**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

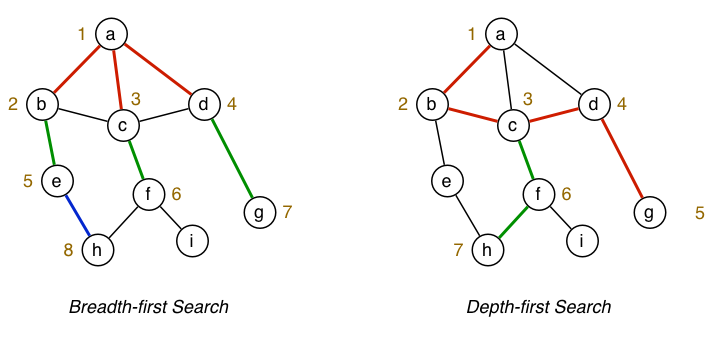
**GRAPH TRAVERSAL**

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**

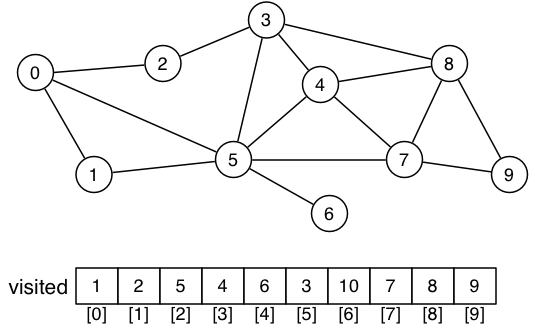
Steps (recursively):

* 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

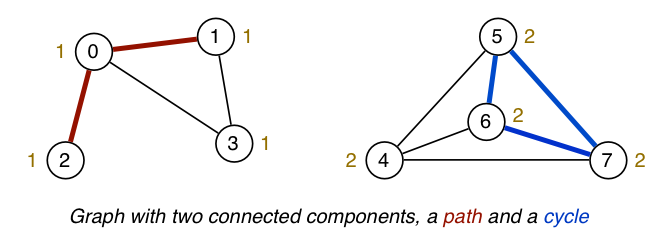
Neighbours are considered in ascending order

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.

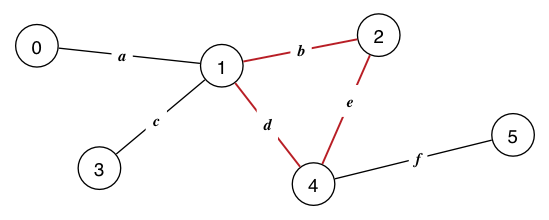
**CHECKING FOR CYCLES**

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