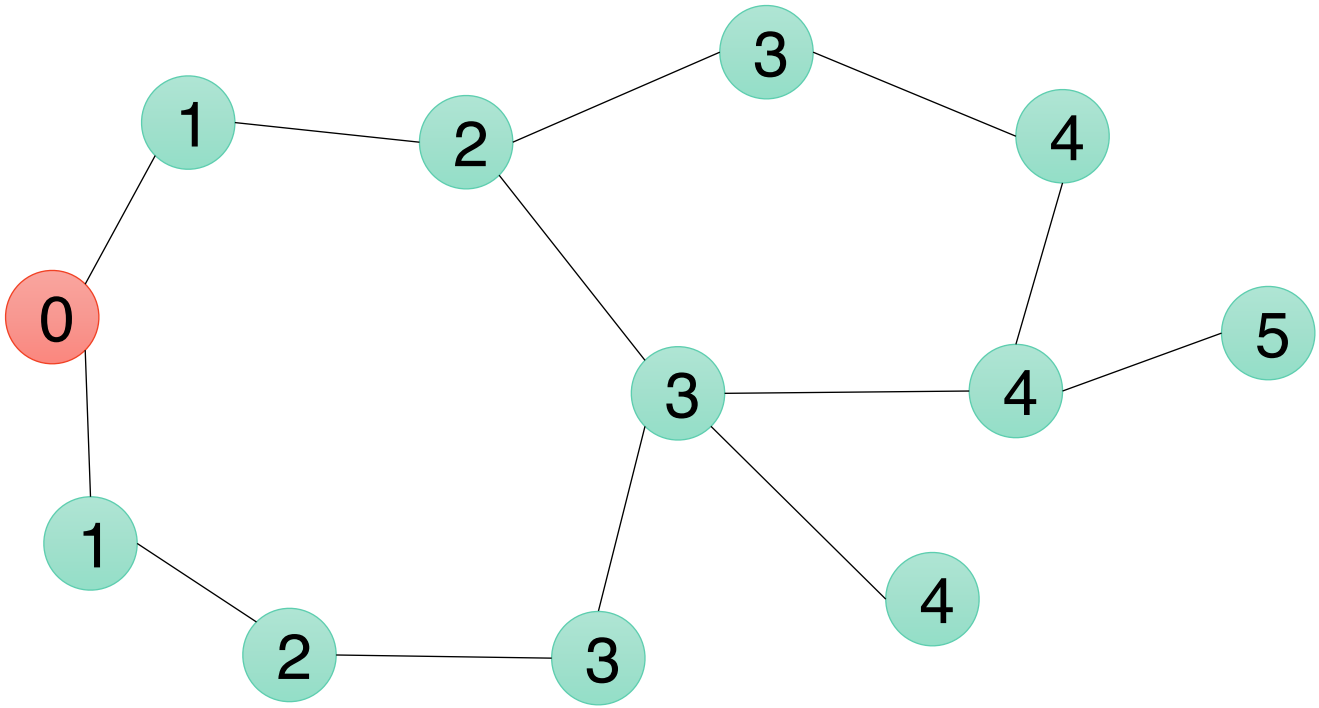
Length of 12 is not shortest path!

Single Source Shortest Path

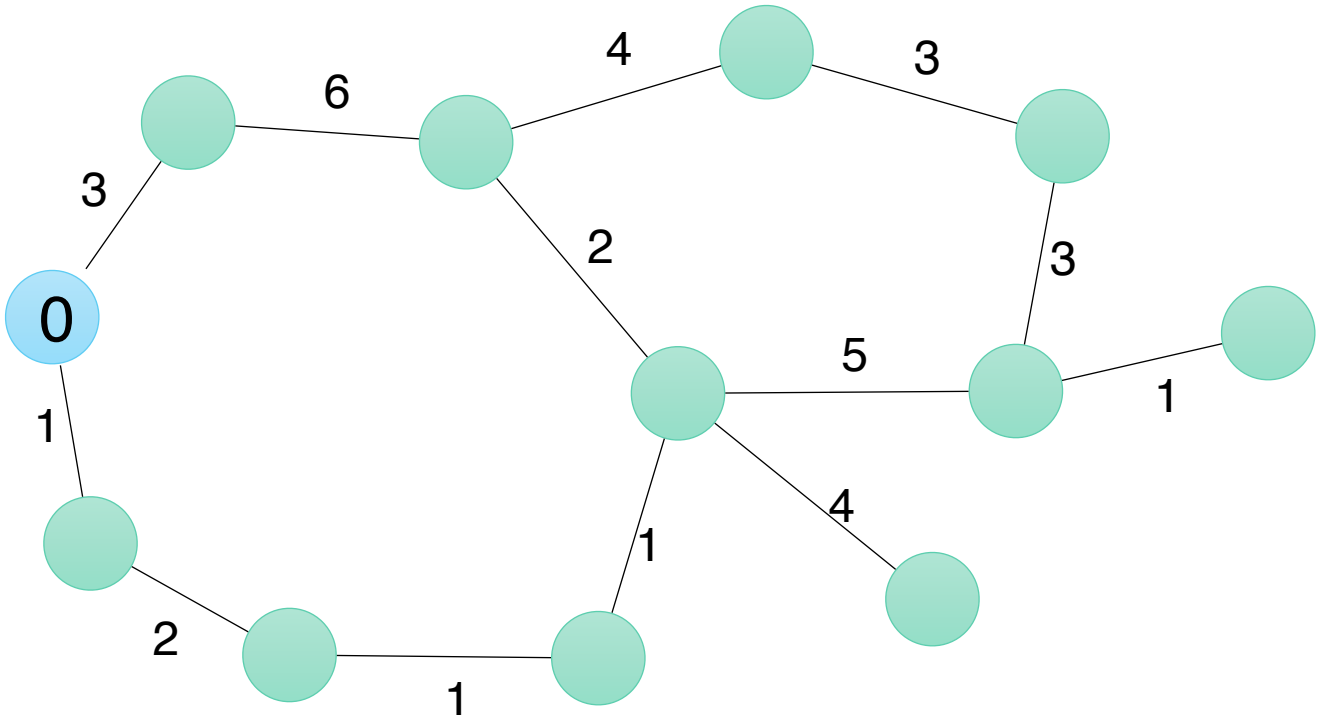
- Shortest distance from one source vertex to all destination vertices
- Is there a simple way to solve this?
- …Say if you had a unit-weighted graph?

- Just do Breadth First Search (BFS)! ☺

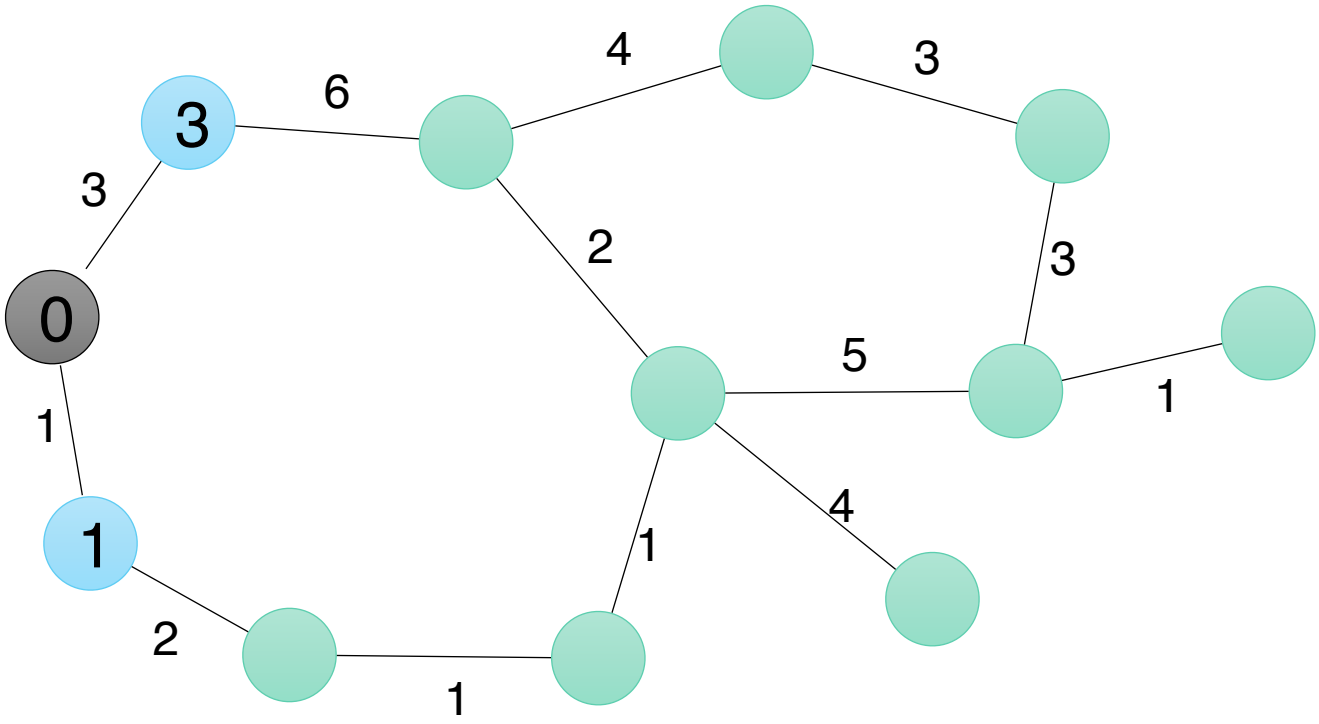
SSSP: BFS on Unweighted Graphs



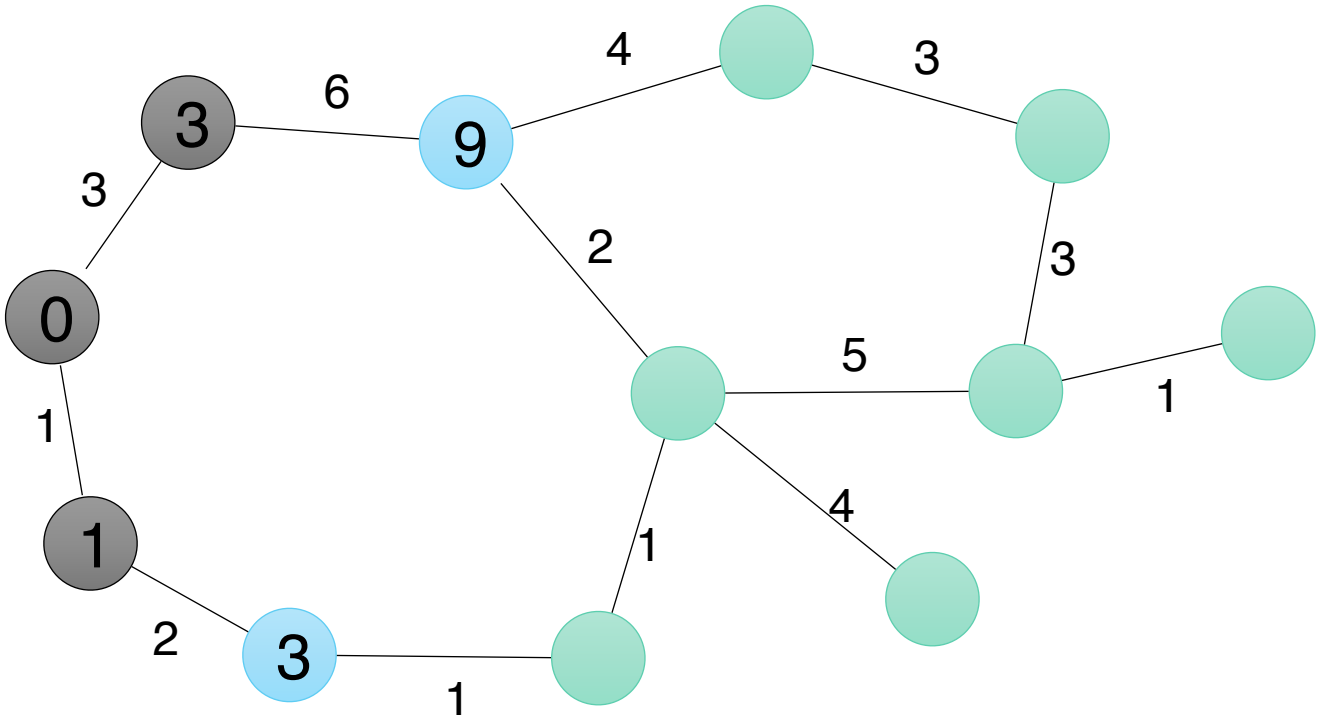
SSSP: BFS on Weighted Graphs?



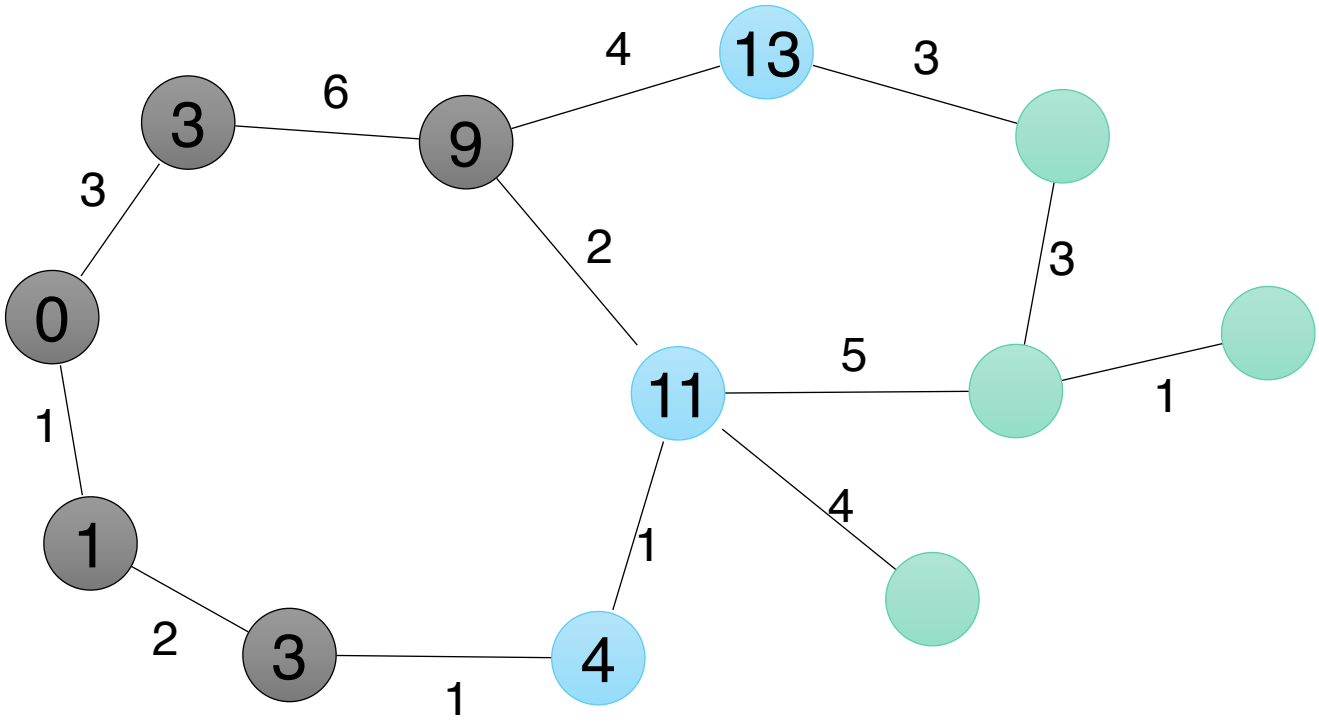
SSSP: BFS on Weighted Graphs?



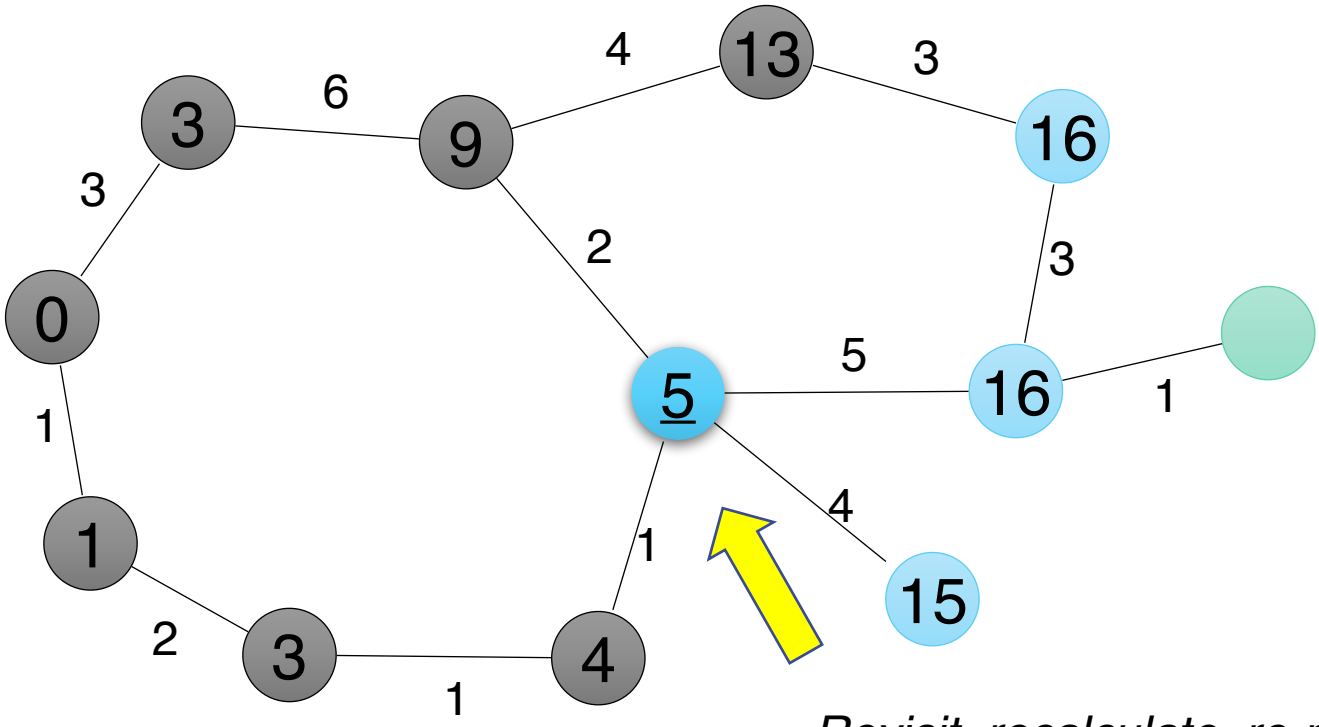
SSSP: BFS on Weighted Graphs?



SSSP: BFS on Weighted Graphs?

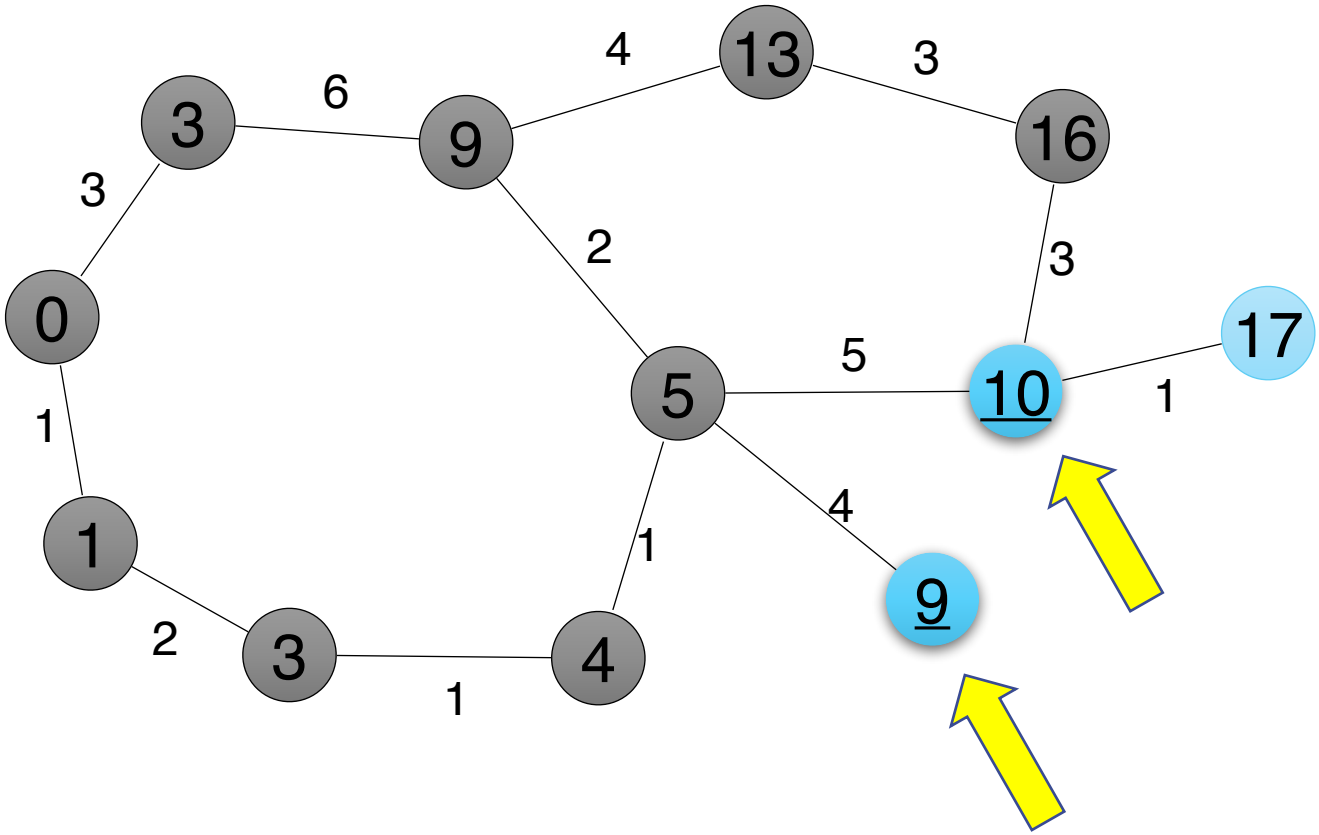


SSSP: BFS on Weighted Graphs?

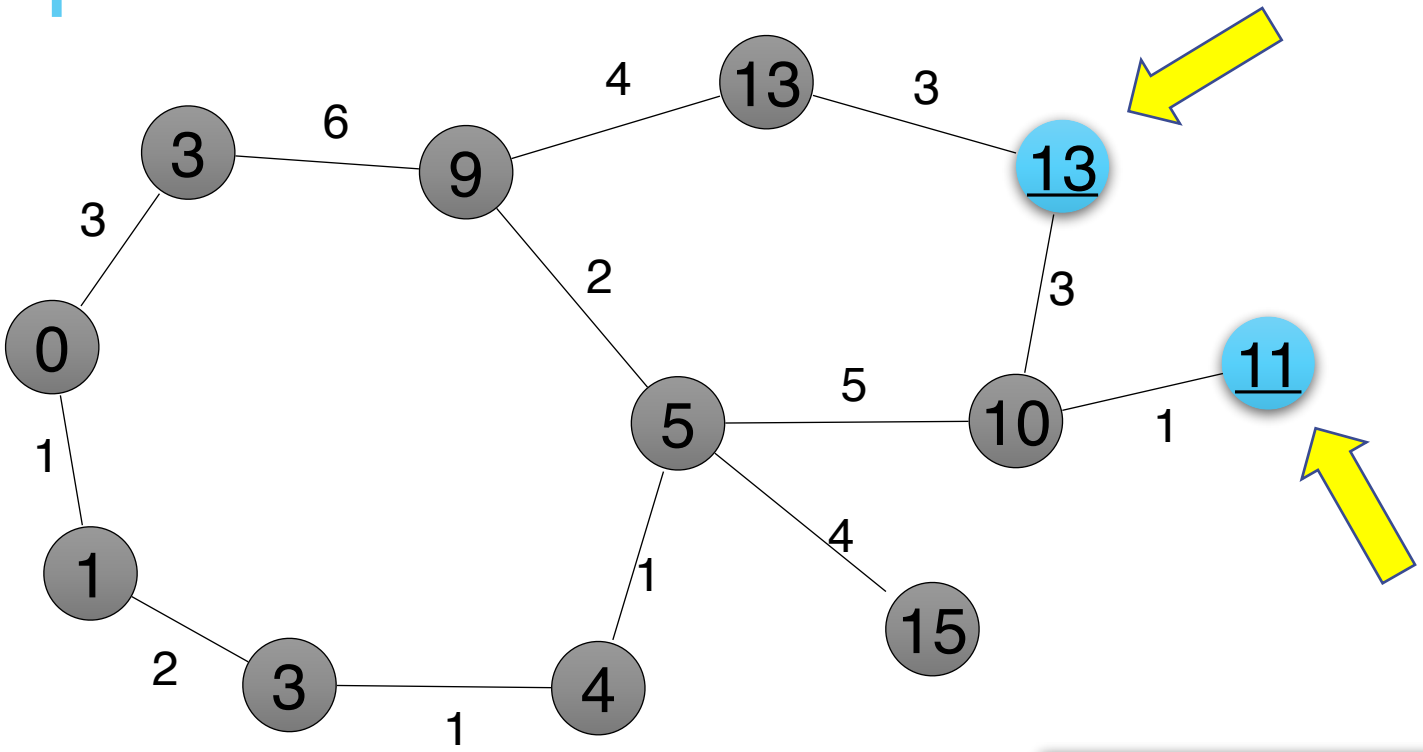


Revisit, recalculate, re-propagate…
cascading effect

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)

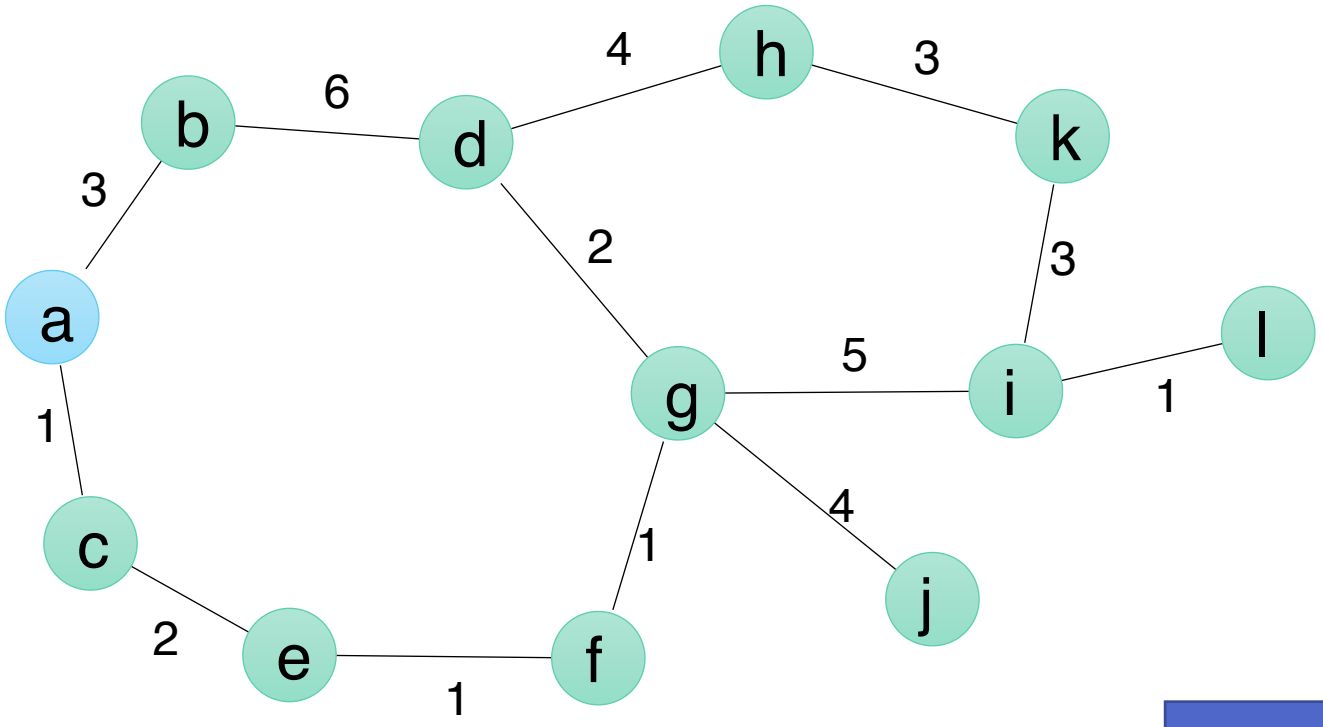
- Similar to BFS, but prioritise the vertices to visit next
  - Pick "unvisited" vertex with *shortest distance* from source
- Do not revisit vertices that have already been visited
  - Avoids false propagation of distances

Dijkstra's Single Source Shortest Path (SSSP)

- Let **w[u,v]** be array with weight of edge from u to v
- Initialise distance vector **d[ ]** for all vertices to infinity, except for source which is set to 0
- Add source vertex to container **Q**
- while(Q is not empty)
  - Remove **u** from **Q** such that **d[u]** is the smallest in **Q**
  - Mark u as **visited**
  - for each **v** adjacent to **u** that is not **visited**
    - Add **v** to container **Q** if not present in it
    - Set d[v] = min(d[v], d[u] + w[u,v])
- **Runtime?**

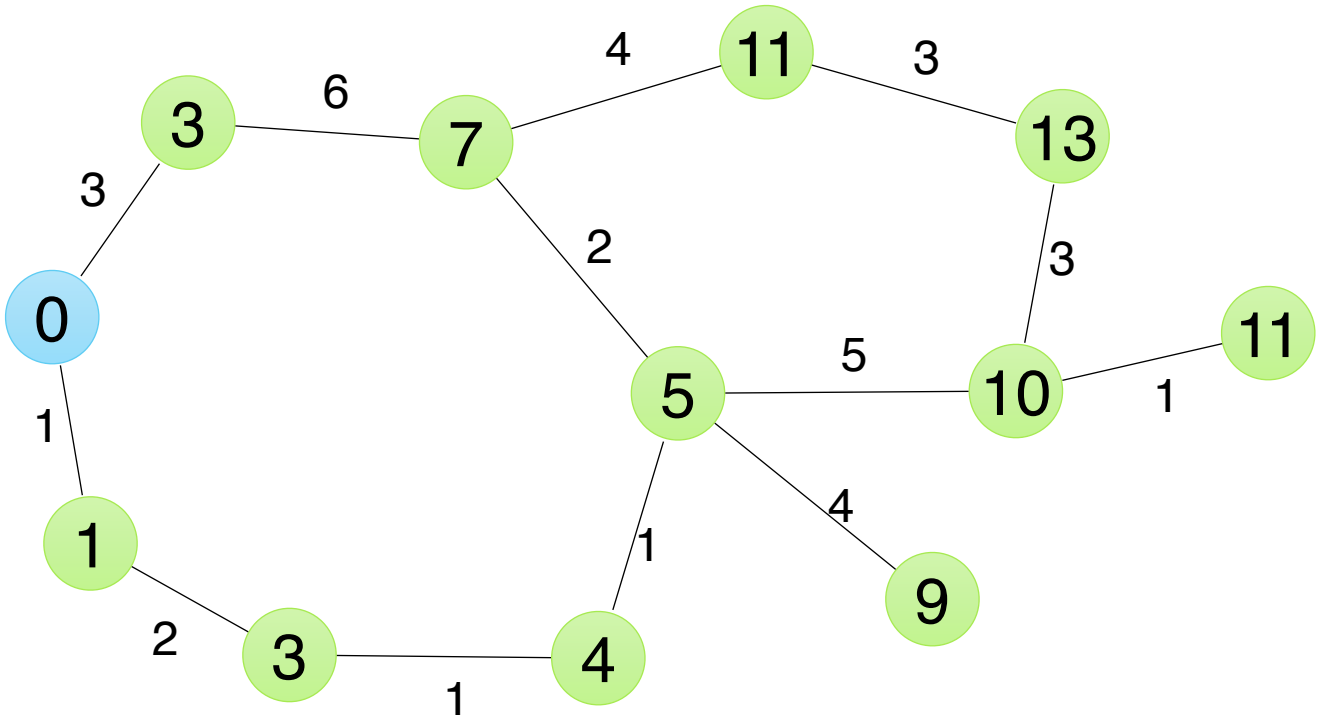
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

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 **$O((v + e) \log v)$**
  - 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)

Tasks

- Self study
  - **Read**: Graphs and graph algorithms (online sources)

Questions?