**COMPLEXITY CLASSES**

Some problems can be solved via. a simple algorithm

Some have **polynomial O(V2)** worst-case performance

Some have **exponential O(2V)** worst-case performance

Classes of algorithms

* **P** = algorithm can compute answer in polynomial time E.g. sorting algorithms: nlogn, n2
* **NP** = no P algorithm is known for solving the problem

The P and NP classes suggest “level of difficulty”

* **P**… “easy”… has known polynomial-time algorithms
* **NP**… “hard”.. no P algorithm known

Levels of “difficulty”

* **Easy** = have a polynomial-time algorithm (useful in practice)
* **Tractable** = have an algorithm, feasible only for small *N* (small input-data)
* **Intractable** = no tractable algorithm is known (NP-hard)
* **Non-computable** = no algorithm can exist (no solution)

**GRAPH ALGORITHMS**

**Connectivity** = Simple Graphs

**Path Finding** = Simple / directed graphs

**Minimum Spanning Trees** = Weighted graphs

**Shortest Path** = Weighted Graphs

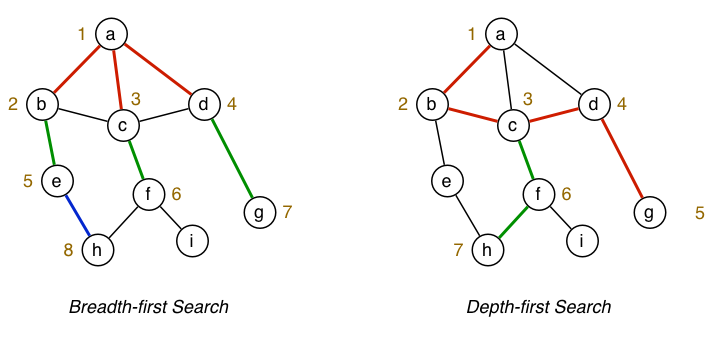
**GRAPH TRAVERSAL**

A common class of graph algorithms involves:

* Walking along edges and visiting vertices
* Recording e.g. path taken, vertices visited etc.

Two strategies for graph traversal / search: **Depth-first + Breadth-first**

* DFS follows one path to completion before considering others
* DFS uses recursion or a stack, and backtracking.
* BFS “fans-out” from the starting vertex (“spreading” subgraph)
* BFS maintains a queue of to-be-visited vertices



Comparison of **DFS / BFS** for **isPath(a,h)**

Both approaches ignore some edges by remembering previously visited vertices.

Previous expression of algorithms used (assumes an adjacency matrix):

* **g->nV** to get #vertices / **g->nE** to get #edges
* **g->edges[v][w]** to check for existence of edge(e,w)

To make things more representation-independent use:

* **nV(g)** to get #vertices / **nE(g)** to get #edges // Function to look up #Vertices / #Edges
* **hasEdge(g,v,w)** to check for existence of edge (v,w) // Function to check if there is an edge between 2 V’s

**DEPTH-FIRST TRAVERSAL**

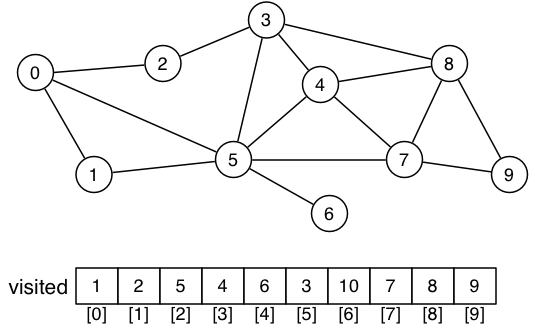
Steps (recursively):

* Do a DFS traversal starting from V
* Mark V as visited
* For each neighbour W of V: If W is already visited, ignore. Otherwise, do a DFS of neighbour W

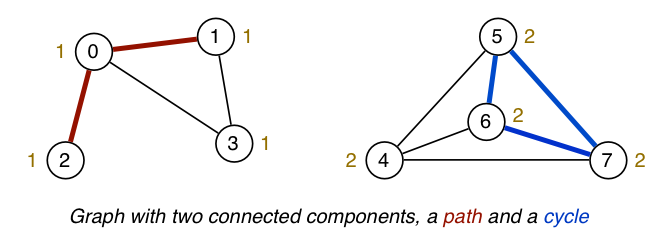
Neighbours are considered in ascending order

The recursion induces **back-tracking**.

**NOTE: For graphs that are un-connected, to visit all vertices, use variation of the “wrapper function”**



Some problems to solve via. DFS graph search



Check for existence of a cycle

* Scan through graph, if you see a vertex that you’ve visited before, there is most likely a cycle.

Determine which connected component each vertex is in.

Finding a path between two vertices.

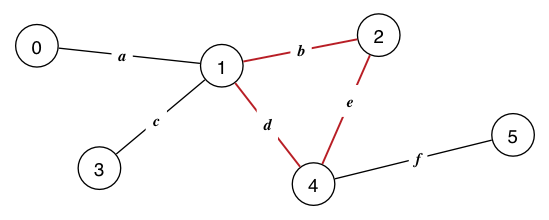
**CHECKING FOR CYCLES**

A graph has a cycle if:

* It has a path of length > 1
* With start vertex src = end vertex dest

Function **bool hasCycle(Graph g)** tells us this.

We are not required to give the path, just indicate its presence.



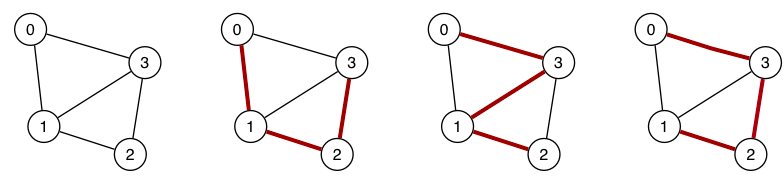
**BREADTH-FIRST SEARCH**

**PATHS: SIMPLE, HAMILTON, EULER**

**Simple path** = A sequence of vertices from SOURCE 🡪 DESTINATION, where no edge / vertex appears twice.

**Shortest path algorithm** = finds a “shortest path” based on minimum # edges between SRC 🡪 DEST

**Hamilton Path and Tour**



* Find a simple path connecting two vertices v,w in graph G
* Such that the path **includes** **each vertex exactly once**
* If v = w, then we have a **Hamilton Tour**

APPROACH: Use **Depth-First Search** to generate paths v..w and check if its a Hamilton Path each time.

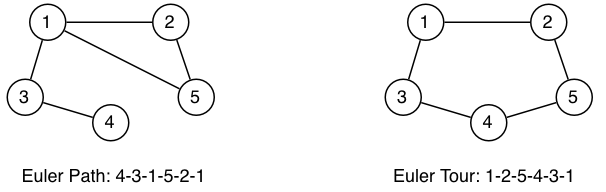
COMPLEXITY CLASS: **NP-Hard** (it is exponential, no polynomial-time solution)

**Euler Path and Tour**

* Find a path connecting two vertices v,w in graph G
* Such that the path **includes each edge exactly once**

(can visit vertices more than once)

* If v = w, then we have **Euler Tour**
* Avoid trying this if we know no such path exists.



APPROACH: Use **Depth-First Search** to generate paths v..w and then check if its an Euler Path each time.

COMPLEXITY CLASS: **NP-Hard**

KNOWN PROPERTIES TO CHECK:

* **Graph has an Euler Path** if it is connected and EXACTLY two vertices have odd degree (Odd # branches)
* **Graph has an Euler Tour** if it is connected and all vertices have even degree (Even # branches)

**CONNECTED COMPONENTS**

Problems:

* How many connected subgraphs are there?
* Are two vertices in the same connected subgraph?

Both of the above can be solved if we can:

* Build an array, one element for each vertex V, indicating which connected component V is in

Consider additional maintenance of such a graph representation:

* Initially, **nC == nV** (because no edges)
* Adding an edge may **reduce nC** / Removing an edge may **increase nC**
* **cc[ ]** can simplify path checking (ensure v,w are in the same component before starting search)

**Having the cc[] array is better than checking the entire graph constantly to see which components are connected**

**DIRECTED GRAPHS (DIGRAPHS)**

In most real-world application of graphs;

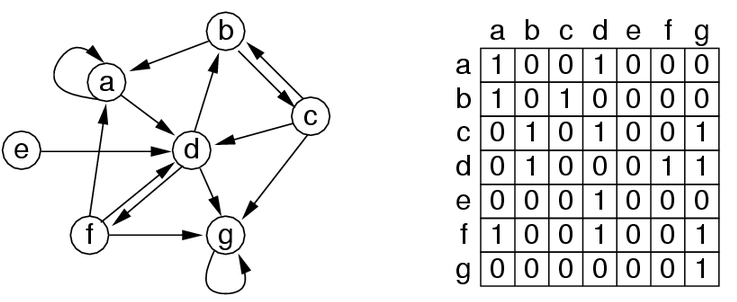
* Edges are **directional** (v🡪w != w🡪v)
* Edges have a **weight** (cost to go from v🡪w)

An adjacency matrix representation of this wouldn’t be symmetric.

Self-connected vertices can exist = non-zero values in the adjacency matrix.

**Traversal = same algos as non-directed (BFS,DFS)**

Example digraph and adjacency matrix representation:

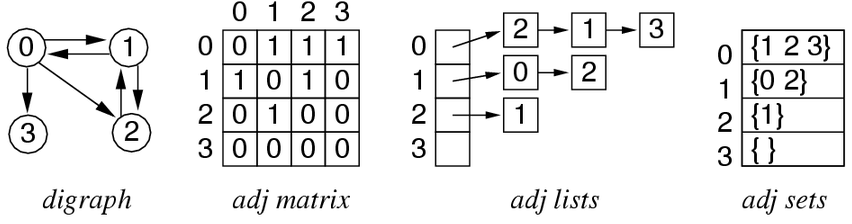
Terminology for Digraphs

* **Directed Path**: List of n>=2 vertices V1,V2…Vn
* Where (Vi , Vi + 1)
* **Directed Cycle**: V1 = Vn (starting vertex = end vertex)
* **Degree of Vertex:** How many edges LEAVE the vertex
* **Indegree of Vertex**: How many edges ENTER the vertex
* **Reachability**: Is W is reachable from V?
* Previously, in a non-directed graph, we just check if two vertices are in the same component
* Now, two vertices in the same component may not be reachable (E.g. Vertice G in the diagram is a sink)
* **Strong connectivity**: Is every vertex is reachable from every other vertex? (in the same connected component).
* **Directed Acyclic Graph (DAG)**: Graph containing no directed cycles

DiGraph applications / Problems to solve on DiGraphs:

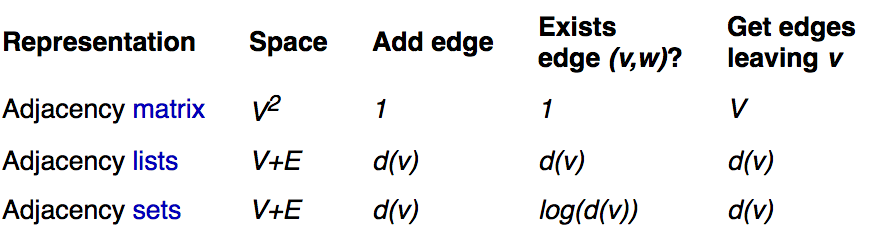
* Is there a directed path from S to T? **(Transitive Closure)**
* What is the shortest path from S to T? **(Shortest Path, #edges)**
* Are all vertices manually reachable? **(Strong connectivity)**
* How to organise a set of tasks? **(topological sort)**
* **Which web pages are “important? (PageRank)**
* How to build a web crawler? **(Graph traversal)**

DiGraph Representation



* V vertices identified by 0..V-1
* Vertex-indexed adjacency matrix (non-symmetric)
* Vertex-indexed adjacency lists (not ordered)
  + No need to add it to two lists, just add it to the list that occurs in the appropriate direction
* Vertex-indexed adjacency sets (some set ADT)
* Treat list of vertices that you’re connected to as a set of vertices

Cost of DiGraph Representations

****

Overall, adjacency sets representation (using sorted arrays) is best.

* Real graphs tend to be **sparse**  
  (Large V, small average d(v))
* Algorithms frequently iterate over edges from V