14.1.1 The Graph ADT

A graph is a collection of vertices and edges. We model the abstraction as a combination of three data types: Vertex, Edge, and Graph. A Vertex is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code); we assume it supports a method, element( ), to retrieve the stored element. An Edge also stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method. In addition, we assume that an Edge supports the following methods:

endpoints( ): Return a tuple (u,v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.

opposite(v): Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

The primary abstraction for a graph is the Graph ADT. We presume that a graph can be either ***undirected*** or ***directed***, with the designation declared upon construction; recall that a mixed graph can be represented as a directed graph, modeling edge $\{u,v\}$ as a pair of directed edges (u,v) and (v,u). The Graph ADT includes the following methods:

vertex_count( ): Return the number of vertices of the graph.

vertices( ): Return an iteration of all the vertices of the graph.

edge_count( ): Return the number of edges of the graph.

edges( ): Return an iteration of all the edges of the graph.

get_edge(u,v): Return the edge from vertex u to vertex v, if one exists; otherwise return None. For an undirected graph, there is no difference between get_edge(u,v) and get_edge(v,u).

degree(v, out=True): For an undirected graph, return the number of edges incident to vertex v. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex v, as designated by the optional parameter.

incident_edges(v, out=True): Return an iteration of all edges incident to vertex v. In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to False.

insert_vertex(x=None): Create and return a new Vertex storing element x.

insert_edge(u, v, x=None): Create and return a new Edge from vertex u to vertex v, storing element x (None by default).

remove_vertex(v): Remove vertex v and all its incident edges from the graph.

remove_edge(e): Remove edge e from the graph.

A ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex. A ***cycle*** is a path that starts and ends at the same vertex, and that includes at least one edge. We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one. A ***directed path*** is a path such that all edges are directed and are traversed along their direction. A ***directed cycle*** is similarly defined. For example, in Figure 14.2, (BOS, NW 35, JFK, AA 1387, DFW) is a directed simple path, and (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a directed simple cycle. Note that a directed graph may have a cycle consisting of two edges with opposite direction between the same pair of vertices, for example (ORD, UA 877, DFW, DL 335, ORD) in Figure 14.2. A directed graph is ***acyclic*** if it has no directed cycles. For example, if we were to remove the edge UA 877 from the graph in Figure 14.2, the remaining graph is acyclic. If a graph is simple, we may omit the edges when describing path P or cycle C, as these are well defined, in which case P is a list of adjacent vertices and C is a cycle of adjacent vertices.

Example 14.6: *Given a graph G representing a city map (see Example 14.3), we can model a couple driving to dinner at a recommended restaurant as traversing a path though G. If they know the way, and do not accidentally go through the same intersection twice, then they traverse a simple path in G. Likewise, we can model the entire trip the couple takes, from their home to the restaurant and back, as a cycle. If they go home from the restaurant in a completely different way than how they went, not even going through the same intersection twice, then their entire round trip is a simple cycle. Finally, if they travel along one-way streets for their entire trip, we can model their night out as a directed cycle.*

Given vertices u and v of a (directed) graph G, we say that u ***reaches*** v, and that v is ***reachable*** from u, if G has a (directed) path from u to v. In an undirected graph, the notion of ***reachability*** is symmetric, that is to say, u reaches v if an only if v reaches u. However, in a directed graph, it is possible that u reaches v but v does not reach u, because a directed path must be traversed according to the respective directions of the edges. A graph is ***connected*** if, for any two vertices, there is a path between them. A directed graph $\vec{G}$ is ***strongly connected*** if for any two vertices u and v of $\vec{G}$, u reaches v and v reaches u. (See Figure 14.3 for some examples.)

A ***subgraph*** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively. A ***spanning subgraph*** of G is a subgraph of G that contains all the vertices of the graph G. If a graph G is not connected, its maximal connected subgraphs are called the ***connected components*** of G. A ***forest*** is a graph without cycles. A ***tree*** is a connected forest, that is, a connected graph without cycles. A ***spanning tree*** of a graph is a spanning subgraph that is a tree. (Note that this definition of a tree is somewhat different from the one given in Chapter 8, as there is not necessarily a designated root.)

14.4 Transitive Closure

We have seen that graph traversals can be used to answer basic questions of reachability in a directed graph. In particular, if we are interested in knowing whether there is a path from vertex u to vertex v in a graph, we can perform a DFS or BFS traversal starting at u and observe whether v is discovered. If representing a graph with an adjacency list or adjacency map, we can answer the question of reachability for u and v in $O(n+m)$ time (see Propositions 14.15 and 14.17).

In certain applications, we may wish to answer many reachability queries more efficiently, in which case it may be worthwhile to precompute a more convenient representation of a graph. For example, the first step for a service that computes driving directions from an origin to a destination might be to assess whether the destination is reachable. Similarly, in an electricity network, we may wish to be able to quickly determine whether current flows from one particular vertex to another. Motivated by such applications, we introduce the following definition. The ***transitive closure*** of a directed graph $\vec{G}$ is itself a directed graph $\vec{G}^*$ such that the vertices of $\vec{G}^*$ are the same as the vertices of $\vec{G}$, and $\vec{G}^*$ has an edge (u,v), whenever $\vec{G}$ has a directed path from u to v (including the case where (u,v) is an edge of the original $\vec{G}$).

If a graph is represented as an adjacency list or adjacency map, we can compute its transitive closure in $O(n(n+m))$ time by making use of n graph traversals, one from each starting vertex. For example, a DFS starting at vertex u can be used to determine all vertices reachable from u, and thus a collection of edges originating with u in the transitive closure.

In the remainder of this section, we explore an alternative technique for computing the transitive closure of a directed graph that is particularly well suited for when a directed graph is represented by a data structure that supports $O(1)$-time lookup for the get_edge(u,v) method (for example, the adjacency-matrix structure). Let $\vec{G}$ be a directed graph with n vertices and m edges. We compute the transitive closure of $\vec{G}$ in a series of rounds. We initialize $\vec{G}_0 = \vec{G}$. We also arbitrarily number the vertices of $\vec{G}$ as $v_1, v_2, \ldots, v_n$. We then begin the computation of the rounds, beginning with round 1. In a generic round k, we construct directed graph $\vec{G}_k$ starting with $\vec{G}_k = \vec{G}_{k-1}$ and adding to $\vec{G}_k$ the directed edge (v_i, v_j) if directed graph $\vec{G}_{k-1}$ contains both the edges (v_i, v_k) and (v_k, v_j). In this way, we will enforce a simple rule embodied in the proposition that follows.

Proposition 14.18: *For $i = 1, \ldots, n$, directed graph $\vec{G}_k$ has an edge (v_i, v_j) if and only if directed graph $\vec{G}$ has a directed path from v_i to v_j, whose intermediate vertices (if any) are in the set $\{v_1, \ldots, v_k\}$. In particular, $\vec{G}_n$ is equal to $\vec{G}^*$, the transitive closure of $\vec{G}$.*

Example 14.2: *We can associate with an object-oriented program a graph whose vertices represent the classes defined in the program, and whose edges indicate inheritance between classes. There is an edge from a vertex v to a vertex u if the class for v inherits from the class for u. Such edges are directed because the inheritance relation only goes in one direction (that is, it is* **asymmetric***).*

If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***. Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a ***mixed graph***. Note that an undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u,v) by the pair of directed edges (u,v) and (v,u). It is often useful, however, to keep undirected and mixed graphs represented as they are, for such graphs have several applications, as in the following example.

Example 14.3: *A city map can be modeled as a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, in this way, a graph modeling a city map is a mixed graph.*

Example 14.4: *Physical examples of graphs are present in the electrical wiring and plumbing networks of a building. Such networks can be modeled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge. Such graphs are actually components of much larger graphs, namely the local power and water distribution networks. Depending on the specific aspects of these graphs that we are interested in, we may consider their edges as undirected or directed, for, in principle, water can flow in a pipe and current can flow in a wire in either direction.*

The two vertices joined by an edge are called the ***end vertices*** (or ***endpoints***) of the edge. If an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge. Two vertices u and v are said to be ***adjacent*** if there is an edge whose end vertices are u and v. An edge is said to be ***incident*** to a vertex if the vertex is one of the edge's endpoints. The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex. The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex. The ***degree*** of a vertex v, denoted $\deg(v)$, is the number of incident edges of v. The ***in-degree*** and ***out-degree*** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted $\text{indeg}(v)$ and $\text{outdeg}(v)$, respectively.

```
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for undirected
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected.

    Property is based on the original declaration of the graph, not its contents.
    """
    return self._incoming is not self._outgoing # directed if maps are distinct

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)

  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys( )

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count edges
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )           # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values( ):
      result.update(secondary_map.values( ))        # add edges to resulting set
    return result
```

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

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.

```
def DFS_complete(g):
  """Perform DFS for entire graph and return forest as a dictionary.

  Result maps each vertex v to the edge that was used to discover it.
  (Vertices that are roots of a DFS tree are mapped to None.)
  """
  forest = { }
  for u in g.vertices( ):
    if u not in forest:
      forest[u] = None                    # u will be the root of a tree
      DFS(g, u, forest)
  return forest
```

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

R-14.12 Explain why the DFS traversal runs in $O(n^2)$ time on an n-vertex simple graph that is represented with the adjacency matrix structure.

R-14.13 In order to verify that all of its nontree edges are back edges, redraw the graph from Figure 14.8b so that the DFS tree edges are drawn with solid lines and oriented downward, as in a standard portrayal of a tree, and with all nontree edges drawn using dashed lines.

R-14.14 A simple undirected graph is ***complete*** if it contains an edge between every pair of distinct vertices. What does a depth-first search tree of a complete graph look like?

R-14.15 Recalling the definition of a complete graph from Exercise R-14.14, what does a breadth-first search tree of a complete graph look like?

R-14.16 Let G be an undirected graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the table above.

a. Draw G.

b. Give the sequence of vertices of G visited using a DFS traversal starting at vertex 1.

c. Give the sequence of vertices visited using a BFS traversal starting at vertex 1.

R-14.17 Draw the transitive closure of the directed graph shown in Figure 14.2.

R-14.18 If the vertices of the graph from Figure 14.11 are numbered as ($v_1 =$ JFK, $v_2 =$ LAX, $v_3 =$ MIA, $v_4 =$ BOS, $v_5 =$ ORD, $v_6 =$ SFO, $v_7 =$ DFW), in what order would edges be added to the transitive closure during the Floyd-Warshall algorithm?

R-14.19 How many edges are in the transitive closure of a graph that consists of a simple directed path of n vertices?

R-14.20 Given an n-node complete binary tree T, rooted at a given position, consider a directed graph $\vec{G}$ having the nodes of T as its vertices. For each parent-child pair in T, create a directed edge in $\vec{G}$ from the parent to the child. Show that the transitive closure of $\vec{G}$ has $O(n \log n)$ edges.

C-14.41 Our solution to reporting a path from u to v in Code Fragment 14.6 could be made more efficient in practice if the DFS process ended as soon as v is discovered. Describe how to modify our code base to implement this optimization.

C-14.42 Let G be an undirected graph G with n vertices and m edges. Describe an $O(n+m)$-time algorithm for traversing each edge of G exactly once in each direction.

C-14.43 Implement an algorithm that returns a cycle in a directed graph $\vec{G}$, if one exists.

C-14.44 Write a function, components(g), for undirected graph g, that returns a dictionary mapping each vertex to an integer that serves as an identifier for its connected component. That is, two vertices should be mapped to the same identifier if and only if they are in the same connected component.

C-14.45 Say that a maze is ***constructed correctly*** if there is one path from the start to the finish, the entire maze is reachable from the start, and there are no loops around any portions of the maze. Given a maze drawn in an $n \times n$ grid, how can we determine if it is constructed correctly? What is the running time of this algorithm?

C-14.46 Computer networks should avoid single points of failure, that is, network vertices that can disconnect the network if they fail. We say an undirected, connected graph G is ***biconnected*** if it contains no vertex whose removal would divide G into two or more connected components. Give an algorithm for adding at most n edges to a connected graph G, with $n \geq 3$ vertices and $m \geq n-1$ edges, to guarantee that G is biconnected. Your algorithm should run in $O(n+m)$ time.

C-14.47 Explain why all nontree edges are cross edges, with respect to a BFS tree constructed for an undirected graph.

C-14.48 Explain why there are no forward nontree edges with respect to a BFS tree constructed for a directed graph.

C-14.49 Show that if T is a BFS tree produced for a connected graph G, then, for each vertex v at level i, the path of T between s and v has i edges, and any other path of G between s and v has at least i edges.

C-14.50 Justify Proposition 14.16.

C-14.51 Provide an implementation of the BFS algorithm that uses a FIFO queue, rather than a level-by-level formulation, to manage vertices that have been discovered until the time when their neighbors are considered.

C-14.52 A graph G is ***bipartite*** if its vertices can be partitioned into two sets X and Y such that every edge in G has one end vertex in X and the other in Y. Design and analyze an efficient algorithm for determining if an undirected graph G is bipartite (without knowing the sets X and Y in advance).

C-14.58 Let $\vec{G}$ be a weighted directed graph with n vertices. Design a variation of Floyd-Warshall's algorithm for computing the lengths of the shortest paths from each vertex to every other vertex in $O(n^3)$ time.

C-14.59 Design an efficient algorithm for finding a ***longest*** directed path from a vertex s to a vertex t of an acyclic weighted directed graph $\vec{G}$. Specify the graph representation used and any auxiliary data structures used. Also, analyze the time complexity of your algorithm.

C-14.60 An independent set of an undirected graph $G = (V, E)$ is a subset I of V such that no two vertices in I are adjacent. That is, if u and v are in I, then (u, v) is not in E. A ***maximal independent set*** M is an independent set such that, if we were to add any additional vertex to M, then it would not be independent any more. Every graph has a maximal independent set. (Can you see this? This question is not part of the exercise, but it is worth thinking about.) Give an efficient algorithm that computes a maximal independent set for a graph G. What is this method's running time?

C-14.61 Give an example of an n-vertex simple graph G that causes Dijkstra's algorithm to run in $\Omega(n^2 \log n)$ time when its implemented with a heap.

C-14.62 Give an example of a weighted directed graph $\vec{G}$ with negative-weight edges, but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex s.

C-14.63 Consider the following greedy strategy for finding a shortest path from vertex *start* to vertex *goal* in a given connected graph.

```
1: Initialize path to start.
2: Initialize set visited to {start}.
3: If start=goal, return path and exit. Otherwise, continue.
4: Find the edge (start,v) of minimum weight such that v is adjacent to
   start and v is not in visited.
5: Add v to path.
6: Add v to visited.
7: Set start equal to v and go to step 3.
```

Does this greedy strategy always find a shortest path from *start* to *goal*? Either explain intuitively why it works, or give a counterexample.

C-14.64 Our implementation of shortest_path_lengths in Code Fragment 14.13 relies on use of "infinity" as a numeric value, to represent the distance bound for vertices that are not (yet) known to be reachable from the source. Reimplement that function without such a sentinel, so that vertices, other than the source, are not added to the priority queue until it is evident that they are reachable.

C-14.65 Show that if all the weights in a connected weighted graph G are distinct, then there is exactly one minimum spanning tree for G.