Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 17.

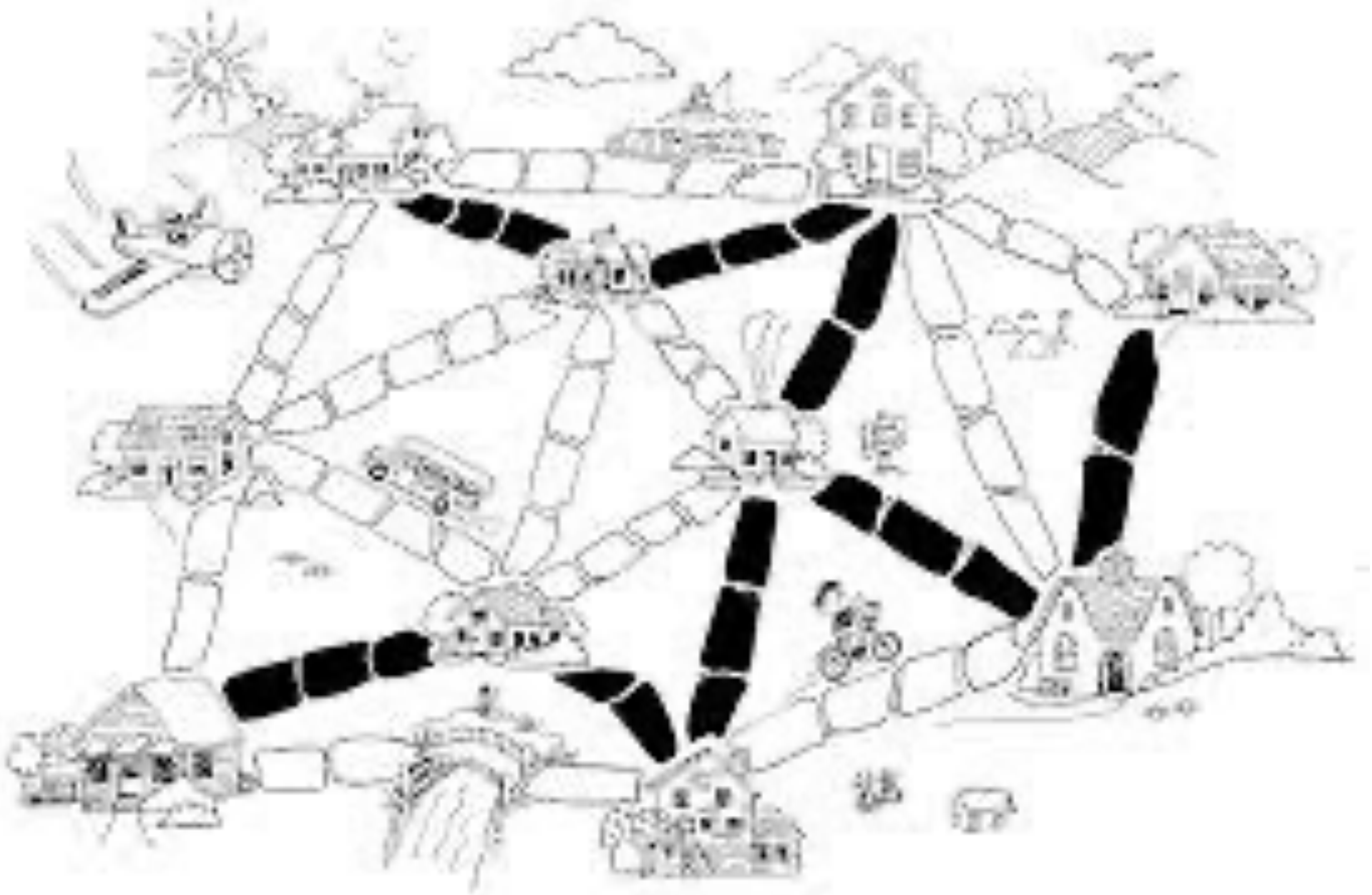
Graph Data Structures and Algorithms - I

M. Oguzhan Kulekci

- Graphs
- How to represent graphs
- Graph Traversals

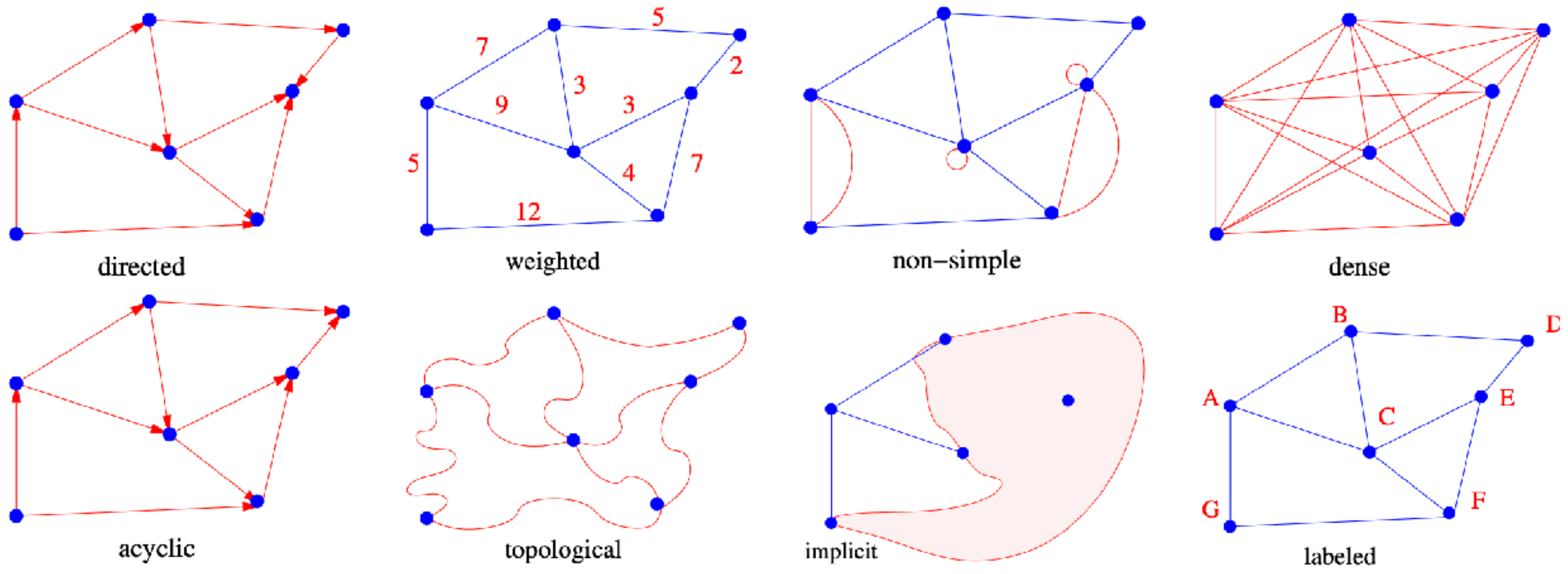
Graphs

- Graphs encode many practical real life scenarios
- Major usage: Modeling our problem with a graph representation and investigating which graph algorithms can help us to solve it.
- Notice that it is very hard to come up with a novel graph algorithm



Some terminology

- G(V,E) represents a graph.



Graph Algorithms - General View

Polynomial Time

- Connected components
- Minimum spanning tree
- Shortest path
- Eulerian cycle
- Edge/vertex connectivity
- Transitive closure
- Network flow
- Planarity testing
- Graph drawing
- .....

NP-Hard

- Clique
- Independent set/vertex cover
- Traveling salesman
- Hamiltonian cycle
- Vertex/edge coloring
- Graph partitioning
- Graph isomorphism
- ....

Representing Graphs

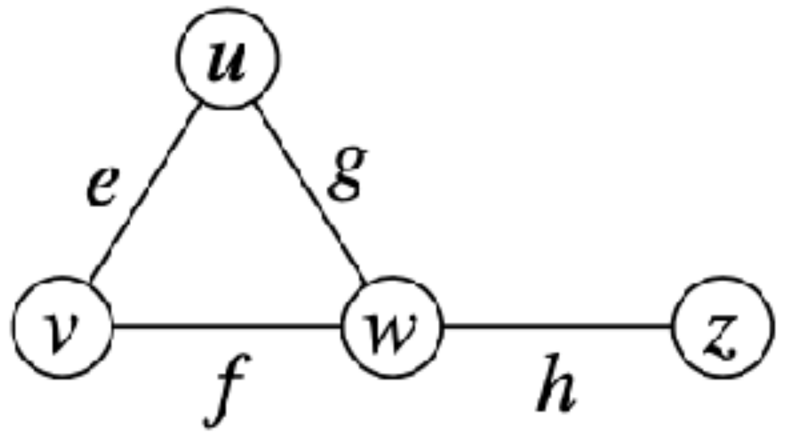
- Different data structures
- Important notice: The complexity of graph algorithms strongly depend on the data structure used to represent the graph.

Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

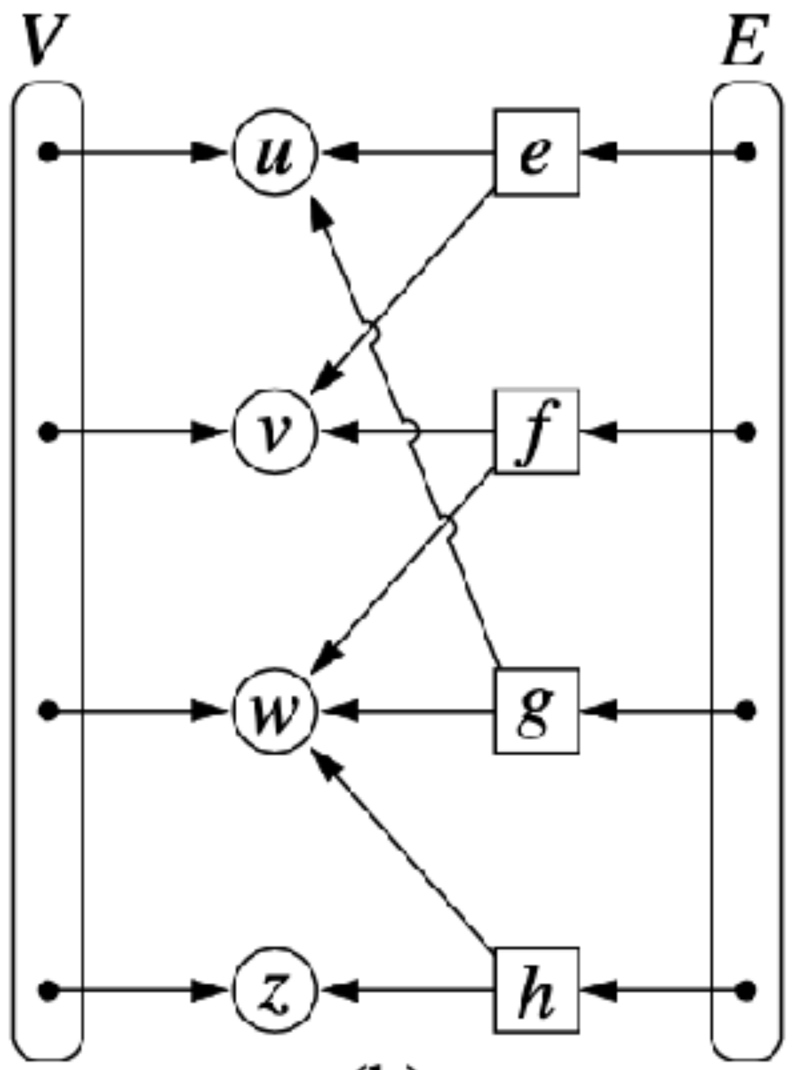
Table 14.1: A summary of the running times for the methods of the graph ADT, using the graph representations discussed in this section. We let n denote the number of vertices, m the number of edges, and d_v the degree of vertex v. Note that the adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n+m)$ space.

- Edge list
- Adjacency list/map
- Adjacency matrix

Edge List



(a)



(b)

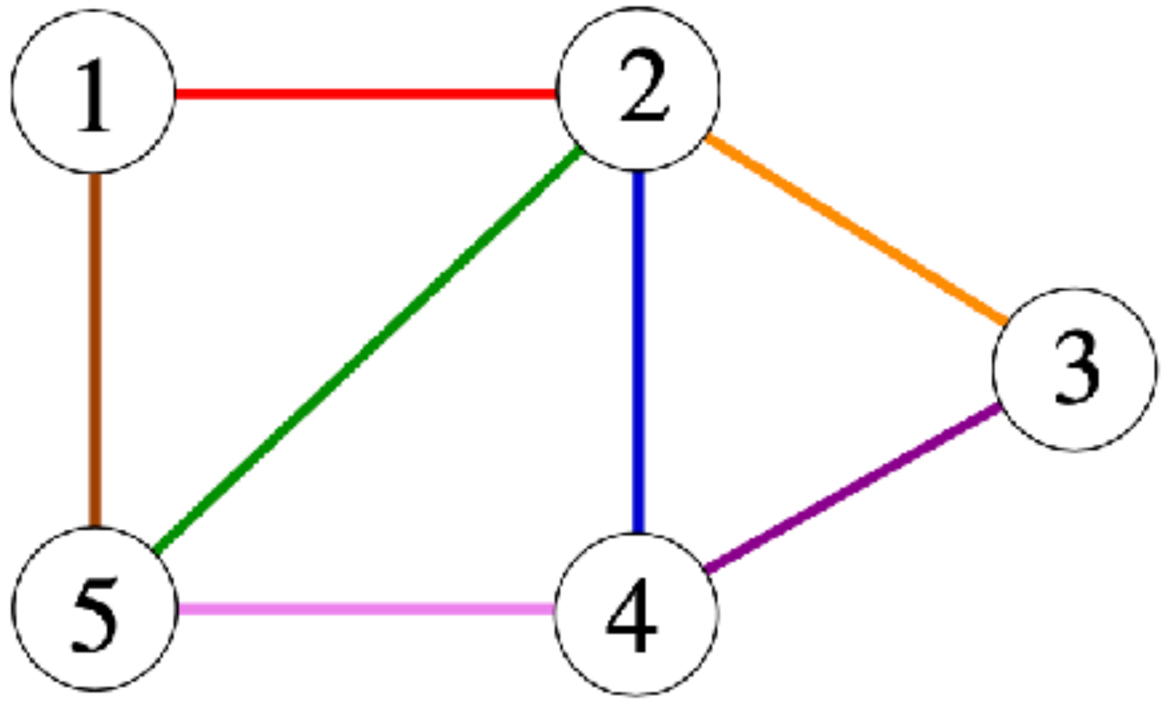
Operation	Running Time
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

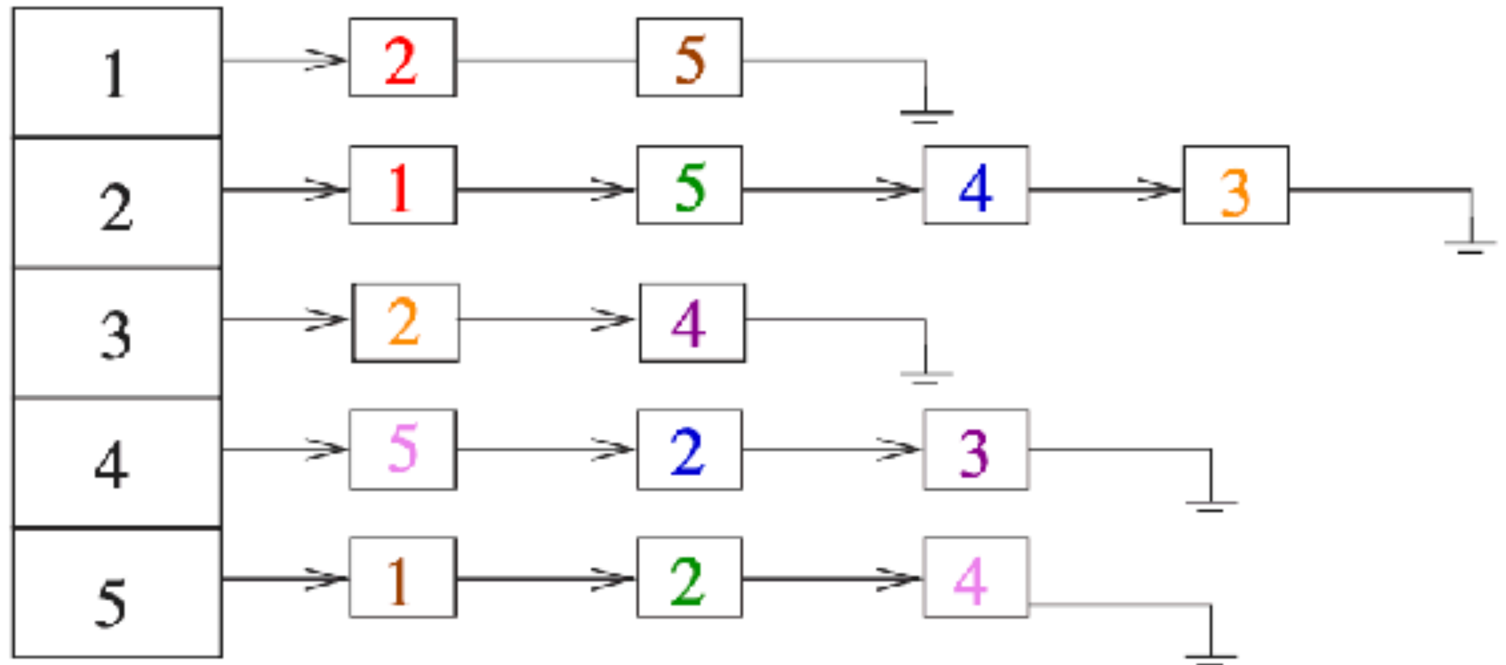
Adjacency Matrix / List

Comparison	Winner
Faster to test if (x, y) is in graph?	adjacency matrices
Faster to find the degree of a vertex?	adjacency lists
Less memory on sparse graphs?	adjacency lists $(m + n)$ vs. (n^2)
Less memory on dense graphs?	adjacency matrices (a small win)
Edge insertion or deletion?	adjacency matrices $O(1)$ vs. $O(d)$
Faster to traverse the graph?	adjacency lists $\Theta(m + n)$ vs. $\Theta(n^2)$
Better for most problems?	adjacency lists

?

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





Adjacency Matrix

Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

R/S Dictionaries, O(1)

Adjacency List

Operation	Running Time
vertex_count(), edge_count()	$O(1)$
vertices() edges()	$O(n)$ $O(m)$
get_edge(u,v)	$O(\min(\deg(u), \deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

Graph Traversal

- Visit every vertex in a graph
- A key operation in many graph algorithms, e.g., connected components, reachability, shortest-path, spanning-tree, cycle-detection etc…
- We need a systematic way to avoid again and again visiting the same vertex or get stuck somewhere in the graph
- So, keep track of visited vertices
- Breadth-first or depth-first

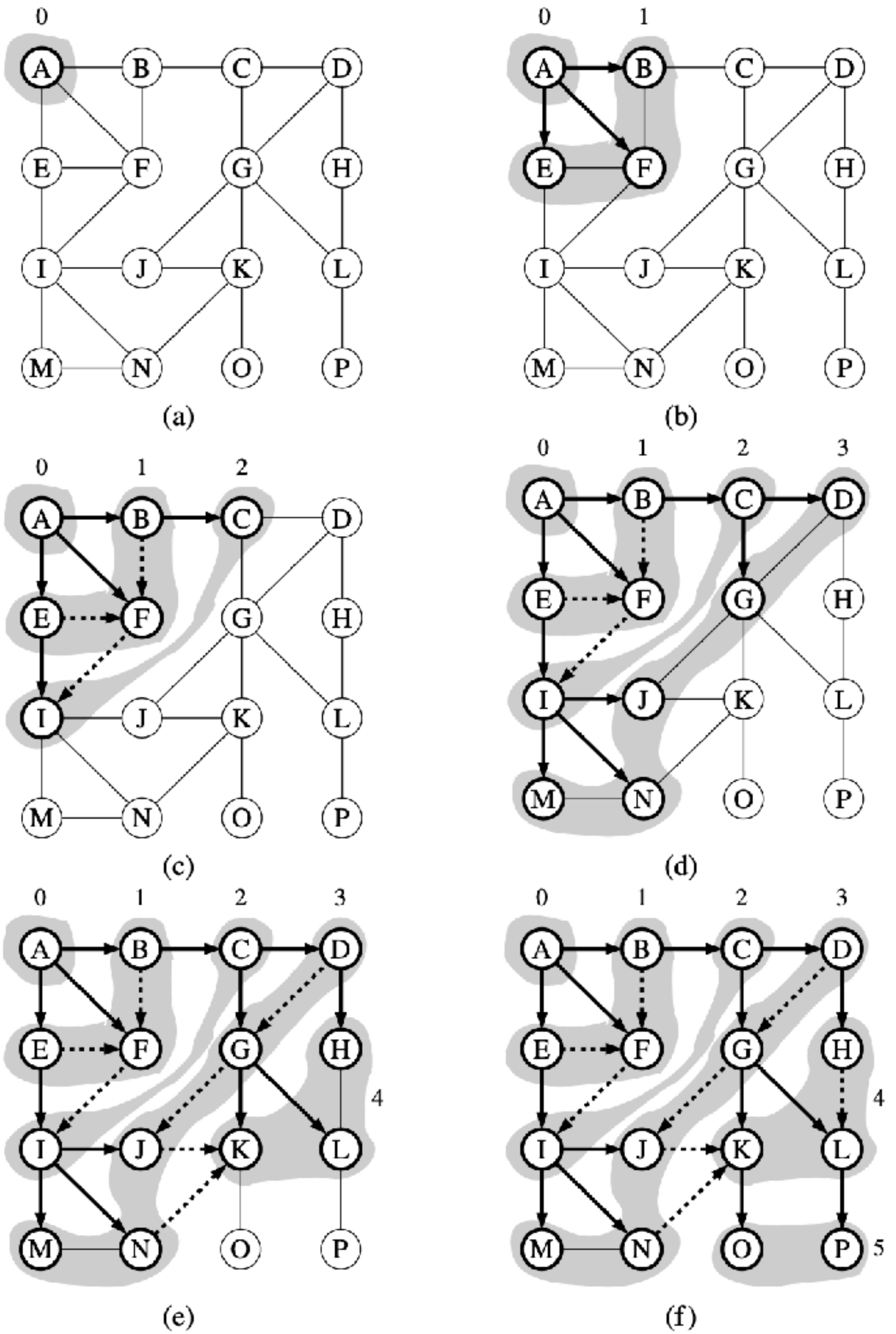
Breadth-First Graph Traversal

```
def BFS(g, s, discovered):
  """Perform BFS of the undiscovered portion of Graph g starting at Vertex s.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the BFS (s should be mapped to None prior to the call).
  Newly discovered vertices will be added to the dictionary as a result.
  """
  level = [s]                          # first level includes only s
  while len(level) > 0:
    next_level = [ ]                   # prepare to gather newly found vertices
    for u in level:
      for e in g.incident_edges(u):    # for every outgoing edge from u
        v = e.opposite(u)
        if v not in discovered:        # v is an unvisited vertex
          discovered[v] = e            # e is the tree edge that discovered v
          next_level.append(v)         # v will be further considered in next pass
    level = next_level                 # relabel 'next' level to become current
```

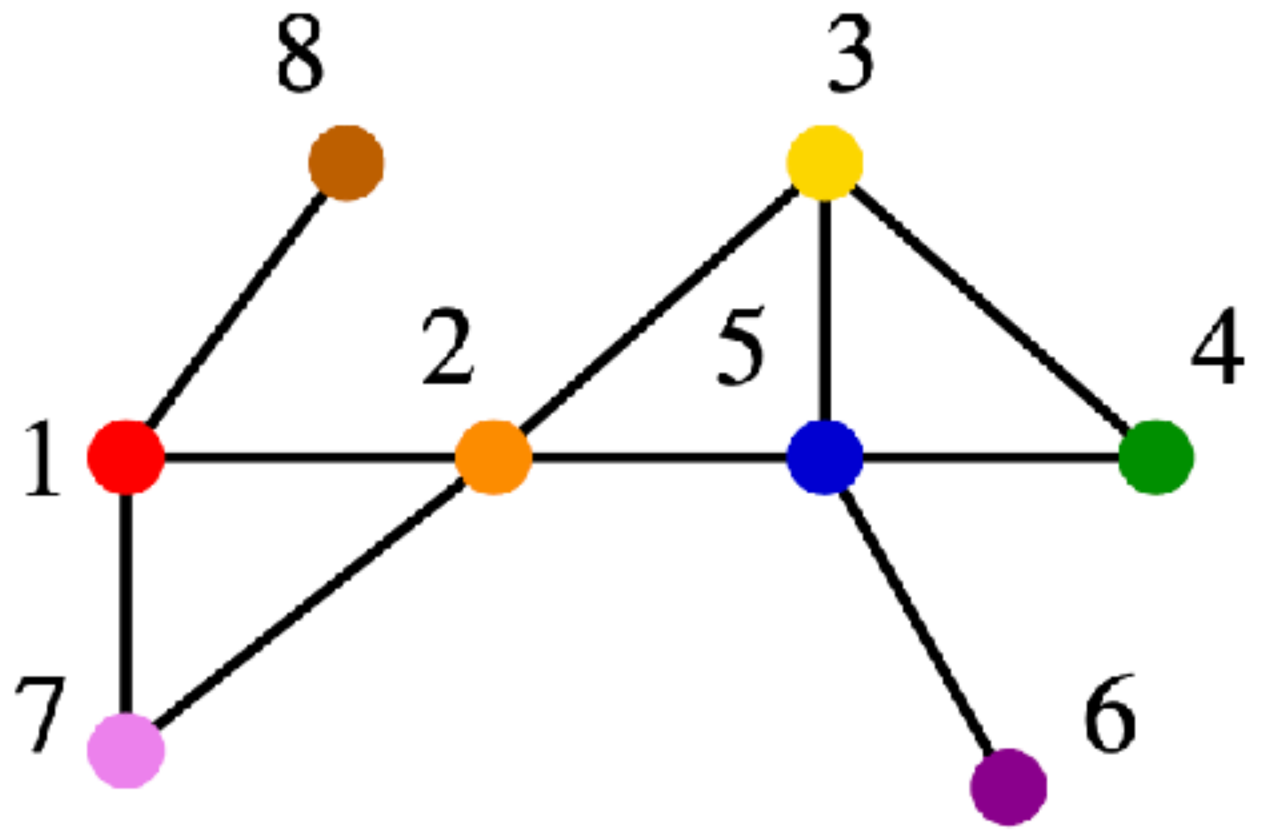
$O(n + m)$-time with adjacency list data structure

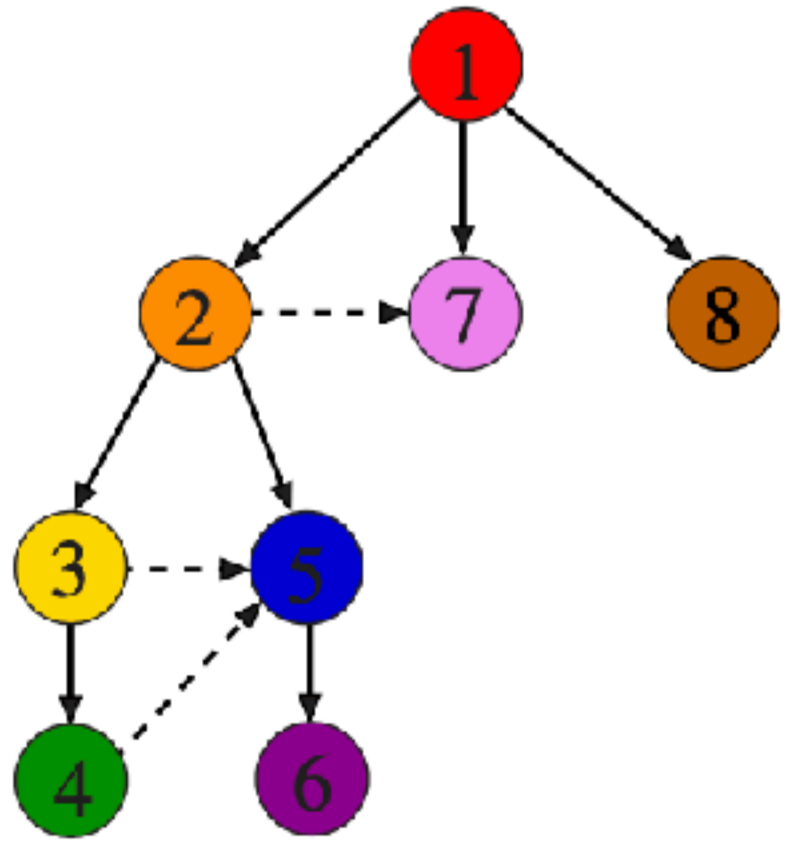
Breadth-first search procedure has some interesting properties….



(a) (b) (c) (d) (e) (f)

Breadth-First Graph Traversal





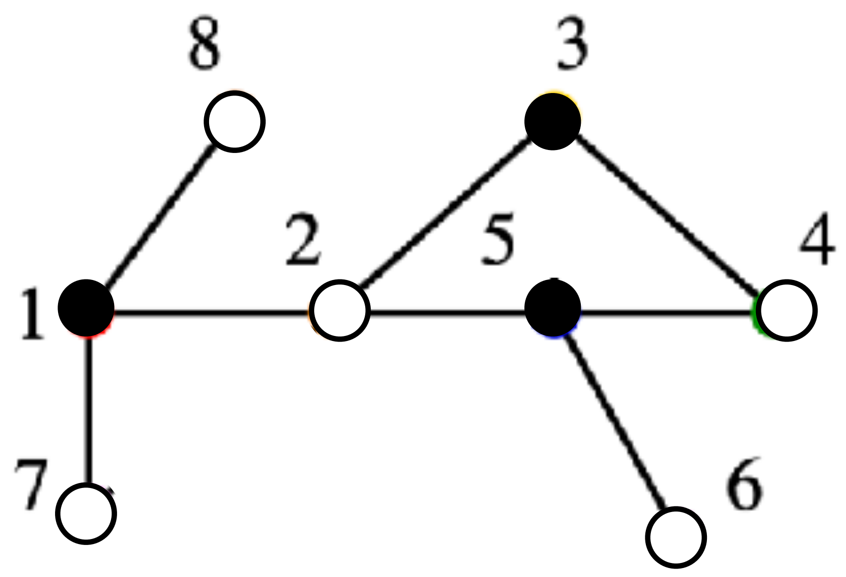
Proposition 14.16: *Let G be an undirected or directed graph on which a BFS traversal starting at vertex s has been performed. Then*

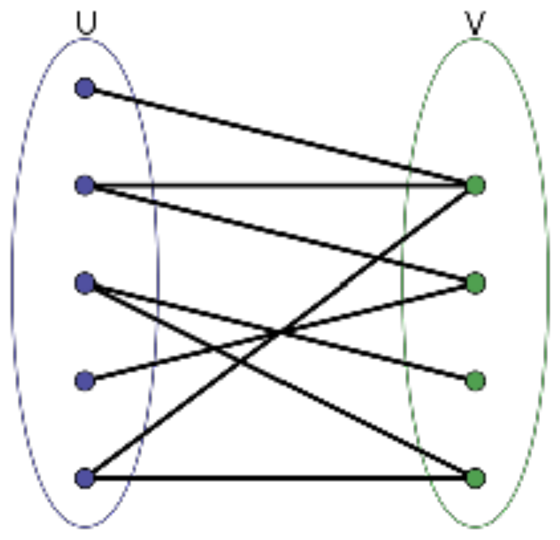
- *The traversal visits all vertices of G that are reachable from s.*
- *For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G from s to v has at least i edges.*
- *If (u, v) is an edge that is not in the BFS tree, then the level number of v can be at most 1 greater than the level number of u.*

"a path in a breadth- first search tree rooted at vertex s to any other vertex v is guaranteed to be the **shortest** such path from s to v **in terms of the number of edges** "

BFS can also be used in

- detecting the connected components of a graph
- two-coloring verification of the graph (is it a bipartite graph)





Depth-First Graph Traversal

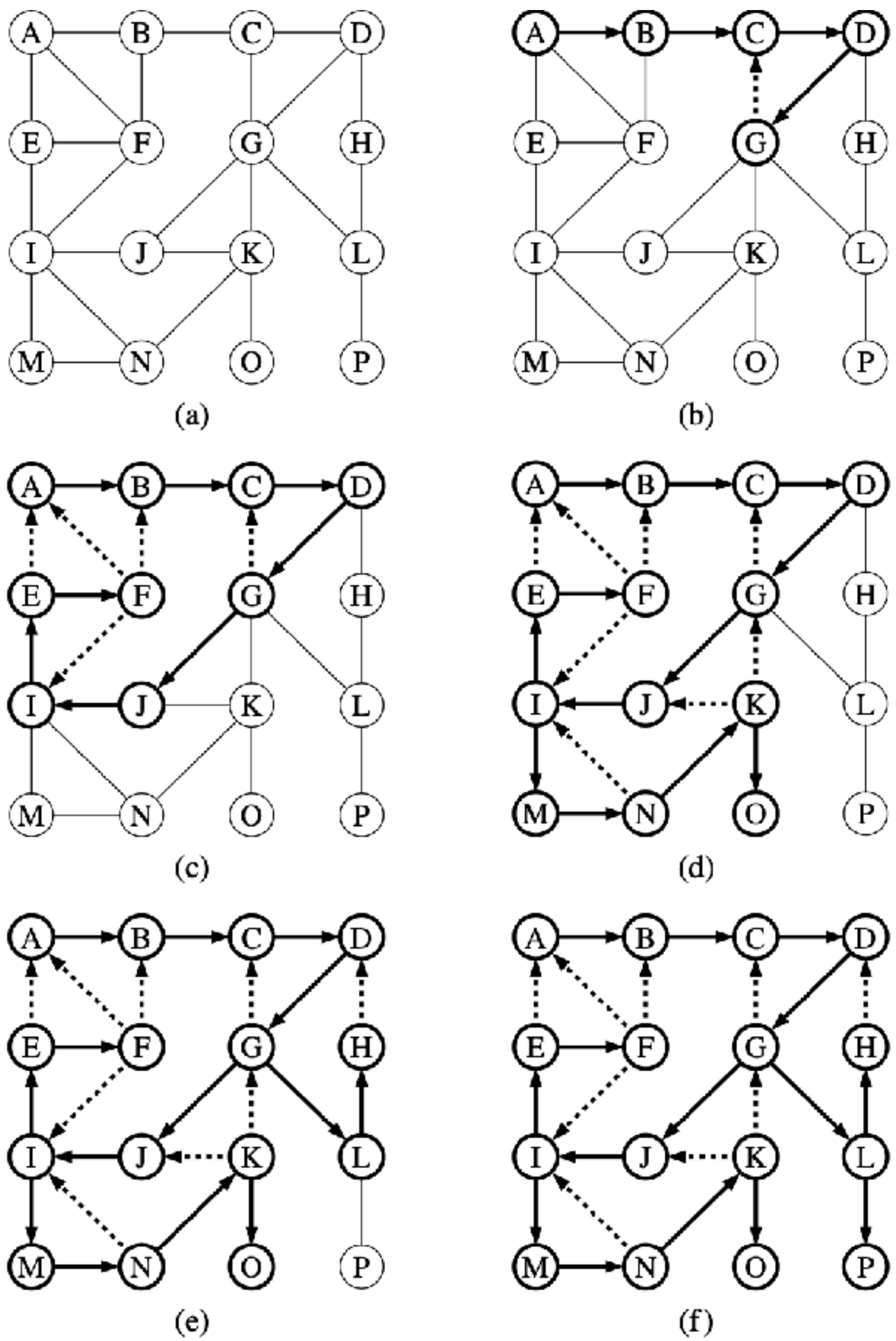
Algorithm DFS(G,u): {We assume u has already been marked as visited}

Input: A graph G and a vertex u of G

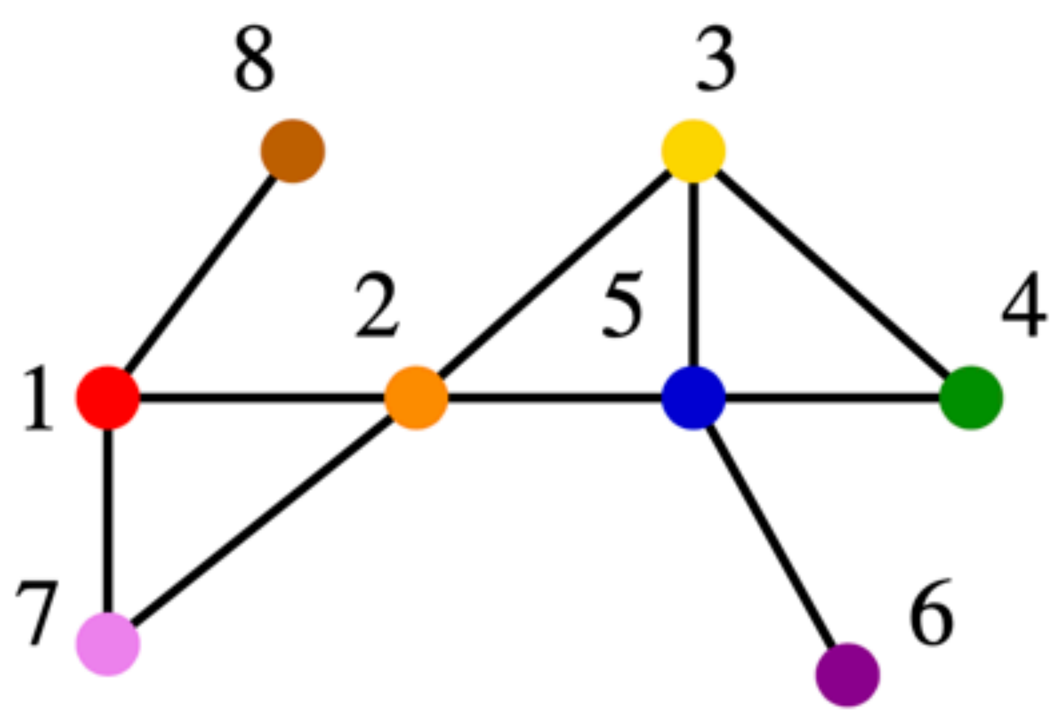
Output: A collection of vertices reachable from u, with their discovery edges

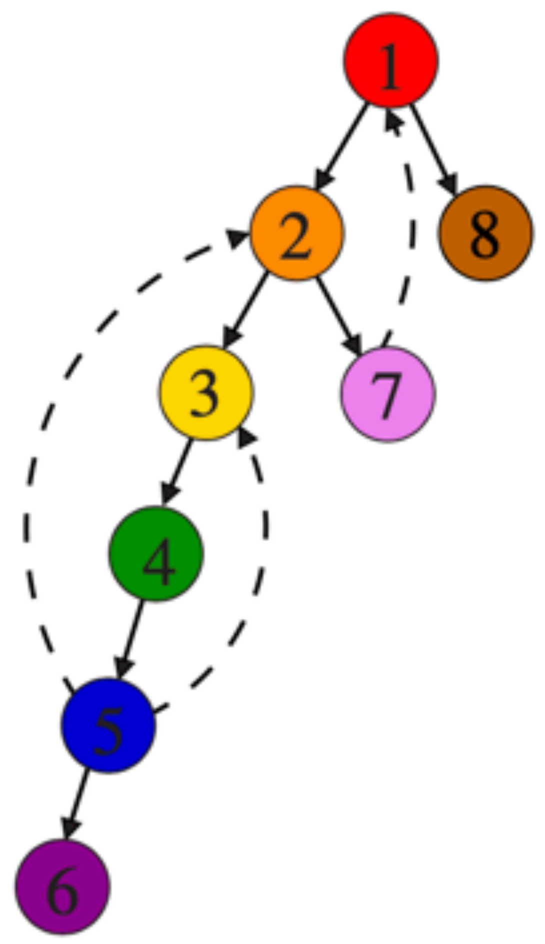
for each outgoing edge e = (u,v) of u **do**
  if vertex v has not been visited **then**
    Mark vertex v as visited (via edge e).
    Recursively call DFS(G,v).

$O(n+m)$-time with adjacency list data structure



Depth-First Graph Traversal





Proposition 14.14: *Let G be an undirected graph with n vertices and m edges. A DFS traversal of G can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a path between two given vertices of G, if one exists.*
- *Testing whether G is connected.*
- *Computing a spanning tree of G, if G is connected.*
- *Computing the connected components of G.*
- *Computing a cycle in G, or reporting that G has no cycles.*

Proposition 14.15: *Let $\vec{G}$ be a directed graph with n vertices and m edges. A DFS traversal of $\vec{G}$ can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a directed path between two given vertices of $\vec{G}$, if one exists.*
- *Computing the set of vertices of $\vec{G}$ that are reachable from a given vertex s.*
- *Testing whether $\vec{G}$ is strongly connected.*
- *Computing a directed cycle in $\vec{G}$, or reporting that $\vec{G}$ is acyclic.*
- *Computing the* **transitive closure** *of $\vec{G}$ (see Section 14.4).*

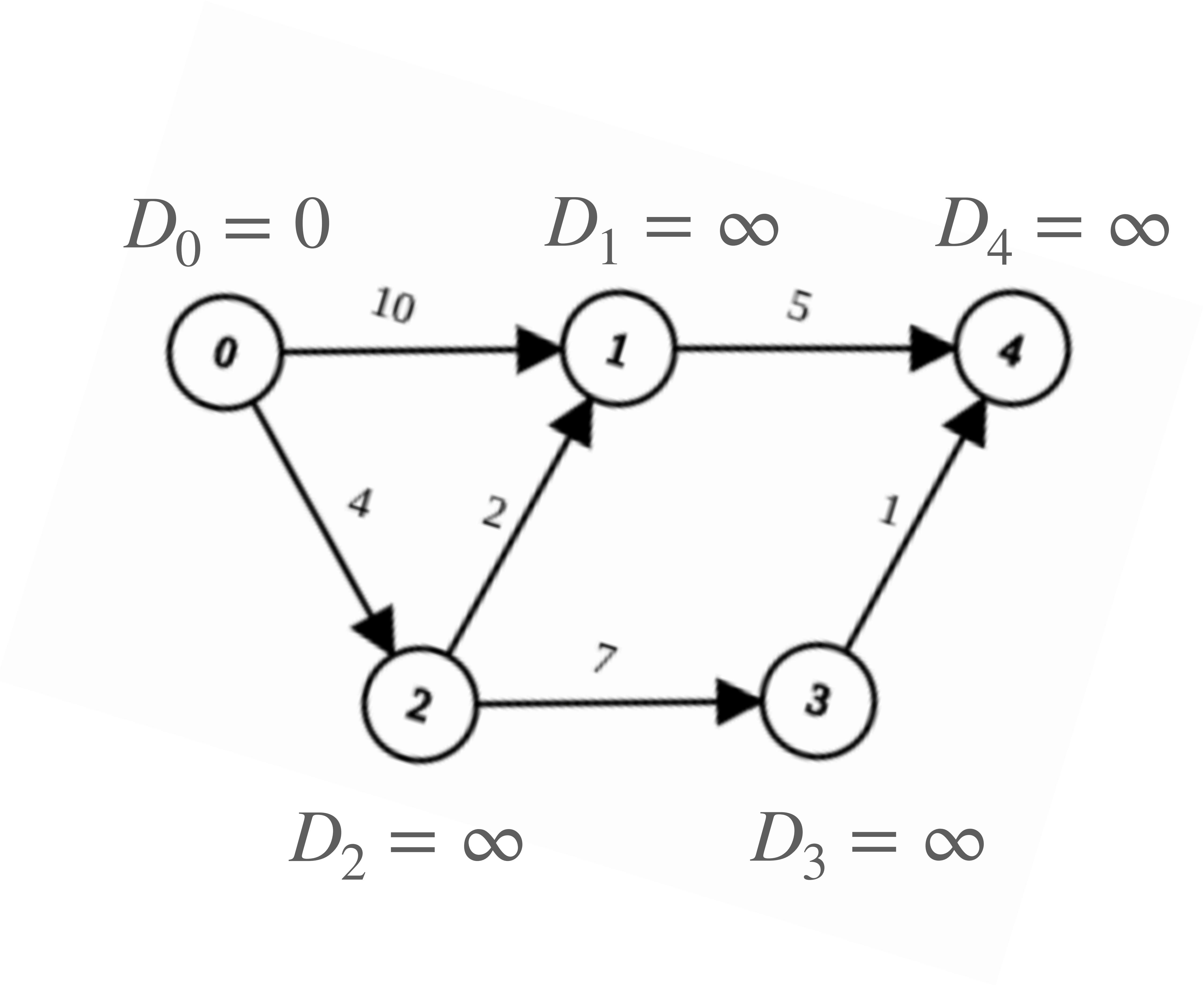
DFS can also be used in

- Testing for connectivity
  - Undirected graphs: Start from a random vertex and see all vertices are reached or not
  - Directed graphs: Run DFS from a random point, reverse all directions of the graph and run DFS again. If all are visited on both cases, it is strongly connected.
- Computing all connected components: Run DFS from a random point. If unvisited vertices are left at the end, repeat until all are visited.
- Detecting cycles: Through the creation of the DFS tree…

Reading assignment

- Skiena chapter 7.

- Goodrich et al. 14.1, 14.2, 14.3


$D_0 = 0$
$D_1 = \infty$
$D_4 = \infty$
10
5
0
1
4
4
2
1
7
2
3
$D_2 = \infty$
$D_3 = \infty$