23.1-5 – The square *of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, w) \in E^2$ iff for some $v \in V$, both $(u, v) \in E$ and $(v, w) \in E$; ie. there is a path of exactly two edges.*

Give efficient algorithms for both adjacency lists and matricies.

Given an adjacency matrix, we can check in constant time whether a given edge exists. To discover whether there is an edge $(u, w) \in G^2$, for each possible intermediate vertex v we can check whether (u, v) and (v, w) exist in $O(1)$.

Since there are at most n intermediate vertices to check, and n^2 pairs of vertices to ask about, this takes $O(n^3)$ time.

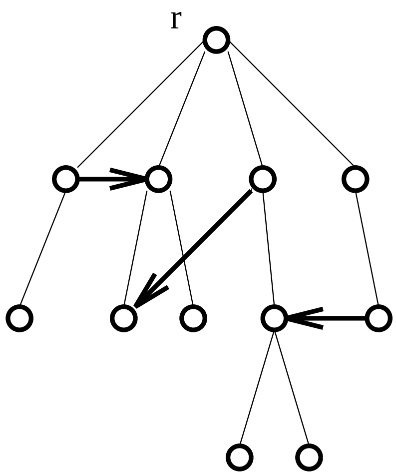
With adjacency lists, we have a list of all the edges in the graph. For a given edge (u, v), we can run through all the edges from v in $O(n)$ time, and fill the results into an adjacency matrix of G^2, which is initially empty.

It takes $O(mn)$ to construct the edges, and $O(n^2)$ to initialize and read the adjacency matrix, a total of $O((n + m)n)$. Since $n \leq m$ unless the graph is disconnected, this is usually simplified to $O(mn)$, and is faster than the previous algorithm on sparse graphs.

Why is it called the square of a graph? Because the square of the adjacency matrix is the adjacency matrix of the square! This provides a theoretically faster algorithm.

BFS Trees

If BFS is performed on a connected, undirected graph, a tree is defined by the edges involved with the discovery of new nodes:



This tree defines a shortest path from the root to every other node in the tree.

The proof is by induction on the length of the shortest path from the root:

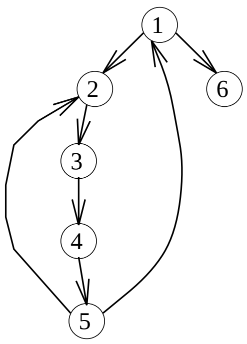
- *Length = 1* First step of BFS explores all neighbors of the root. In an unweighted graph one edge must be the shortest path to any node.

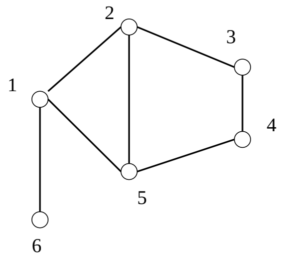
- *Length = s* Assume the BFS tree has the shortest paths up to length $s-1$. Any node at a distance of s will first be discovered by expanding a distance $s-1$ node.

The *key* idea about DFS

A depth-first search of a graph organizes the edges of the graph in a precise way.

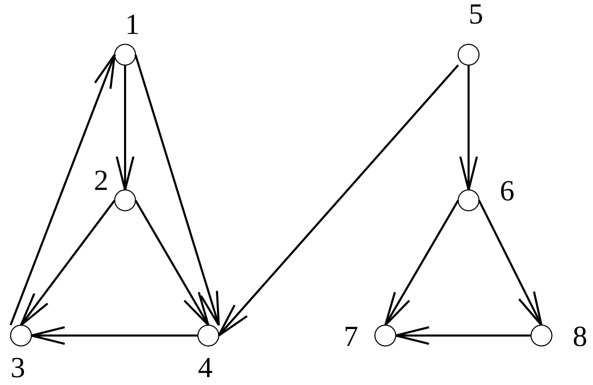
In a DFS of an undirected graph, we assign a direction to each edge, from the vertex which discover it:





In a DFS of a directed graph, every edge is either a tree edge or a black edge.

In a DFS of a directed graph, no cross edge goes to a higher numbered or rightward vertex. Thus, no edge from 4 to 5 is possible:

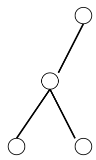


Edge Classification for DFS

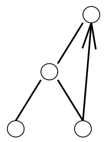
What about the other edges in the graph? Where can they go on a search?

Every edge is either:

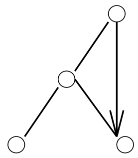
1. A Tree Edge



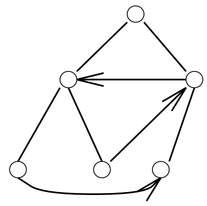
2. A Back Edge
to an ancestor



3. A Forward Edge
to a decendant



4. A Cross Edge
to a different node



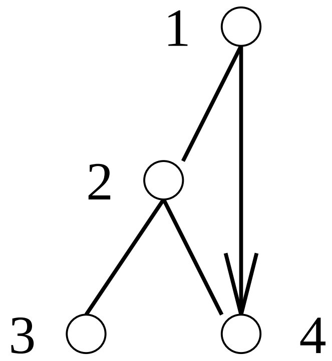
On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

DFS Trees

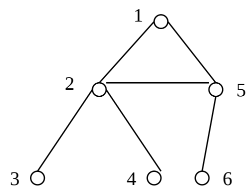
The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.

In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

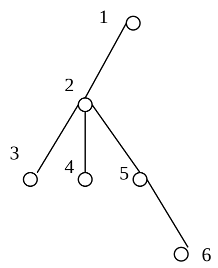
Why? Suppose we have a forward edge. We would have encountered $(4, 1)$ when expanding 4, so this is a back edge.



Suppose we have a cross-edge

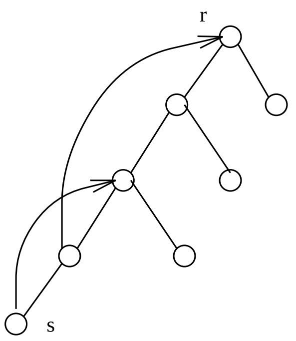


When expanding 2, we would discover 5, so the tree would look like:



Paths in search trees

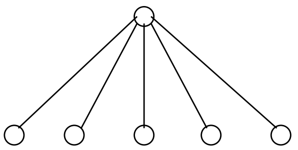
Where is the shortest path in a DFS?



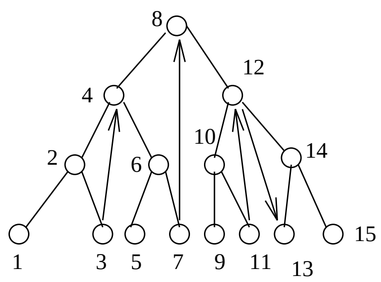
It could use multiple back and tree edges, where BFS only used tree edges.

It could use multiple back and tree edges, where BFS only uses tree edges.

DFS gives a better approximation of the longest path than BFS.



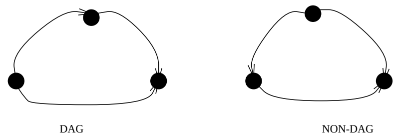
The BFS tree can have height 1, independant of the length of the longest path.



The DFS must always have height >= log P, where P is the length of the longest path.

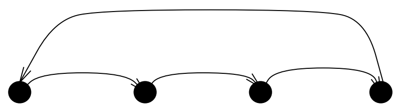
Topological Sorting

A directed, acyclic graph is a directed graph with no directed cycles.



A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.

Only a DAG can have a topological sort.



Any DAG has (at least one) topological sort.

Applications of Topological Sorting

Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given constraints, such as a set of left-right constraints on the positions of objects.

Example: Dressing schedule from CLR.

Example: Identifying errors in DNA fragment assembly.

Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.

```
A B R A C            A B R A C A D A B R A
                     ---------------------
A C A D A            A B R A C
                         R A C A D
A D A B R                  A C A D A
D A B R A                      A D A B R
                                 D A B R A
R A C A D
```

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistant ordering. If there are cycles, there must be errors.

A DFS can test if a graph is a DAG (it is iff there are no back edges - forward edges are allowed for DFS on directed graph).

Algorithm

Theorem: Arranging vertices in decreasing order of DFS finishing time gives a topological sort of a DAG.

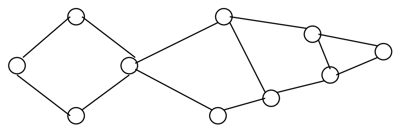
Proof: Consider any directed edge u, v, when we encounter it during the exploration of vertex u:

- If v is white - we then start a DFS of v before we continue with u.
- If v is grey - then u, v is a back edge, which cannot happen in a DAG.
- If v is black - we have already finished with v, so $f[v] < f[u]$.

Thus we can do topological sorting in $O(n + m)$ time.

Articulation Vertices

Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



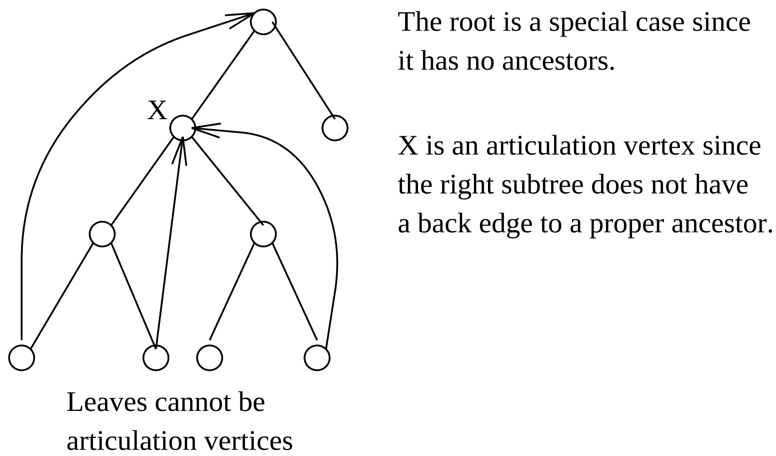
An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.

Clearly connectivity is an important concern in the design of any network.

Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

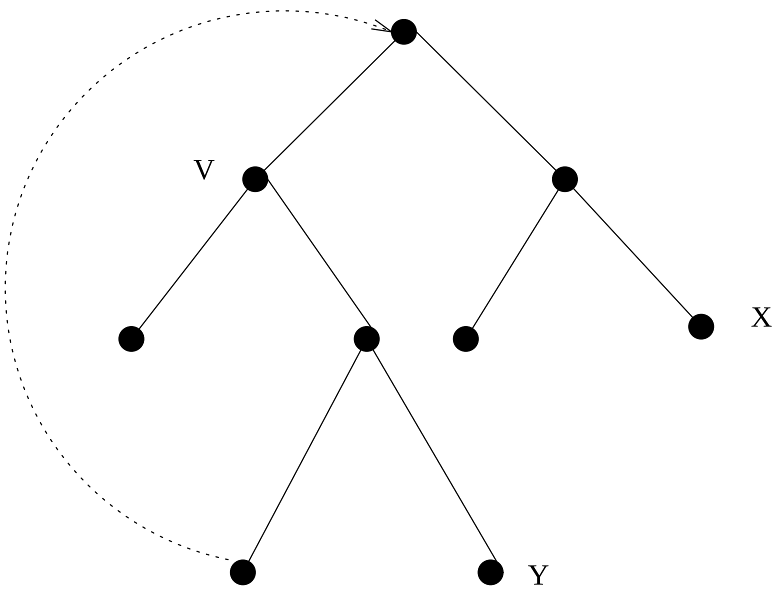
A Faster $O(n+m)$ DFS Algorithm

Theorem: In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident until a proper ancestor of v.



Proof: (1) v is an articulation vertex $\rightarrow$ v cannot be a leaf.

Why? Deleting v must seperate a pair of vertices x and y. Because of the other tree edges, this cannot happen unless y is a decendant of v.



v separating x, y implies there is no back edge in the subtree of y to a proper ancestor of v.

(2) Conditions $\rightarrow$ v is a non-root articulation vertex. v separates any ancestor of v from any decendant in the appropriate subtree.

Actually implementing this test in $O(n+m)$ is tricky – but believable once you accept this theorem.