14.2 Data Structures for Graphs

In this section, we introduce four data structures for representing a graph. In each representation, we maintain a collection to store the vertices of a graph. However, the four representations differ greatly in the way they organize the edges.

- In an ***edge list***, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v.
- In an ***adjacency list***, we maintain, for each vertex, a separate list containing those edges that are incident to the vertex. The complete set of edges can be determined by taking the union of the smaller sets, while the organization allows us to more efficiently find all edges incident to a given vertex.
- An ***adjacency map*** is very similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows for access to a specific edge (u,v) in $O(1)$ expected time.
- An ***adjacency matrix*** provides worst-case $O(1)$ access to a specific edge (u,v) by maintaining an $n \times n$ matrix, for a graph with n vertices. Each entry is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the entry will be None.

A summary of the performance of these structures is given in Table 14.1. We give further explanation of the structures in the remainder of this section.

Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Table 14.1: A summary of the running times for the methods of the graph ADT, using the graph representations discussed in this section. We let n denote the number of vertices, m the number of edges, and d_v the degree of vertex v. Note that the adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n+m)$ space.

14.2.1 Edge List Structure

The ***edge list*** structure is possibly the simplest, though not the most efficient, representation of a graph G. All vertex objects are stored in an unordered list V, and all edge objects are stored in an unordered list E. We illustrate an example of the edge list structure for a graph G in Figure 14.4.

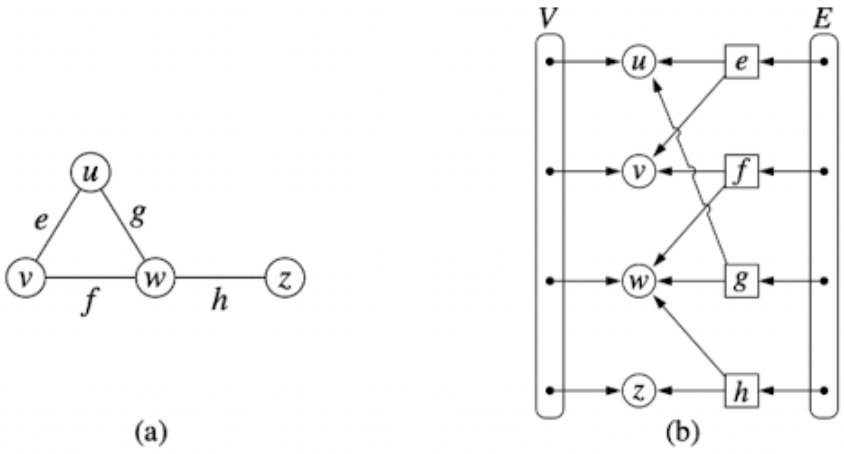


Figure 14.4: (a) A graph G; (b) schematic representation of the edge list structure for G. Notice that an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

To support the many methods of the Graph ADT (Section 14.1), we assume the following additional features of an edge list representation. Collections V and E are represented with doubly linked lists using our PositionalList class from Chapter 7.

Vertex Objects

The vertex object for a vertex v storing element x has instance variables for:

- A reference to element x, to support the element( ) method.
- A reference to the position of the vertex instance in the list V, thereby allowing v to be efficiently removed from V if it were removed from the graph.

Edge Objects

The edge object for an edge e storing element x has instance variables for:

- A reference to element x, to support the element( ) method.
- References to the vertex objects associated with the endpoint vertices of e. These allow the edge instance to provide constant-time support for methods endpoints( ) and opposite(v).
- A reference to the position of the edge instance in list E, thereby allowing e to be efficiently removed from E if it were removed from the graph.

Performance of the Edge List Structure

The performance of an edge list structure in fulfilling the graph ADT is summarized in Table 14.2. We begin by discussing the space usage, which is $O(n+m)$ for representing a graph with n vertices and m edges. Each individual vertex or edge instance uses $O(1)$ space, and the additional lists V and E use space proportional to their number of entries.

In terms of running time, the edge list structure does as well as one could hope in terms of reporting the number of vertices or edges, or in producing an iteration of those vertices or edges. By querying the respective list V or E, the vertex_count and edge_count methods run in $O(1)$ time, and by iterating through the appropriate list, the methods vertices and edges run respectively in $O(n)$ and $O(m)$ time.

The most significant limitations of an edge list structure, especially when compared to the other graph representations, are the $O(m)$ running times of methods get_edge(u,v), degree(v), and incident_edges(v). The problem is that with all edges of the graph in an unordered list E, the only way to answer those queries is through an exhaustive inspection of all edges. The other data structures introduced in this section will implement these methods more efficiently.

Finally, we consider the methods that update the graph. It is easy to add a new vertex or a new edge to the graph in $O(1)$ time. For example, a new edge can be added to the graph by creating an Edge instance storing the given element as data, adding that instance to the positional list E, and recording its resulting Position within E as an attribute of the edge. That stored position can later be used to locate and remove this edge from E in $O(1)$ time, and thus implement the method remove_edge(e)

It is worth discussing why the remove_vertex(v) method has a running time of $O(m)$. As stated in the graph ADT, when a vertex v is removed from the graph, all edges incident to v must also be removed (otherwise, we would have a contradiction of edges that refer to vertices that are not part of the graph). To locate the incident edges to the vertex, we must examine all edges of E.

Operation	Running Time
vertex_count(), edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

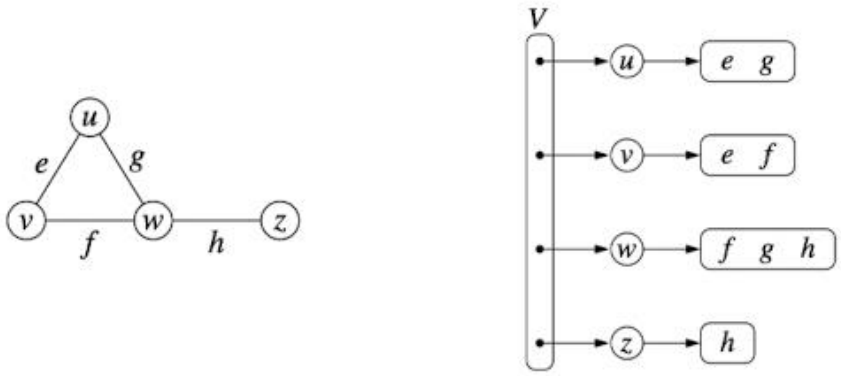
Table 14.2: Running times of the methods of a graph implemented with the edge list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

14.2.2 Adjacency List Structure

In contrast to the edge list representation of a graph, the ***adjacency list*** structure groups the edges of a graph by storing them in smaller, secondary containers that are associated with each individual vertex. Specifically, for each vertex v, we maintain a collection $I(v)$, called the ***incidence collection*** of v, whose entries are edges incident to v. (In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{\text{out}}(v)$ and $I_{\text{in}}(v)$.) Traditionally, the incidence collection $I(v)$ for a vertex v is a list, which is why we call this way of representing a graph the ***adjacency list*** structure.

We require that the primary structure for an adjacency list maintain the collection V of vertices in a way so that we can locate the secondary structure $I(v)$ for a given vertex v in $O(1)$ time. This could be done by using a positional list to represent V, with each Vertex instance maintaining a direct reference to its $I(v)$ incidence collection; we illustrate such an adjacency list structure of a graph in Figure 14.5. If vertices can be uniquely numbered from 0 to $n-1$, we could instead use a primary array-based structure to access the appropriate secondary lists.

The primary benefit of an adjacency list is that the collection $I(v)$ contains exactly those edges that should be reported by the method incident_edges(v). Therefore, we can implement this method by iterating the edges of $I(v)$ in $O(\deg(v))$ time, where $\deg(v)$ is the degree of vertex v. This is the best possible outcome for any graph representation, because there are $\deg(v)$ edges to be reported.



(a) (b)

Figure 14.5: (a) An undirected graph G; (b) a schematic representation of the adjacency list structure for G. Collection V is the primary list of vertices, and each vertex has an associated list of incident edges. Although not diagrammed as such, we presume that each edge of the graph is represented with a unique Edge instance that maintains references to its endpoint vertices.

Performance of the Adjacency List Structure

Table 14.3 summarizes the performance of the adjacency list structure implementation of a graph, assuming that the primary collection V and all secondary collections $I(v)$ are implemented with doubly linked lists.

Asymptotically, the space requirements for an adjacency list are the same as an edge list structure, using $O(n+m)$ space for a graph with n vertices and m edges. The primary list of vertices uses $O(n)$ space. The sum of the lengths of all secondary lists is $O(m)$, for reasons that were formalized in Propositions 14.8 and 14.9. In short, an undirected edge (u,v) is referenced in both $I(u)$ and $I(v)$, but its presence in the graph results in only a constant amount of additional space.

We have already noted that the incident_edges(v) method can be achieved in $O(\deg(v))$ time based on use of $I(v)$. We can achieve the degree(v) method of the graph ADT to use $O(1)$ time, assuming collection $I(v)$ can report its size in similar time. To locate a specific edge for implementing get_edge(u,v), we can search through either $I(u)$ and $I(v)$. By choosing the smaller of the two, we get $O(\min(\deg(u),\deg(v)))$ running time.

The rest of the bounds in Table 14.3 can be achieved with additional care. To efficiently support deletions of edges, an edge (u,v) would need to maintain a reference to its positions within both $I(u)$ and $I(v)$, so that it could be deleted from those collections in $O(1)$ time. To remove a vertex v, we must also remove any incident edges, but at least we can locate those edges in $O(\deg(v))$ time.

The easiest way to support edges( ) in $O(m)$ and count_edges( ) in $O(1)$ is to maintain an auxiliary list E of edges, as in the edge list representation. Otherwise, we can implement the edges method in $O(n+m)$ time by accessing each secondary list and reporting its edges, taking care not to report an undirected edge (u,v) twice.

Operation	Running Time
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v)	$O(\min(\deg(u),\deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

Table 14.3: Running times of the methods of a graph implemented with the adjacency list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

14.2.3 Adjacency Map Structure

In the adjacency list structure, we assume that the secondary incidence collections are implemented as unordered linked lists. Such a collection $I(v)$ uses space proportional to $O(\deg(v))$, allows an edge to be added or removed in $O(1)$ time, and allows an iteration of all edges incident to vertex v in $O(\deg(v))$ time. However, the best implementation of get_edge(u,v) requires $O(\min(\deg(u), \deg(v)))$ time, because we must search through either $I(u)$ or $I(v)$.

We can improve the performance by using a hash-based map to implement $I(v)$ for each vertex v. Specifically, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value. We call such a graph representation an ***adjacency map***. (See Figure 14.6.) The space usage for an adjacency map remains $O(n+m)$, because $I(v)$ uses $O(\deg(v))$ space for each vertex v, as with the adjacency list.

The advantage of the adjacency map, relative to an adjacency list, is that the get_edge(u,v) method can be implemented in ***expected*** $O(1)$ time by searching for vertex u as a key in $I(v)$, or vice versa. This provides a likely improvement over the adjacency list, while retaining the worst-case bound of $O(\min(\deg(u), \deg(v)))$.

In comparing the performance of adjacency map to other representations (see Table 14.1), we find that it essentially achieves optimal running times for all methods, making it an excellent all-purpose choice as a graph representation.

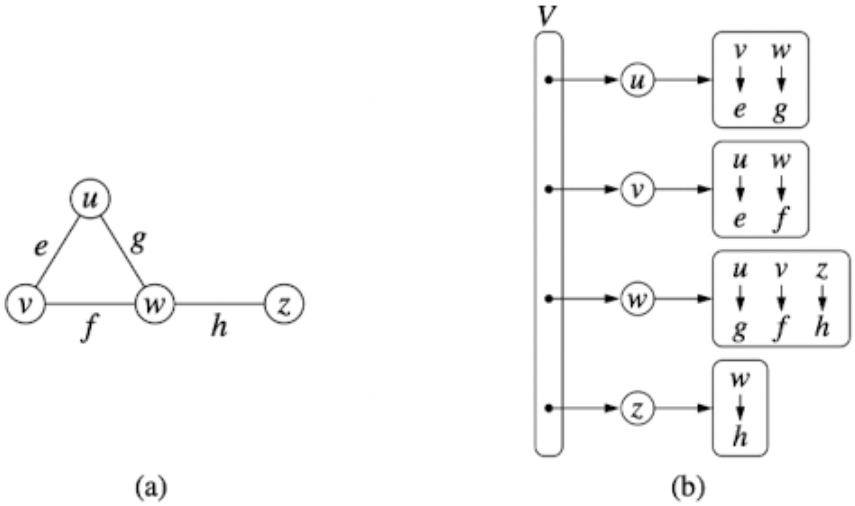


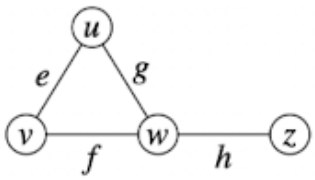
Figure 14.6: (a) An undirected graph G; (b) a schematic representation of the adjacency map structure for G. Each vertex maintains a secondary map in which neighboring vertices serve as keys, with the connecting edges as associated values. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge of the graph, and that it maintains references to its endpoint vertices.

14.2.4 Adjacency Matrix Structure

The ***adjacency matrix*** structure for a graph G augments the edge list structure with a matrix A (that is, a two-dimensional array, as in Section 5.6), which allows us to locate an edge between a given pair of vertices in *worst-case* constant time. In the adjacency matrix representation, we think of the vertices as being the integers in the set $\{0,1,\ldots,n-1\}$ and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional $n \times n$ array A. Specifically, the cell $A[i,j]$ holds a reference to the edge (u,v), if it exists, where u is the vertex with index i and v is the vertex with index j. If there is no such edge, then $A[i,j] =$ None. We note that array A is symmetric if graph G is undirected, as $A[i,j] = A[j,i]$ for all pairs i and j. (See Figure 14.7.)

The most significant advantage of an adjacency matrix is that any edge (u,v) can be accessed in worst-case $O(1)$ time; recall that the adjacency map supports that operation in $O(1)$ *expected* time. However, several operation are less efficient with an adjacency matrix. For example, to find the edges incident to vertex v, we must presumably examine all n entries in the row associated with v; recall that an adjacency list or map can locate those edges in optimal $O(\deg(v))$ time. Adding or removing vertices from a graph is problematic, as the matrix must be resized.

Furthermore, the $O(n^2)$ space usage of an adjacency matrix is typically far worse than the $O(n+m)$ space required of the other representations. Although, in the worst case, the number of edges in a ***dense*** graph will be proportional to n^2, most real-world graphs are ***sparse***. In such cases, use of an adjacency matrix is inefficient. However, if a graph is dense, the constants of proportionality of an adjacency matrix can be smaller than that of an adjacency list or map. In fact, if edges do not have auxiliary data, a Boolean adjacency matrix can use one bit per edge slot, such that $A[i,j] =$ True if and only if associated (u,v) is an edge.



(a)

	0	1	2	3
u → 0		e	g	
v → 1	e		f	
w → 2	g	f		h
z → 3			h	

(b)

Figure 14.7: (a) An undirected graph G; (b) a schematic representation of the auxiliary adjacency matrix structure for G, in which n vertices are mapped to indices 0 to $n-1$. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge, and that it maintains references to its endpoint vertices. We also assume that there is a secondary edge list (not pictured), to allow the edges( ) method to run in $O(m)$ time, for a graph with m edges.

14.2.5 Python Implementation

In this section, we provide an implementation of the Graph ADT. Our implementation will support directed or undirected graphs, but for ease of explanation, we first describe it in the context of an undirected graph.

We use a variant of the ***adjacency map*** representation. For each vertex v, we use a Python dictionary to represent the secondary incidence map $I(v)$. However, we do not explicitly maintain lists V and E, as originally described in the edge list representation. The list V is replaced by a top-level dictionary D that maps each vertex v to its incidence map $I(v)$; note that we can iterate through all vertices by generating the set of keys for dictionary D. By using such a dictionary D to map vertices to the secondary incidence maps, we need not maintain references to those incidence maps as part of the vertex structures. Also, a vertex does not need to explicitly maintain a reference to its position in D, because it can be determined in $O(1)$ expected time. This greatly simplifies our implementation. However, a consequence of our design is that some of the worst-case running time bounds for the graph ADT operations, given in Table 14.1, become ***expected*** bounds. Rather than maintain list E, we are content with taking the union of the edges found in the various incidence maps; technically, this runs in $O(n+m)$ time rather than strictly $O(m)$ time, as the dictionary D has n keys, even if some incidence maps are empty.

Our implementation of the graph ADT is given in Code Fragments 14.1 through 14.3. Classes Vertex and Edge, given in Code Fragment 14.1, are rather simple, and can be nested within the more complex Graph class. Note that we define the `__hash__` method for both Vertex and Edge so that those instances can be used as keys in Python's hash-based sets and dictionaries. The rest of the Graph class is given in Code Fragments 14.2 and 14.3. Graphs are undirected by default, but can be declared as directed with an optional parameter to the constructor.

Internally, we manage the directed case by having two different top-level dictionary instances, `_outgoing` and `_incoming`, such that `_outgoing[v]` maps to another dictionary representing $I_{\text{out}}(v)$, and `_incoming[v]` maps to a representation of $I_{\text{in}}(v)$. In order to unify our treatment of directed and undirected graphs, we continue to use the `_outgoing` and `_incoming` identifiers in the undirected case, yet as aliases to the same dictionary. For convenience, we define a utility named `is_directed` to allow us to distinguish between the two cases.

For methods `degree` and `incident_edges`, which each accept an optional parameter to differentiate between the outgoing and incoming orientations, we choose the appropriate map before proceeding. For method `insert_vertex`, we always initialize `_outgoing[v]` to an empty dictionary for new vertex v. In the directed case, we independently initialize `_incoming[v]` as well. For the undirected case, that step is unnecessary as `_outgoing` and `_incoming` are aliases. We leave the implementations of methods `remove_vertex` and `remove_edge` as exercises (C-14.37 and C-14.38).

```
1   #---------------------- nested Vertex class ----------------------
2   class Vertex:
3     """Lightweight vertex structure for a graph."""
4     __slots__ = '_element'
5
6     def __init__(self, x):
7       """Do not call constructor directly. Use Graph's insert_vertex(x)."""
8       self._element = x
9
10    def element(self):
11      """Return element associated with this vertex."""
12      return self._element
13
14    def __hash__(self):             # will allow vertex to be a map/set key
15      return hash(id(self))
16
17  #---------------------- nested Edge class ----------------------
18  class Edge:
19    """Lightweight edge structure for a graph."""
20    __slots__ = '_origin', '_destination', '_element'
21
22    def __init__(self, u, v, x):
23      """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
24      self._origin = u
25      self._destination = v
26      self._element = x
27
28    def endpoints(self):
29      """Return (u,v) tuple for vertices u and v."""
30      return (self._origin, self._destination)
31
32    def opposite(self, v):
33      """Return the vertex that is opposite v on this edge."""
34      return self._destination if v is self._origin else self._origin
35
36    def element(self):
37      """Return element associated with this edge."""
38      return self._element
39
40    def __hash__(self):             # will allow edge to be a map/set key
41      return hash( (self._origin, self._destination) )
```

Code Fragment 14.1: Vertex and Edge classes (to be nested within Graph class).

```
1  class Graph:
2    """Representation of a simple graph using an adjacency map."""
3
4    def __init__(self, directed=False):
5      """Create an empty graph (undirected, by default).
6
7      Graph is directed if optional paramter is set to True.
8      """
9      self._outgoing = { }
10     # only create second map for directed graph; use alias for undirected
11     self._incoming = { } if directed else self._outgoing
12
13   def is_directed(self):
14     """Return True if this is a directed graph; False if undirected.
15
16     Property is based on the original declaration of the graph, not its contents.
17     """
18     return self._incoming is not self._outgoing # directed if maps are distinct
19
20   def vertex_count(self):
21     """Return the number of vertices in the graph."""
22     return len(self._outgoing)
23
24   def vertices(self):
25     """Return an iteration of all vertices of the graph."""
26     return self._outgoing.keys( )
27
28   def edge_count(self):
29     """Return the number of edges in the graph."""
30     total = sum(len(self._outgoing[v]) for v in self._outgoing)
31     # for undirected graphs, make sure not to double-count edges
32     return total if self.is_directed( ) else total // 2
33
34   def edges(self):
35     """Return a set of all edges of the graph."""
36     result = set( )          # avoid double-reporting edges of undirected graph
37     for secondary_map in self._outgoing.values( ):
38       result.update(secondary_map.values( ))      # add edges to resulting set
39     return result
```

Code Fragment 14.2: Graph class definition (continued in Code Fragment 14.3).

```
40    def get_edge(self, u, v):
41      """Return the edge from u to v, or None if not adjacent."""
42      return self._outgoing[u].get(v)          # returns None if v not adjacent
43
44    def degree(self, v, outgoing=True):
45      """Return number of (outgoing) edges incident to vertex v in the graph.
46
47      If graph is directed, optional parameter used to count incoming edges.
48      """
49      adj = self._outgoing if outgoing else self._incoming
50      return len(adj[v])
51
52    def incident_edges(self, v, outgoing=True):
53      """Return all (outgoing) edges incident to vertex v in the graph.
54
55      If graph is directed, optional parameter used to request incoming edges.
56      """
57      adj = self._outgoing if outgoing else self._incoming
58      for edge in adj[v].values( ):
59        yield edge
60
61    def insert_vertex(self, x=None):
62      """Insert and return a new Vertex with element x."""
63      v = self.Vertex(x)
64      self._outgoing[v] = { }
65      if self.is_directed( ):
66        self._incoming[v] = { }          # need distinct map for incoming edges
67      return v
68
69    def insert_edge(self, u, v, x=None):
70      """Insert and return a new Edge from u to v with auxiliary element x."""
71      e = self.Edge(u, v, x)
72      self._outgoing[u][v] = e
73      self._incoming[v][u] = e
```

Code Fragment 14.3: Graph class definition (continued from Code Fragment 14.2). We omit error-checking of parameters for brevity.

14.3 Graph Traversals

Greek mythology tells of an elaborate labyrinth that was built to house the monstrous Minotaur, which was part bull and part man. This labyrinth was so complex that neither beast nor human could escape it. No human, that is, until the Greek hero, Theseus, with the help of the king's daughter, Ariadne, decided to implement a ***graph traversal*** algorithm. Theseus fastened a ball of thread to the door of the labyrinth and unwound it as he traversed the twisting passages in search of the monster. Theseus obviously knew about good algorithm design, for, after finding and defeating the beast, Theseus easily followed the string back out of the labyrinth to the loving arms of Ariadne.

Formally, a ***traversal*** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of ***reachability***, that is, in determining how to travel from one vertex to another while following paths of a graph. Interesting problems that deal with reachability in an undirected graph G include the following:

- Computing a path from vertex u to vertex v, or reporting that no such path exists.
- Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Computing a cycle in G, or reporting that G has no cycles.

Interesting problems that deal with reachability in a directed graph $\vec{G}$ include the following:

- Computing a directed path from vertex u to vertex v, or reporting that no such path exists.
- Finding all the vertices of $\vec{G}$ that are reachable from a given vertex s.
- Determine whether $\vec{G}$ is acyclic.
- Determine whether $\vec{G}$ is strongly connected.

In the remainder of this section, we present two efficient graph traversal algorithms, called ***depth-first search*** and ***breadth-first search***, respectively.

14.3.1 Depth-First Search

The first traversal algorithm we consider in this section is ***depth-first search*** (DFS). Depth-first search is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

Depth-first search in a graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost. We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "visited." The vertex s is now our "current" vertex—call our current vertex u. We then traverse G by considering an (arbitrary) edge (u,v) incident to the current vertex u. If the edge (u,v) leads us to a vertex v that is already visited (that is, painted), we ignore that edge. If, on the other hand, (u,v) leads to an unvisited vertex v, then we unroll our string, and go to v. We then paint v as "visited," and make it the current vertex, repeating the computation above. Eventually, we will get to a "dead end," that is, a current vertex v such that all the edges incident to v lead to vertices already visited. To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to v, going back to a previously visited vertex u. We then make u our current vertex and repeat the computation above for any edges incident to u that we have not yet considered. If all of u's incident edges lead to visited vertices, then we again roll up our string and backtrack to the vertex we came from to get to u, and repeat the procedure at that vertex. Thus, we continue to backtrack along the path that we have traced so far until we find a vertex that has yet unexplored edges, take one such edge, and continue the traversal. The process terminates when our backtracking leads us back to the start vertex s, and there are no more unexplored edges incident to s.

The pseudo-code for a depth-first search traversal starting at a vertex u (see Code Fragment 14.4) follows our analogy with string and paint. We use recursion to implement the string analogy, and we assume that we have a mechanism (the paint analogy) to determine whether a vertex or edge has been previously explored.

```
Algorithm DFS(G,u):          {We assume u has already been marked as visited}
    Input: A graph G and a vertex u of G
    Output: A collection of vertices reachable from u, with their discovery edges
    for each outgoing edge e = (u,v) of u do
        if vertex v has not been visited then
            Mark vertex v as visited (via edge e).
            Recursively call DFS(G,v).
```

Code Fragment 14.4: The DFS algorithm.

Classifying Graph Edges with DFS

An execution of depth-first search can be used to analyze the structure of a graph, based upon the way in which edges are explored during the traversal. The DFS process naturally identifies what is known as the ***depth-first search tree*** rooted at a starting vertex s. Whenever an edge $e = (u,v)$ is used to discover a new vertex v during the DFS algorithm of Code Fragment 14.4, that edge is known as a ***discovery edge*** or ***tree edge***, as oriented from u to v. All other edges that are considered during the execution of DFS are known as ***nontree edges***, which take us to a previously visited vertex. In the case of an undirected graph, we will find that all nontree edges that are explored connect the current vertex to one that is an ancestor of it in the DFS tree. We will call such an edge a ***back edge***. When performing a DFS on a directed graph, there are three possible kinds of nontree edges:

- ***back edges***, which connect a vertex to an ancestor in the DFS tree
- ***forward edges***, which connect a vertex to a descendant in the DFS tree
- ***cross edges***, which connect a vertex to a vertex that is neither its ancestor nor its descendant.

An example application of the DFS algorithm on a directed graph is shown in Figure 14.8, demonstrating each type of nontree edge. An example application of the DFS algorithm on an undirected graph is shown in Figure 14.9.

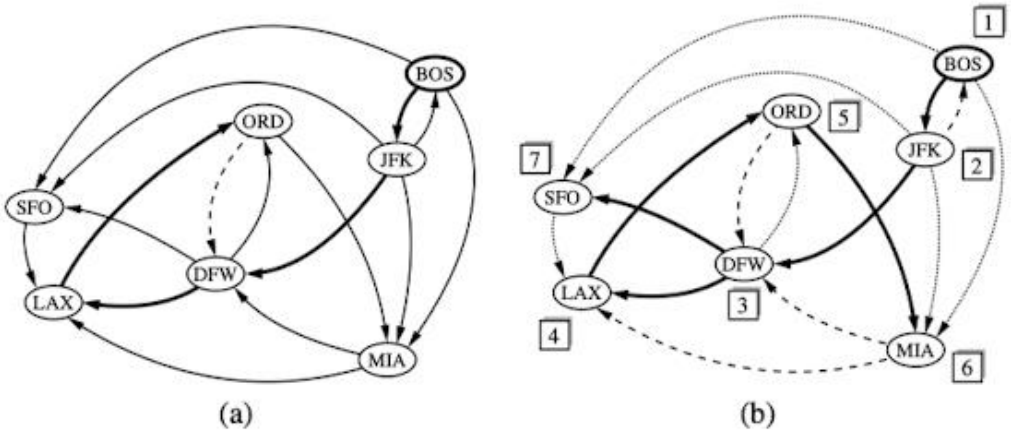


Figure 14.8: An example of a DFS in a directed graph, starting at vertex (BOS): (a) intermediate step, where, for the first time, a considered edge leads to an already visited vertex (DFW); (b) the completed DFS. The tree edges are shown with thick lines, the back edges are shown with dashed lines, and the forward and cross edges are shown with dotted lines. The order in which the vertices are visited is indicated by a label next to each vertex. The edge (ORD,DFW) is a back edge, but (DFW,ORD) is a forward edge. Edge (BOS,SFO) is a forward edge, and (SFO,LAX) is a cross edge.

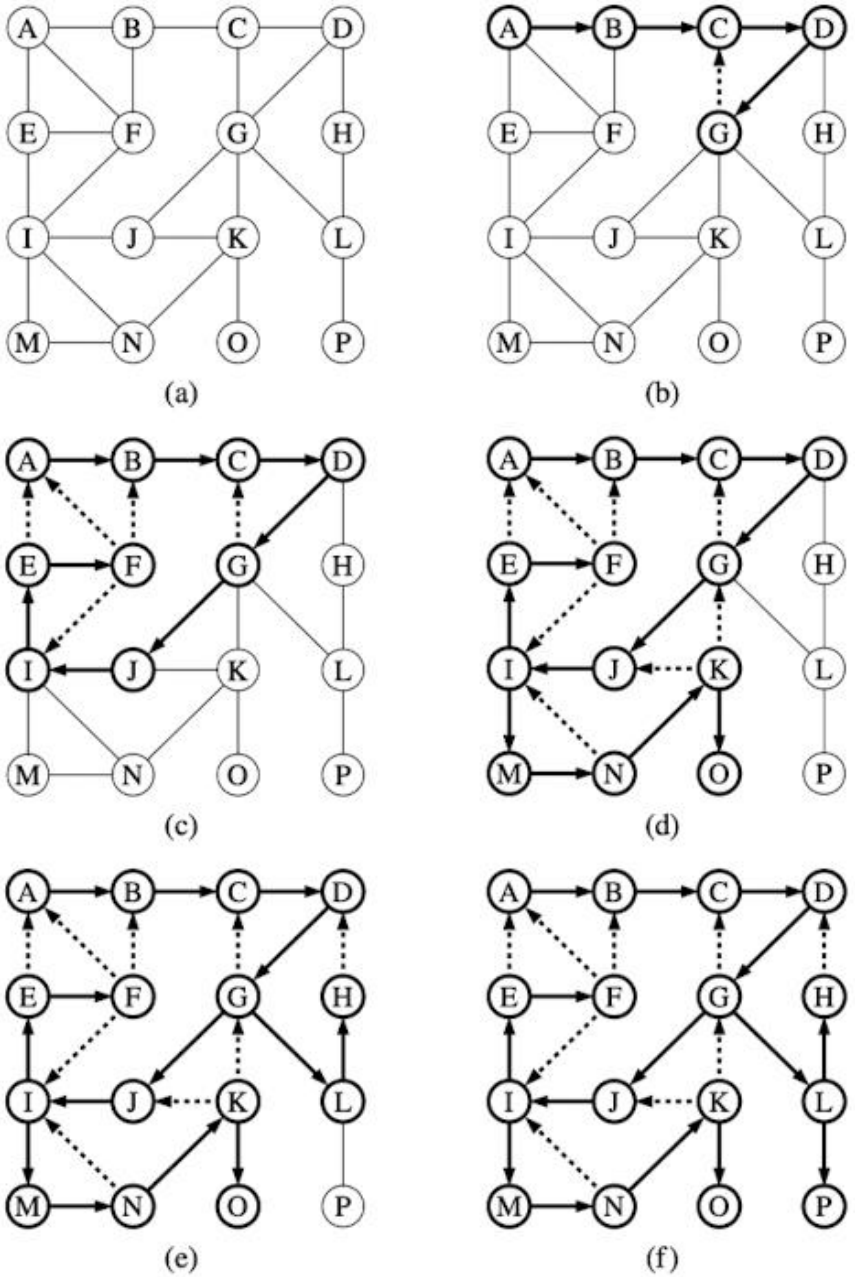


Figure 14.9: Example of depth-first search traversal on an undirected graph starting at vertex *A*. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph; (b) path of tree edges, traced from A until back edge (G,C) is examined; (c) reaching F, which is a dead end; (d) after backtracking to I, resuming with edge (I,M), and hitting another dead end at O; (e) after backtracking to G, continuing with edge (G,L), and hitting another dead end at H; (f) final result.

Properties of a Depth-First Search

There are a number of observations that we can make about the depth-first search algorithm, many of which derive from the way the DFS algorithm partitions the edges of a graph G into groups. We begin with the most significant property.

Proposition 14.12: *Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s.*

Justification: Suppose there is at least one vertex w in s's connected component not visited, and let v be the first unvisited vertex on some path from s to w (we may have $v = w$). Since v is the first unvisited vertex on this path, it has a neighbor u that was visited. But when we visited u, we must have considered the edge (u, v); hence, it cannot be correct that v is unvisited. Therefore, there are no unvisited vertices in s's connected component.

Since we only follow a discovery edge when we go to an unvisited vertex, we will never form a cycle with such edges. Therefore, the discovery edges form a connected subgraph without cycles, hence a tree. Moreover, this is a spanning tree because, as we have just seen, the depth-first search visits each vertex in the connected component of s. ■

Proposition 14.13: *Let $\vec{G}$ be a directed graph. Depth-first search on $\vec{G}$ starting at a vertex s visits all the vertices of $\vec{G}$ that are reachable from s. Also, the DFS tree contains directed paths from s to every vertex reachable from s.*

Justification: Let V_s be the subset of vertices of $\vec{G}$ visited by DFS starting at vertex s. We want to show that V_s contains s and every vertex reachable from s belongs to V_s. Suppose now, for the sake of a contradiction, that there is a vertex w reachable from s that is not in V_s. Consider a directed path from s to w, and let (u, v) be the first edge on such a path taking us out of V_s, that is, u is in V_s but v is not in V_s. When DFS reaches u, it explores all the outgoing edges of u, and thus must reach also vertex v via edge (u, v). Hence, v should be in V_s, and we have obtained a contradiction. Therefore, V_s must contain every vertex reachable from s.

We prove the second fact by induction on the steps of the algorithm. We claim that each time a discovery edge (u, v) is identified, there exists a directed path from s to v in the DFS tree. Since u must have previously been discovered, there exists a path from s to u, so by appending the edge (u, v) to that path, we have a directed path from s to v. ■

Note that since back edges always connect a vertex v to a previously visited vertex u, each back edge implies a cycle in G, consisting of the discovery edges from u to v plus the back edge (u, v).

Running Time of Depth-First Search

In terms of its running time, depth-first search is an efficient method for traversing a graph. Note that DFS is called at most once on each vertex (since it gets marked as visited), and therefore every edge is examined at most twice for an undirected graph, once from each of its end vertices, and at most once in a directed graph, from its origin vertex. If we let $n_s \leq n$ be the number of vertices reachable from a vertex s, and $m_s \leq m$ be the number of incident edges to those vertices, a DFS starting at s runs in $O(n_s + m_s)$ time, provided the following conditions are satisfied:

- The graph is represented by a data structure such that creating and iterating through the incident_edges(v) takes $O(\deg(v))$ time, and the e.opposite(v) method takes $O(1)$ time. The adjacency list structure is one such structure, but the adjacency matrix structure is not.
- We have a way to "mark" a vertex or edge as explored, and to test if a vertex or edge has been explored in $O(1)$ time. We discuss ways of implementing DFS to achieve this goal in the next section.

Given the assumptions above, we can solve a number of interesting problems.

Proposition 14.14: *Let G be an undirected graph with n vertices and m edges. A DFS traversal of G can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a path between two given vertices of G, if one exists.*
- *Testing whether G is connected.*
- *Computing a spanning tree of G, if G is connected.*
- *Computing the connected components of G.*
- *Computing a cycle in G, or reporting that G has no cycles.*

Proposition 14.15: *Let $\vec{G}$ be a directed graph with n vertices and m edges. A DFS traversal of $\vec{G}$ can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a directed path between two given vertices of $\vec{G}$, if one exists.*
- *Computing the set of vertices of $\vec{G}$ that are reachable from a given vertex s.*
- *Testing whether $\vec{G}$ is strongly connected.*
- *Computing a directed cycle in $\vec{G}$, or reporting that $\vec{G}$ is acyclic.*
- *Computing the* **transitive closure** *of $\vec{G}$ (see Section 14.4).*

The justification of Propositions 14.14 and 14.15 is based on algorithms that use slightly modified versions of the DFS algorithm as subroutines. We will explore some of those extensions in the remainder of this section.

14.3.2 DFS Implementation and Extensions

We begin by providing a Python implementation of the basic depth-first search algorithm, originally described with pseudo-code in Code Fragment 14.4. Our DFS function is presented in Code Fragment 14.5.

```
1  def DFS(g, u, discovered):
2    """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.
3
4    discovered is a dictionary mapping each vertex to the edge that was used to
5    discover it during the DFS. (u should be "discovered" prior to the call.)
6    Newly discovered vertices will be added to the dictionary as a result.
7    """
8    for e in g.incident_edges(u):          # for every outgoing edge from u
9      v = e.opposite(u)
10     if v not in discovered:              # v is an unvisited vertex
11       discovered[v] = e                  # e is the tree edge that discovered v
12       DFS(g, v, discovered)              # recursively explore from v
```

Code Fragment 14.5: Recursive implementation of depth-first search on a graph, starting at a designated vertex u.

In order to track which vertices have been visited, and to build a representation of the resulting DFS tree, our implementation introduces a third parameter, named discovered. This parameter should be a Python dictionary that maps a vertex of the graph to the tree edge that was used to discover that vertex. As a technicality, we assume that the source vertex u occurs as a key of the dictionary, with None as its value. Thus, a caller might start the traversal as follows:

```
result = {u : None}          # a new dictionary, with u trivially discovered
DFS(g, u, result)
```

The dictionary serves two purposes. Internally, the dictionary provides a mechanism for recognizing visited vertices, as they will appear as keys in the dictionary. Externally, the DFS function augments this dictionary as it proceeds, and thus the values within the dictionary are the DFS tree edges at the conclusion of the process.

Because the dictionary is hash-based, the test, “**if** v **not in** discovered,” and the record-keeping step, “discovered[v] = e,” run in $O(1)$ ***expected*** time, rather than worst-case time. In practice, this is a compromise we are willing to accept, but it does violate the formal analysis of the algorithm, as given on page 643. If we could assume that vertices could be numbered from 0 to $n-1$, then those numbers could be used as indices into an array-based lookup table rather than a hash-based map. Alternatively, we could store each vertex’s discovery status and associated tree edge directly as part of the vertex instance.

Reconstructing a Path from u to v

We can use the basic DFS function as a tool to identify the (directed) path leading from vertex u to v, if v is reachable from u. This path can easily be reconstructed from the information that was recorded in the discovery dictionary during the traversal. Code Fragment 14.6 provides an implementation of a secondary function that produces an ordered list of vertices on the path from u to v.

To reconstruct the path, we begin at the *end* of the path, examining the discovery dictionary to determine what edge was used to reach vertex v, and then what the other endpoint of that edge is. We add that vertex to a list, and then repeat the process to determine what edge was used to discover it. Once we have traced the path all the way back to the starting vertex u, we can reverse the list so that it is properly oriented from u to v, and return it to the caller. This process takes time proportional to the length of the path, and therefore it runs in $O(n)$ time (in addition to the time originally spent calling DFS).

```
1  def construct_path(u, v, discovered):
2    path = [ ]                                  # empty path by default
3    if v in discovered:
4      # we build list from v to u and then reverse it at the end
5      path.append(v)
6      walk = v
7      while walk is not u:
8        e = discovered[walk]                    # find edge leading to walk
9        parent = e.opposite(walk)
10       path.append(parent)
11       walk = parent
12     path.reverse( )                           # reorient path from u to v
13   return path
```

Code Fragment 14.6: Function to reconstruct a directed path from u to v, given the trace of discovery from a DFS started at u. The function returns an ordered list of vertices on the path.

Testing for Connectivity

We can use the basic DFS function to determine whether a graph is connected. In the case of an undirected graph, we simply start a depth-first search at an arbitrary vertex and then test whether len(discovered) equals n at the conclusion. If the graph is connected, then by Proposition 14.12, all vertices will have been discovered; conversely, if the graph is not connected, there must be at least one vertex v that is not reachable from u, and that will not be discovered.

For directed graph, $\vec{G}$, we may wish to test whether it is ***strongly connected***, that is, whether for every pair of vertices u and v, both u reaches v and v reaches u. If we start an independent call to DFS from each vertex, we could determine whether this was the case, but those n calls when combined would run in $O(n(n+m))$. However, we can determine if $\vec{G}$ is strongly connected much faster than this, requiring only two depth-first searches.

We begin by performing a depth-first search of our directed graph $\vec{G}$ starting at an arbitrary vertex s. If there is any vertex of $\vec{G}$ that is not visited by this traversal, and is not reachable from s, then the graph is not strongly connected. If this first depth-first search visits each vertex of $\vec{G}$, we need to then check whether s is reachable from all other vertices. Conceptually, we can accomplish this by making a copy of graph $\vec{G}$, but with the orientation of all edges reversed. A depth-first search starting at s in the reversed graph will reach every vertex that could reach s in the original. In practice, a better approach than making a new graph is to reimplement a version of the DFS method that loops through all ***incoming*** edges to the current vertex, rather than all ***outgoing*** edges. Since this algorithm makes just two DFS traversals of $\vec{G}$, it runs in $O(n+m)$ time.

Computing all Connected Components

When a graph is not connected, the next goal we may have is to identify all of the ***connected components*** of an undirected graph, or the ***strongly connected components*** of a directed graph. We begin by discussing the undirected case.

If an initial call to DFS fails to reach all vertices of a graph, we can restart a new call to DFS at one of those unvisited vertices. An implementation of such a comprehensive DFS_all method is given in Code Fragment 14.7.

```
1  def DFS_complete(g):
2    """Perform DFS for entire graph and return forest as a dictionary.
3
4    Result maps each vertex v to the edge that was used to discover it.
5    (Vertices that are roots of a DFS tree are mapped to None.)
6    """
7    forest = { }
8    for u in g.vertices( ):
9      if u not in forest:
10       forest[u] = None                # u will be the root of a tree
11       DFS(g, u, forest)
12   return forest
```

Code Fragment 14.7: Top-level function that returns a DFS forest for an entire graph.

Although the DFS_complete function makes multiple calls to the original DFS function, the total time spent by a call to DFS_complete is $O(n+m)$. For an undirected graph, recall from our original analysis on page 643 that a single call to DFS starting at vertex s runs in time $O(n_s+m_s)$ where n_s is the number of vertices reachable from s, and m_s is the number of incident edges to those vertices. Because each call to DFS explores a different component, the sum of n_s+m_s terms is $n+m$. The $O(n+m)$ total bound applies to the directed case as well, even though the sets of reachable vertices are not necessarily disjoint. However, because the same discovery dictionary is passed as a parameter to all DFS calls, we know that the DFS subroutine is called once on each vertex, and then each outgoing edge is explored only once during the process.

The DFS_complete function can be used to analyze the connected components of an undirected graph. The discovery dictionary it returns represents a ***DFS forest*** for the entire graph. We say this is a forest rather than a tree, because the graph may not be connected. The number of connected components can be determined by the number of vertices in the discovery dictionary that have None as their discovery edge (those are roots of DFS trees). A minor modification to the core DFS method could be used to tag each vertex with a component number when it is discovered. (See Exercise C-14.44.)

The situation is more complex for finding strongly connected components of a directed graph. There exists an approach for computing those components in $O(n+m)$ time, making use of two separate depth-first search traversals, but the details are beyond the scope of this book.

Detecting Cycles with DFS

For both undirected and directed graphs, a cycle exists if and only if a ***back edge*** exists relative to the DFS traversal of that graph. It is easy to see that if a back edge exists, a cycle exists by taking the back edge from the descendant to its ancestor and then following the tree edges back to the descendant. Conversely, if a cycle exists in the graph, there must be a back edge relative to a DFS (although we do not prove this fact here).

Algorithmically, detecting a back edge in the undirected case is easy, because all edges are either tree edges or back edges. In the case of a directed graph, additional modifications to the core DFS implementation are needed to properly categorize a nontree edge as a back edge. When a directed edge is explored leading to a previously visited vertex, we must recognize whether that vertex is an ancestor of the current vertex. This requires some additional bookkeeping, for example, by tagging vertices upon which a recursive call to DFS is still active. We leave details as an exercise (C-14.43).

The importance of the Floyd-Warshall algorithm is that it is much easier to implement than DFS, and much faster in practice because there are relatively few low-level operations hidden within the asymptotic notation. The algorithm is particularly well suited for the use of an adjacency matrix, as a single bit can be used to designate the reachability modeled as an edge (u,v) in the transitive closure.

However, note that repeated calls to DFS results in better asymptotic performance when the graph is sparse and represented using an adjacency list or adjacency map. In that case, a single DFS runs in $O(n+m)$ time, and so the transitive closure can be computed in $O(n^2+nm)$ time, which is preferable to $O(n^3)$.

Python Implementation

We conclude with a Python implementation of the Floyd-Warshall algorithm, as presented in Code Fragment 14.10. Although the original algorithm is described using a series of directed graphs $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_n$, we create a single copy of the original graph (using the deepcopy method of Python's copy module) and then repeatedly add new edges to the closure as we progress through rounds of the Floyd-Warshall algorithm.

The algorithm requires a canonical numbering of the graph's vertices; therefore, we create a list of the vertices in the closure graph, and subsequently index that list for our order. Within the outermost loop, we must consider all pairs i and j. Finally, we optimize by only iterating through all values of j after we have verified that i has been chosen such that (v_i, v_k) exists in the current version of our closure.

```
1  def floyd_warshall(g):
2    """Return a new graph that is the transitive closure of g."""
3    closure = deepcopy(g)                    # imported from copy module
4    verts = list(closure.vertices())         # make indexable list
5    n = len(verts)
6    for k in range(n):
7      for i in range(n):
8        # verify that edge (i,k) exists in the partial closure
9        if i != k and closure.get_edge(verts[i],verts[k]) is not None:
10         for j in range(n):
11           # verify that edge (k,j) exists in the partial closure
12           if i != j != k and closure.get_edge(verts[k],verts[j]) is not None:
13             # if (i,j) not yet included, add it to the closure
14             if closure.get_edge(verts[i],verts[j]) is None:
15               closure.insert_edge(verts[i],verts[j])
16   return closure
```

Code Fragment 14.10: Python implementation of the Floyd-Warshall algorithm.

```
1  def topological_sort(g):
2    """Return a list of verticies of directed acyclic graph g in topological order.
3
4    If graph g has a cycle, the result will be incomplete.
5    """
6    topo = [ ]                  # a list of vertices placed in topological order
7    ready = [ ]                 # list of vertices that have no remaining constraints
8    incount = { }               # keep track of in-degree for each vertex
9    for u in g.vertices( ):
10     incount[u] = g.degree(u, False)    # parameter requests incoming degree
11     if incount[u] == 0:                # if u has no incoming edges,
12       ready.append(u)                  # it is free of constraints
13   while len(ready) > 0:
14     u = ready.pop( )                   # u is free of constraints
15     topo.append(u)                     # add u to the topological order
16     for e in g.incident_edges(u):      # consider all outgoing neighbors of u
17       v = e.opposite(u)
18       incount[v] -= 1                  # v has one less constraint without u
19       if incount[v] == 0:
20         ready.append(v)
21   return topo
```

Code Fragment 14.11: Python implementation for the topological sorting algorithm. (We show an example execution of this algorithm in Figure 14.13.)

Performance of Topological Sorting

Proposition 14.22: *Let $\vec{G}$ be a directed graph with n vertices and m edges, using an adjacency list representation. The topological sorting algorithm runs in $O(n+m)$ time using $O(n)$ auxiliary space, and either computes a topological ordering of $\vec{G}$ or fails to include some vertices, which indicates that $\vec{G}$ has a directed cycle.*

Justification: The initial recording of the n in-degrees uses $O(n)$ time based on the degree method. Say that a vertex u is ***visited*** by the topological sorting algorithm when u is removed from the ready list. A vertex u can be visited only when incount(u) is 0, which implies that all its predecessors (vertices with outgoing edges into u) were previously visited. As a consequence, any vertex that is on a directed cycle will never be visited, and any other vertex will be visited exactly once. The algorithm traverses all the outgoing edges of each visited vertex once, so its running time is proportional to the number of outgoing edges of the visited vertices. In accordance with Proposition 14.9, the running time is $(n+m)$. Regarding the space usage, observe that containers topo, ready, and incount have at most one entry per vertex, and therefore use $O(n)$ space. ■