DS286 | 2016-11-02,04

L22-23: Graph Algorithms

Yogesh Simmhan
simmhan@cds.iisc.ac.in

Slides courtesy:
Venkatesh Babu, CDS, IISc

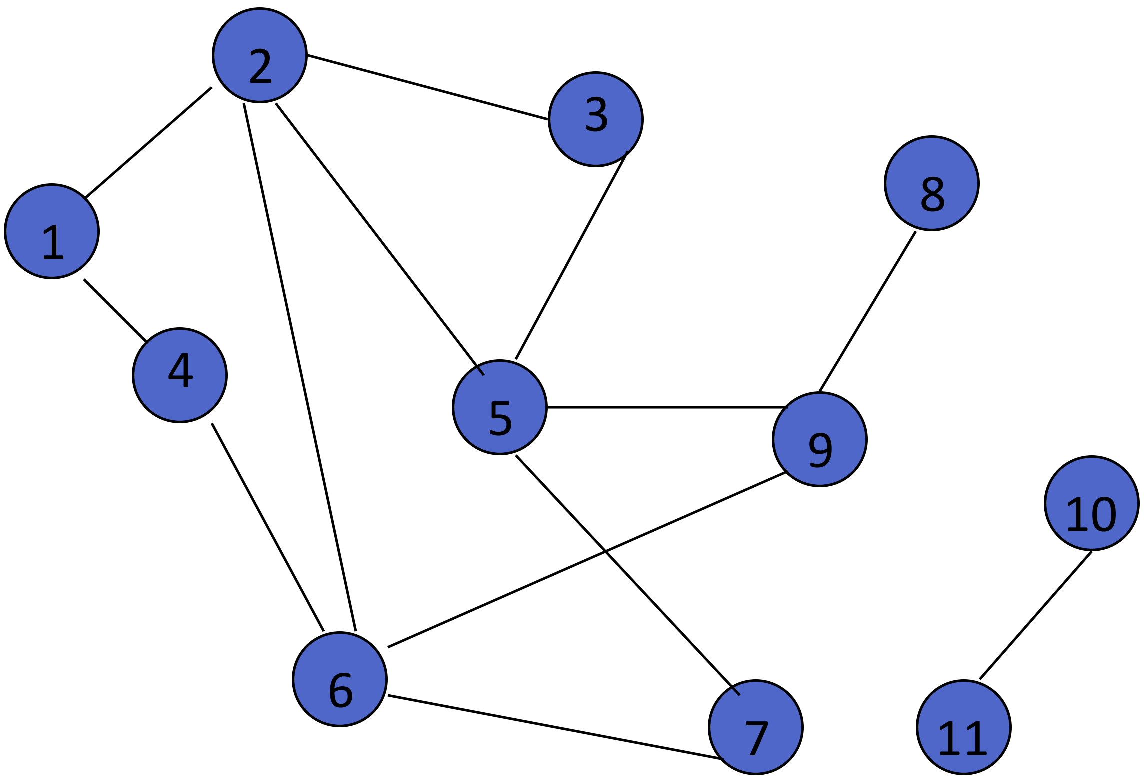
Sample Graph Problems

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

Graph Search & Traversal

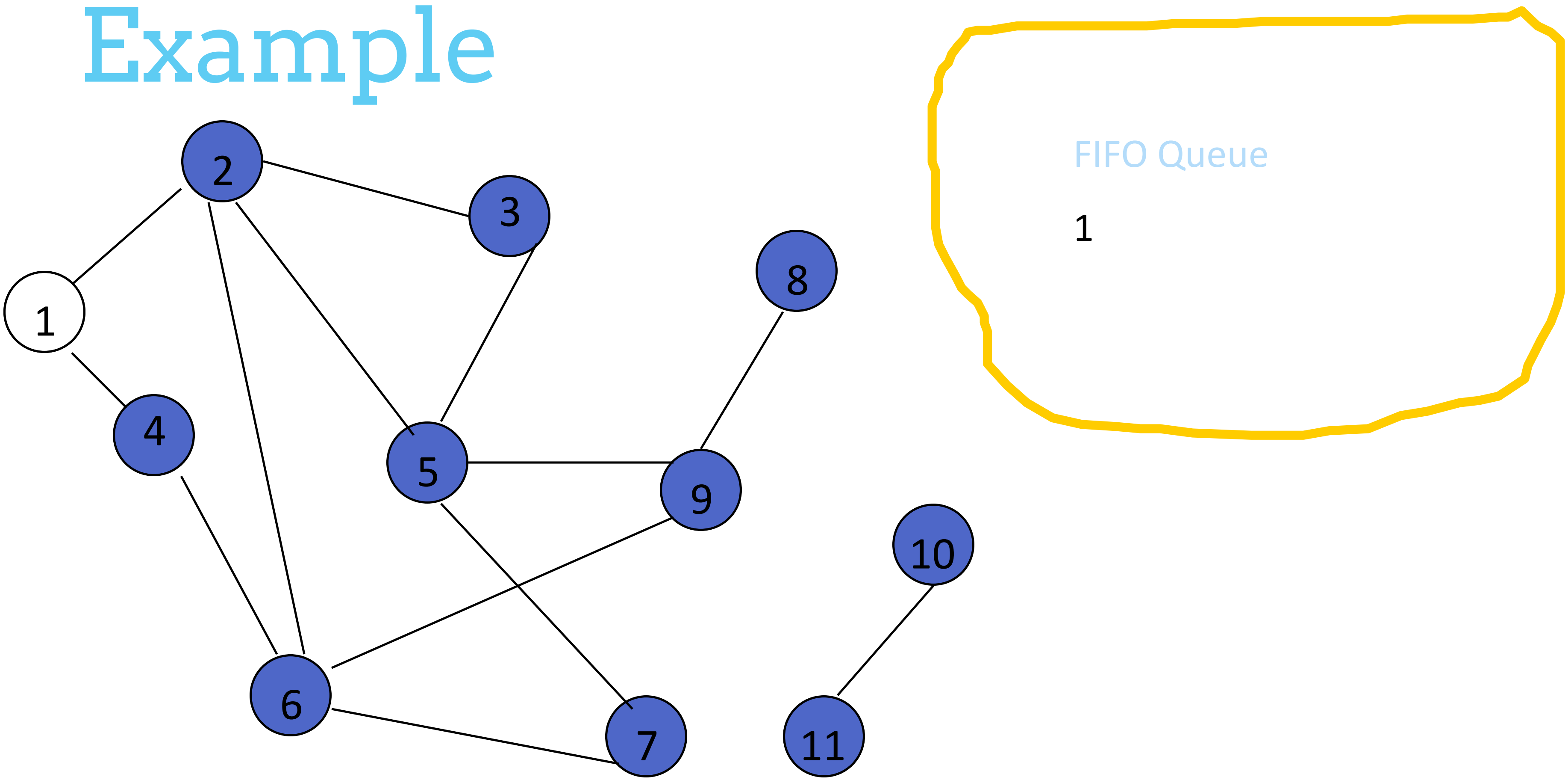
- Find a vertex (or edge) with a given ID or value
  - If list of vertices/edges is available, linear scan!
  - BUT, goal here is to traverse the neighbors of the graph, not scan the list
- Traverse through the graph to list all vertices in a particular order
  - Finding the item can be side-effect of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

Breadth-First Search Example



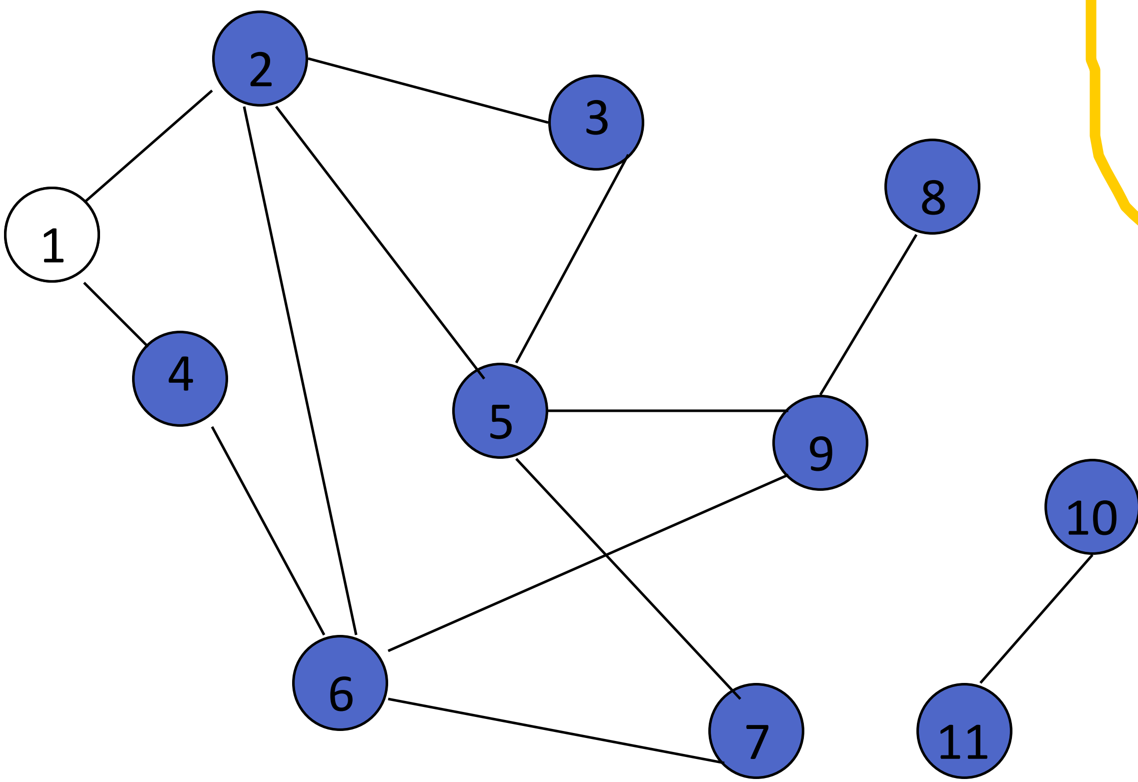
Start search at vertex 1.

Breadth-First Search Example



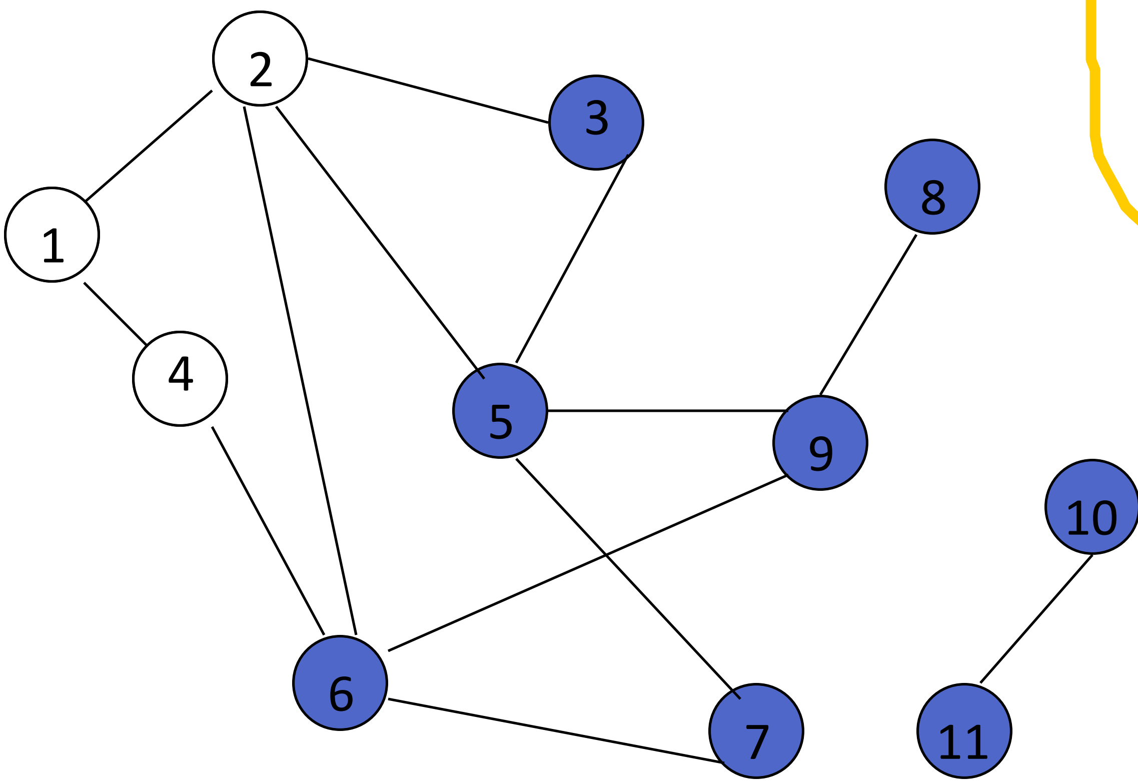
Visit/mark/label start vertex and put in a FIFO queue.

Breadth-First Search Example



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

Breadth-First Search Example

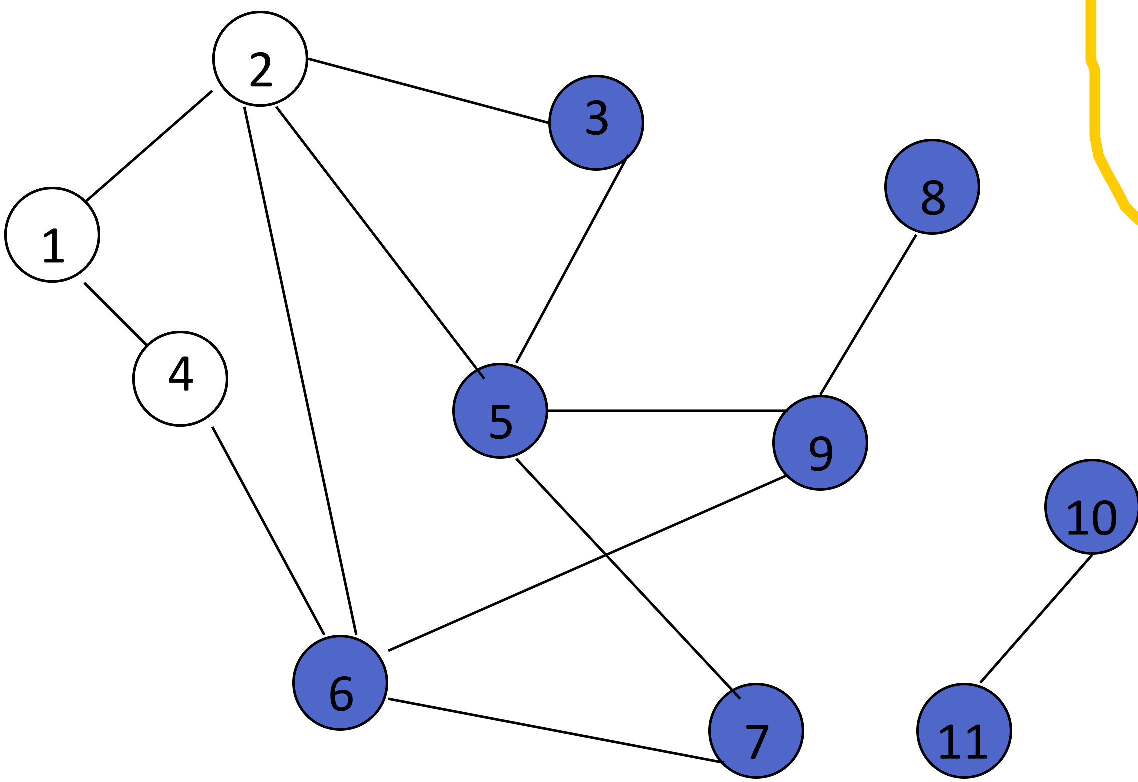


FIFO Queue

2 4

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

Breadth-First Search Example

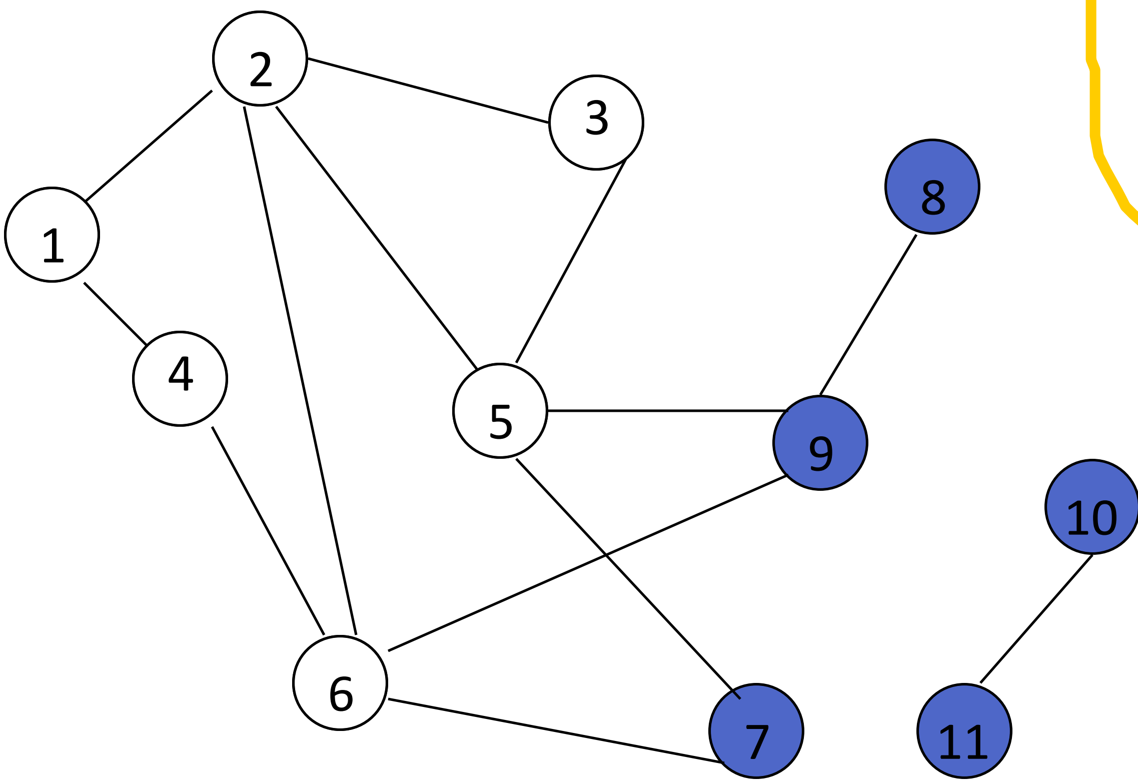


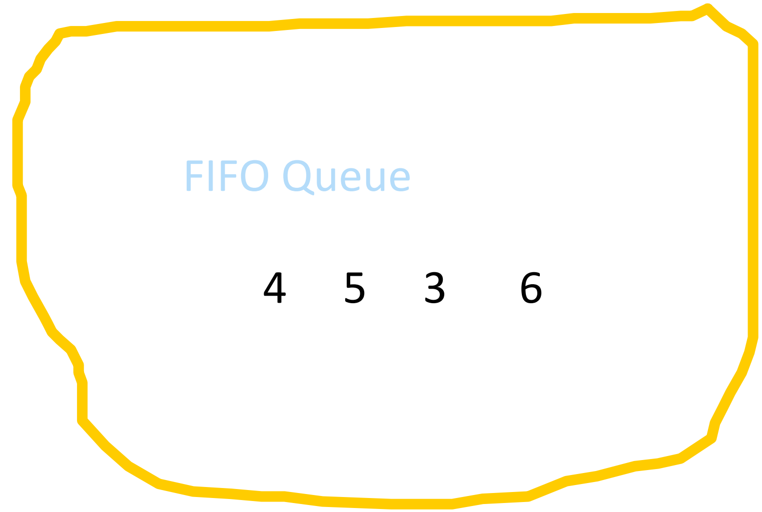
FIFO Queue

2 4

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

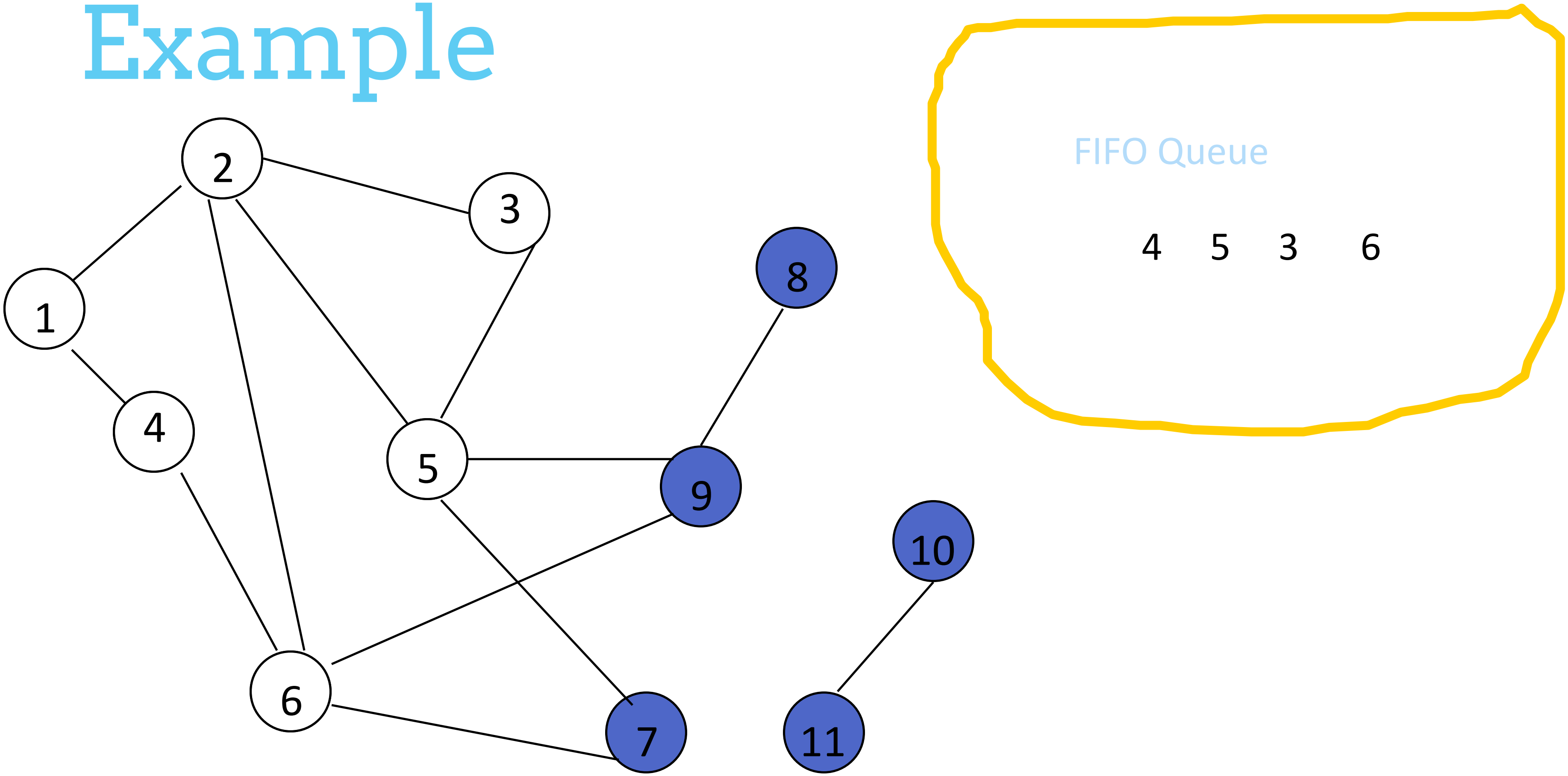
Breadth-First Search Example





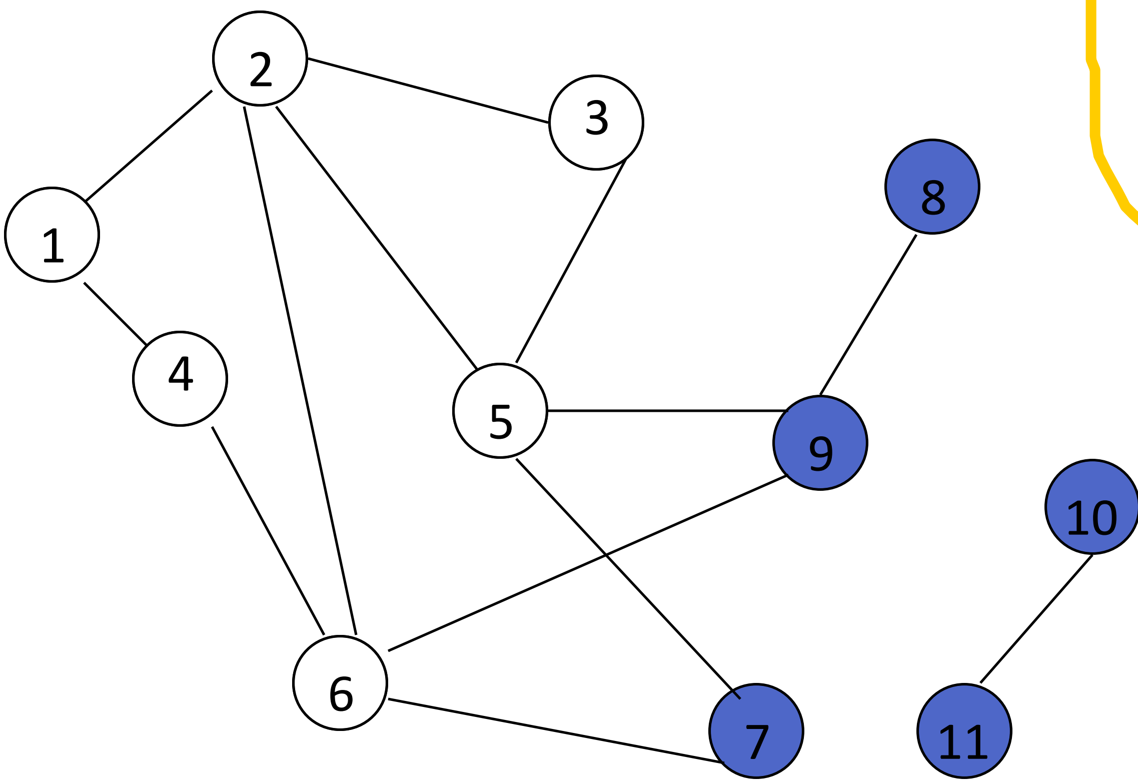
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

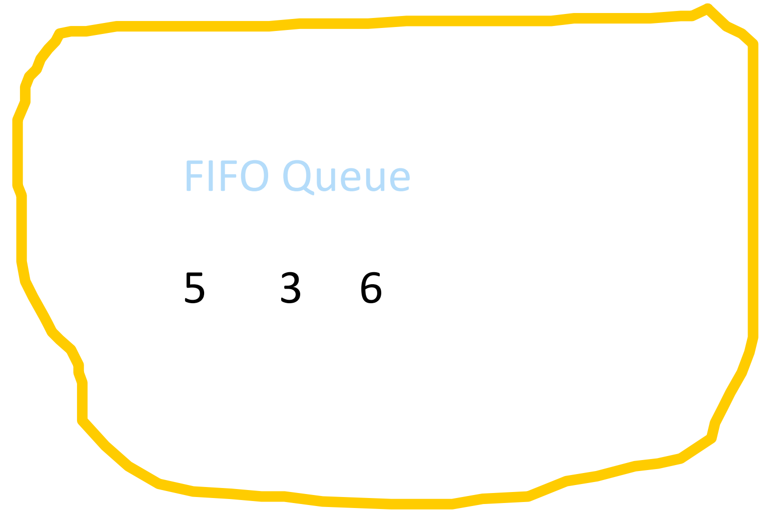
Breadth-First Search Example



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

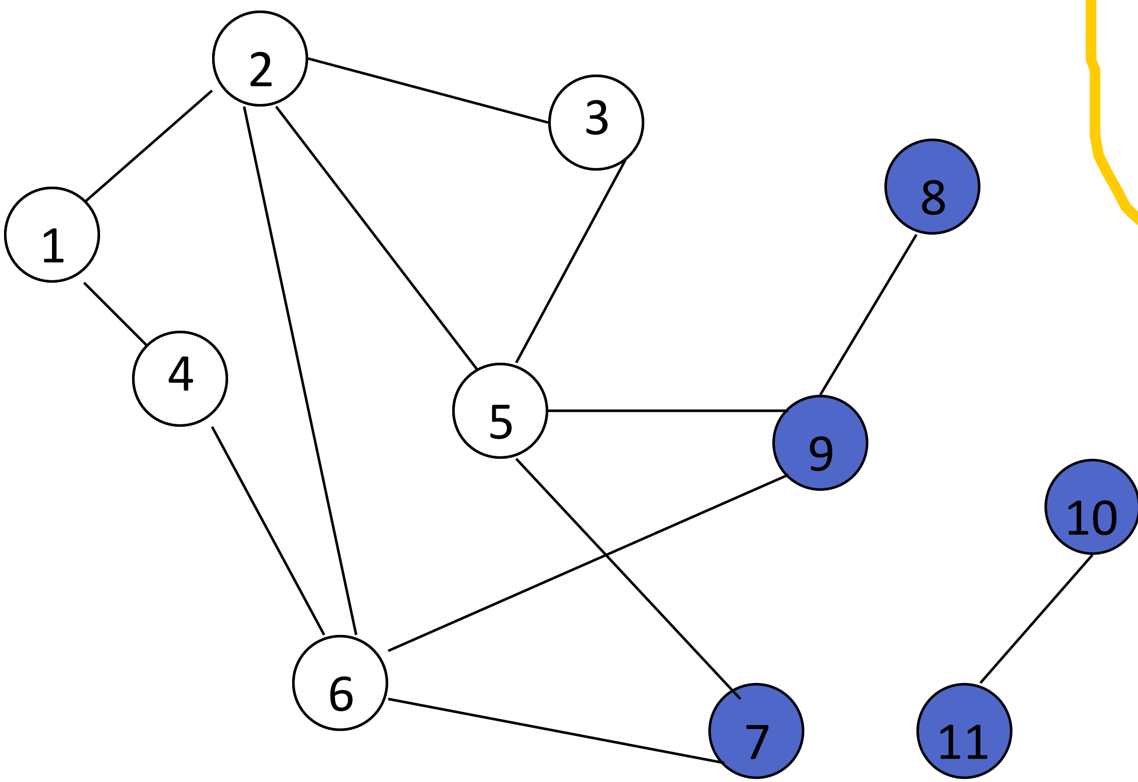
Breadth-First Search Example

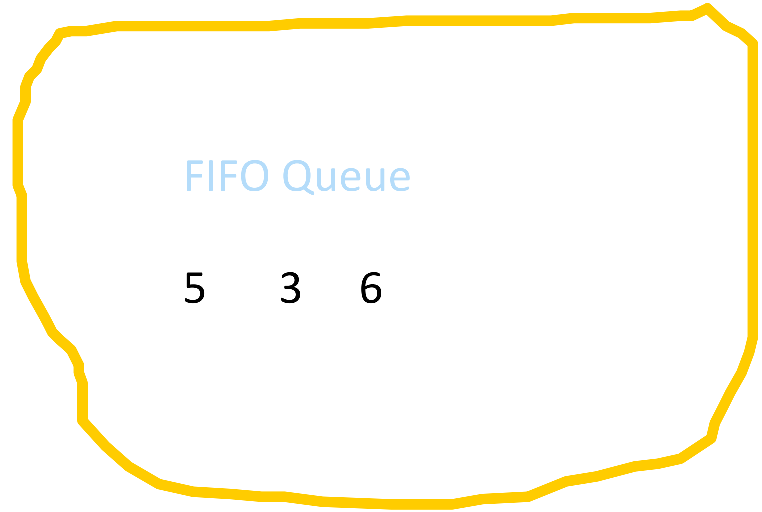




Remove 4 from Q; visit adjacent unvisited vertices;
put in Q.

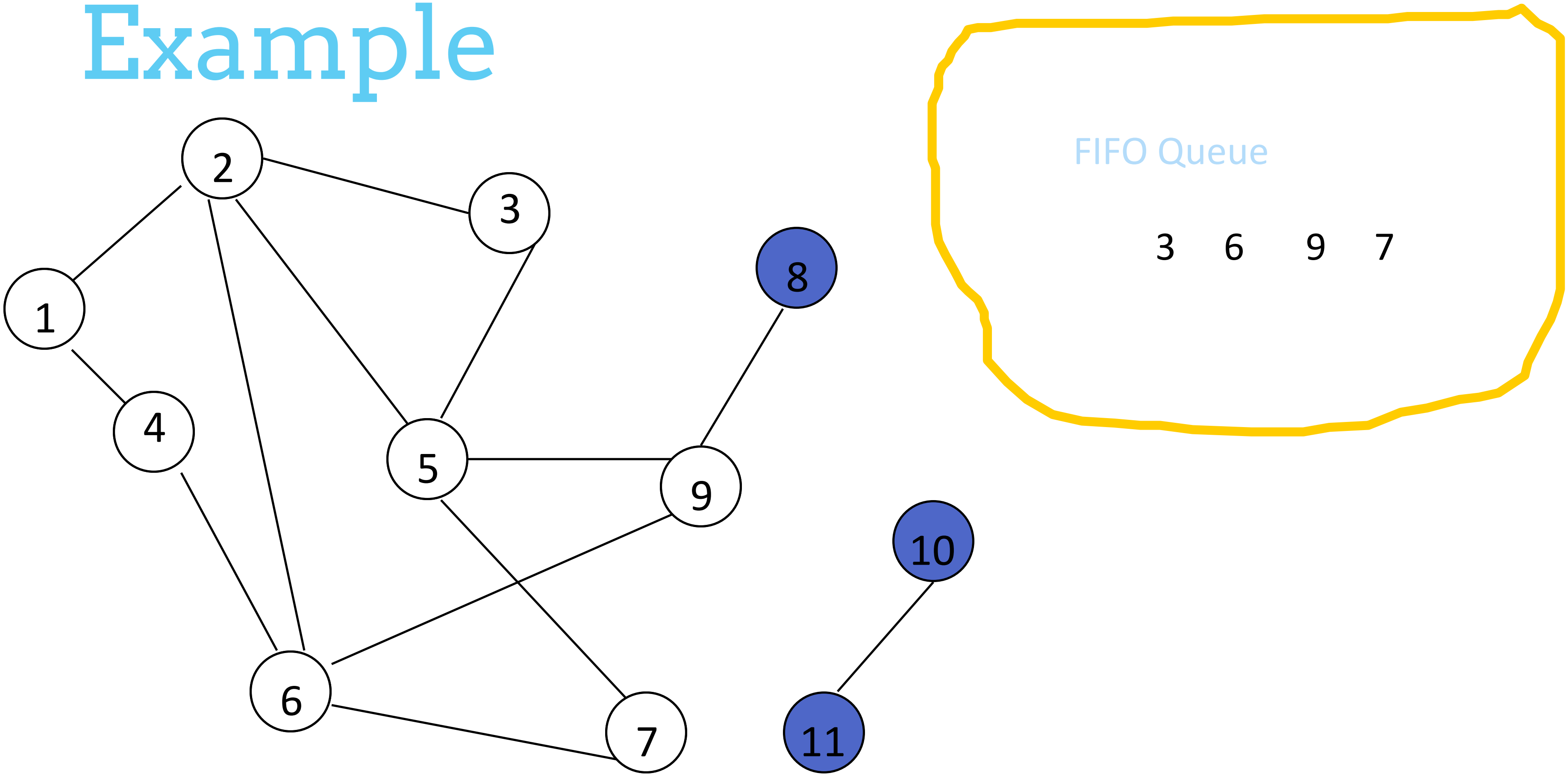
Breadth-First Search Example





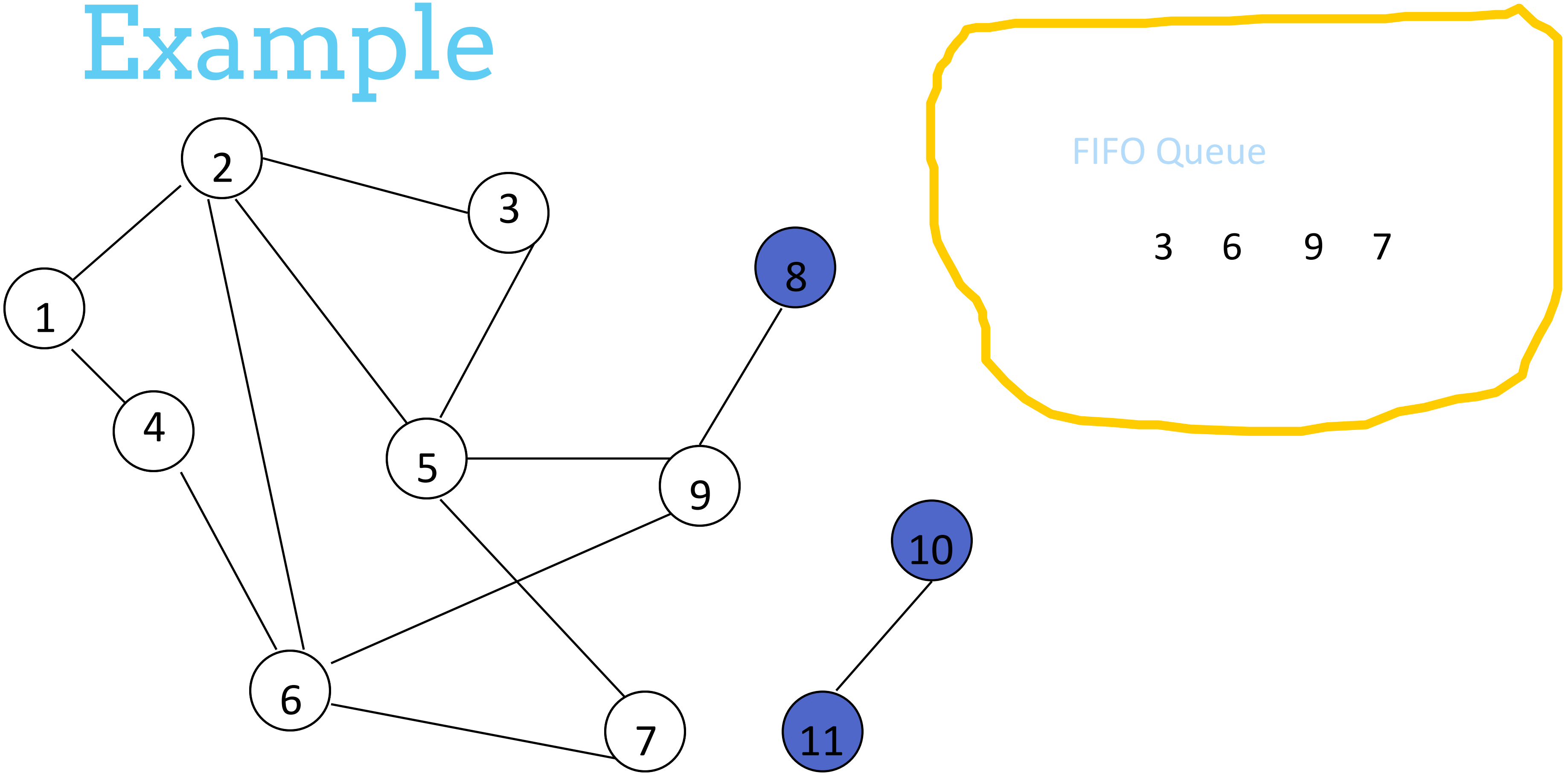
Remove 5 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



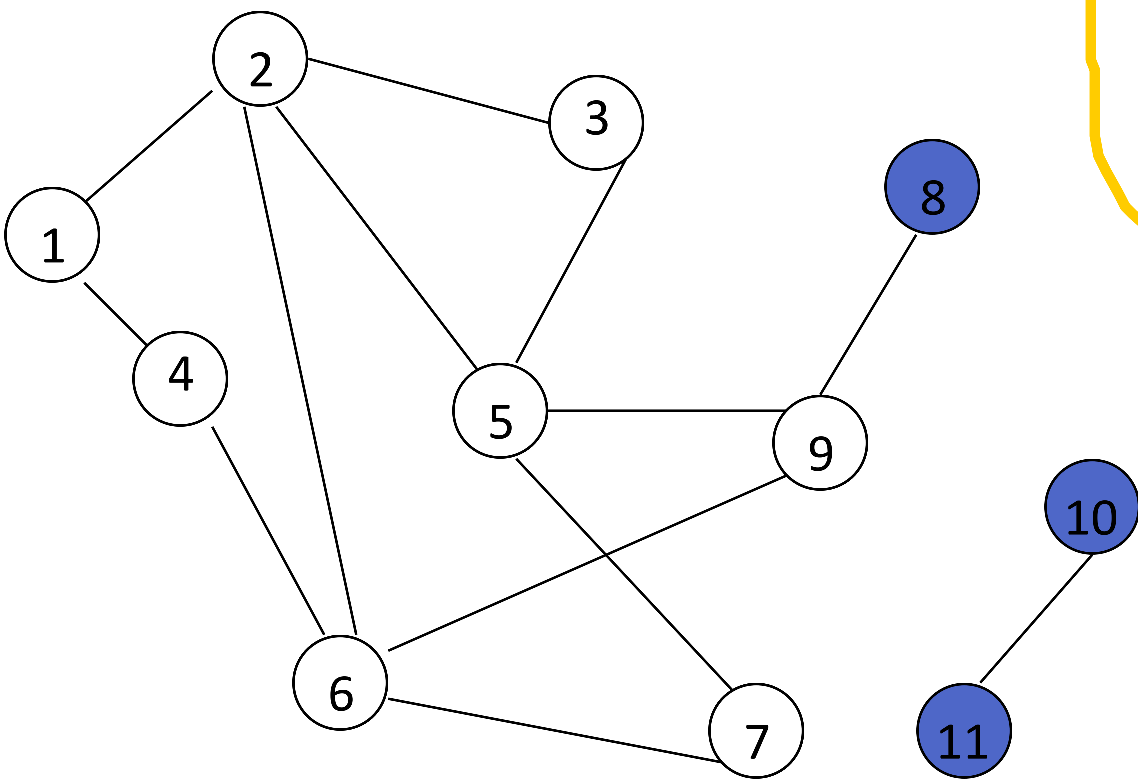
Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

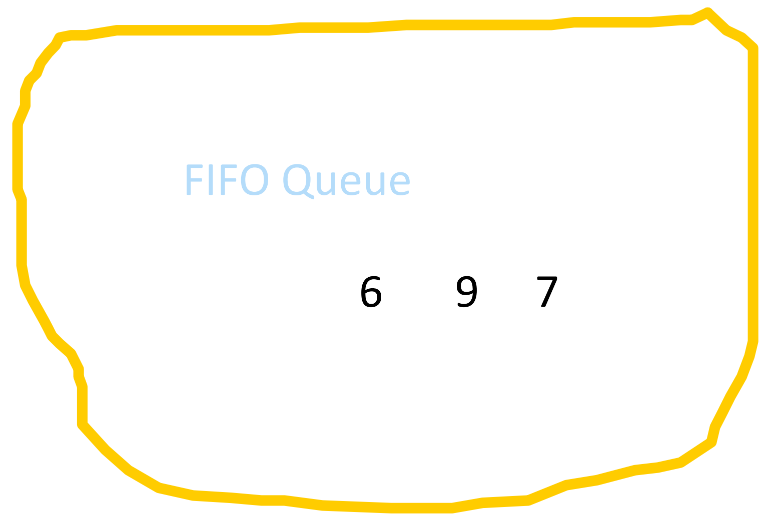
Breadth-First Search Example



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

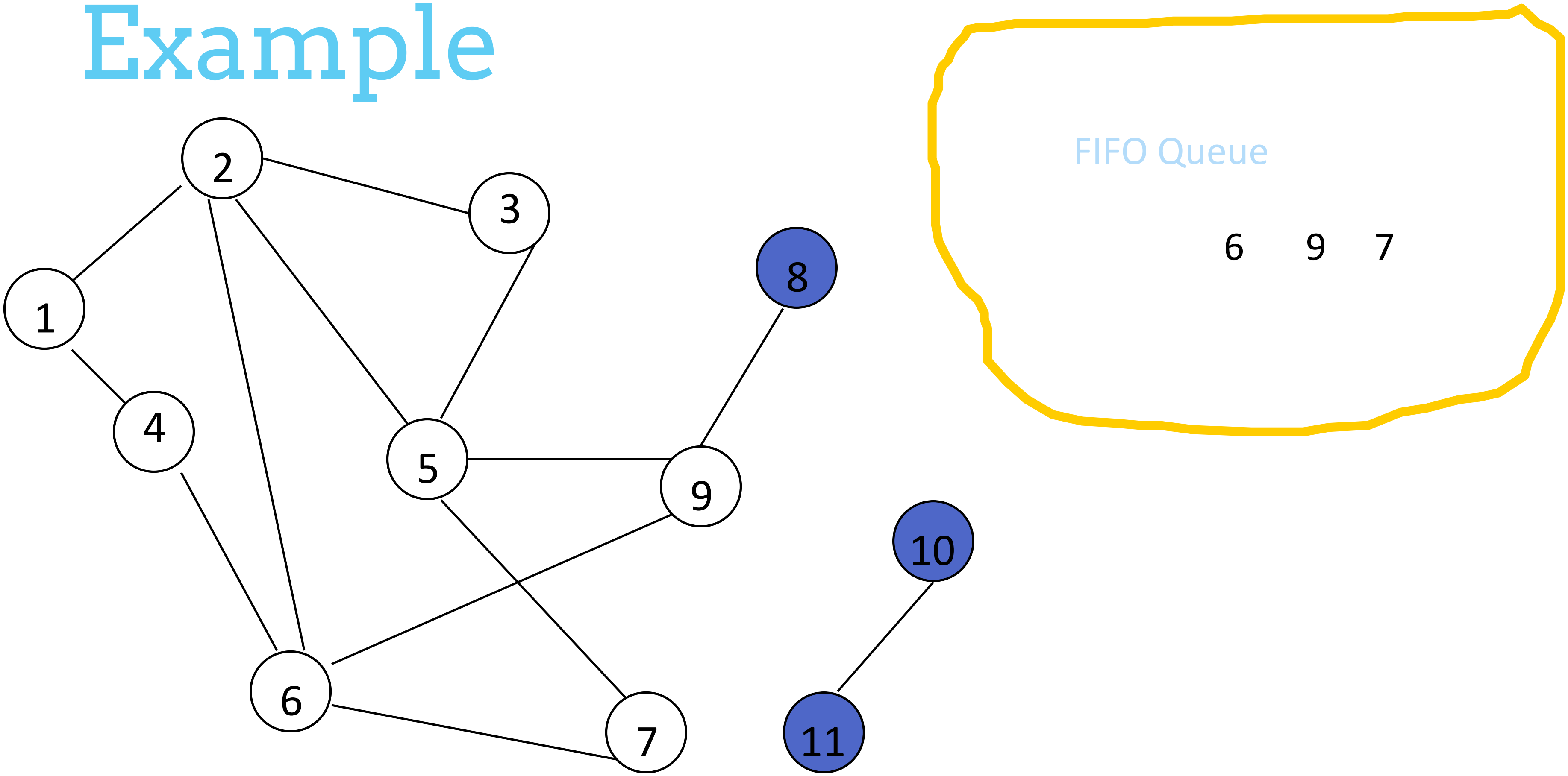
Breadth-First Search Example





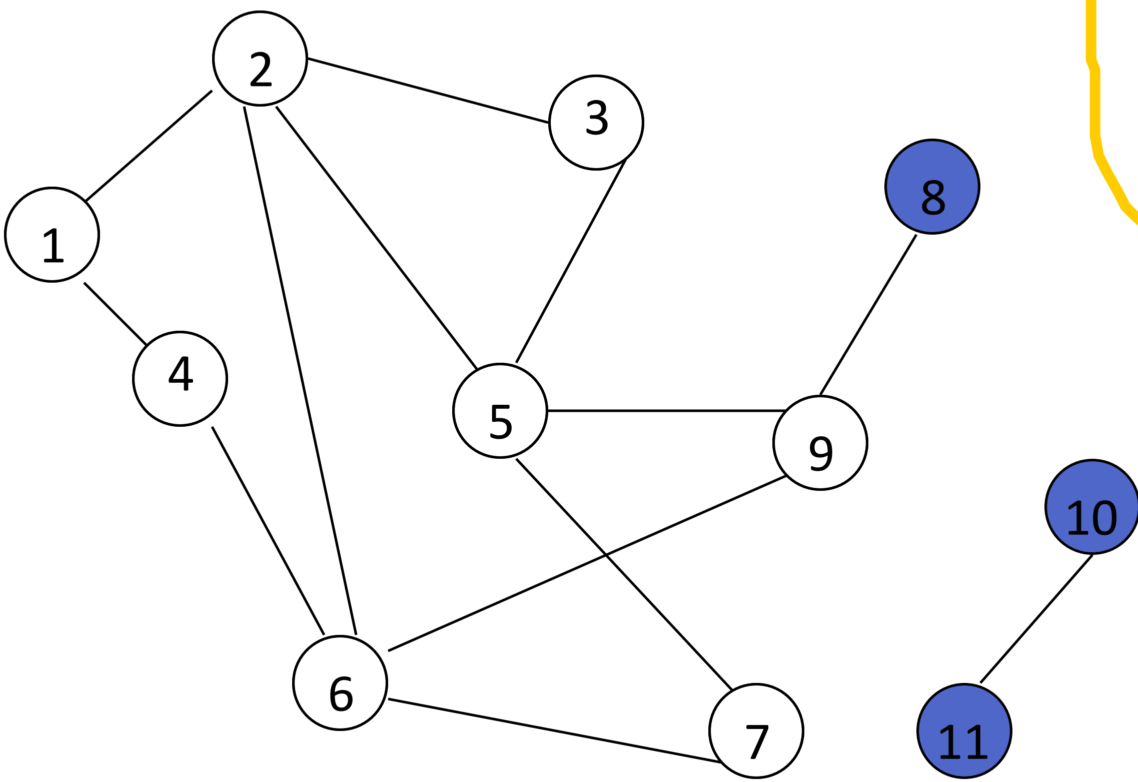
Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

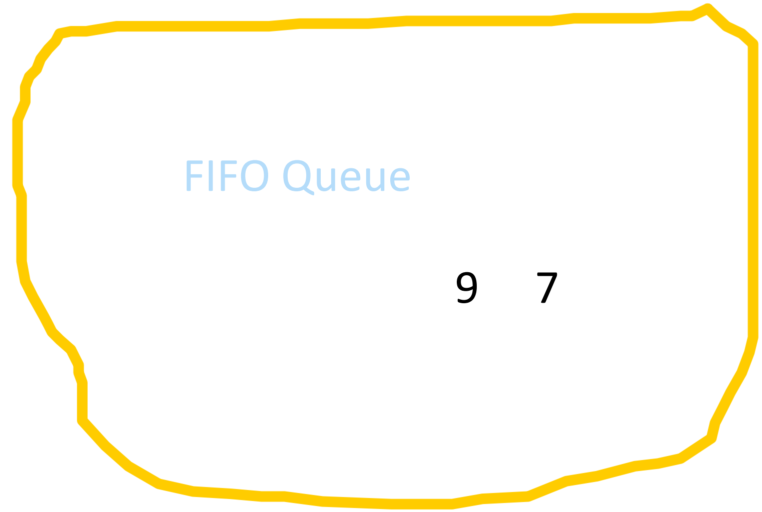
Breadth-First Search Example



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

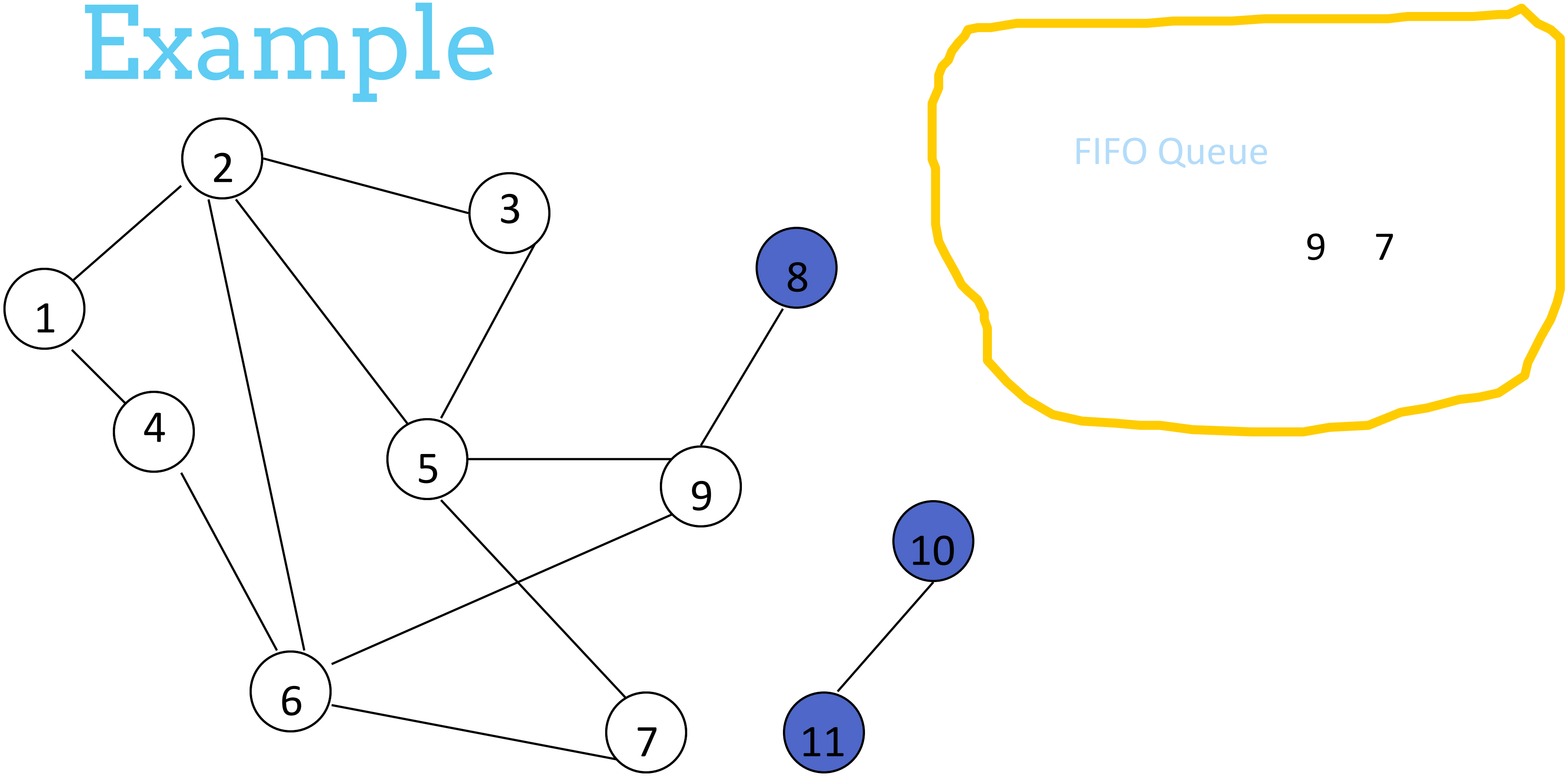
Breadth-First Search Example





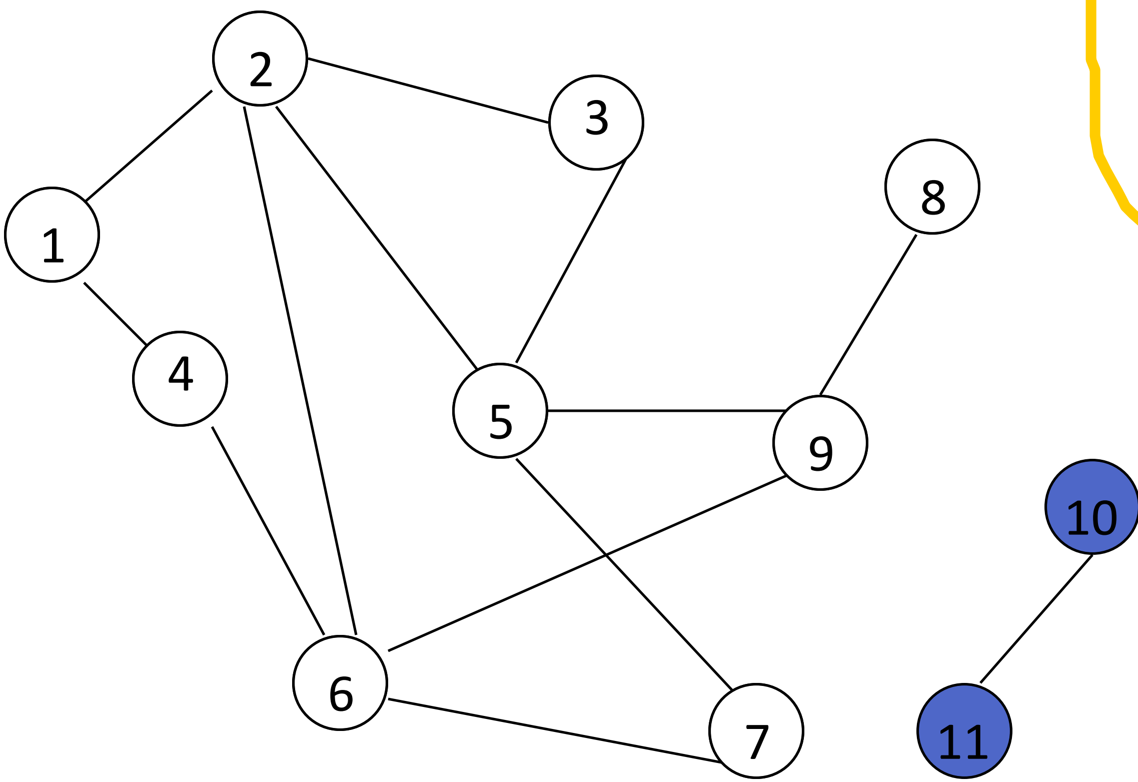
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

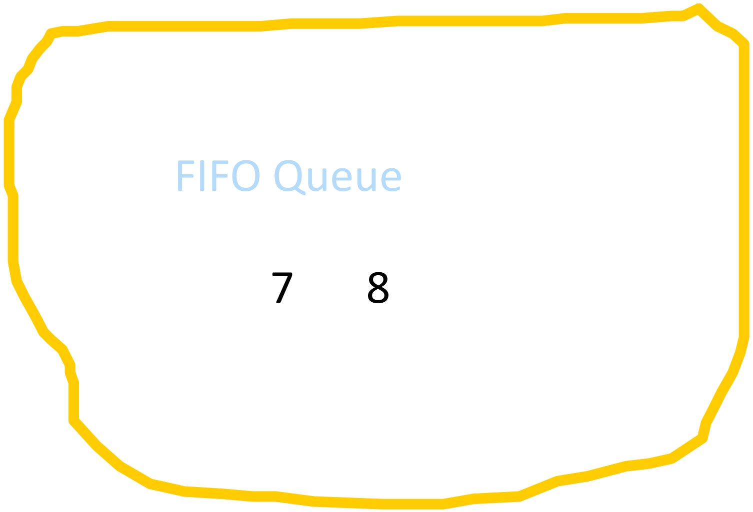
Breadth-First Search Example



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

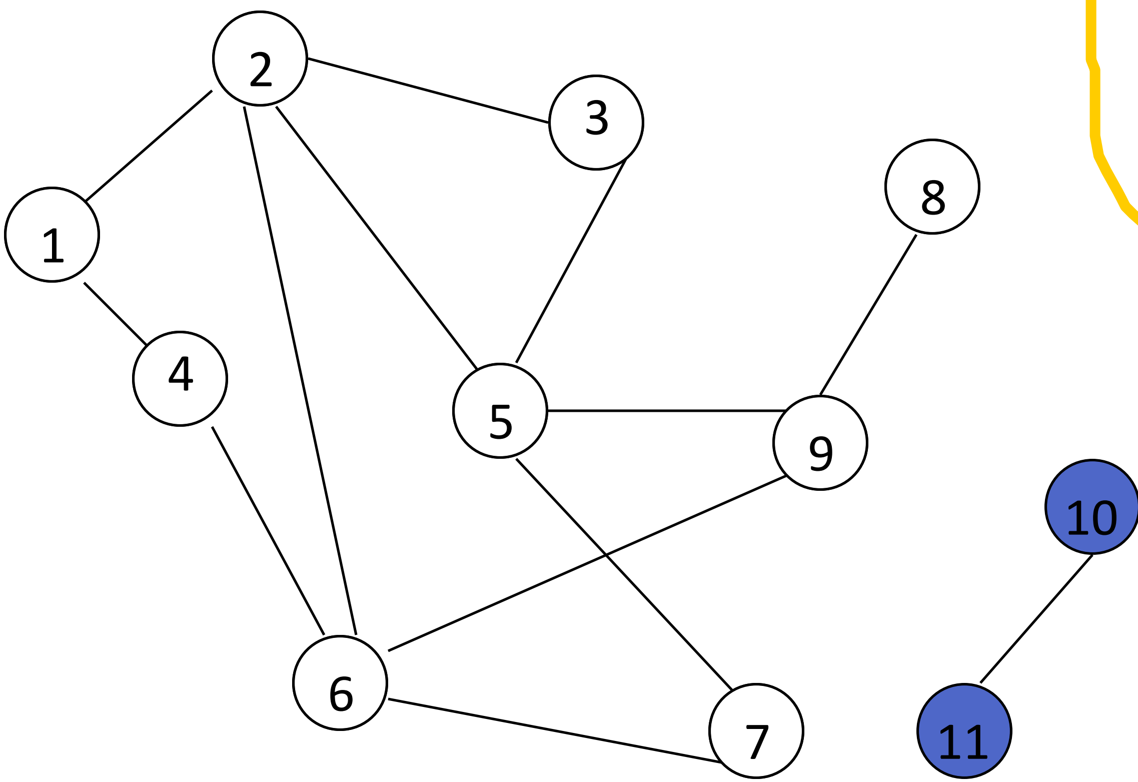
Breadth-First Search Example

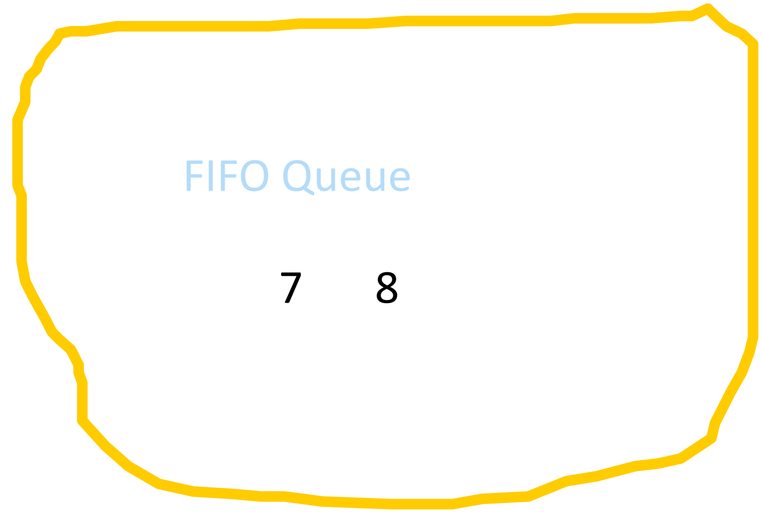




Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

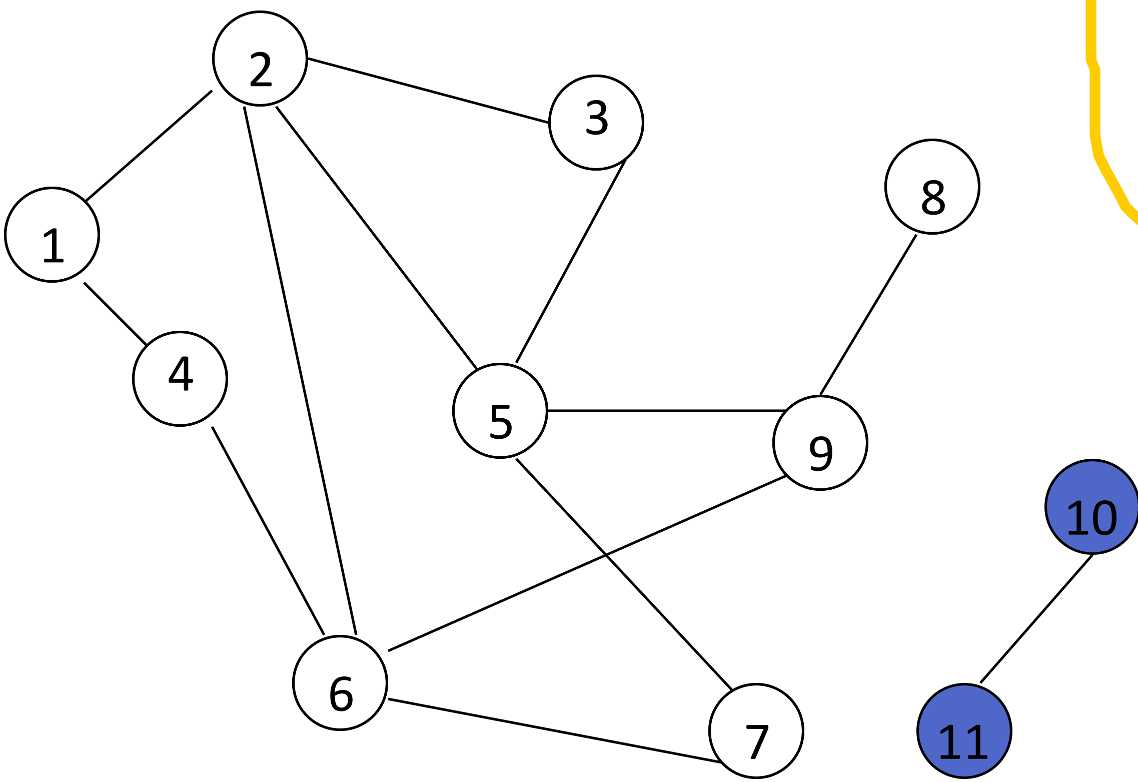
Breadth-First Search Example





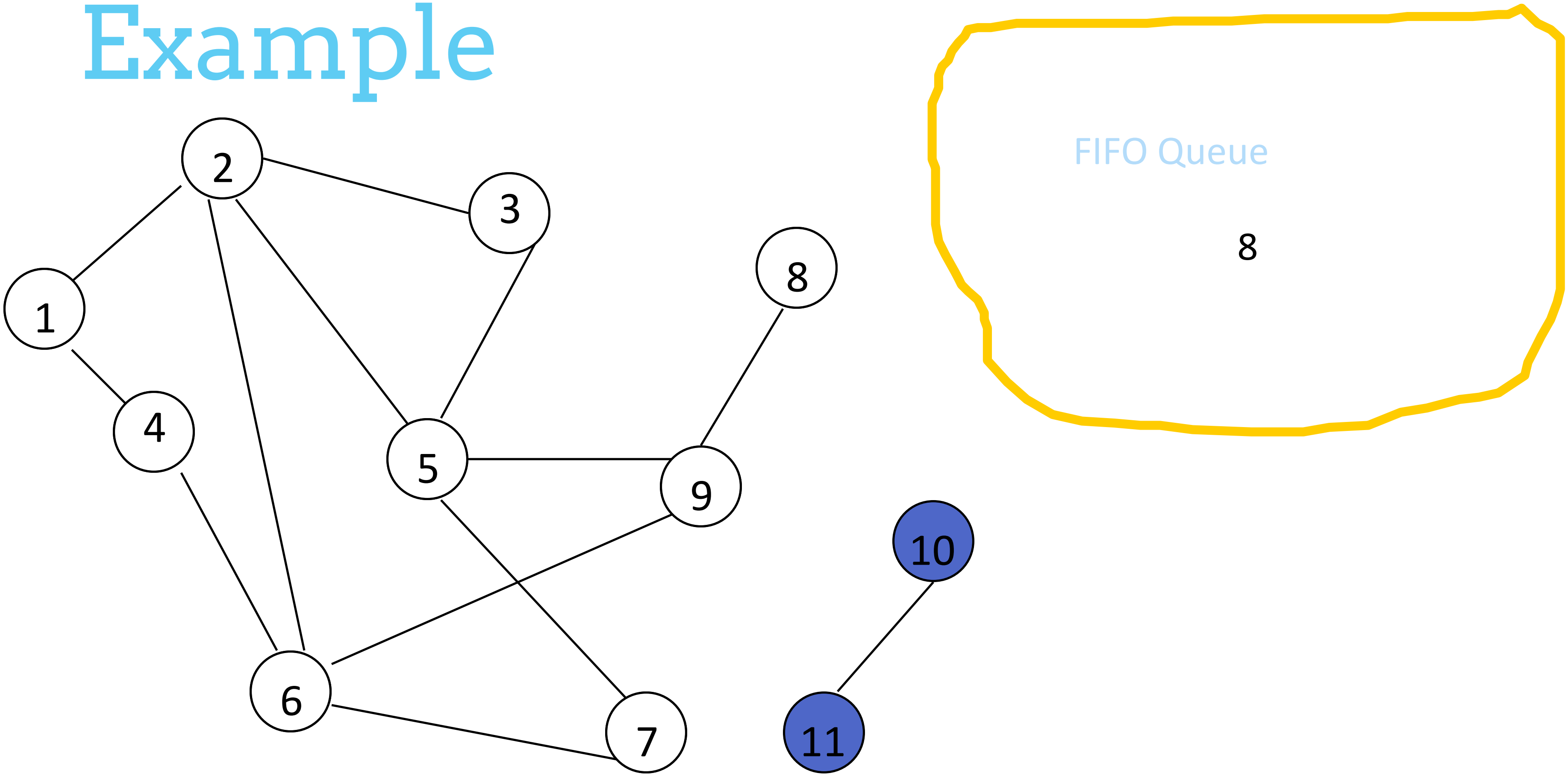
Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



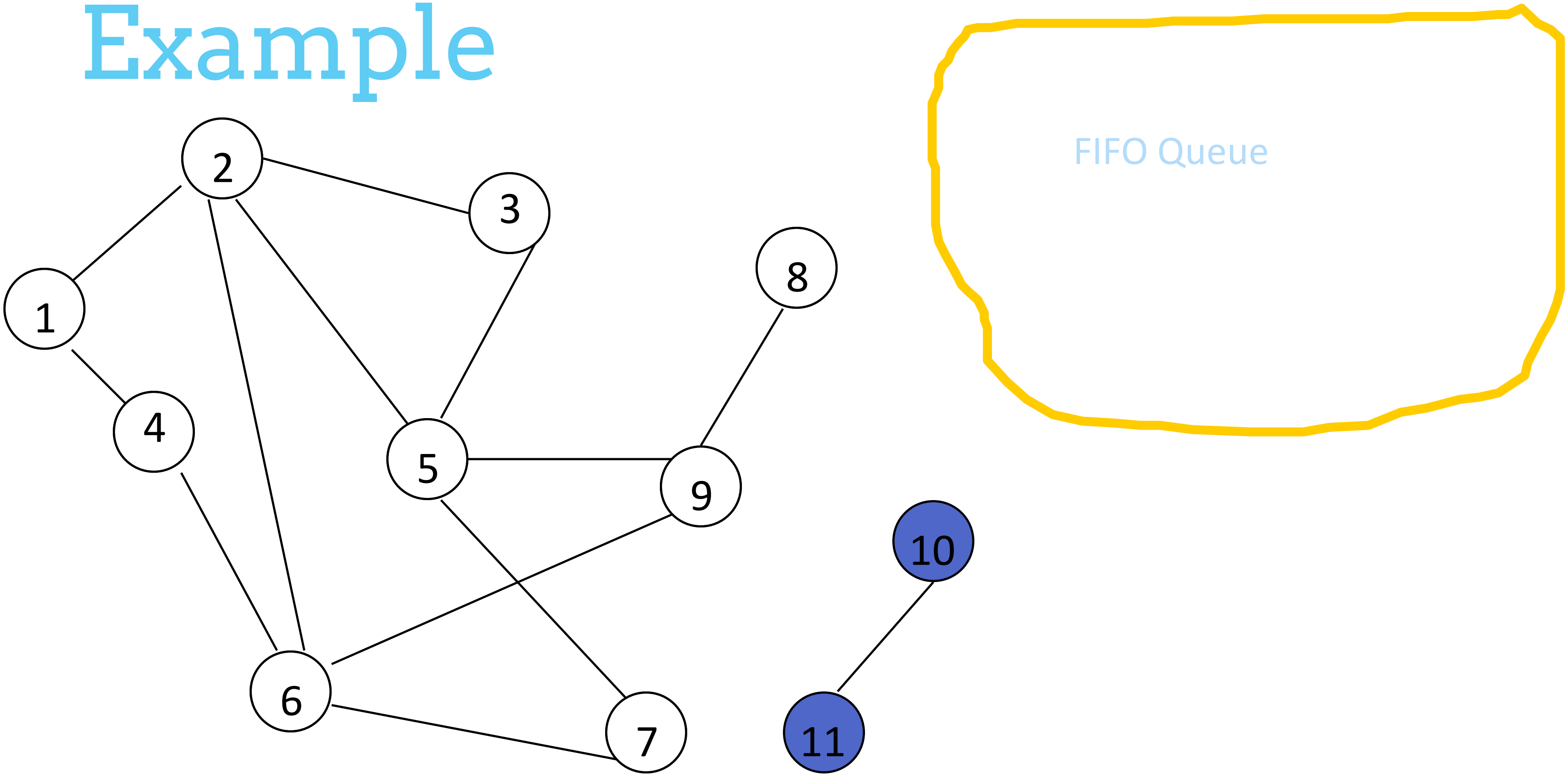
Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

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.

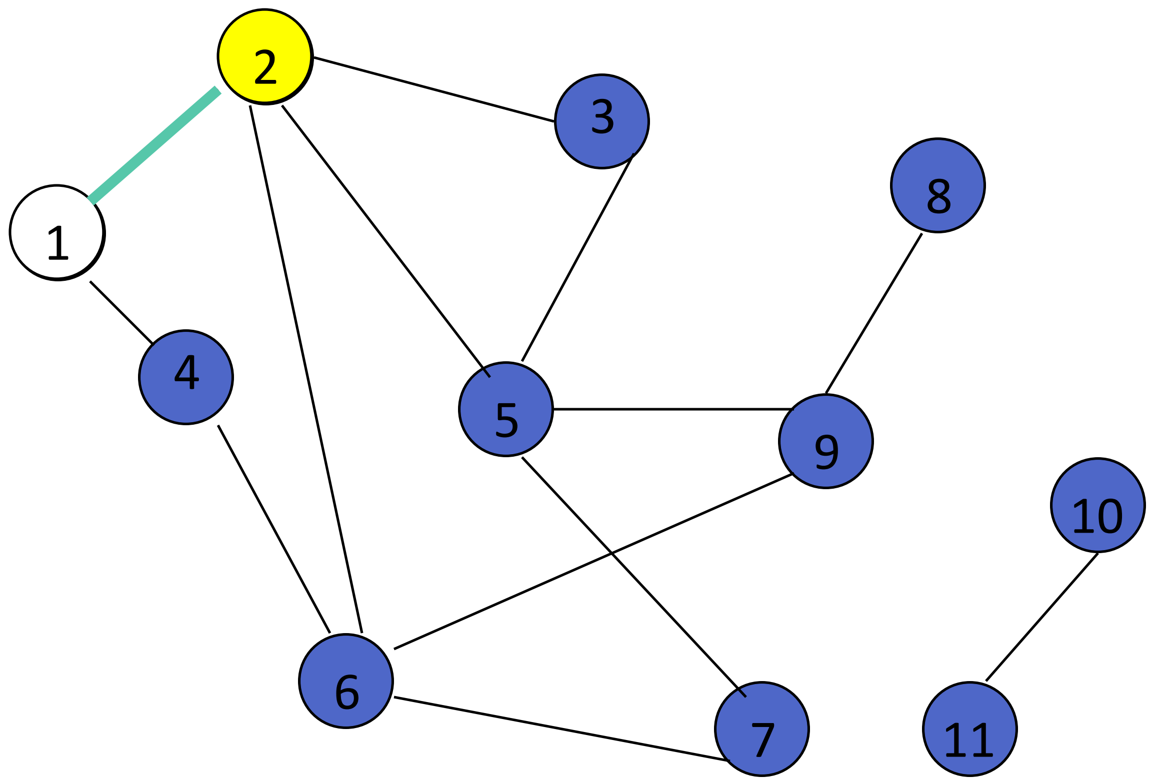
Time Complexity

- 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

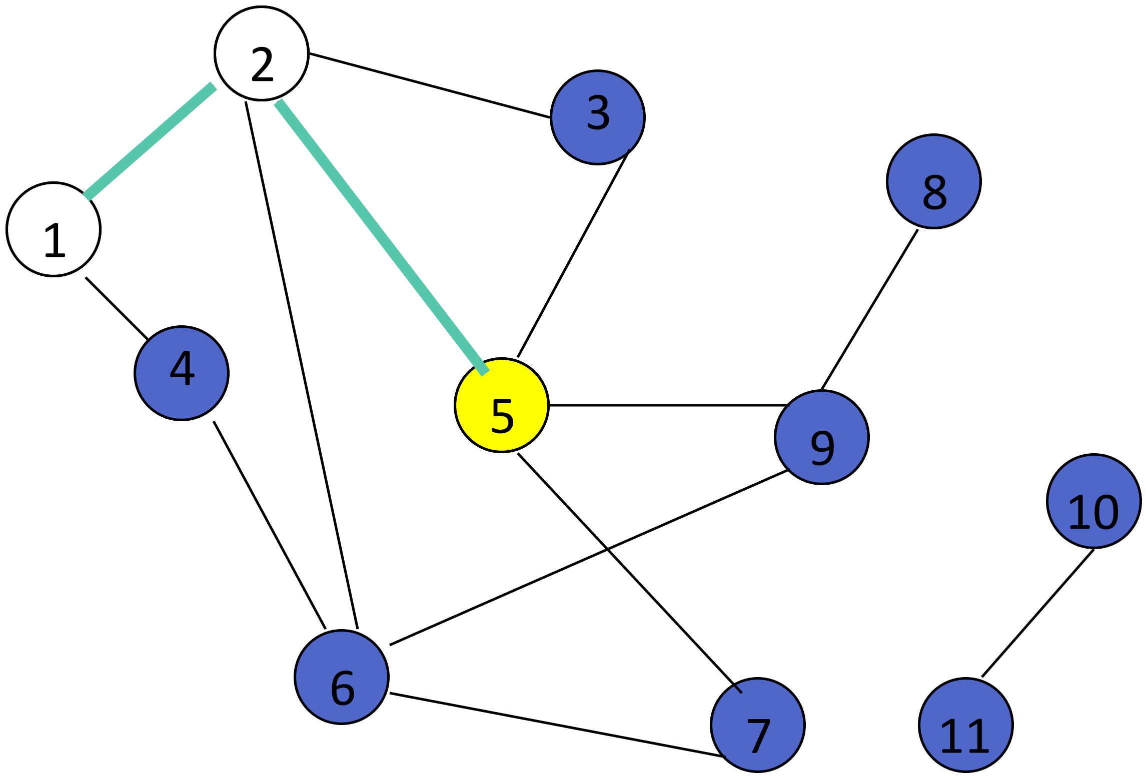


Start search at vertex 1.

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

Suppose that vertex 2 is selected.

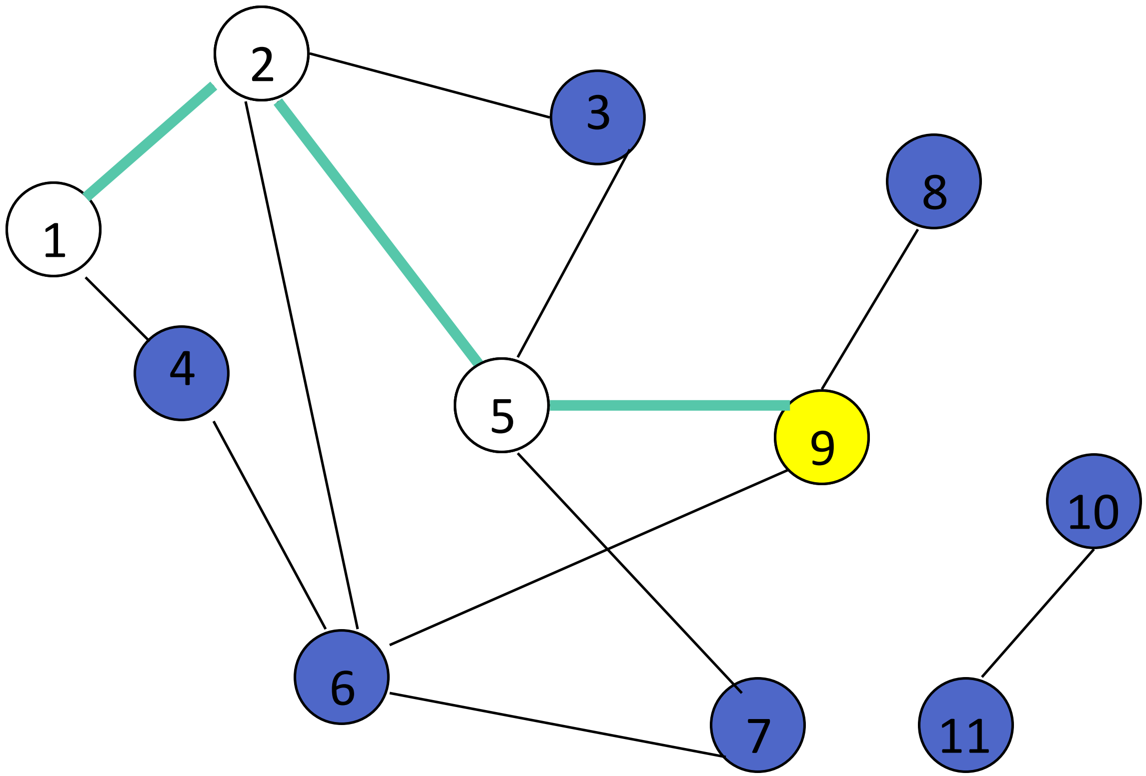
Depth-First Search Example



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

Suppose that vertex 5 is selected.

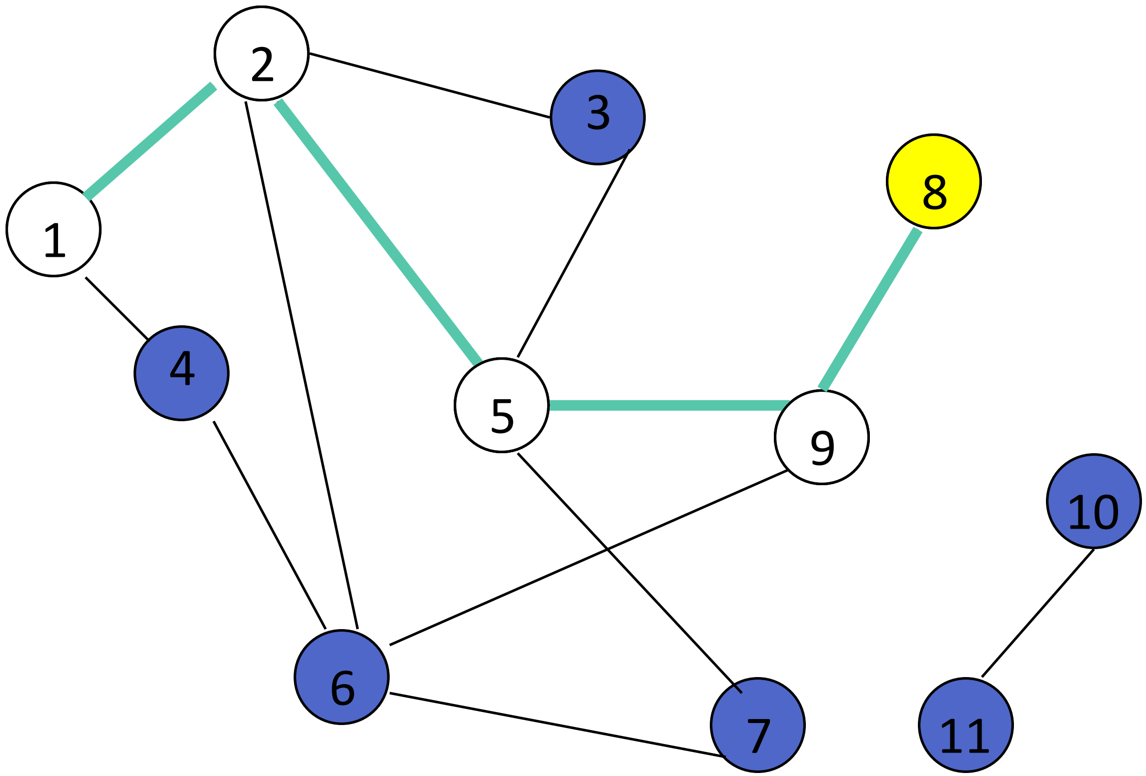
Depth-First Search



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

Suppose that vertex 9 is selected.

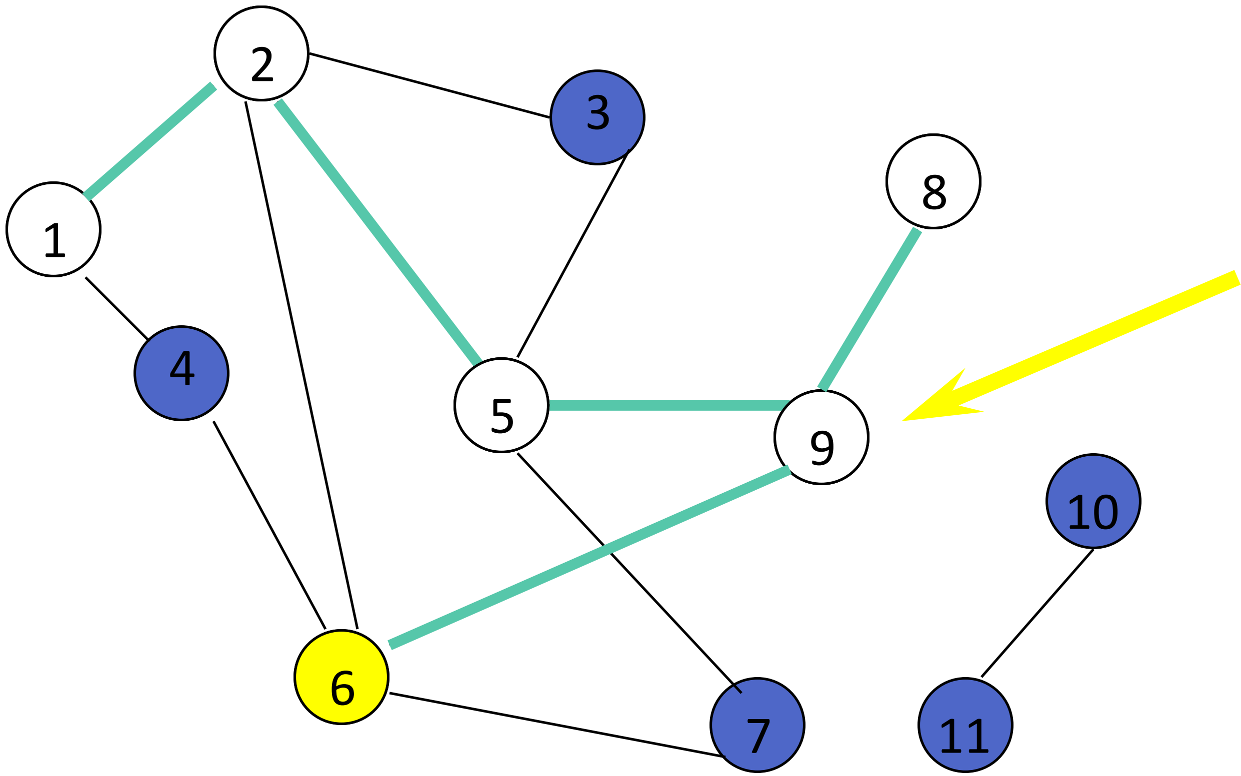
Depth-First Search



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

Suppose that vertex 8 is selected.

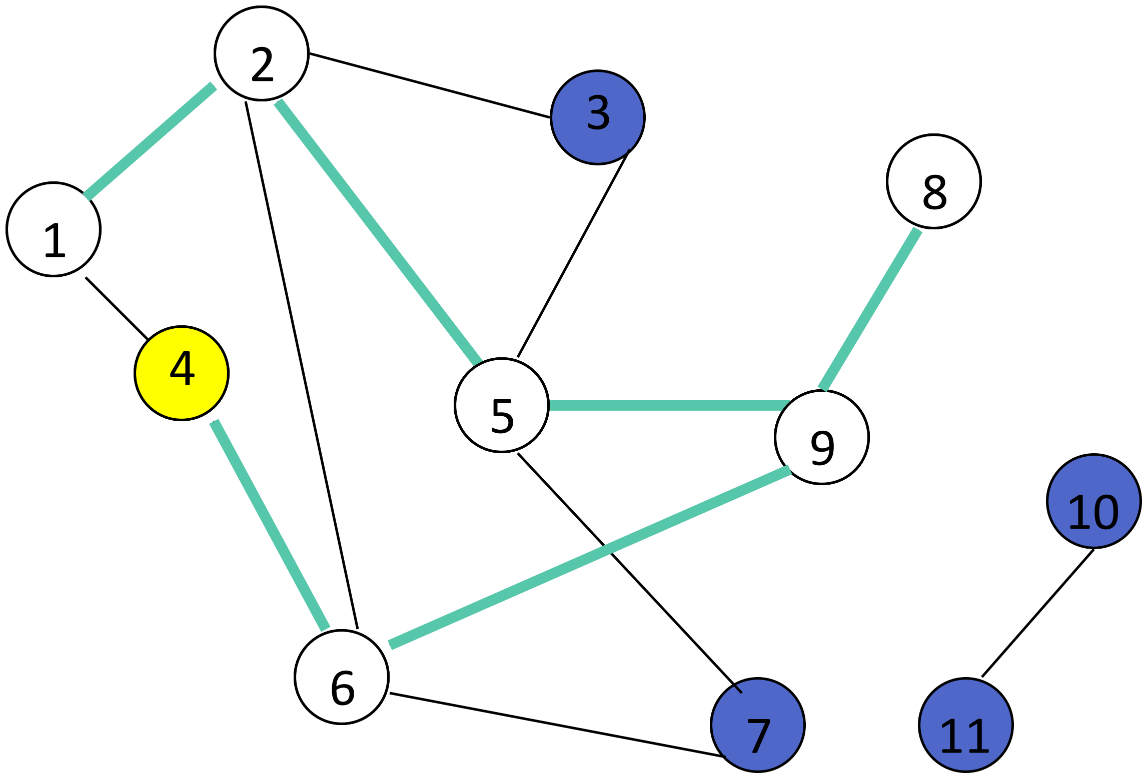
Depth-First Search



Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6)

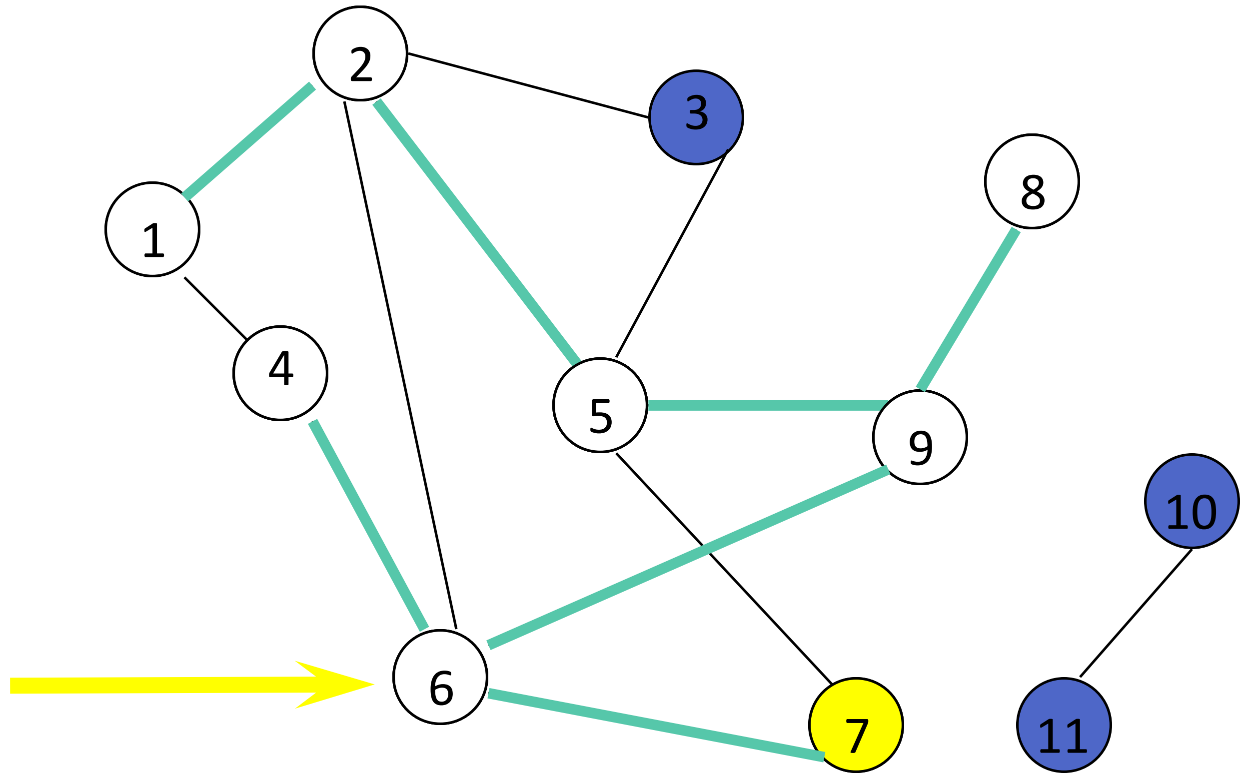
Depth-First Search



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

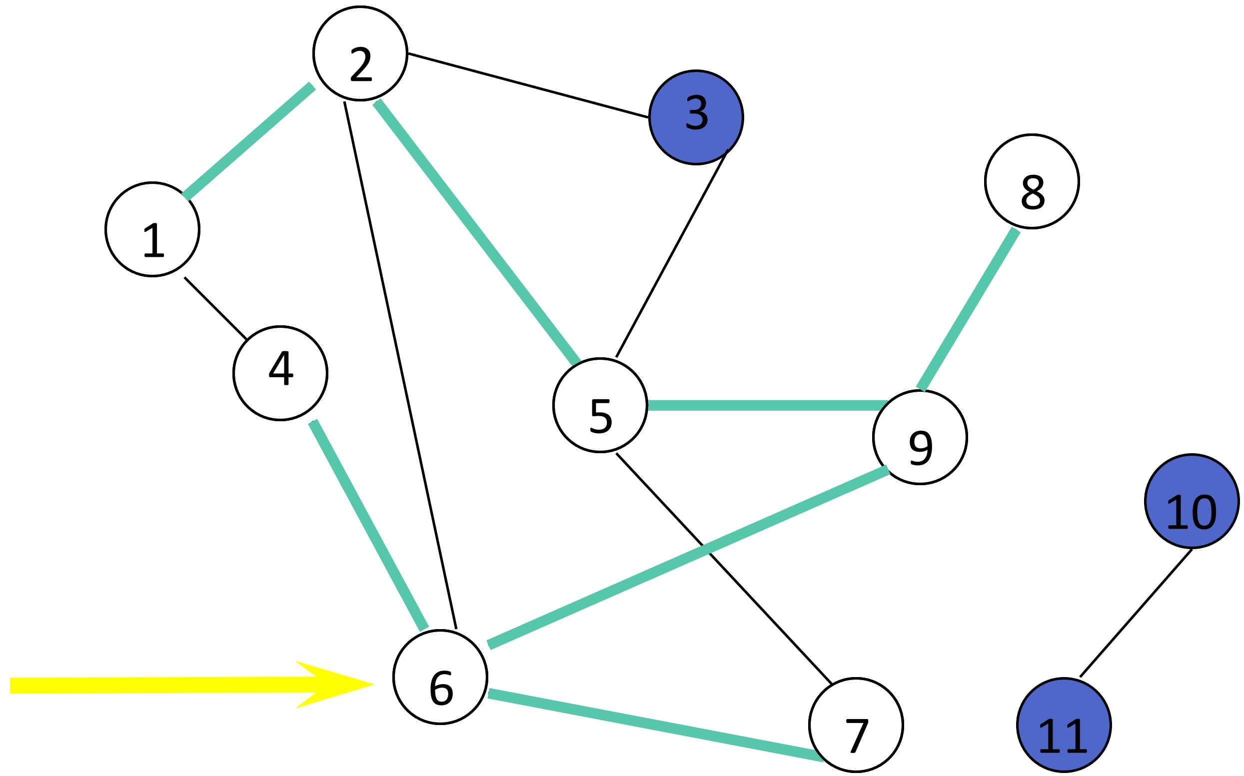
Suppose that vertex 4 is selected.

Depth-First Search



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

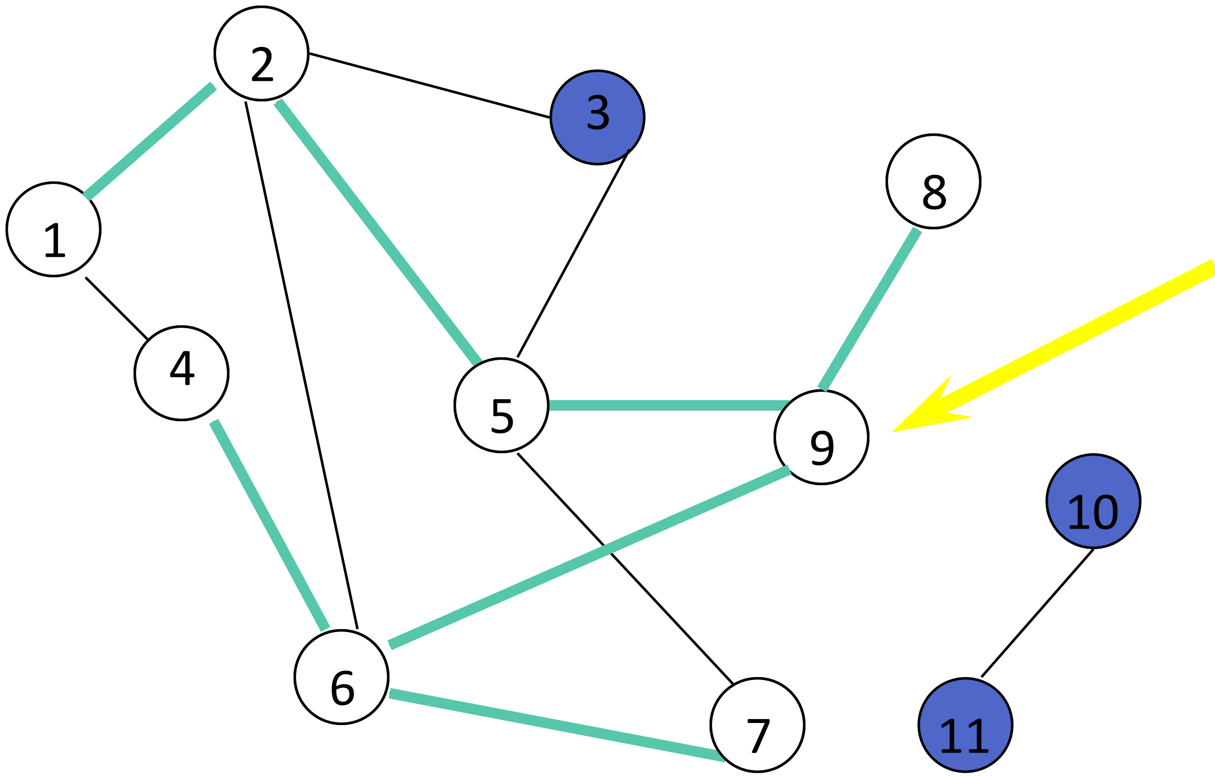
Depth-First Search



Label vertex 7 and return to 6.

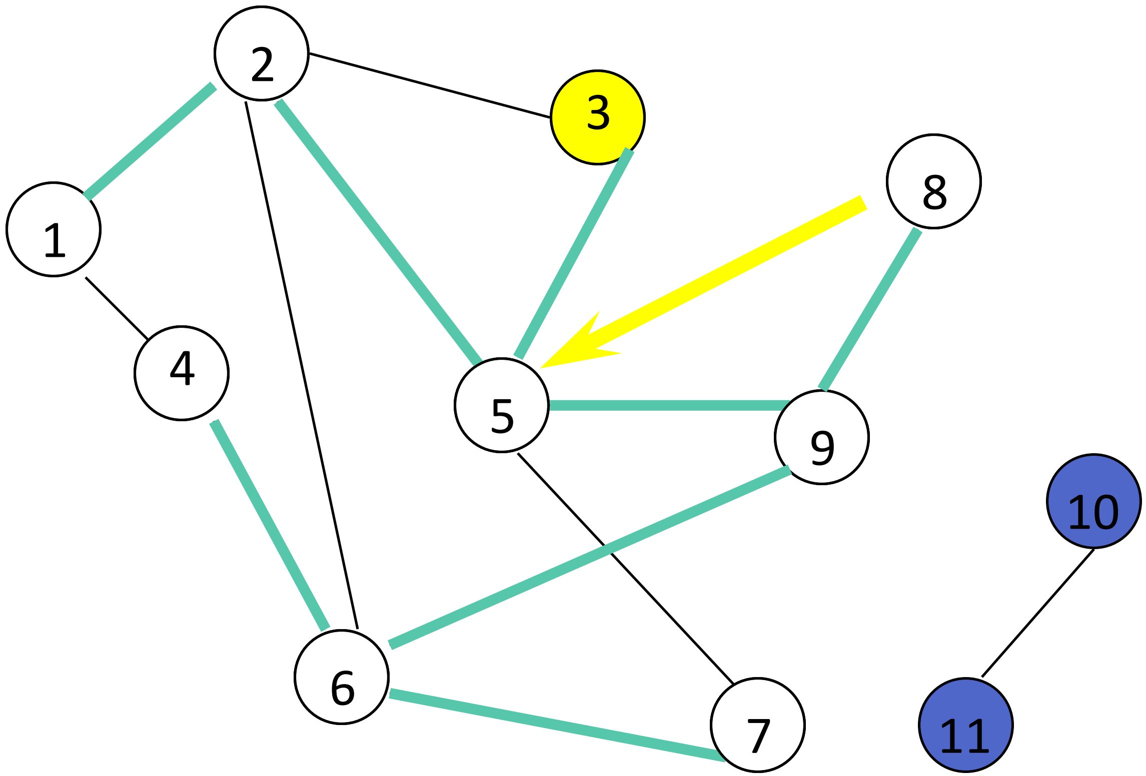
Return to 9.

Depth-First Search



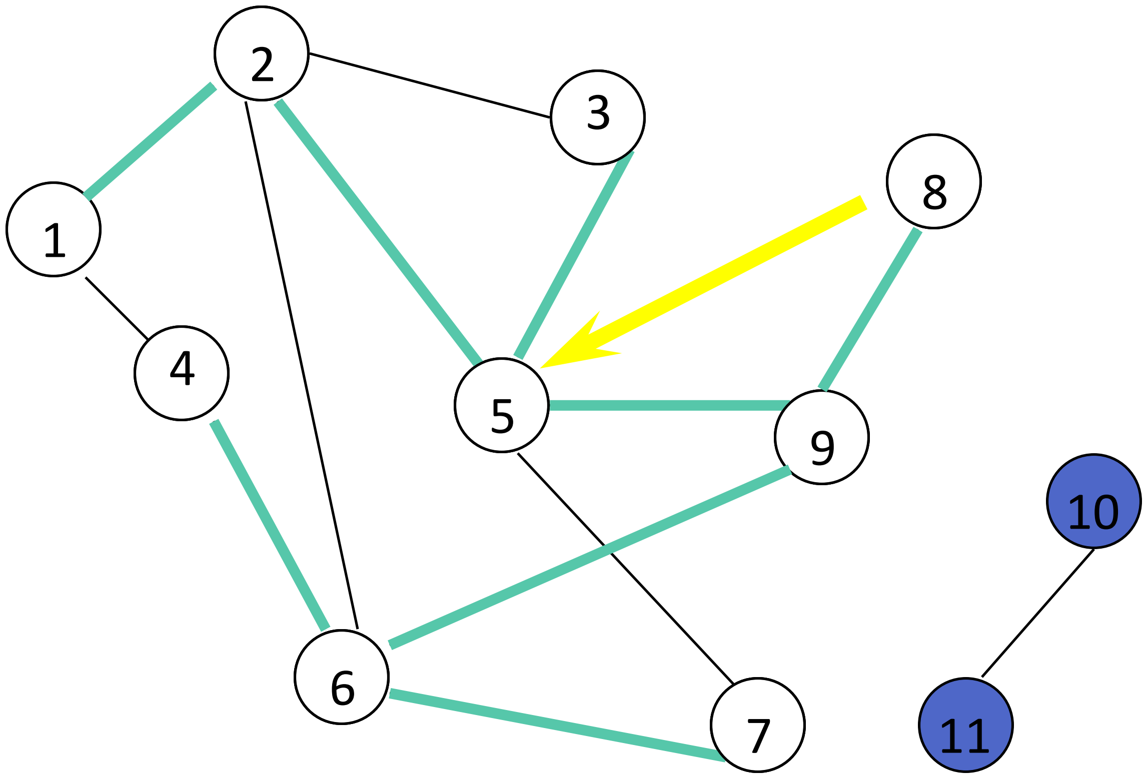
Return to 5.

Depth-First Search



Do a dfs(3).

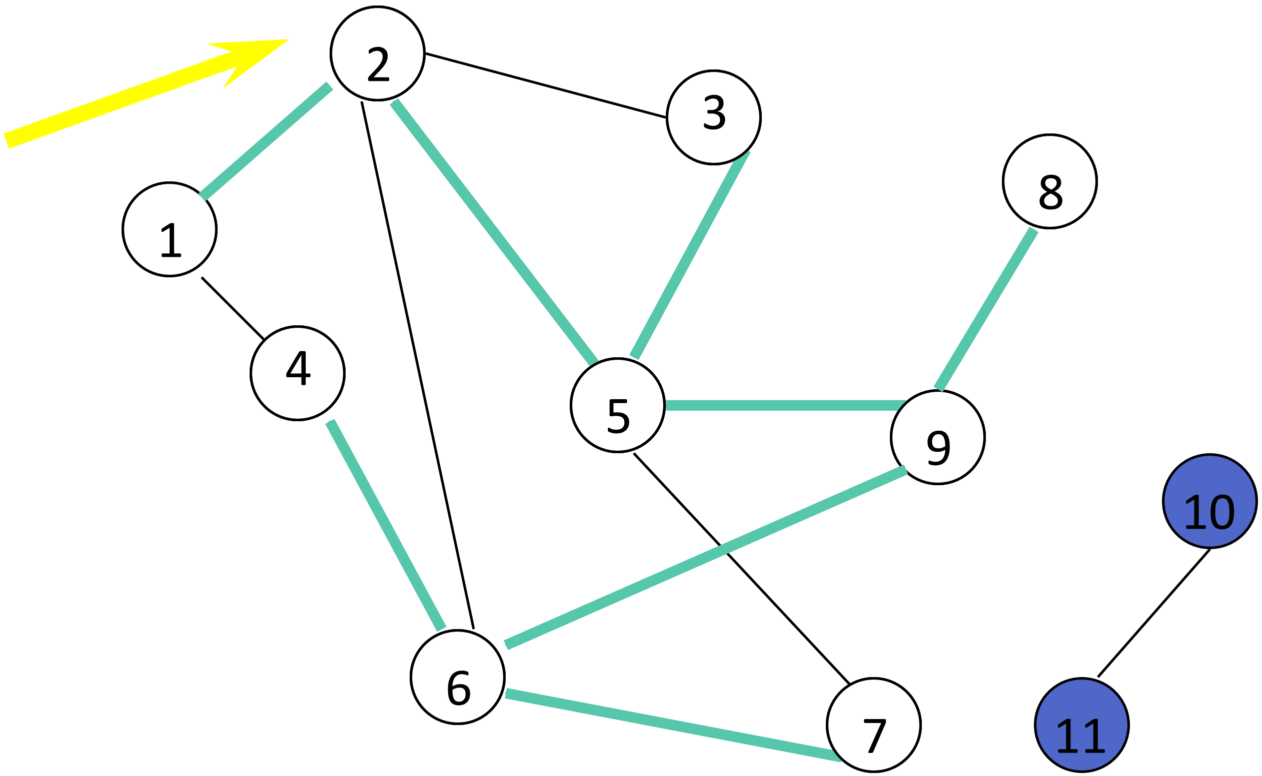
Depth-First Search



Label 3 and return to 5.

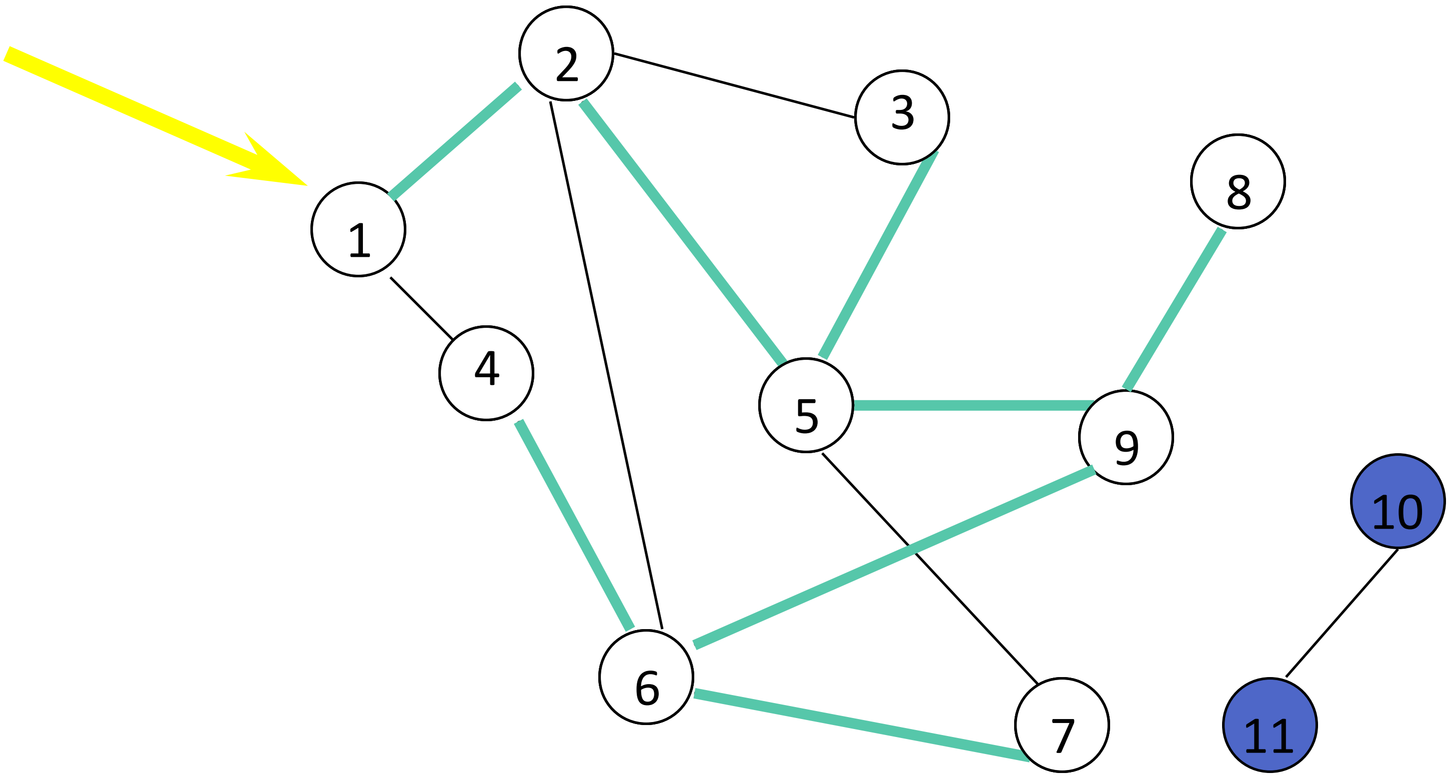
Return to 2.

Depth-First Search



Return to 1.

Depth-First Search



Return to invoking method.

DFS Properties

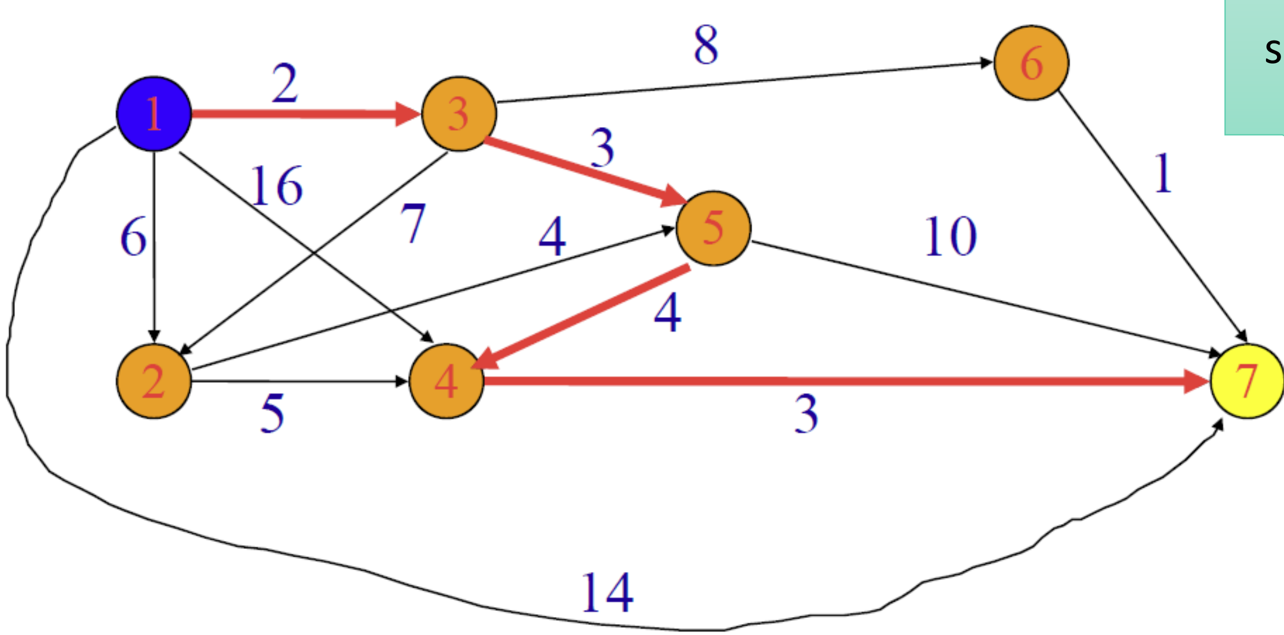
- DFS has same time complexity as BFS
- DFS requires O(h) memory for recursive function stack calls while BFS requires O(w) queue capacity
- Same properties with respect to path finding, connected components, and spanning trees.
  - Edges used to reach unlabeled vertices define a depth-first spanning 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



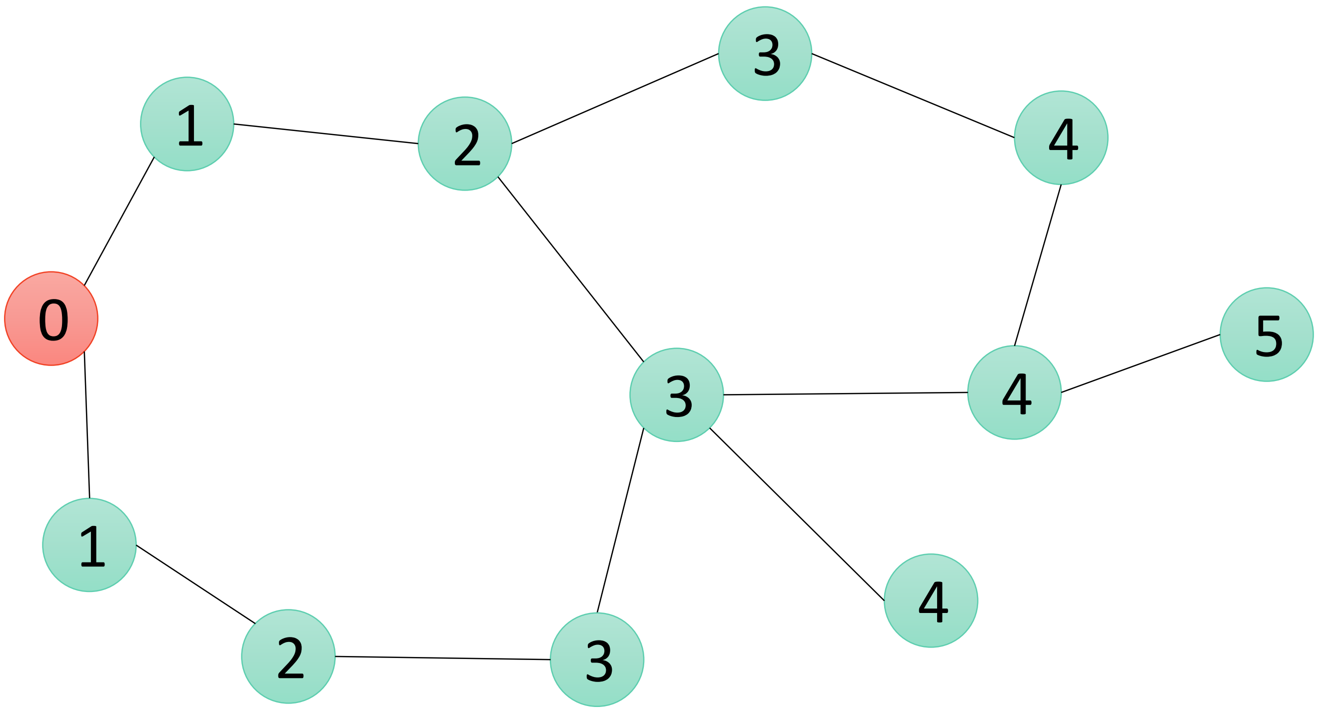
Length of 12 is not shortest path!

Single Source Shortest Path

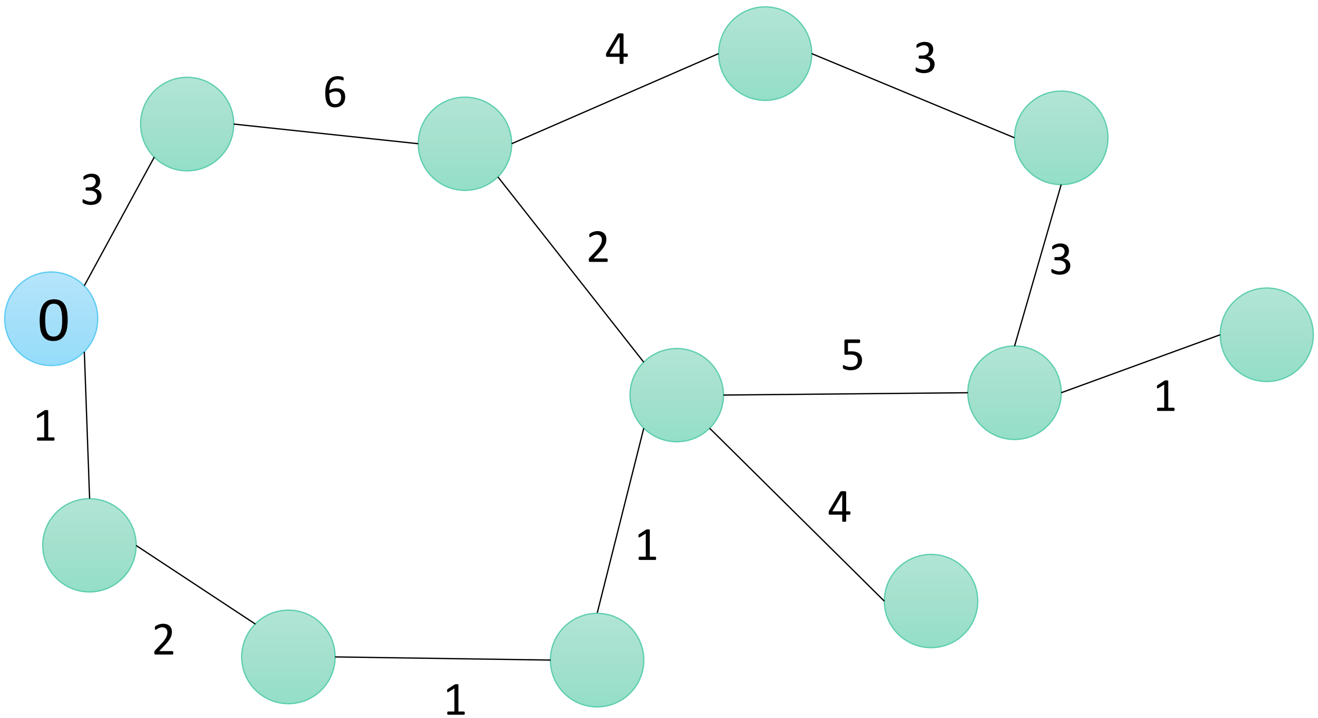
- Shortest distance from one source vertex to all destination vertices
- Is there a simple way to solve this?
- …Say if you had an unit-weighted graph?

- Just do Breadth First Search (BFS)! ☺

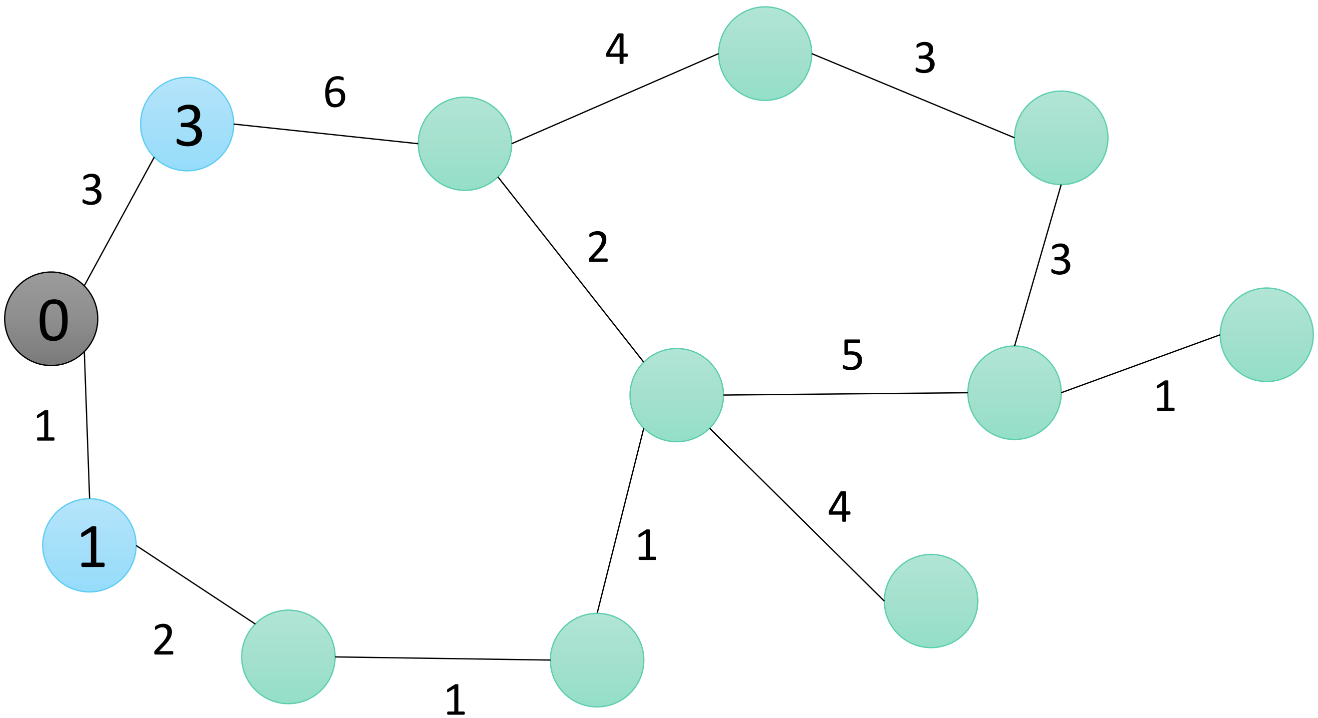
SSSP: BFS on Unweighted Graphs



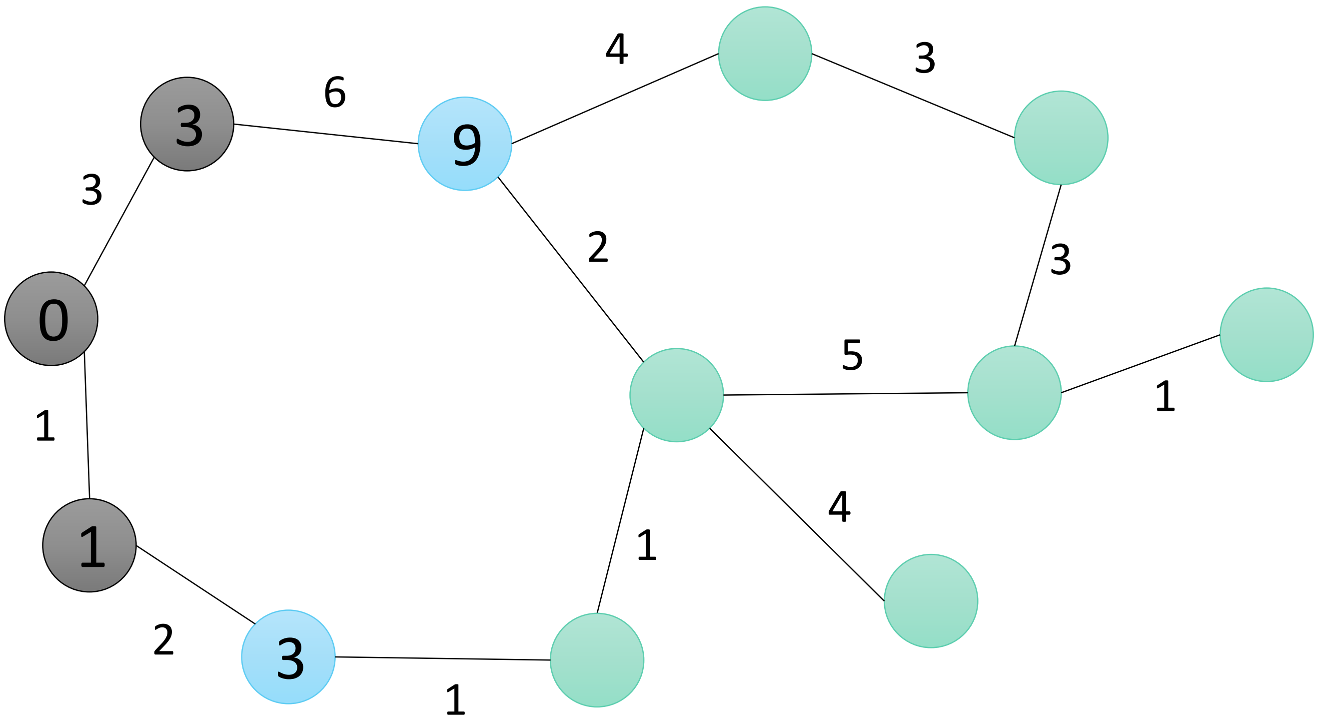
SSSP: *BFS on Weighted Graphs?*



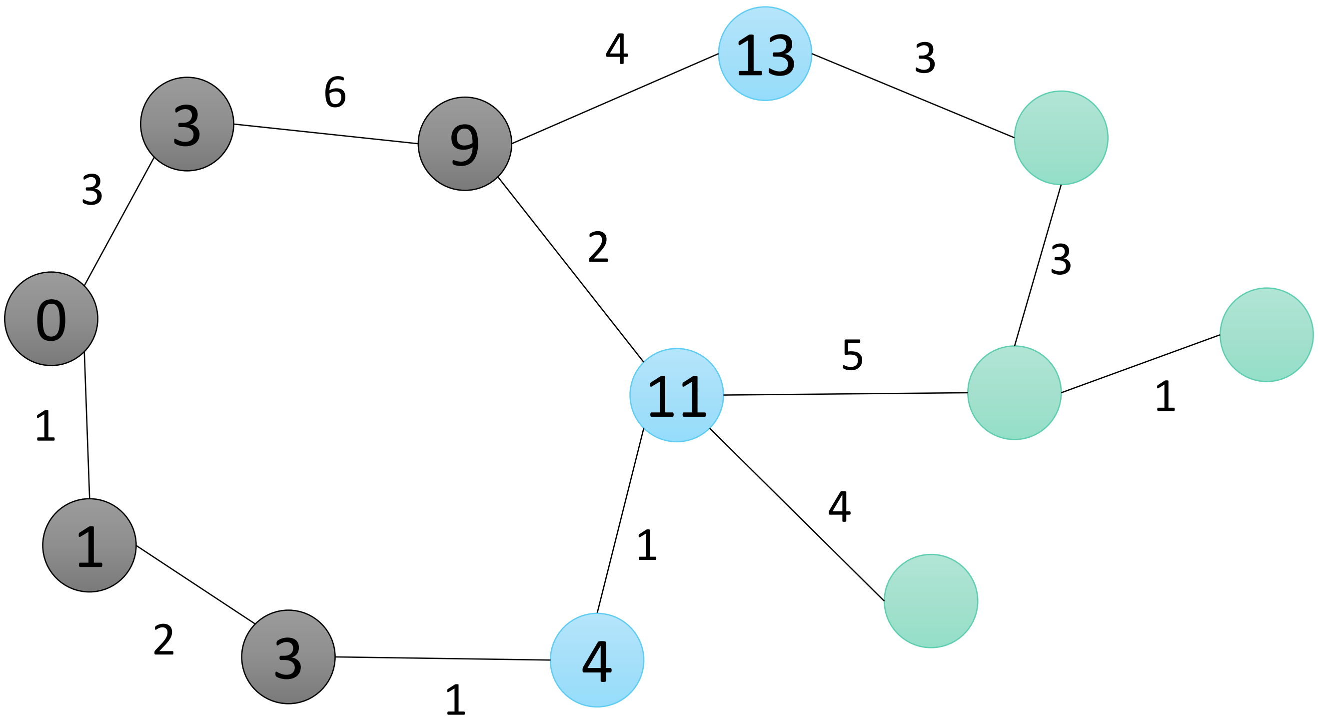
SSSP: BFS on Weighted Graphs?



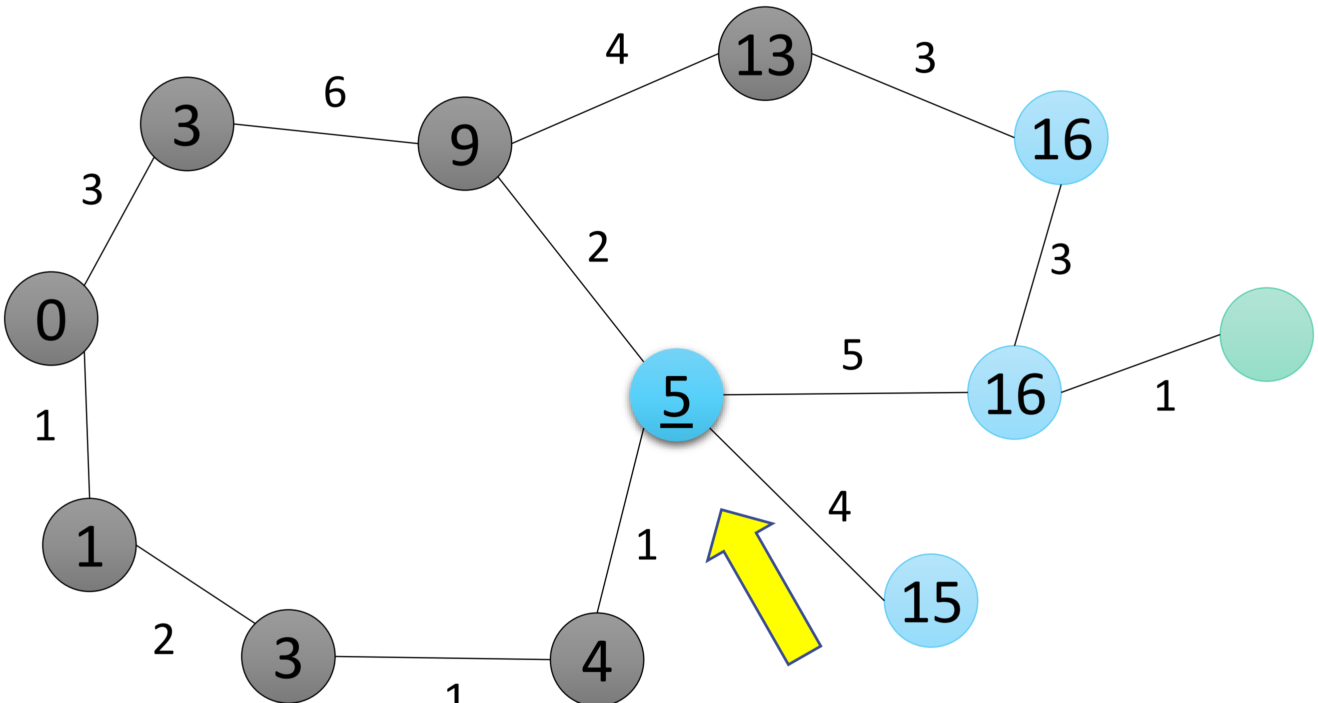
SSSP: BFS on Weighted Graphs?



SSSP: BFS on Weighted Graphs?

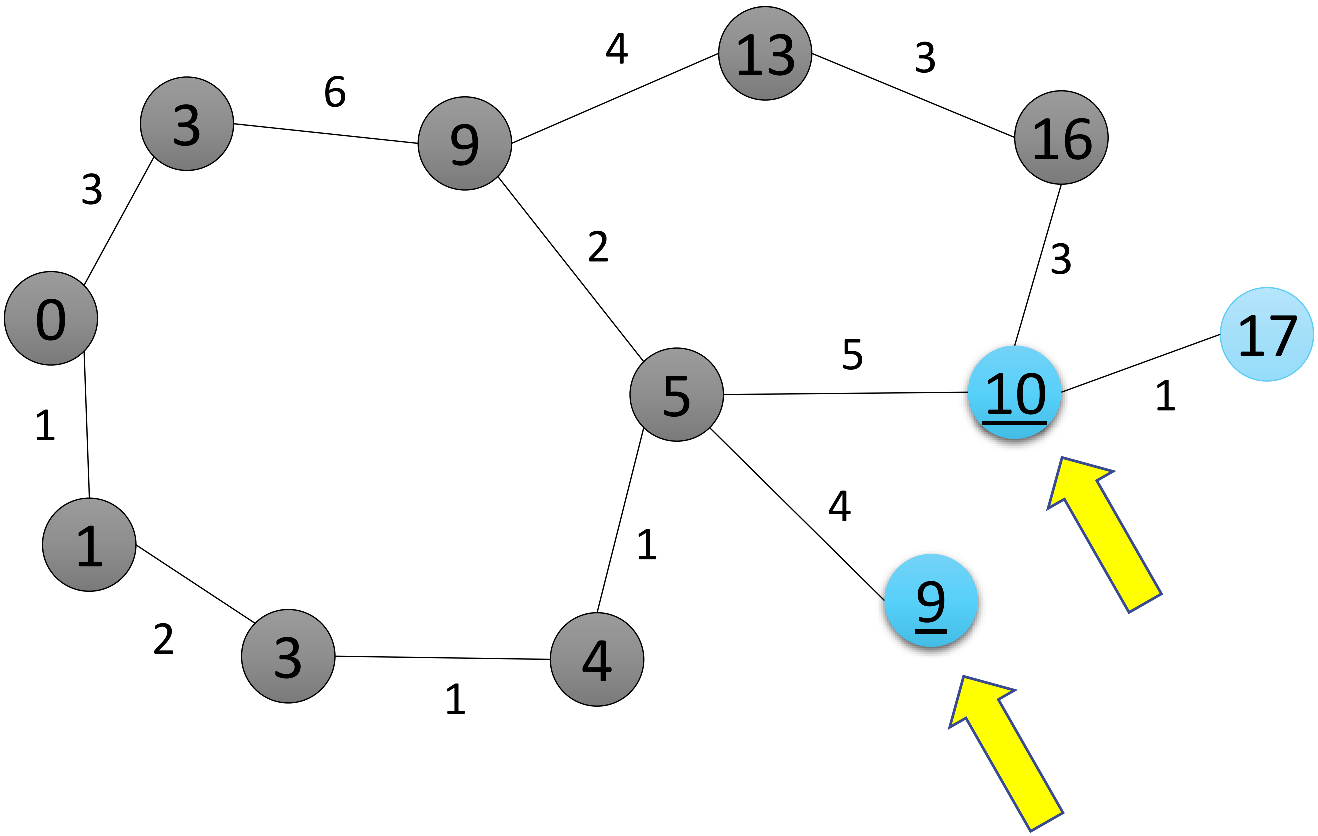


SSSP: BFS on Weighted Graphs?

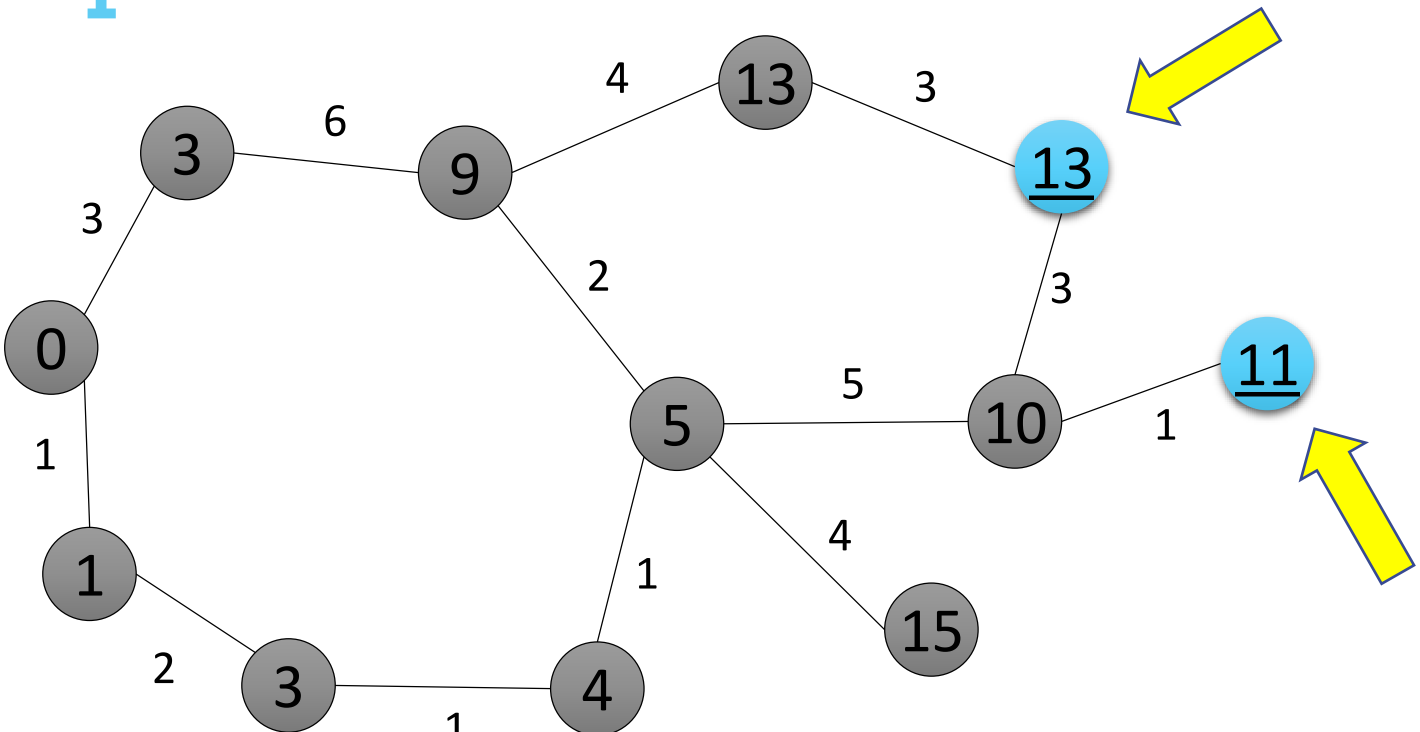


Revisit, recalculate, re-propagate…
cascading effect

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)

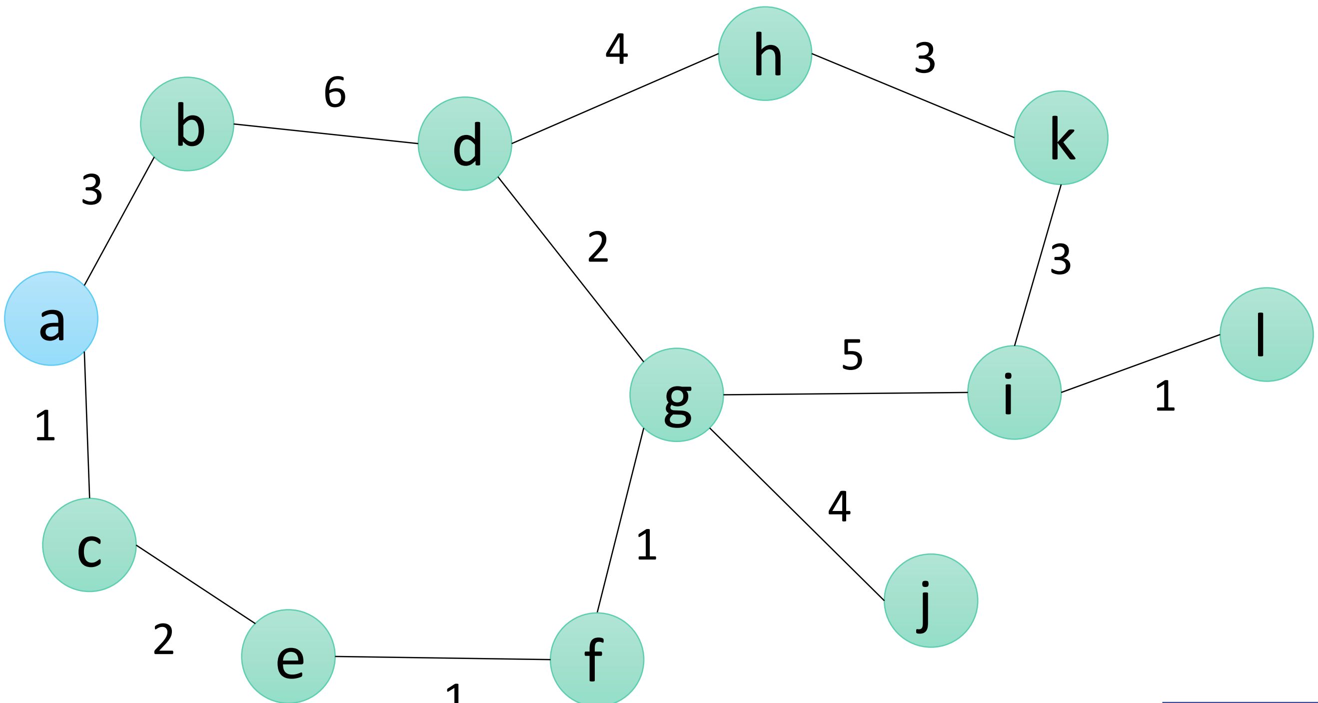
- Prioritize the vertices to visit next
  - Pick "unvisited" vertex with *shortest distance* from source
- Do not visit vertices that have already been visited
  - Avoids false propagation of distances

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

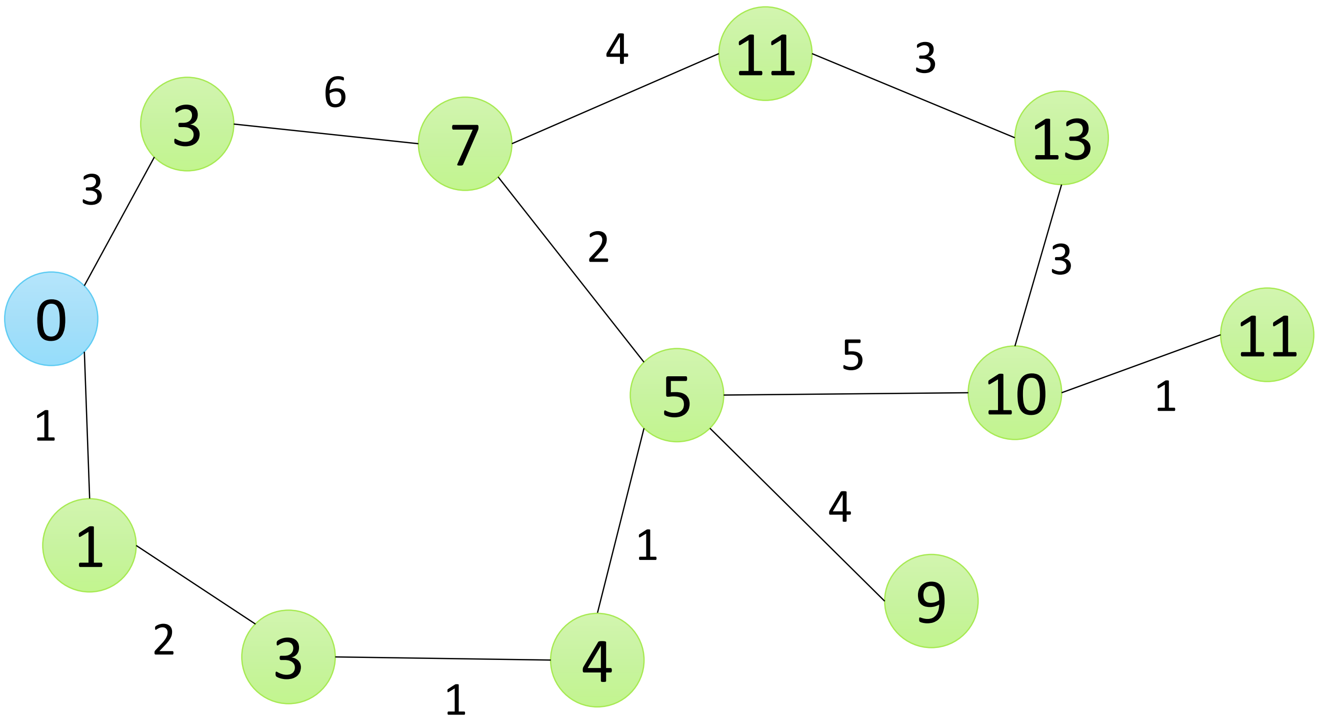
Only change relative to BFS!

SSSP on Weighted Graphs



Work out!

SSSP on Weighted Graphs

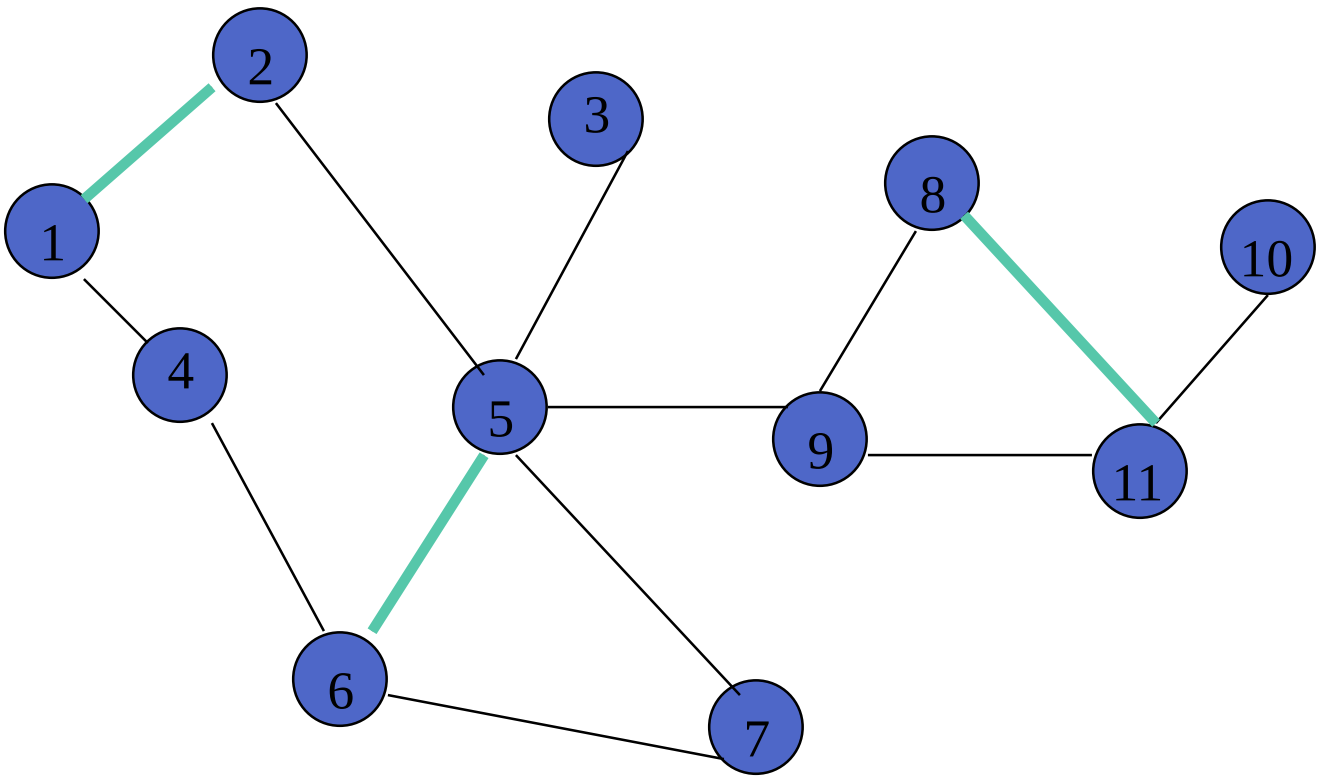


Complexity

- 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)

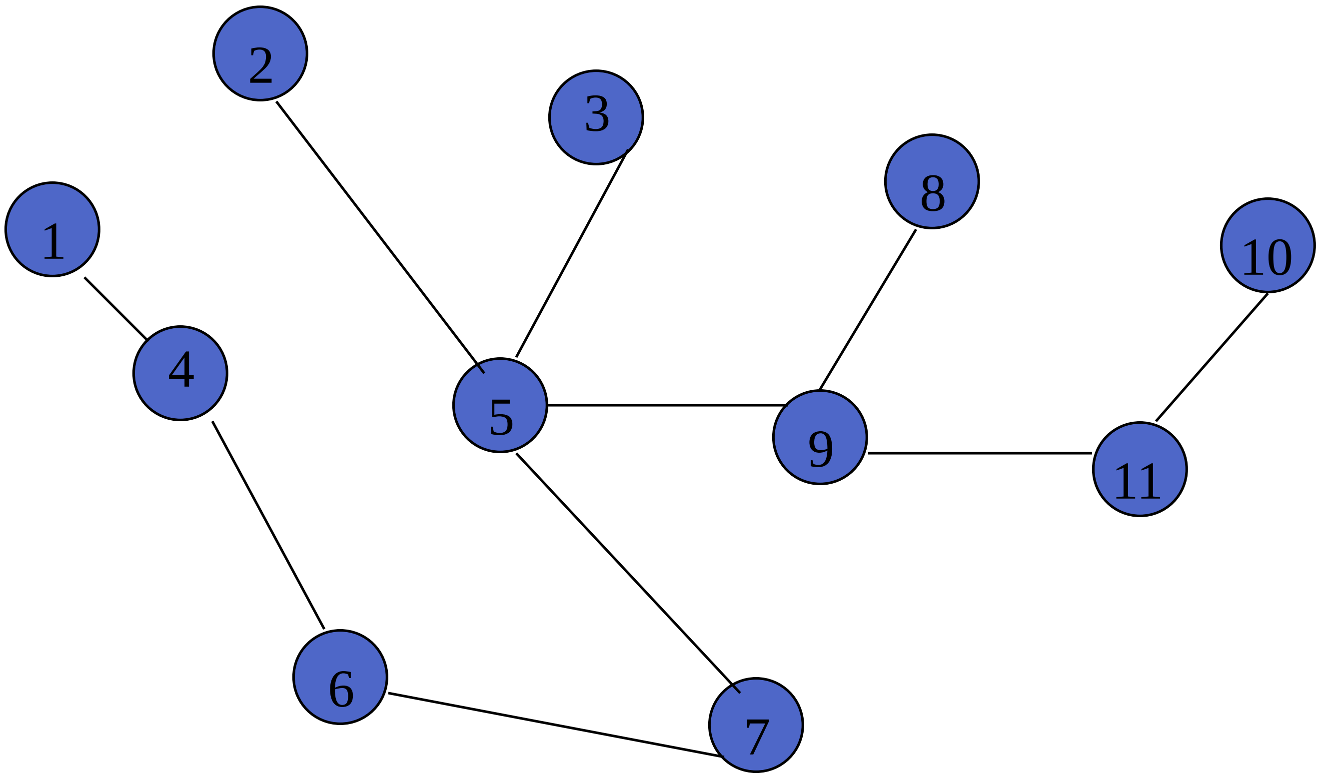
Cycles And Connectedness

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.



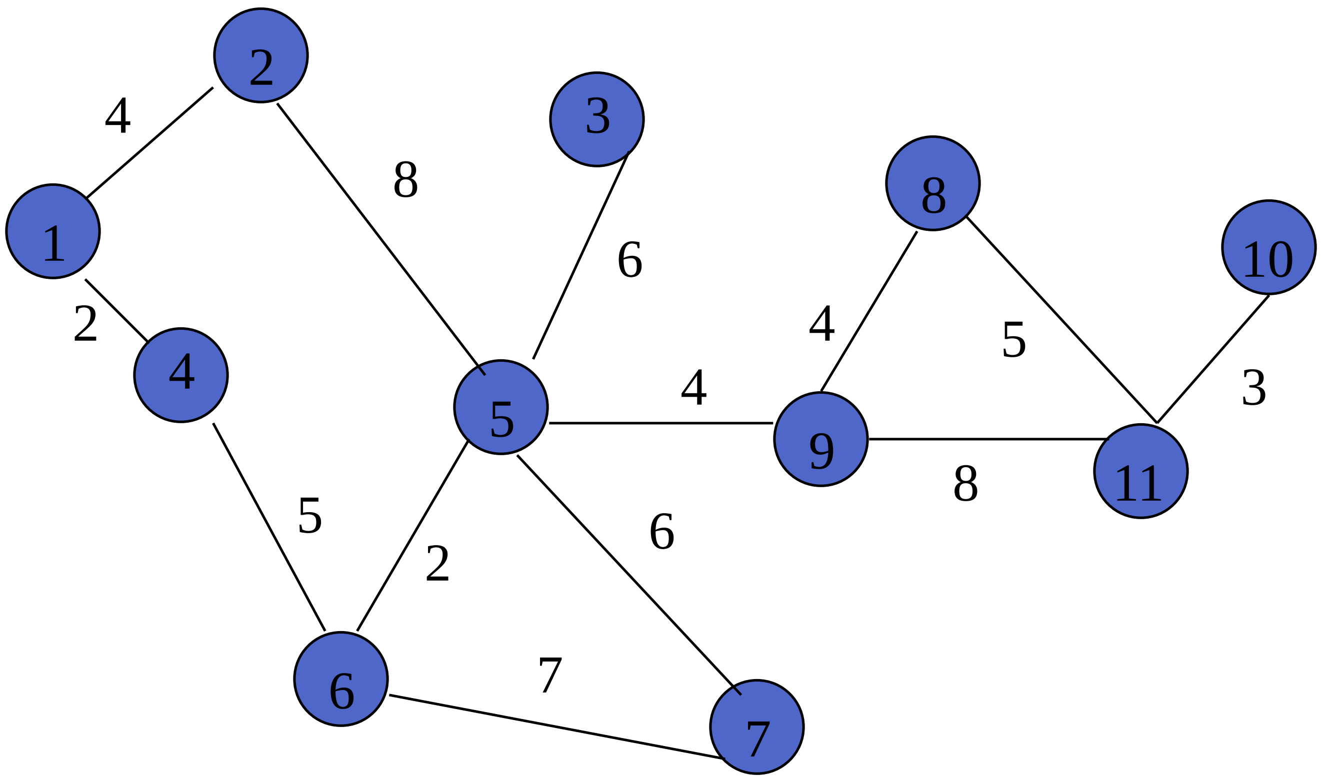
Spanning Tree

- 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.

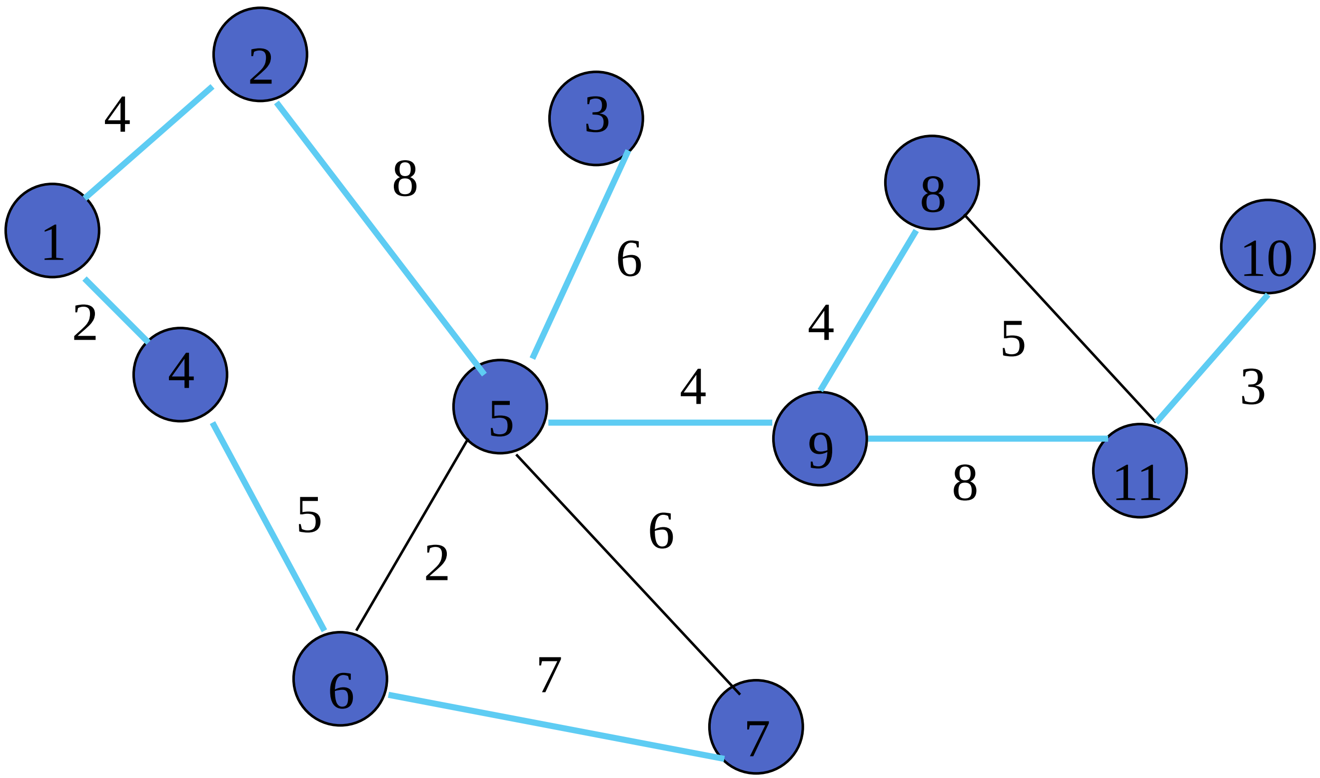
Minimum Cost Spanning Tree

- Tree cost is sum of edge weights/costs.



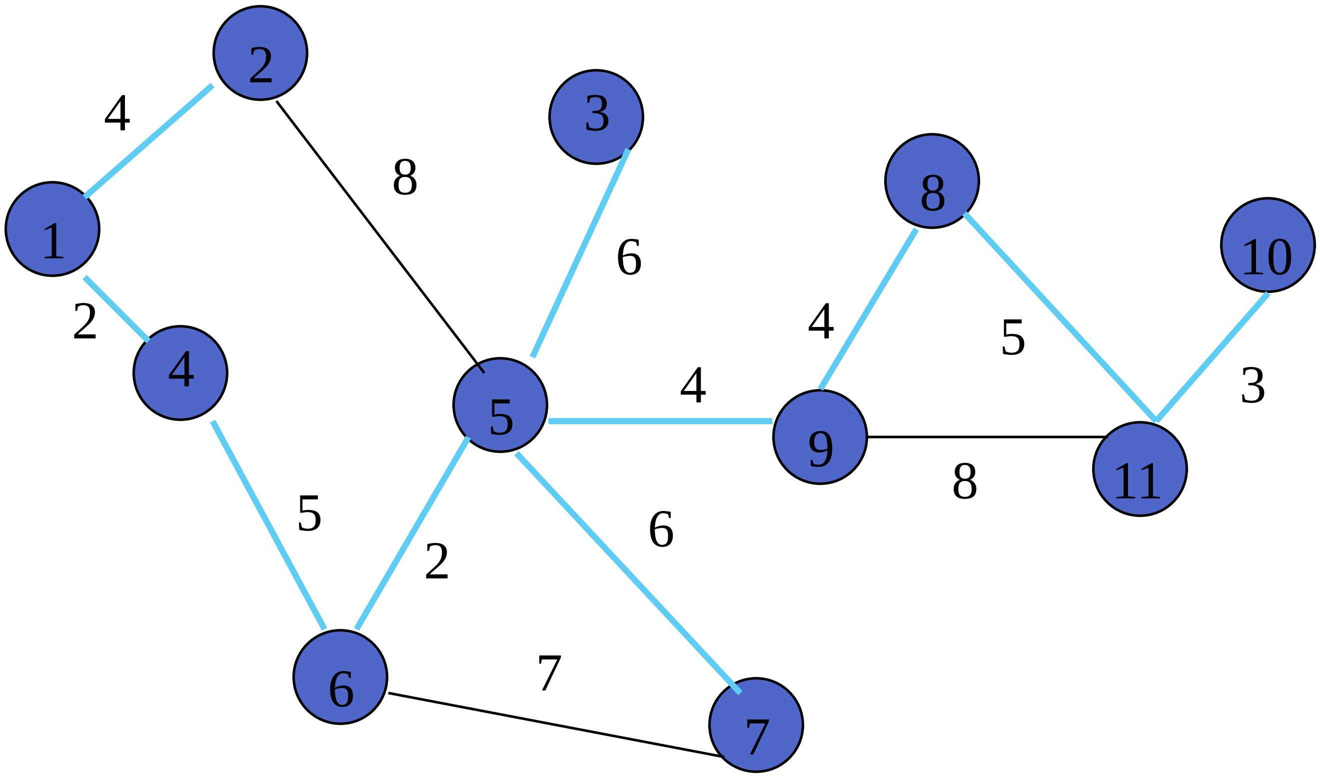
A Spanning Tree

A Spanning tree, cost = 51.



Minimum Cost Spanning Tree

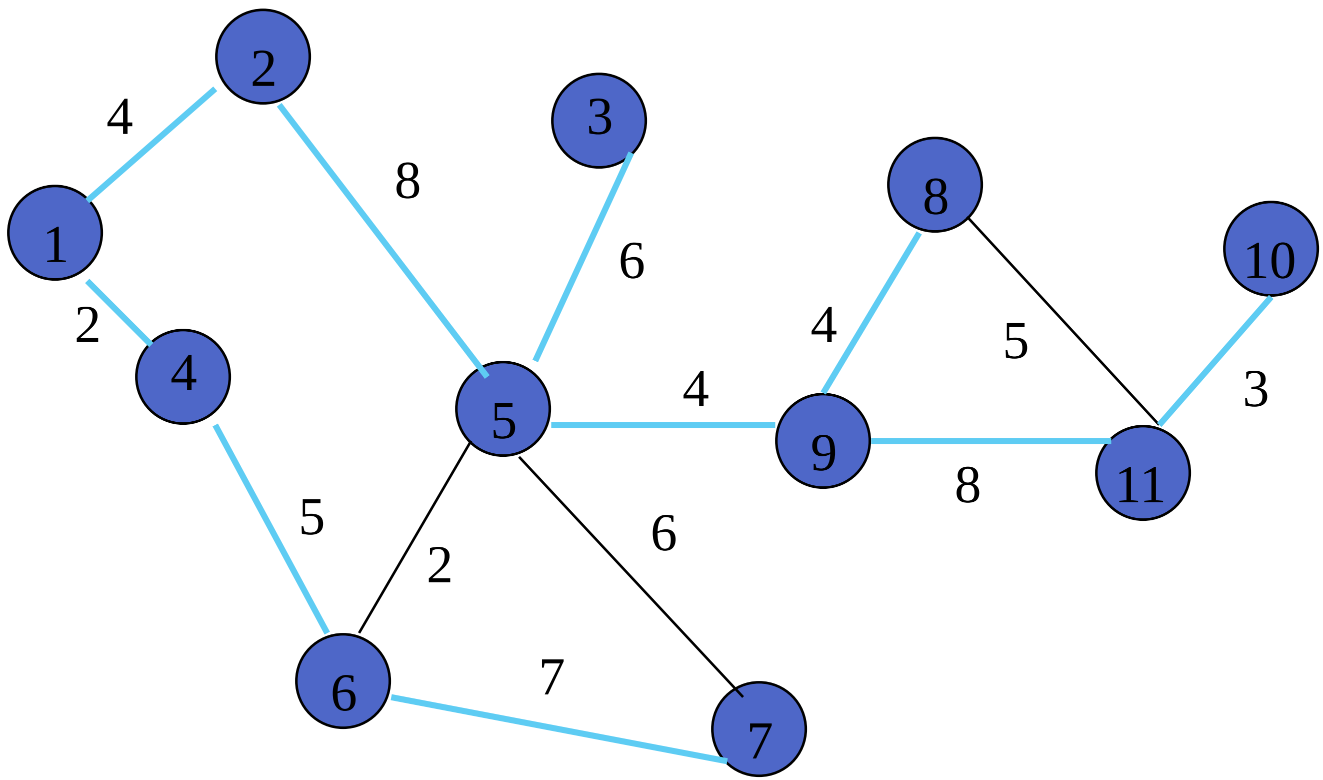
Minimum Spanning tree, cost = 41.



A Wireless Broadcast Tree

Source = 1, weights = needed power.

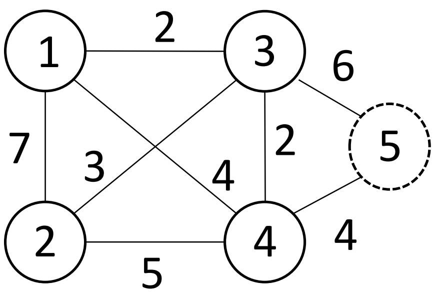
Cost = 4 + 8 + 5 + 6 + 7 + 8 + 3 = 41.



Prim's Minimum Spanning Tree (MST) … Self Study

Given Input Graph

G



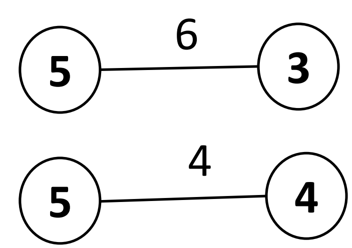
Select Vertex Randomly
e.g., Vertex 5

Initialize Empty Graph T with Vertex 5

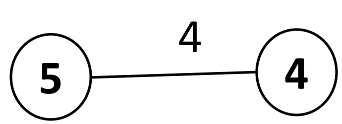
T

Repeat until all vertices are added to T

Select a list of edges L from G such that at most ONE vertex of each edge is in T

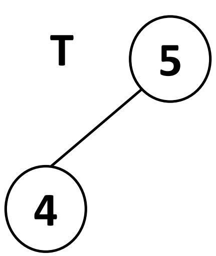


From L select the edge X with minimum weight



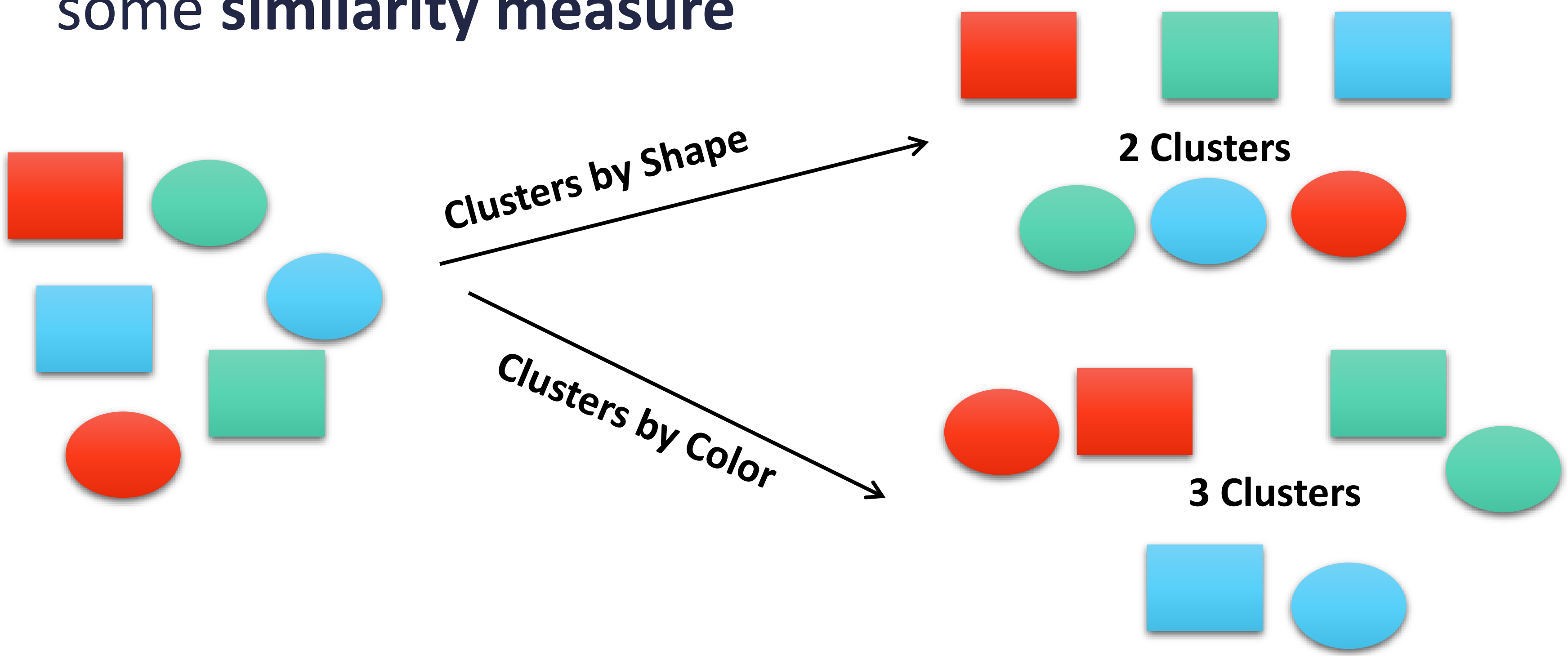
Add X to T

T



Graph Clustering

- **Clustering**: 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**



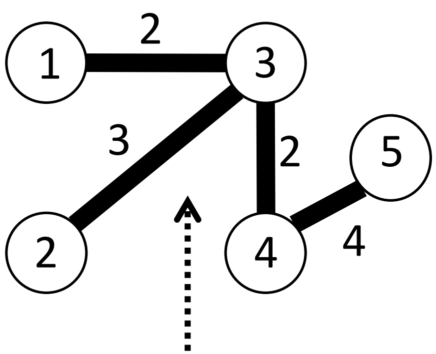
Introduction to Graph Cluster Analysis, https://www.csc.ncsu.edu/faculty/samatova

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

Introduction to Graph Cluster Analysis, https://www.csc.ncsu.edu/faculty/samatova

Graph Clustering: k-spanning Tree

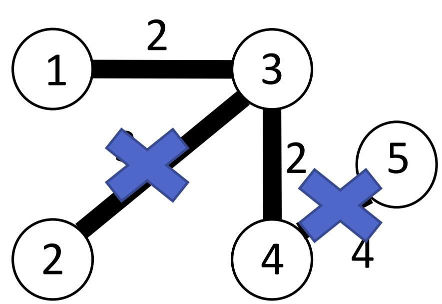


Minimum Spanning Tree

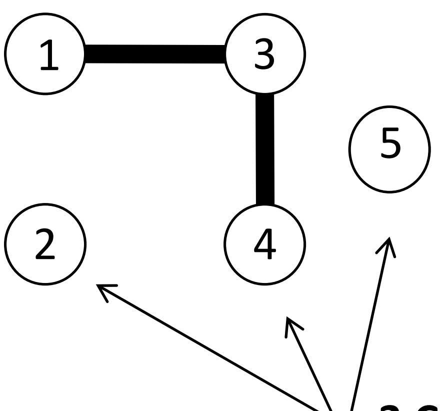
Remove k-1 edges with highest weight

Note: *k – is the number of clusters*

E.g., k=3



E.g., k=3



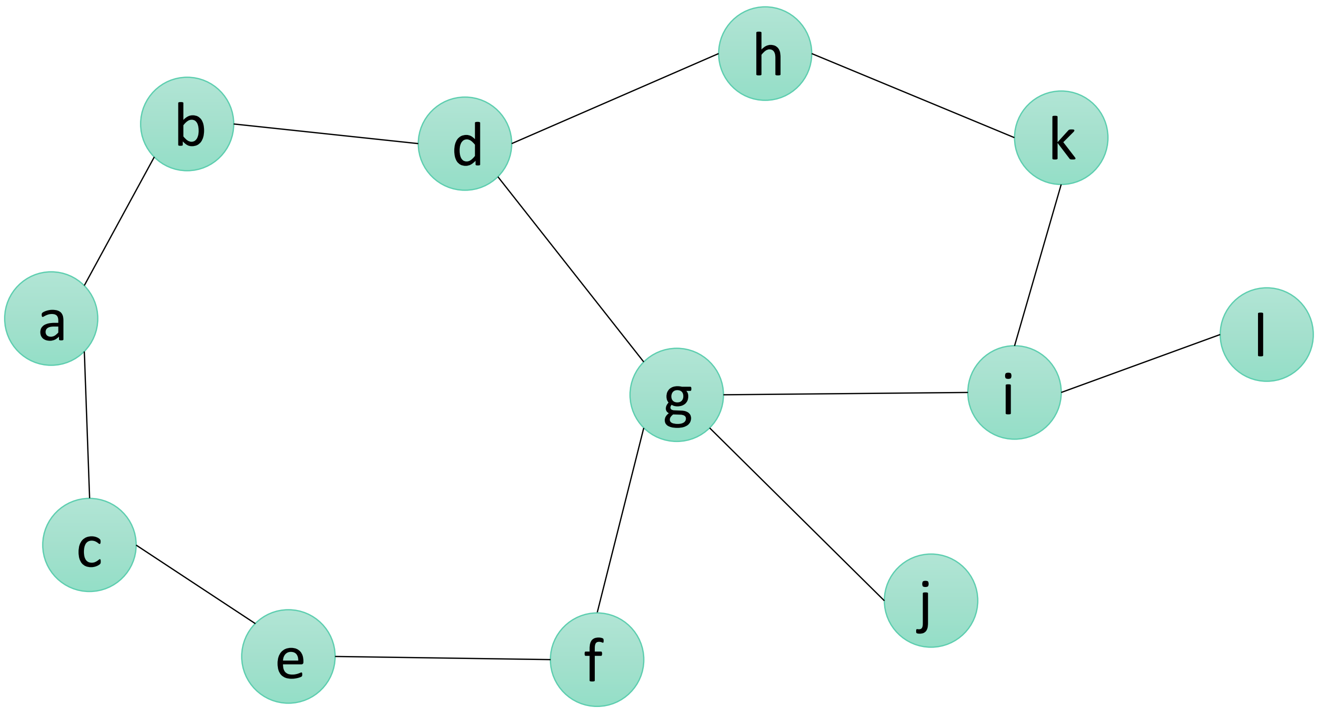
3 Clusters

Graph Clustering: k-means Clustering

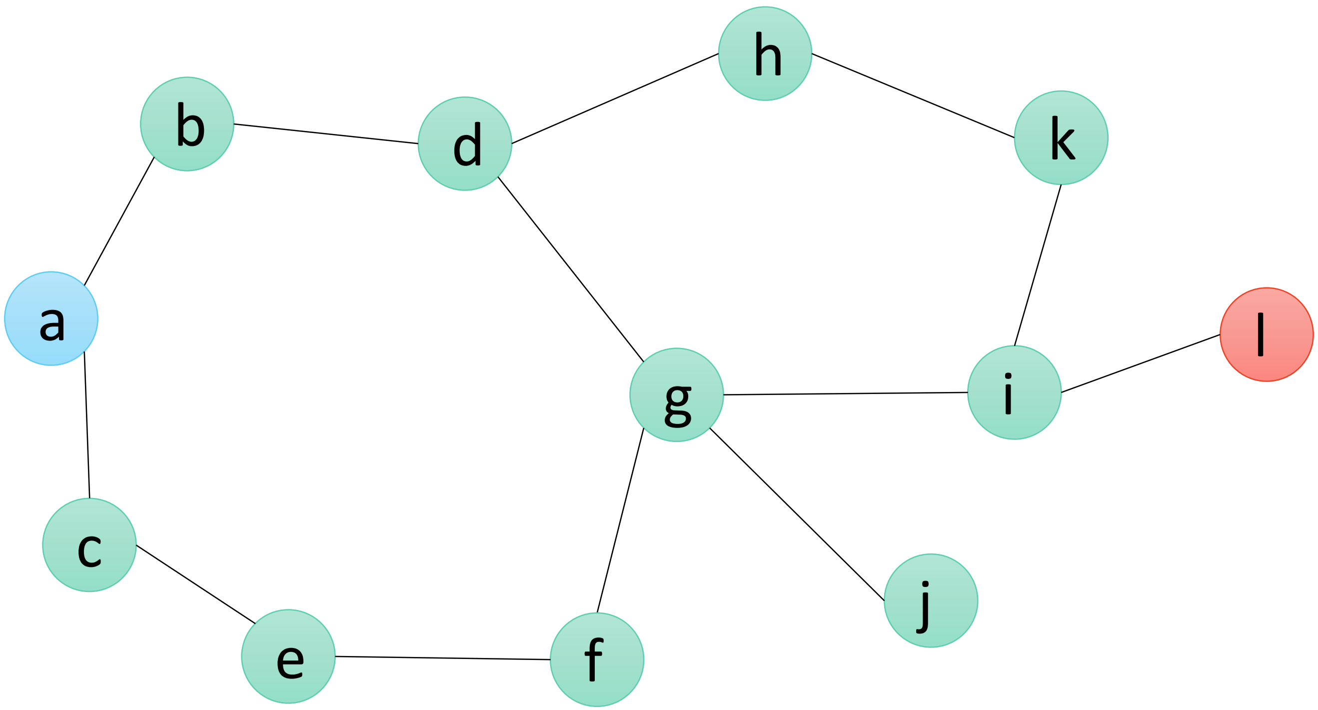
1. Identify k random vertices as centers, label them with unique colors
2. Start BFS traversal from each center, one level at a time
3. Label the vertices reached from each BFS center with its colors
4. If multiple centers reach the same vertex at same level, pick one of the colors
5. Continue propagation 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

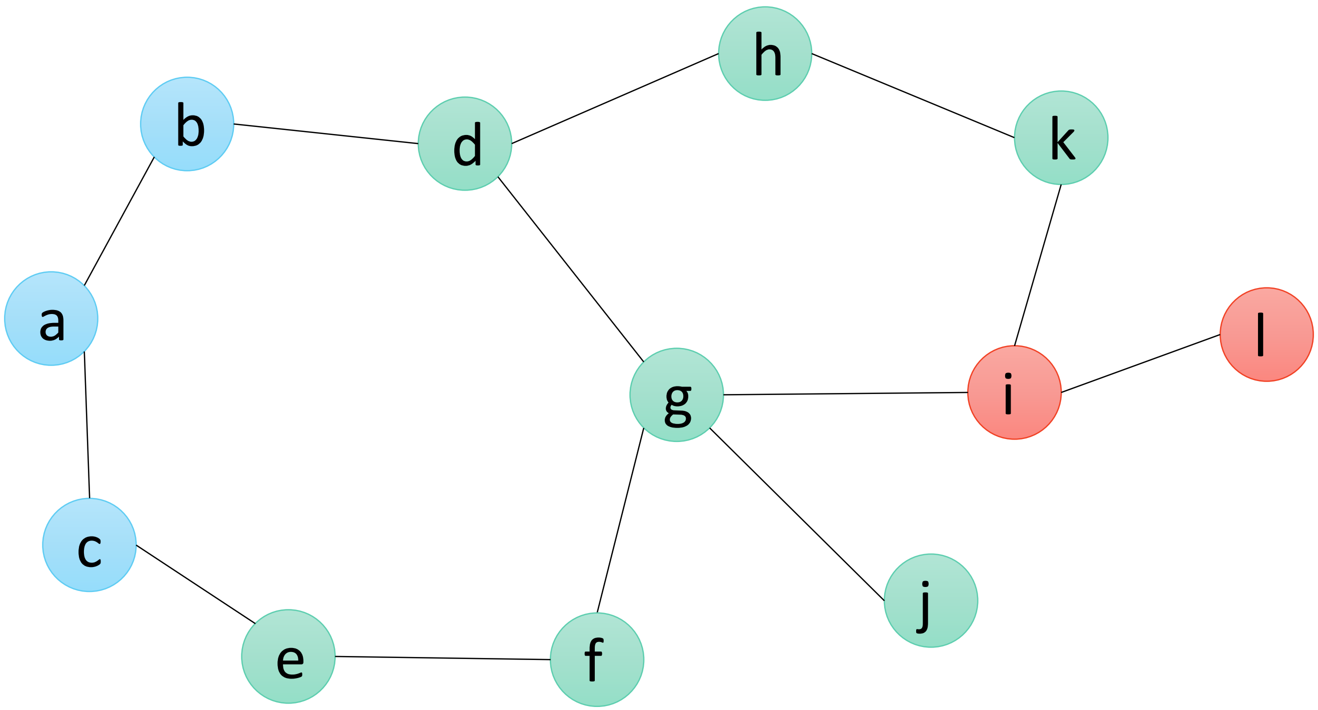


K-Means Clustering



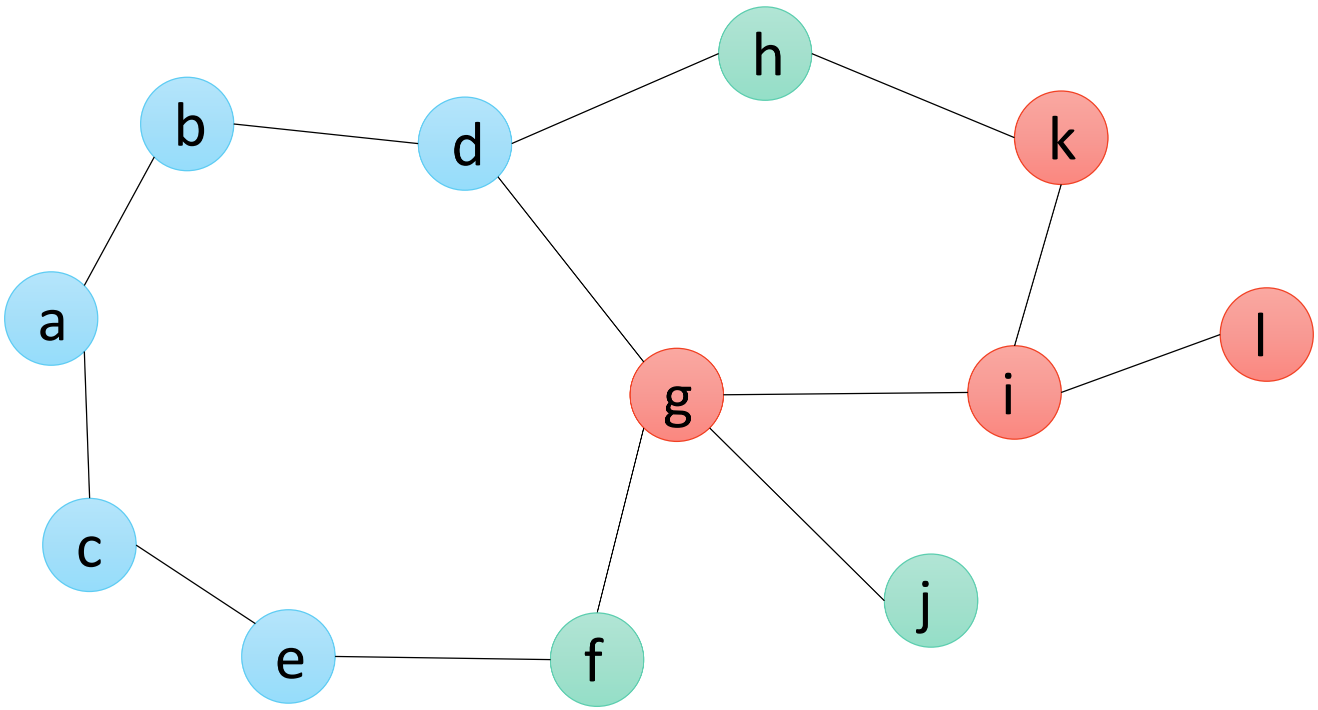
Pick *k* random vertices

K-Means Clustering



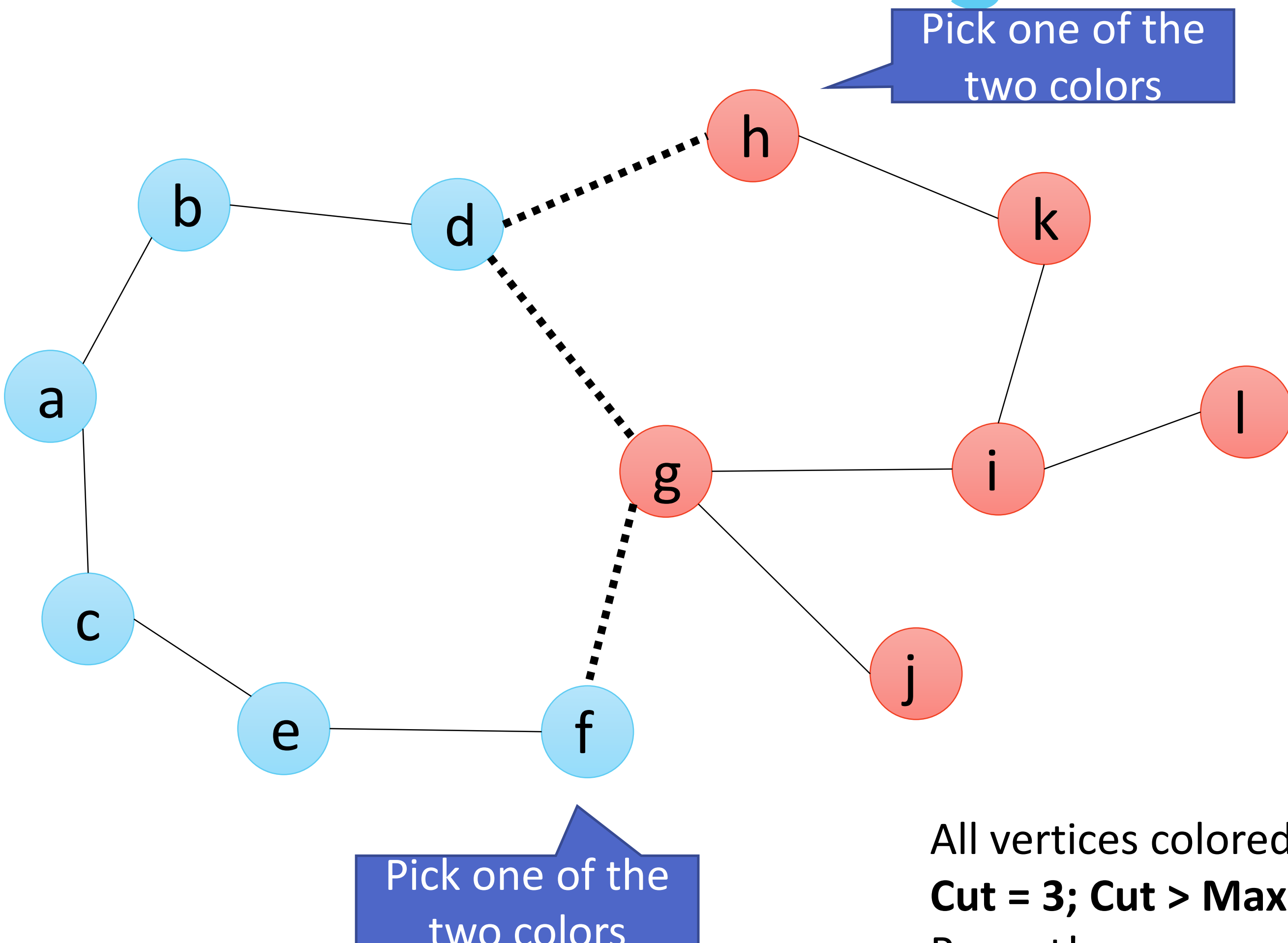
Perform k BFS simultaneously

K-Means Clustering



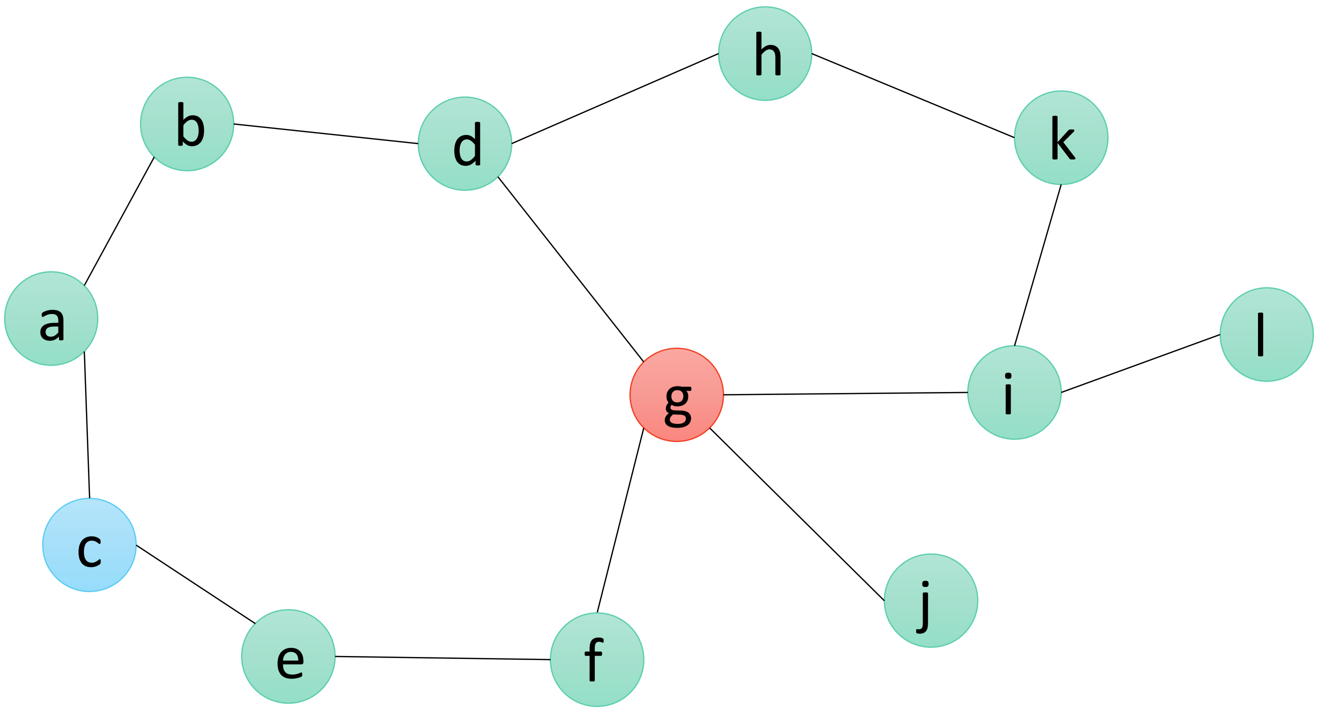
Perform k BFS simultaneously

K-Means Clustering



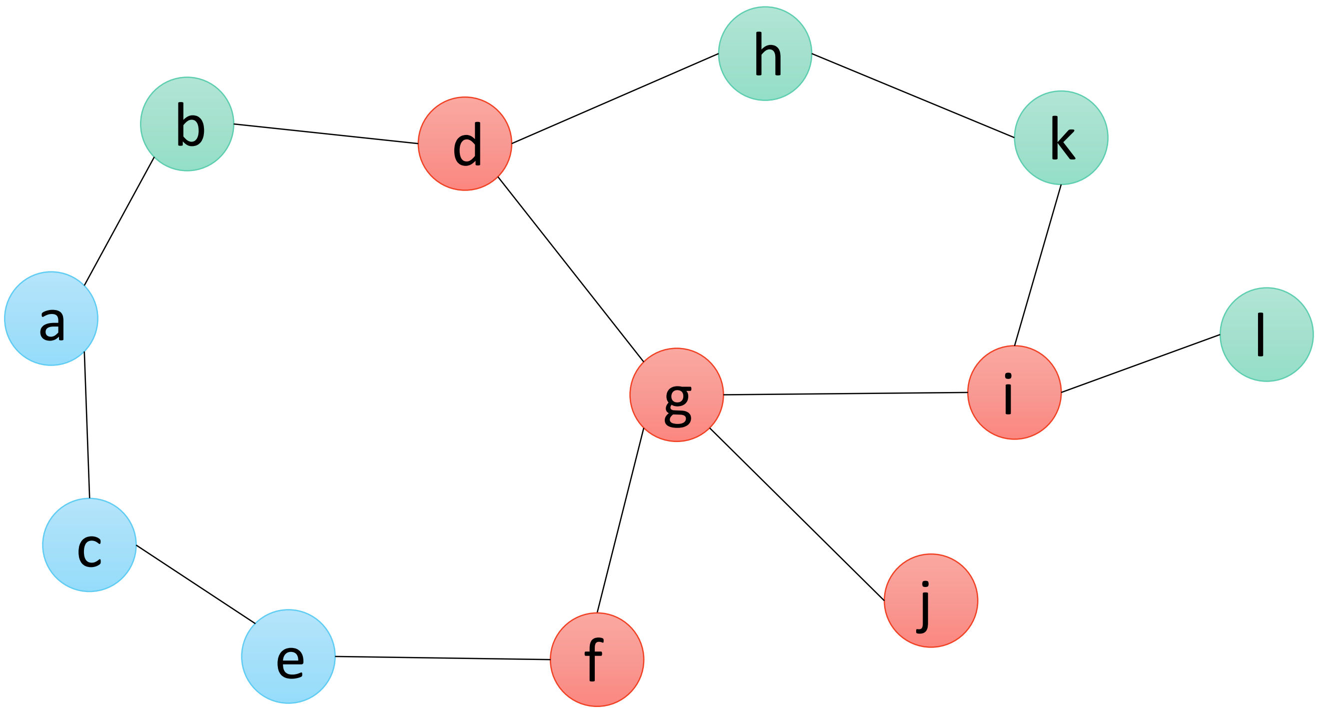
All vertices colored
Cut = 3; Cut > MaxCut
Repeat!

K-Means Clustering



Pick *k* random vertices

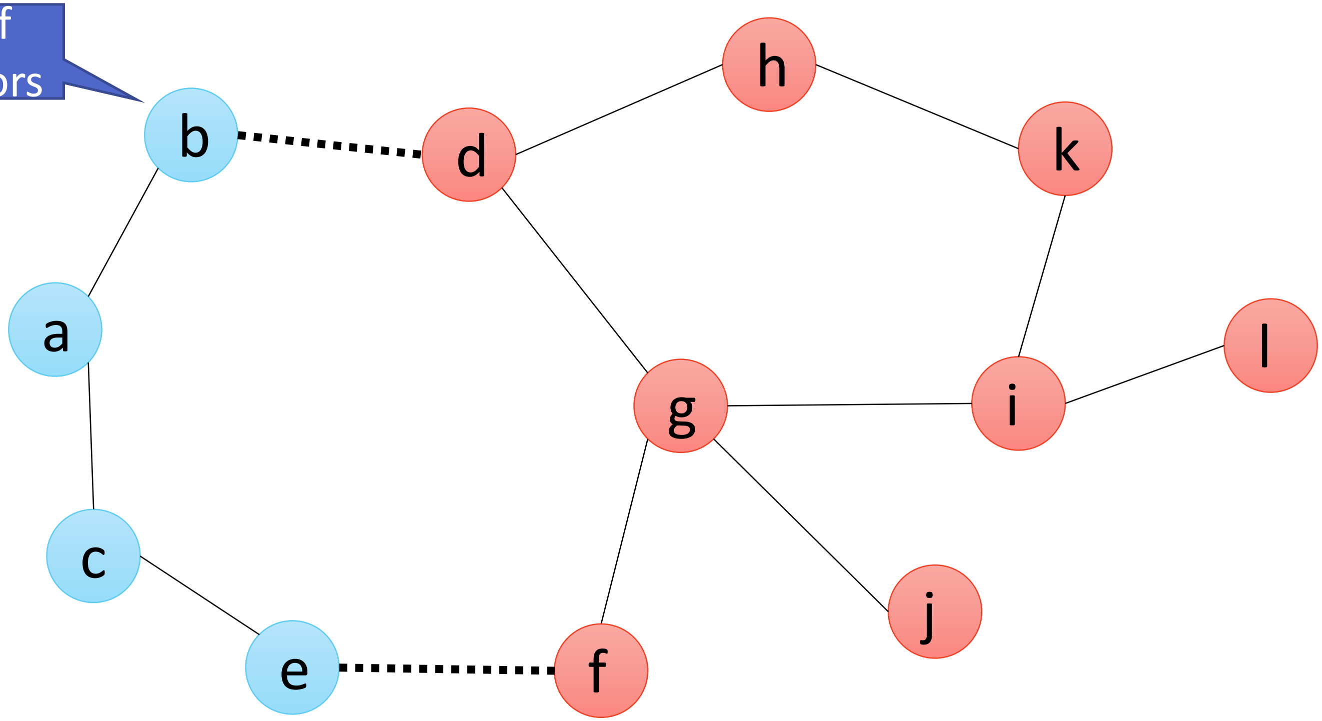
K-Means Clustering



Perform k BFS simultaneously

K-Means Clustering

Pick one of the two colors



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

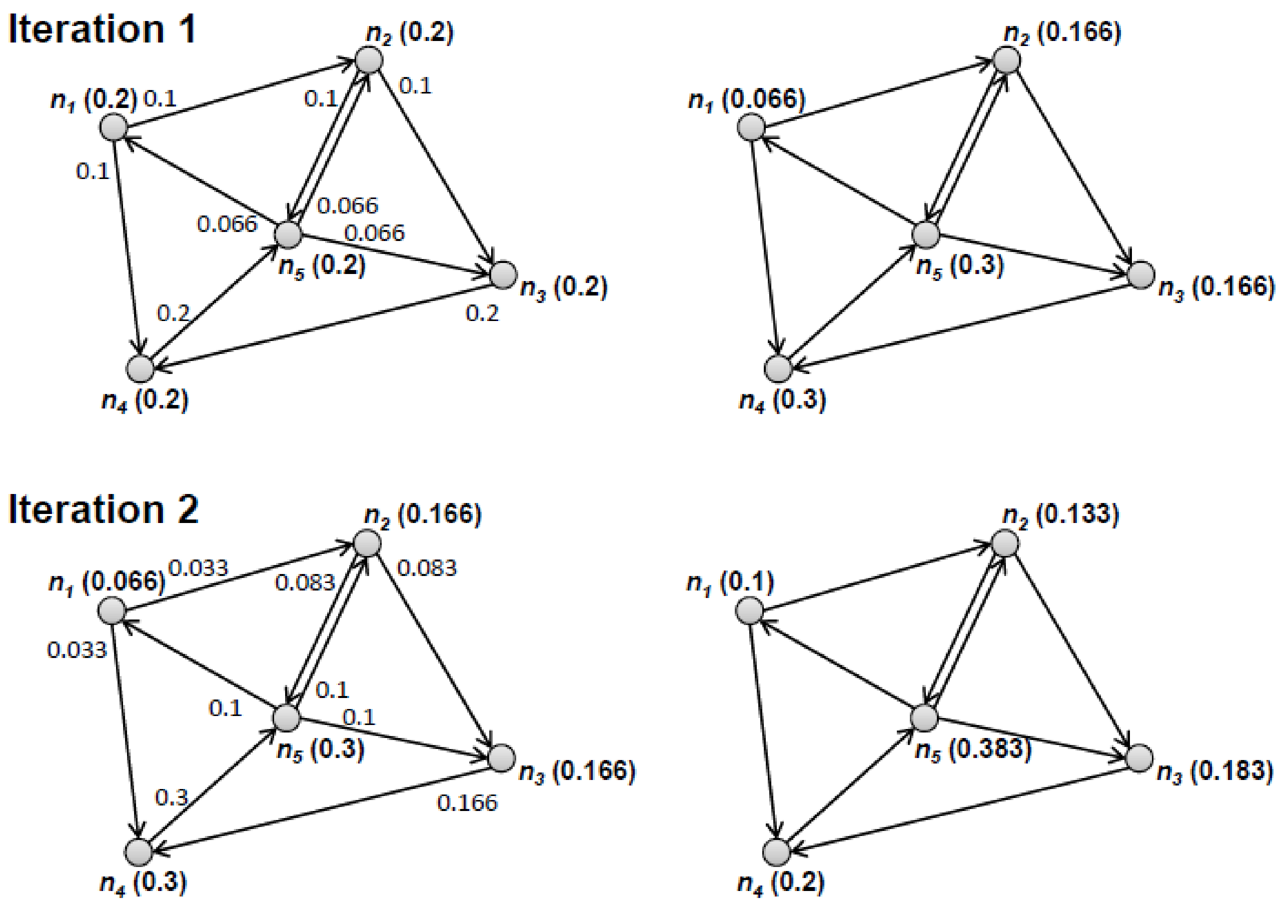
$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

- 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|



Lin, Fig 5.7

Tasks

- Self study
  - **Read**: Graphs and graph algorithms (online sources)
- Finish Assignment 5 by Mon Nov 14 *(100 points)*
- Make progress on CodeChef (100 points)

Questions?