Data structures for graphs. We generally assume a given graph $G = (V, E)$ is represented in *adjacency list* representation. That is, the vertex set V is represented as a collection of vertices (for example, a linked list or array), and associated with each vertex $v \in V$ is a collection (again, a linked list or array) containing the edges leaving v. This representation allows us to store a graph using space proportional to $|V| + |E|$, to enumerate the vertices in time $O(|V|)$, and to enumerate the edges leaving any given vertex v in time proportional to the out-degree of v.

In some cases (for a directed graph), we may also preprocess the graph (in linear time) to compute, for every vertex v, a collection containing the edges *entering* v. Then we can enumerate the vertices into a given v in time proportional to the in-degree of v.

We don't use the *adjacency matrix* representation (which stores the edge set as a 2-dimensional array A such that $A[i, j]$ is 1 if there is an edge from the ith vertex to the jth vertex, and 0 otherwise). It can save some space (a constant factor) for dense graphs (where most pairs of vertices have an edge). But it is much less efficient for sparse graphs.

It can be convenient to store the collection of vertices and the collections of edges in hash tables. This allows checking in constant time whether a given edge is present.

Generally, we store an *undirected* graph $G = (V, E)$ as a directed graph that has both directed edges (u, w) and (w, u) for each (undirected) edge $\{u, w\} \in E$.

When designing algorithms that traverse a graph, keep in mind that the algorithm does not have a "global" view of the graph. Given a vertex v, all it has access to is the immediate neighbors of v.

Breadth-first search. Given an unweighted digraph $G = (V, E)$ and a source vertex s, we already know one way to find all the vertices reachable from s. We could consider each edge to have weight 1, then use Dijkstra's algorithm. This would take time $O((|V| + |E|) \log |V|)$. When every edge weight is 1, the algorithm is particularly simple. It starts at s, then finds all out-neighbors of s (at distance 1), then finds all out-neighbors of the out-neighbors (at distance 2 from s), and so on. In place of the priority queue (heap), we can use a FIFO (first-in first-out) queue. We get the next vertex to visit from the head of the queue, and when we first discover a new vertex v, we add it to the tail of the queue. Each queue operation takes constant time, so the total time is reduced to $O(|V| + |E|)$ — linear time:

```
BFS(G = (V, E), s)                                   G is a directed or undirected graph
1. initialize FIFO queue Q = (s); initialize distance array d with d[s] = 0
2. while Q is not empty:
3.1. pop the next vertex, say u, from the head of the queue Q
3.2. for each edge (u, w) out of u:
3.3.1. if d[w] is not set, then set d[w] = d[u] + 1 and add w to the tail of the queue Q
4. for each vertex v such that d[v] is not set, set d[v] = ∞
5. return d
```

We omit the proof of correctness, which is similar (but easier) than the proof for Dijkstra's.

Each vertex w is added to the queue at most once, because (by line 3.3.1) $d[w]$ is set the first time w is added, so won't be added again. So each vertex w is popped from the queue in line 3.1 at most once, and the total time for the algorithm is $O(\sum_{w \in V} 1 + \mathsf{in\text{-}degree}(w)) = O(|V| + |E|)$.

We can easily augment the algorithm to compute a breadth-first-search tree — a shortest-path tree T rooted at s, where neighbors of s are children of s, neighbors of those neighbors are grandchildren, etc. For each edge $(u, w) \in E$, note that $d[w] \leq d[u] + 1$, so non-tree edges can't jump more than one level forward in T.

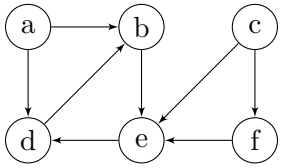
Depth-first search. Breadth-first search expands the search along an even frontier, postponing the exploration of a newly discovered vertex until all vertices closer to s have been explored. In contrast, depth-first-search explores each vertex immediately, as soon as it is discovered.

Depth-first-search does not solve a specific computational problem. Rather, it is used as the basis for designing algorithms for many problems. Here is pseudo-code. No start vertex is given.

```
DFS(G = (V, E))                          G is a directed or undirected graph
1. initialize visited ← ∅                (vertices visited so far, initially the empty set)
2. define DFS1(u):                       (definition of internal subroutine DFS1)
3.1.   add u to visited
3.2.   for each edge (u, w) ∈ E out of u:
3.3.     if w ∉ visited: DFS1(w)
4. for each vertex u ∈ V:                (outer loop that actually calls DFS1)
5.     if u ∉ visited: DFS1(u)
```

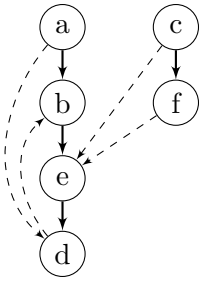
Consider executing DFS on any graph $G = (V, E)$. During the execution, when a call to DFS1(u) is made, call this *visiting* vertex u. Each vertex $u \in V$ is visited exactly once, and when u is visited, the inner loop iterates over the edges out of u, so the total time for the execution is $O(\sum_{u \in V} 1 + \text{out-degree}(u)) = O(|V| + |E|)$.

The DFS forest. Executing DFS on a graph implicitly defines a *DFS forest*. To explain, consider executing DFS on the graph G shown to the right with $V = \{a, b, c, d, e, f\}$. Assume that in Lines 3.2 and 4, vertices are chosen in alphabetical order. The recursive calls to DFS1 are nested as shown below.




```
DFS1(a)        --- called in outer loop, Line 5.1
  DFS1(b)      --- called from DFS1(a), Line 3.3.1
    DFS1(e)    --- called from DFS1(b), Line 3.3.1
      DFS1(d)  --- called from DFS1(e), Line 3.3.1
DFS1(c)        --- called in outer loop, Line 5.1
  DFS1(f)      --- called from DFS1(c), Line 3.3.1
```

the DFS forest



The resulting DFS forest is shown to the right. It consists of two *DFS trees*. The vertices of the DFS forest are the vertices u of G, each representing the single call to DFS1(u) made during the execution. If the call to DFS1(u) recursively calls DFS1(w) (following the edge (u, w)), then w is the child of u in the DFS forest. Then edge (u, w) is called a *tree edge*. In the diagram, the tree edges are the edges drawn with solid lines. The remaining (non-tree) edges in E are not part of the forest. They are nonetheless shown in the diagram, but with dashed lines. Each non-tree edge $(u, w) \in E$ is classified as one of three types, depending on how it interacts with the DFS forest:

If w is a descendant of u in the forest (in the example, (a, d)), then (u, w) is a *forward edge*.

If u is a descendant of w (in the example, (d, b)), then (u, w) is a *back edge*.

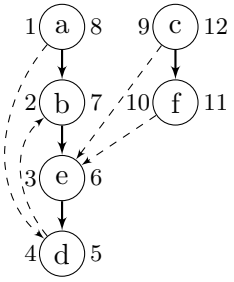
The remaining non-tree edges (in the example, (c, e) and (f, e)), are *cross edges*.

Cross edges can only go "right to left" — from a node u to a node w that was visited before u — because otherwise the edge (u, w) would have been traversed when u was visited.

What about *undirected* graphs? For undirected graphs there cannot be any cross edges! Also, each forward edge is also a back edge.

Discovery times and finish times. Along with the DFS tree, executing DFS on a graph implicitly defines a *discovery time* $d[u]$ and a *finish time* $f[u]$ for each vertex $u \in V$. The discovery time is the time when u is first visited, that is, when DFS1(u) is called. The finish time is the time when DFS1(u) completes. To explicitly compute these times, we could augment DFS as follows:

```
DFS(G = (V, E))
1. initialize visited ← ∅; initialize t ← 1
2. define DFS1(u):
3.1. add u to visited; set d[u] = t; increment t    (set discovery time for u)
3.2. for each edge (u, w) ∈ E out of u:
3.3.     if w ∉ visited: DFS1(w)
3.4. set f[u] = t; increment t                      (set finish time for u)
4. for each vertex u ∈ V:
5.     if u ∉ visited: DFS1(u)
```



The discovery and finish times for the example are shown above to the right. The discovery time for each vertex v is shown just to the left of v; the finish time is shown just to the right.

For each edge, the discovery and finish times of the endpoints are related as follows, depending on the type of the edge:

Observation 1. *In any execution of DFS on a graph $G = (V, E)$, for any edge $(u, w) \in E$,*

If (u, w) is a tree or forward edge, then $d[u] < d[w] < f[w] < f[u]$.
If (u, w) is a back edge, then $d[w] < d[u] < f[u] < f[w]$.
If (u, w) is a cross edge, then $d[w] < f[w] < d[u] < f[u]$.

Finding cycles or topologically sorting in linear time. Recall that a directed graph $G = (V, E)$ is *topologically ordered* if the vertex set V is given as a sequence $V = (v_1, v_2, \ldots, v_n)$ such that each edge $(v_i, v_j) \in E$ has $i < j$. Topological Sorting is the following problem: *Given G, reorder G's vertices so that G is topologically ordered.*

Lemma 1. *If G has a cycle, then it cannot be topologically sorted.*

Proof. Suppose for contradiction that G has been topologically sorted, and has a cycle C. Let v_i be the vertex in C with largest index i, so that the edge out of (v_i, v_j) in C must have $i > j$, contradicting that G is topologically ordered. □

So let's amend our problem to the following: *Given a digraph $G = (V, E)$, determine whether G has a cycle, and, if not, topologically sort the vertices of G.* We will use DFS to solve it in linear time. Consider any execution of DFS on G. We start with the following observation:

Observation 2. *(i) If G has a back edge (u, w), then G has a cycle.*
(ii) If G has no back edge, then $f[w] < f[u]$ for each edge $(u, w) \in E$.

For Part (i), note that the back edge (u, w) forms a cycle with the path from u to its descendant w in the DFS forest. For Part (ii), if there are no back edges, each edge (u, w) is either a tree, forward, or cross edge, so, by inspection of Observation 1, $f[u] < f[w]$.

This gives us the following linear-time algorithm. Run DFS on G. If there is a back edge, then G has a cycle. Otherwise topologically sort G by reordering the vertices by reverse finishing times. (E.g. push each u on the front of a globally maintained list when $f[u]$ is set in Line 3.4 of DFS1(u).)

Finding connected components in undirected graphs in linear time. In an undirected graph $G = (V, E)$, the *connected components* are defined as follows. Consider two vertices u and w to be *equivalent* if there is a path between u and w. This relation is transitive and symmetric, and partitions the vertex set into equivalence classes $C_1, C_2, \ldots, C_k$, each called a *connected component*. For every two vertices u and w, there is a path between u and w if and only if u and w are in the same component C_i. There are no edges between connected components in G.

Given an undirected graph $G = (V, E)$, the connected components of G can be found in linear time by doing a single DFS on G. Then (since there are no cross edges) each tree in the DFS forest is a single connected component.

Strongly connected components in directed graphs. In a directed graph the situation is more complicated, because it can be that there is a path from u to w, but no path from w to u. We generalize the notion of connected components as follows. Say that two vertices u and w are *strongly connected* if there is both a path from u to w *and* a path from w to u. Now consider two vertices u and w to be equivalent if they are strongly connected. This relation is transitive and symmetric, so it partitions the vertex set into equivalence classes $C_1, C_2, \ldots, C_k$, each called a *strongly connected component*. For each pair of vertices $u, w \in V$, the pair is strongly connected if and only if they are in the same component C_i. There can be edges between strongly connected components, but any cycle in G must be wholly contained within one component.

The following algorithm computes strongly connected components in linear time.

$\mathsf{SCC}(G = (V, E), s)$ *G is any directed graph*

1. Run DFS on G. Record the finish time $f[u]$ of each vertex u.
2. Obtain digraph $G' = (V, E')$ from G by reversing each edge ($E' = \{(w, u) : (u, w) \in E\}$).
3. Run DFS on G', ordering V in the outer loop (Line 4) by decreasing $f[u]$ (from the first DFS).
4. Let $T_1, T_2, \ldots, T_k$ be the DFS trees in the resulting DFS forest.
5. Return $(C_1, C_2, \ldots, C_k)$, where C_i contains the vertices in tree T_i.

Theorem 1. *The above algorithm returns the strongly connected components of G.*

For a proof, see Chapter 22.5 of the text (CLRS).

External resources on linear-time graph traversal

- CLRS Chapter 22. Dasgupta et al. Chapters 3 and 4.3. Kleinberg & Tardos Chapter 3.
- Jeff Edmond's Lectures 18 and 19:

  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/18-graphs.pdf

  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/19-dfs.pdf
- MIT lecture videos
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-14-depth-first-search-dfs-topological-sort/