

Lecture Note 6

Graph Traversal

draft of September 29, 2026, by [N. Young](#)

© 2018–2026 [Neal E. Young](#). Text: CC BY-NC-SA 4.0; code: MIT; except where noted. See <https://github.com/nealeyoung/algs101/blob/main/LICENSE>.

Contents

6.1	Graph terminology and data structures	1
6.2	Breadth-first Search	1
6.3	Depth-first Search	2
6.3.1	The DFS forest: forward, back, cross, and tree edges	3
6.3.2	Finding connected components in undirected graphs in linear time	4
6.3.3	Finding a cycle in an undirected graph in linear time	4
6.3.4	Discovery times and finish times	4
6.3.5	Topologically sorting a directed graph in linear time	5
6.3.6	Strongly connected components in directed graphs	5
6.3.7	Exercises	6
6.3.8	External sources on graphs and linear-time graph traversal	9
6.3.9	Appendix: Proof of correctness for Breadth-First Search	9
6.4	Python code	11

6.1 Graph terminology and data structures

Definitions and terminology related to graphs. We assume familiarity with graphs and the following graph-related terms: *edge*, *vertex*, *node*, *endpoint* of an edge, *directed graph*, *path*, *simple path*, *cycle*, *simple cycle*, *connected graph*, *strongly connected graph*, *distance* between two nodes. We also assume familiarity with trees and the following tree-related terms: *rooted tree*, *parent*, *child*, *descendant*, *ancestor*. Given a graph $G = (V, E)$, vertices $u, w \in V$ are *neighbors* if they are connected by an edge (that is, if $(u, w) \in E$). Given a digraph $G = (V, E)$, if the directed edge (u, w) is in E , then w is an *out-neighbor* of u and u is an *in-neighbor* of w .

Data structures for graphs. We generally assume that graphs are stored in *adjacency-list* representation. For given graph $G = (V, E)$, then, 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 . The space (memory) required for representing G this way is proportional to the *size* of G , that is, the number of edges and vertices, $|V| + |E|$. The representation allows us to enumerate V in time $O(|V|)$, and to enumerate the edges leaving a 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 i th vertex to the j th 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 a *hash table* (dictionary). 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 $\{u, w\} \in E$.

For detailed background information on graphs and graph representations, see [JE](#) Chapter 5, [KT](#) Chapter 3, [DPV](#) Chapters 3 and 4, and/or [CLRS](#) Chapter 22.

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 .

6.2 Breadth-first Search

Given an graph $G = (V, E)$ and a source vertex s , how can we efficiently find all the vertices reachable from s ? *Breadth-first search* starts at s . It then discovers all out-neighbors of s (at distance 1 from s). From those it finds all out-neighbors of those out-neighbors (at distance 2 from s), and so on. The algorithm uses two data structures: an array d and a FIFO queue Q . The queue holds the vertices that have been discovered, but whose out-edges have not yet been explored. As the algorithm proceeds, for each vertex v that has been discovered so far, the algorithm records the distance from s to v (that is, $\text{dist}(s, v)$) in the array entry $d[v]$.

$\text{BFS}(G = (V, E), s)$ G is a directed or undirected graph, in adjacency-list representation

1. initialize FIFO queue $Q = (s)$; initialize distance array d with $d[s] = 0$; initialize array π
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. if $d[w]$ is not set, then set $d[w] \leftarrow d[u] + 1$, set $\pi[w] \leftarrow u$, and add w to the tail of Q
4. for each vertex v such that $d[v]$ is not set, set $d[v] = \infty$
5. return d and π

Lemma 6.1 (run time). *Given any directed or undirected graph G and source s , $\text{BFS}(G = (V, E), s)$ runs in time linear in the size of G (that is, in time $O(|V| + |E|)$).*

Proof. Each vertex w is added to the queue at most once, because (by Line 3.3) $d[w]$ is set the first time w is added, so won't be added again. So each vertex u is popped from the queue in line 3.1 at most once, and the total time for the algorithm is $O(\sum_{u \in V} 1 + \text{out-degree}(u)) = O(|V| + |E|)$. $\square$

Before we sketch the correctness proof, we formally prove some intuitive properties of distances. Fix an input $G = (V, E)$ with source s . For each integer $\ell \geq 0$, define *level* ℓ to be the set of vertices v with $\text{dist}(s, v) = \ell$.

Lemma 6.2 (levels). *Level 0 contains just the source s . Each level $\ell > 0$ consists of the vertices w such that (a) w is not in any level less than ℓ , and (b) there is an edge $(u, w) \in E$ from some vertex u in level $\ell - 1$. That is, each level $\ell > 0$ is $\{w \in V : \text{dist}(s, w) \geq \ell, (\exists(u, w) \in E) \text{dist}(s, u) = \ell - 1\}$.*

A detailed long-form proof (which just requires reasoning carefully about the definitions) is in an appendix to this note. Using the lemma, we can prove that BFS computes distances correctly:

Lemma 6.3 (correctness). *Given any directed or undirected graph G and source s , the array d returned by $\text{BFS}(G = (V, E), s)$ satisfies $d[v] = \text{dist}(s, v)$ for all $v \in V$.*

Roughly, the proof shows by induction that $\text{BFS}(G, s)$ discovers the vertices level by level. It starts with the queue containing just $\{s\}$ (level 0), with $d[s]$ set to 0. When it pops s off the queue, it puts the vertices in level 1 in the queue, and sets $d[w]$ for each level-1 vertex to 1. Then, as it pops the level-1 vertices off the queue, it appends the level-2 vertices to the queue (setting $d[w]$ for each to 2) and so on. In general, as it pops the vertices in any given level $\ell - 1$ off the queue, it appends the vertices in level ℓ to the queue (and correctly sets $d[w]$ for each such vertex to ℓ). A detailed long-form proof is in an appendix to this note.

Note that the edges $\{(\pi[v], v) : 0 < d[v] < \infty\}$ form the so-called *shortest-path tree* T —rooted at s , in which neighbors of s in G are children of s , neighbors of those neighbors are grandchildren, etc. so that, for each $\ell \geq 0$, the nodes at depth ℓ in T are the vertices in level ℓ of G , and the path from s to any node v in T is a shortest path from s to v .

Python code: [graph_traversal/breadth_first_search.py](#) (Section 6.4).

External sources on Breadth-First Search

- [JE](#) Sections 5.5–5.6; [KT](#) Sections 3.2–3.3; [DPV](#) Section 4.2; [CLRS](#) Section 22.2.

6.3 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 $\text{visited} \leftarrow \emptyset$	(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 $(u, w) \in E$ out of u :	
3.3. if $w \notin \text{visited}$: DFS1(w)	
4. for each vertex $u \in V$:	(outer loop that actually calls DFS1)
5. if $u \notin \text{visited}$: 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 $O(\sum_{u \in V} 1 + \text{out-degree}(u)) = O(|V| + |E|)$. That is, DFS runs in linear time.

Python code: [graph_traversal/depth_first_search.py](#) (Section 6.4).

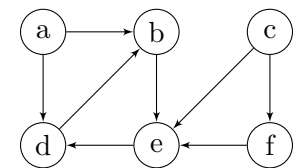
6.3.1 The DFS forest: forward, back, cross, and tree edges

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.

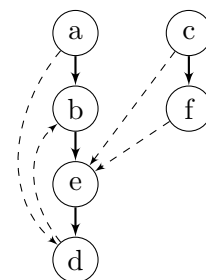
```

DFS1(a)      --- called in outer loop, Line 5
  DFS1(b)    --- called from DFS1(a), Line 3.3
    DFS1(e)  --- called from DFS1(b), Line 3.3
      DFS1(d) --- called from DFS1(e), Line 3.3
    DFS1(c)  --- called in outer loop, Line 5
      DFS1(f) --- called from DFS1(c), Line 3.3

```



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.

6.3.2 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 reflexive, transitive and symmetric, and partitions the vertex set into equivalence classes $C_1, C_2, \dots, C_k$, 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* different 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.

Python code: [graph_traversal/depth_first_search.py](#) (Section 6.4).

6.3.3 Finding a cycle in an undirected graph in linear time

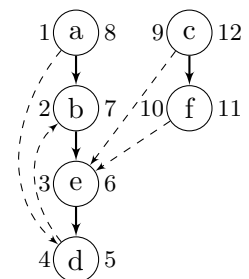
Given an undirected graph $G = (V, E)$, DFS can be used to detect in linear time whether G has a cycle, as follows. After running DFS on G , if there is a back edge (u, w) other than the reverse of a tree edge (that is, with w not the parent of u), then that back edge forms a cycle with the path of tree edges between u and w . Conversely, suppose the DFS produces no such back edge. Since G is undirected, any forward edge would also be such a back edge, so there are no forward edges either. And DFS of an undirected graph cannot produce any cross edges. So, if the DFS produces no such back edge, all of the edges in G are tree edges (or reverses of tree edges). Since the trees in the DFS forest don't have cycles, G has no cycle. It follows that an undirected graph G has a cycle if and only if DFS on G produces such a back edge.

6.3.4 Discovery times and finish times

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 $\text{DFS1}(u)$ is called. The finish time is the time when $\text{DFS1}(u)$ completes. To explicitly compute these times, we could augment DFS as follows:

```

DFS( $G = (V, E)$ )
1. initialize visited  $\leftarrow \emptyset$ ; initialize  $t \leftarrow 1$ 
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  $(u, w) \in E$  out of  $u$ :
3.3.   if  $w \notin$  visited: DFS1( $w$ )
3.4. set  $f[u] = t$ ; increment  $t$     (set finish time for  $u$ )
4. for each vertex  $u \in V$ :
5.   if  $u \notin$  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. Discovery and finish times of the endpoints of an edge are related, depending on the type of the edge:

Observation 6.1. 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 $d[w] < f[w] < d[u] < f[u]$. (Happens only if G is a directed graph.)

6.3.5 Topologically sorting a directed graph in linear time

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

Lemma 6.4. *If digraph G has a cycle, then it cannot be topologically ordered.*

Proof. Suppose for contradiction that G has been topologically ordered, 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. $\square$

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 order 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 6.2. (i) *If digraph G has a back edge (u, w) , then G has a cycle.*
(ii) *If G has no back edge, then $f[u] > f[w]$ 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 6.1, $f[u] > f[w]$.

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).) After reordering, we have $f[v_1] > f[v_2] > \dots > f[v_n]$. By Observation 6.2(ii) each edge (v_i, v_j) has $f[v_i] > f[v_j]$, so has $i < j$. That is, the graph is now topologically ordered.

Python code: [graph_traversal/depth_first_search.py](#) (Section 6.4).

6.3.6 Strongly connected components in directed graphs

What is the meaning of “connected component” in a *directed* graph? The situation is more complicated than in undirected graphs, because it can be that there is a path from u to w , but no path from w to u . Two vertices u and w are said to be *strongly connected* if there is both a path from u to w and a path from w to u . This relation is reflexive, transitive and symmetric, so it partitions the vertex set into equivalence classes $C_1, C_2, \dots, C_k$, 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 from one strongly connected component to another, but any cycle in G must be wholly contained within a single component.

Given a source vertex s , an easy way to compute the strongly connected component containing s is as follows. Run DFS (or BFS) to find all vertices that s has a path to. Then run DFS (or BFS) on the graph *with all edges reversed* to find all vertices that have a path to s . (These will be the vertices that s has a path to in the reversed graph.) Then the strongly connected component containing s contains those vertices v in both sets—such that s has a path to v , and v has a path to s . Thus, one can identify the strongly connected component containing s in linear time.

The following DFS-based algorithm finds *all* the strongly connected components in linear time:

$\text{SCC}(G = (V, E))$

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 $T_1, T_2, \dots, T_k$ be the DFS trees in the resulting DFS forest.
5. Return $(C_1, C_2, \dots, C_k)$, where C_i contains the vertices in tree T_i .

Lemma 6.5. *The above algorithm returns the strongly connected components of G .*

For a proof, see [CLRS](#) Chapter 22.5.

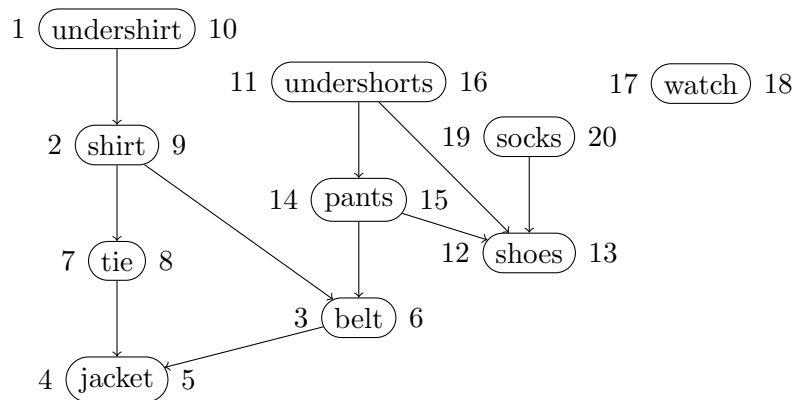
6.3.7 Exercises

External exercises with solutions

- [KT](#) Chapter 3: Solved Exercises 1 (topological ordering) and 2

Exercises without solutions

Exercise 6.1. (*DFS*)



The drawing above shows a directed acyclic graph (DAG) $G = (V, E)$, on which a depth-first search (DFS) has been run. Each node is labeled with its discovery time and its finish time during the DFS. Show the DFS forest for this execution of DFS. Draw all the nodes. Draw only the tree edges, not the non-tree edges. How many forward edges are there? How many cross edges are there? How many back edges are there?

Exercise 6.2. (*Cut vertices in undirected graphs.*) Fix any connected, undirected graph $G = (V, E)$. A vertex $v \in V$ is a *cut vertex* if there are vertices $u, w \in V \setminus \{v\}$ such that every u -to- w path in G goes through v . In other words, removing v from G would increase the number of connected components.

Consider any execution of DFS on G . As observed in Section 6.3, because G is a connected, undirected graph, the DFS forest will consist of just one rooted tree, say T , and each non-tree edge is both a back edge and a forward edge. There are no cross edges. In T , say that any tree edge (v, v') is *bypassed* by a back edge (u, w) if u is a descendant of both v and v' , and w is a proper ancestor of both v and v' . (In T , by definition, each vertex is its own descendant and ancestor. A *proper* ancestor of v is an ancestor other than v .)

Theorem 6.1. Let v be a vertex in V other than the root of the DFS tree T . Then v is a cut vertex if and only if, for some child v' of v in T , no back edge bypasses (v, v') .

Theorem 6.2. The root of the DFS tree T is a cut vertex if and only the root has at least two children in T .

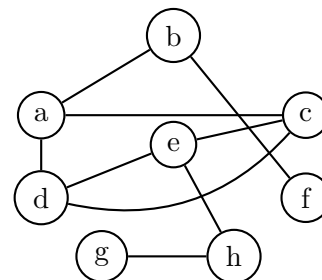
Give a long-form proof of Theorem 6.1.

(For context, these theorems are the basis of the following linear-time algorithm for finding the cut vertices in G . Run DFS on G . For each $v \in V$, define $\ell[v]$ (the *low number* of v) to be the minimum, over all back edges (u, w) out of any descendant u of v , of $d[w]$. Define $\ell[v] = \infty$ if there are no such back edges. When finishing the visit to vertex v , the low number of v can be computed via the recurrence

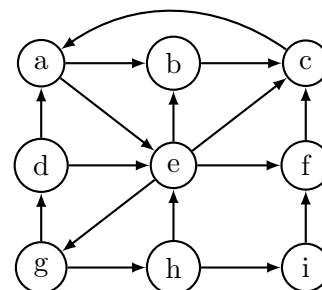
$$\ell[v] = \min \left(\min\{d[w] : (v, w) \text{ is a back edge}\}, \min\{\ell[v'] : v' \text{ is a child of } v\} \right).$$

By Theorem 6.1, (unless v is the root) v is a cut vertex iff it has a child v' such that (v, v') is not bypassed by some back edge (u, w) . This happens iff v has a child v' with $\ell[v'] \geq d[v]$. So, by computing and checking the low numbers as described above, and checking the number of children of the root, one can find all cut vertices in G in linear time.)

Exercise 6.3. (*BFS*) Simulate BFS on the graph to the right, starting from the node a . In your answer, just draw the resulting BFS tree and label each node with its distance from a .



Exercise 6.4. (*DFS*) Simulate DFS (as described in this note) on the graph to the right. When there are multiple choices for the next node to visit, break ties alphabetically (choose a before b , choose b before c , etc.).



- Draw the DFS forest that results.
- Label every edge as either tree, forward, backward, or cross.
- Label each vertex with its post-order number.
- Is this graph a directed acyclic graph (DAG)?
- If so, give the topological ordering that the DFS-based algorithm (from class and this note) would output after doing the DFS that you simulated above.
If not, how many strongly connected components does the graph have? List them all.

Exercise 6.5. (*Street Sweeper*) You have been hired to clean up the streets of Boston using your street sweeper. In order to do the job for the lowest cost, you want to devise a way to sweep each street once in each direction. (That is, for each street, you go up the street once and down the street once to sweep each side of the road.) All roads are two way. To find a good route, you formally define the following problem

STREET SWEEPER

input: a connected, undirected graph $G = (V, E)$

output: a (not necessarily simple) cycle C that traverses each edge (u, w) once in each direction¹

- (a) Give a correct output for the input which is the graph from Exercise 6.3.
- (b) Give a definition for a correct algorithm for STREET SWEEPER. For this part, explain clearly in a few English sentences exactly what your algorithm does.
- (c) Define your algorithm in pseudocode.
- (d) Prove your algorithm correct (i.e., gives the correct output for every input). You may cite (without reproving them) facts about DFS that we’ve observed in these notes. We recommend that you use the long-form proof format, as it will help you catch gaps and errors, and make your proof easier to check.
- (e) Give the best bound you can on the worst-case running time of your algorithm in terms of $n = |V|$ and $m = |E|$. Explain carefully what your reasoning is as to why the bound you give holds for all inputs. You may use (without reproving them) any facts about DFS that we’ve observed in these notes. Your algorithm should run in linear time.
- (f) Implement your algorithm and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyoung-algs101-street-sweeper>.

Exercise 6.6. (*DFS Edge Count*) Fix an undirected graph $G = (V, E)$, run DFS on G , and let F be the resulting DFS forest. For each vertex $u \in V$, let E_u be the set of edges in E that have both endpoints among the descendants of u in F (including u itself). Give a linear-time algorithm that computes $|E_u|$ for every vertex $u \in V$. Implement your algorithm and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyoung-algs101-dfs-edges-and-street-sweeper> (the “DFS Edge Count” challenge, which specifies how the DFS should break ties).

Exercises with solutions

Exercise 6.7. (*implementation*) Implement the algorithm for topological sorting in Section 6.3.5 and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyoung-algs101-dag-paths-and-topological-sort> (the “Topologically sort or find a cycle” challenge).

Solution. Python code: [graph_traversal/depth_first_search.py](#) (Section 6.4).

External exercises without solutions

- [JE](#) Chapter 5: Exercises 1–28
- [CLRS](#) Chapter 22: exercises in Sections 22.1 (background), 22.2 (BFS), 22.3 (DFS), 22.4 (topological sort), 22.5, (strongly connected components), and Problems 22–1 through 22–4 (Page 621)

¹Formally, the cycle C can be specified as a sequence of *directed* edges $(v_1, v_2), (v_2, v_3), \dots, (v_{\ell-1}, v_{\ell}), (v_{\ell}, v_1)$. In any cycle, each edge must start with the vertex that the edge before ends with (except the *first* edge must start with the vertex that the last edge ends with). Here, C should contain each edge $(u, w) \in E$ exactly twice: once as (u, w) and once as (w, u) .

- [DPV](#): exercises in Chapter 3 (DFS) and Chapter 4 Exercise 4.5
- [KT](#) Chapter 3: Exercises 1–12

6.3.8 External sources on graphs and linear-time graph traversal

- [JE](#) Chapter 5
- [CLRS](#) Chapter 22: Sections 22.1 (background), 22.2 (BFS), 22.3 (DFS), 22.4 (topological sort), 22.5, (strongly connected components)
- [DPV](#) Chapters 3 (DFS) and Sections 4.1–4.2 (BFS)
- [KT](#) Chapter 3
- MIT lecture video
 - <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/>

6.3.9 Appendix: Proof of correctness for Breadth-First Search

Here is the detailed proof of correctness for BFS. Fix any directed or undirected graph $G = (V, E)$ and source s . Consider executing $\text{BFS}(G = (V, E), s)$. Recall that *level* ℓ is $\{v \in V : \text{dist}(s, v) = \ell\}$, the set of vertices that are at distance ℓ from s . To start we prove some properties of distances.

Lemma 6.2 (levels). *Level 0 contains just the source s . Each level $\ell > 0$ consists of the vertices w such that (a) w is not in any level less than ℓ , and (b) there is an edge $(u, w) \in E$ from some vertex u in level $\ell - 1$. That is, each level $\ell > 0$ is $\{w \in V : \text{dist}(s, w) \geq \ell, (\exists(u, w) \in E) \text{dist}(s, u) = \ell - 1\}$.*

Proof (long form).

1. Level 0 contains just s because s is the only vertex reachable from s by a path of length 0.
2. Consider any level $\ell > 0$ and any vertex $w \in V$.
 - 3.1. Assume that Properties (a) and (b) hold for w .
 - 3.2. By Property (b), there is an edge (u, w) where $\text{dist}(s, u) = \ell - 1$.
 - 3.3. So there is a length- $(\ell - 1)$ path p from s to some vertex u with $(u, w) \in E$.
 - 3.4. Hence, there is a path from s to w of length ℓ . That is, $\text{dist}(s, w) \leq \ell$.
 - 3.5. This and (a) ($\text{dist}(s, w) \geq \ell$) imply $\text{dist}(s, w) = \ell$. That is, w is in level ℓ .
4. By the preceding block, if Properties (a) and (b) hold, then w is in level ℓ .
5. To finish we show the converse.
 - 6.1. Assume that w is in level ℓ , that is, that $\text{dist}(s, w) = \ell$. Property (a) clearly holds ($\text{dist}(s, w) \geq \ell$).
 - 6.2. Let p be a shortest path from s to w with length ℓ . (Such a path exists because $\text{dist}(s, w) = \ell$.)
 - 6.3. Let $(u, w) \in E$ be the last edge on path p . Let p' be the prefix of p from s to u .
 - 6.4. So p' has length $\ell - 1$, and no path from s to u has length less than $\ell - 1$ (because otherwise that path and (u, w) would form a path from s to w of length less than ℓ).
 - 6.5. So $\text{dist}(s, u) = \ell - 1$. That is, Property (b) also holds.
7. By the preceding block, if w is in level ℓ , then Properties (a) and (b) hold.
8. By this and Step 4, w is in level ℓ if and only if Properties (a) and (b) hold. □

Lemma 6.3 (correctness). *$\text{BFS}(G = (V, E), s)$ returns an array d with $d[v] = \text{dist}(s, v)$ for all $v \in V$.*

Proof (long form).

1. Consider any execution of BFS on some input $(G = (V, E), s)$.
2. We will show, for each level $\ell \geq 0$, by induction on ℓ , that the following three properties hold at some time t_ℓ during the execution of the while loop:
 - A_ℓ — The queue Q contains exactly those vertices in level ℓ .
 - B_ℓ — The set of vertices v for which $d[v]$ has been set is exactly the union of levels $0, 1, \dots, \ell$.
 - C_ℓ — Each array entry $d[v]$ that has been set has been set correctly, to $\text{dist}(s, v)$.
3. For the base case, level $\ell = 0$ is $\{s\}$, so A_0 , B_0 , and C_0 hold just before the first iteration.
- 4.1. Assume inductively that, for some level $\ell \geq 1$, $A_{\ell-1}$, $B_{\ell-1}$, and $C_{\ell-1}$ hold at some time $t_{\ell-1}$.
- 4.2. At time $t_{\ell-1}$, by property $A_{\ell-1}$, the queue contains exactly the vertices in level $\ell - 1$.
- 4.3. And (by $B_{\ell-1}$ and $C_{\ell-1}$) each vertex u in level $\ell - 1$ has $d[u]$ set correctly to $\ell - 1$.
- 4.4. Let t_ℓ be the end-time of the last iteration in which any level- $(\ell - 1)$ vertex is popped from Q , so that, between time $t_{\ell-1}$ and time t_ℓ , the alg. pops every vertex u in level $\ell - 1$ from Q (and, because Q is FIFO, pops no other vertex before time t_ℓ , so that at time t_ℓ the queue contains exactly the vertices added between times $t_{\ell-1}$ and t_ℓ).
- 4.5. For each such vertex u that it pops from Q , it adds each vertex w with an edge (u, w) such that $d[w]$ is not already set, and it sets $d[w]$ to $d[u] + 1 = (\ell - 1) + 1 = \ell$.
- 4.6. By $B_{\ell-1}$, $d[w]$ was set at time $t_{\ell-1}$ if and only if w is in level $\ell - 1$ or less.
- 4.7. So, between time $t_{\ell-1}$ and t_ℓ , the algorithm adds to Q those vertices w with an edge (u, w) such that u is a vertex in level $\ell - 1$ and w is not in a level less than ℓ .
- 4.8. By Lemma 6.2 these added vertices are just those in level ℓ .
- 4.9. So at time t_ℓ Property A_ℓ and B_ℓ hold. And $d[w] = \ell$ for each added w , so Property C_ℓ holds.
5. Inductively, for every level $\ell \geq 0$, Properties A_ℓ , B_ℓ , and C_ℓ hold at some time t_ℓ .
6. B_ℓ and C_ℓ hold for every level ℓ at some time during the while loop, and each $d[v]$ is only set once, for every vertex v with finite $\text{dist}(s, v)$, the while loop sets $d[v] = \text{dist}(s, v)$.
7. Line 4 of the algorithm sets $d[v] = \text{dist}(s, v) = \infty$ for each vertex v that is not in any level.
8. So, at termination, $d[v] = \text{dist}(s, v)$ for every $v \in V$. □

6.4 Python code

This section lists the Python code for this lecture note. Click on a file's name to open the file on GitHub. Copying code from GitHub is more reliable than copying it from this PDF, since some PDF viewers drop the leading spaces (the indentation) of copied lines.

[graph_traversal/graph_library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Hashable, Iterable

def out_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    out_neighbors: dict[T, list[T]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[tuple[int, int]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[tuple[int, int]]:
        if acyclic:
            priority = random.sample(range(n), n)

        edges: set[tuple[int, int]] = set()
        while len(edges) < n_edges:
            u, w = random.choice(vertices), random.choice(vertices)
            if u != w:
                if acyclic and priority[u] < priority[w]:
                    u, w = w, u
                edges.add((u, w))
        return edges

    if m <= n_pairs / 2:
        edges = random_edge_set(m)
    else:
```

```

        # if most edges are present, faster to generate the complement
        all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
        assert len(all_pairs) == n_pairs
        edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
    assert len(edges) == m
    return vertices, list(edges)

def random_graph(n: int = 10, m: int | None = None) -> tuple[list[int], list[tuple[int, int]]]:
    V, E = random_digraph(n, m, acyclic=True)
    E += [(w, u) for u, w in E]
    return V, E

```

graph_traversal/breadth_first_search.py

```

#!/usr/bin/env python3

from collections import deque
from collections.abc import Hashable, Iterable, Mapping
from math import inf

from graph_library import in_neighbor_lists, out_neighbor_lists, random_digraph

def minf(collection: Iterable[float]) -> float:
    return min(collection, default=inf)

def breadth_first_search[T: Hashable](
    V: Iterable[T], E: Iterable[tuple[T, T]], source: T | None = None
) -> dict[T, float]:
    if source is None:
        source = next(v for v in V)

    neighbors = out_neighbor_lists(V, E)

    queue: deque[T] = deque()
    d: dict[T, float] = {source: 0}
    queue.append(source)

    while queue:
        u = queue.popleft()
        for w in neighbors[u]:
            if w not in d:
                d[w] = d[u] + 1
                queue.append(w)

    for v in V:
        if v not in d:
            d[v] = inf

    return d

def check_unweighted_shortest_path_distances[T: Hashable](
    V: Iterable[T], E: Iterable[tuple[T, T]], source: T, d: Mapping[T, float]
) -> bool:
    in_neighbors = in_neighbor_lists(V, E)

```

```

    return d[source] == 0 and all(
        d[w] == minf(d[u] + 1 for u in in_neighbors[w]) for w in V if w != source
    )

if __name__ == "__main__":

    def test_BFS(n: int = 10, m: int | None = None) -> None:
        V, E = random_digraph(n, m)
        source = next(v for v in V)
        d = breadth_first_search(V, E, source)
        assert check_unweighted_shortest_path_distances(V, E, source, d)

    for _ in range(50):
        test_BFS(100)

    print("test passed")

```

[graph_traversal/depth_first_search.py](#)

```

#!/usr/bin/env python3

from collections.abc import Callable, Hashable, Iterable

from graph_library import out_neighbor_lists, random_digraph, random_graph

type Visit[T] = Callable[[T, T | None], object]

def depth_first_search[T: Hashable](
    V: Iterable[T],
    E: Iterable[tuple[T, T]],
    previsit: Visit[T] | None = None,
    postvisit: Visit[T] | None = None,
) -> None:
    neighbors = out_neighbor_lists(V, E)
    visited: set[T] = set()

    def DFS1(u: T, parent: T | None = None) -> None:
        visited.add(u)
        if previsit:
            previsit(u, parent)
        for w in neighbors[u]:
            if w not in visited:
                DFS1(w, u)
        if postvisit:
            postvisit(u, parent)

    for v in V:
        if v not in visited:
            DFS1(v)

def topological_sort[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> list[T] | None:
    """Return the vertices in topological order, or None if the graph has a cycle."""
    E = list(E)
    finished: list[T] = []

    def postvisit(u: T, parent: T | None = None) -> None:

```

```

        finished.append(u)

    depth_first_search(V, E, postvisit=postvisit)

    # G has a cycle iff the DFS has a back edge, i.e., an edge (u, w) with w finishing no earlier
    # than u (for tree, forward and cross edges, w finishes before u; a self-loop is a back edge)
    finish_time = {u: t for t, u in enumerate(finished)}
    if any(finish_time[w] >= finish_time[u] for u, w in E):
        return None

    return list(reversed(finished))

def test_topological_sort(n: int = 10, m: int | None = None) -> None:
    from random import shuffle

    V, E = random_digraph(n, m, acyclic=True)

    shuffle(V)
    order = topological_sort(V, E)
    assert order is not None

    rank = dict((v, i) for i, v in enumerate(order))

    assert all(rank[u] < rank[w] for u, w in E)

    # adding an edge that closes a cycle (or a self-loop) must be detected
    if E:
        u, w = E[0]
        assert topological_sort(V, E + [(w, u)]) is None
        assert topological_sort(V, E + [(V[0], V[0])]) is None

def count_forward_edges[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> int:
    neighbors = out_neighbor_lists(V, E)
    visited: set[T] = set()
    being_visited: set[T] = set()
    n_forward_edges = 0

    def DFS1(u: T) -> None:
        nonlocal n_forward_edges

        visited.add(u)
        being_visited.add(u)

        for w in neighbors[u]:
            if w not in visited:
                DFS1(w)
            elif w not in being_visited:
                n_forward_edges += 1

        being_visited.remove(u)

    for v in V:
        if v not in visited:
            DFS1(v)

    return n_forward_edges

def count_connected_components[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> int:

```

```

    # (V, E) is undirected graph

    n_components = 0

    def previsit(u: T, parent: T | None) -> None:
        nonlocal n_components
        if parent is None:
            n_components += 1

    depth_first_search(V, E, previsit=previsit)

    return n_components

def test_count_forward_edges(n: int = 10, m: int | None = None) -> None:
    V, E = random_graph(n, m)
    assert n == len(V)

    n_components = count_connected_components(V, E)
    n_tree_edges = n - n_components
    n_forward_edges = count_forward_edges(V, E)

    assert n_tree_edges + n_forward_edges == len(E) / 2

# def strong_components(G):
#     assert isinstance(G, DiGraph)

#     finished = []
#     depth_first_search(G, postvisit=lambda u, _: finished.append(u))

#     G_reversed = DiGraph(vertices=list(reversed(finished)),
#                           edges=set((w, u) for (u, w) in G.edges))

#     components = []

#     def previsit(u, tree_edge):
#         if tree_edge is None:
#             components.append([])
#         components[-1].append(u)

#     depth_first_search(G_reversed, previsit)

#     return components

# class DFS_edge_classifier:
#     def __init__(self, G):
#         self.G = G
#         self.start_times = {}
#         self.finish_times = {}

#         counter = 0

#         def time():
#             nonlocal counter
#             counter += 1
#             return counter

#         self.tree_edges = set()

```

```

#         def previsit(u, tree_edge):
#             if tree_edge is not None:
#                 self.tree_edges.add(tree_edge)
#                 self.start_times[u] = time()

#         def postvisit(u, _):
#             self.finish_times[u] = time()

#         depth_first_search(G, previsit, postvisit)

#         self.forward_edges = set()
#         self.back_edges = set()
#         self.cross_edges = set()

#         for e in G.edges:
#             if e not in self.tree_edges:
#                 u, w = e
#                 s_u, f_u = self.start_times[u], self.finish_times[u]
#                 s_w, f_w = self.start_times[w], self.finish_times[w]
#                 if e == {u, w}: # undirected edge
#                     assert s_w < s_u < f_u < f_w or s_u < s_w < f_w < f_u
#                     self.forward_edges.add(e)
#                 elif s_u < s_w < f_w < f_u:
#                     self.forward_edges.add(e)
#                 elif s_w < s_u < f_u < f_w:
#                     self.back_edges.add(e)
#                 elif s_w < f_w < s_u < f_u:
#                     self.cross_edges.add(e)
#                 else:
#                     assert False

if __name__ == "__main__":
    for _ in range(5):
        test_topological_sort(10)

    for _ in range(5):
        test_count_forward_edges(10)

    print("test passed")
    
```