Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For efficiency, we must make sure we visit each edge at most twice.

For correctness, we must do the traversal in a systematic way so that we don’t miss anything.

Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.
Marking Vertices

The idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

For each vertex, we can maintain two flags:

- *discovered* - have we ever encountered this vertex before?

- *completely-explored* - have we finished exploring this vertex yet?

We must also maintain a structure containing all the vertices we have discovered but not completely explored.

Initially, only a single start vertex is considered to be discovered.

To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do.

Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.

very edge and vertex in the connected component is eventually visited.
Suppose not, ie. there exists a vertex which was unvisited whose neighbor was visited. This neighbor will eventually be explored so we would visit it:
Traversals Orders

The order we explore the vertices depends upon what kind of data structure is used:

- **Queue** – by storing the vertices in a first-in, first-out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a so-called breadth-first search.

- **Stack** - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, constantly visiting a new neighbor if one is available, and backing up only if we are surrounded by previously discovered vertices. Thus our explorations quickly wander away from our starting point, defining a so-called depth-first search.

The three possible colors of each node reflect if it is unvisited (white), visited but unexplored (grey) or completely explored (black).
Breadth-First Search

BFS(G,s)
for each vertex \( u \in V[G] - \{s\} \)
do color[u] = white
\( d[u] = \infty \), i.e. the distance from \( s \)
\( p[u] = NIL \), i.e. the parent in the BFS tree
color[s] = grey
d[s] = 0
\( p[s] = NIL \)
\( Q = \{s\} \)
while \( Q \neq \emptyset \)
do \( u = head[Q] \)
for each \( v \in Adj[u] \)
do if color[v] = white then
color[v] = gray
\( d[v] = d[u] + 1 \)
\( p[v] = u \)
enqueue[Q,v]
dequeue[Q]
color[u] = black
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.
Depth-First Search

DFS has a neat recursive implementation which eliminates the need to explicitly use a stack.

Discovery and final times are sometimes a convenience to maintain.

for each vertex \( u \in V[G] - \{s\} \)
   do color[\( u \)] = white
       \[ p[u] = NIL, \text{ i.e. the parent in the DFS tree} \]
   \[ p[s] = NIL \]
N = 0
DFS(G,s)

Recursive Procedure DFS(G,u):
   Do
      N =: N+1
      number(u) =: N
      color[u] = grey
      for each \( v \in Adj[u] \)
         do if color[\( v \)] = white then
             \[ p[v] = u; \text{DFS}(G,v) \]
         \[ color[u] = black \]
   Od
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 undirected graph, every edge is either a tree edge or a black edge.
Directed Graph:

DFS of a Directed Graph:
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?

very edge is either:

1. A Tree Edge

2. A Back Edge to an ancestor

3. A Forward Edge to a descendant

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 uses tree edges.

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

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

The DFS must always have height \( \geq \log P \), where \( P \) is the length of the longest path.
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.

\[
\begin{array}{cccc}
A & B & R & A & C \\
A & C & A & D & A \\
A & D & A & B & R \\
D & A & B & R & A \\
R & A & C & A & D \\
\end{array}
\quad
\begin{array}{cccc}
A & B & R & A & C \\
R & A & C & A & D \\
A & C & A & D & A \\
A & D & A & B & R \\
D & A & B & R & A \\
\end{array}
\]

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistent 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 $v$.

![Diagram](image)

- The root is a special case since it has no ancestors.
- $X$ is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.
- Leaves cannot be articulation vertices

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

Why? Deleting $v$ must separate a pair of vertices $x$ and $y$. Because of the other tree edges, this cannot happen unless $y$ is a descendant 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 descendant in the appropriate subtree.

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