3

It's A Small World After All

Modeling networks with graphs

In the previous chapter, we learn how a tree allows us to represent hierarchical relationships. However, a tree cannot represent all relationships, because many relationships are simply connecting pairs of entities together (without a hierarchical sense). For example, while parent-child relationships are hierarchical, friendships often are just about which two people are related to one another.

In this chapter, we will learn about a new type of non-linear data structure called *graph*, which allows us to represent many different types of relationships.

3.1 Real-world Networks

Many things that we experience in the real world are naturally inter-dependent and inter-connected. Here are just some examples of such connected structures that we usually call "networks":

Transportation. Networks are commonly found in many transportation infrastructures. Airlines fly from one city to the next. The connectivity of departure and arrival cities can be represented as a network. When you take a train, it goes from one stop to the next stop, and we can link together each pair of previous and next stops, resulting in a train network.

Communication. Similarly, in our telecommunications infrastructure, telcos around the world maintains servers that are connected to one another to support our communication needs, be it phone calls, television, or Internet services.

Web. When we view a web page on our browser, the page has hyperlinks that link to other pages on the Web. When you click on a hyperlink, you go on to the next page, which again has a number of other hyperlinks connecting to yet other pages. The collective

pages on the Web, and how they link to one another, represent one of the largest networks that exist today.

Dependency. In a university degree program, to take a particular course, there may be one or more pre-requisite courses. This precedence or dependency relationship can be represented as a network, where we create a link from a course A to a course B, if A is a prerequisite for B. We will revisit this structure later in Section 3.4.

Social The type of network that most of us are probably familiar with is social network. When we visit Facebook, we go there to see the postings of our friends. When we visit Twitter, we see the new tweets that are posted by people that we follow. A social network connects each person to other people whom we have indicated as friends (Facebook), or people whom we follow (Twitter).

These are just but a subset of network structures that we encounter in real life. There are many others. Can you think of a few more?

Six Degrees of Separation

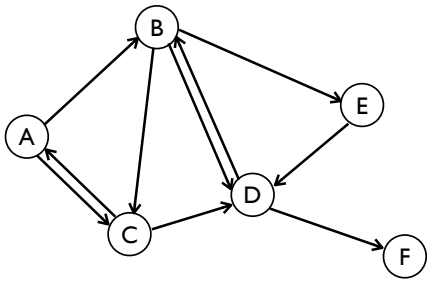
An interesting finding by social network scientists is the "small world" property of many real world networks. What it means is that on average, two entities in a network are connected to each other by a surprisingly small number of sequential edges.

In particular, the term "six degrees of separation" was popularized by an experiment[1] conducted in the 1960's by a psychologist by the name of Stanley Milgram. In this experiment, Milgram sent a number of packets to people in two US cities (Omaha, Nebraska, and Wichita, Kansas). Inside the packet was the name of a target person in Boston, Massachusetts. Each recipient was instructed to forward the packet to the target person if he or she knows the target person. Otherwise, the recipient was to forward it to an acquaintance who were more likely to know the target person. Out of 296 packets sent, eventually 64 reached the target person. The average number of hops from a recipient to the target person was found to be around six, thus suggesting that people in the US were separated by merely six degrees[2].

This finding was popularized further by a game, called "Six Degrees of Kevin Bacon"[3]. Kevin Bacon, being one of the most prolific actors in Hollywood, had collaborated with many other actors. Two actors are directly related if they appear in the same movie together. The game revolves around who can name how an actor (e.g., Johnny Depp) is related to Kevin Bacon, through a series of actor relationships. For example, Johnny Depp and Helena Bonham Carter appeared in Dark Shadows. Helena Bonham Carter and Kevin Bacon appeared in Novocaine. In this case, we say that Bacon number of Johnny Depp is 2, because he is connected to Kevin Bacon in two hops (through Helena Bonham Carter).

In this chapter, we will explore how networks can be modeled by graph data structures, as well as how some operations on the graph allow us to explore the connectivity of different entities.

[1] S. Milgram, The small world problem, Psychology Today, 1967.
[2] `http://en.wikipedia.org/wiki/Small-world_experiment`
[3] `http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon`



(a) Graph

Adjacency List

A	B	C	
B	C	D	E
C	D	A	
D	B	F	
E	D		
F			

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	1	1	0
C	1	0	0	1	0	0
D	0	1	0	0	0	1
E	0	0	0	1	0	0
F	0	0	0	0	0	0

(b) Representation

Figure 3.1: *Example of a directed graph*

3.2 Graph

We will first explore the terminologies associated with graph data structures, before we look at how to represent graphs, as well as identify some variants of graph.

Terminologies

A graph G contains a set of **vertices** V and a set of **edges** E. Figure 3.1(a) shows an example of a graph. This graph contains seven vertices. We refer to each vertex by its label. In this case, V is the set of vertices $\{A, B, C, D, E, F\}$. The set of edges E encode the connectivity structure between pairs of vertices. In this graph, we are modeling **directed edges**, i.e., there is a **source vertex** and a **target vertex** for each edge. When an edge exists from a vertex to another vertex, we draw a line from the source vertex to the target vertex. For example, A (source) has two outgoing edges, to B and to C (targets) respectively.

Adjacency List vs. Adjacency Matrix Representation

Figure 3.1(a) shows a visual representation of a graph with nodes as vertices and lines as edges. To represent graphs in computations, we frequently use one of the following representations.

Adjacency List. The first representation of a graph is adjacency list. In this representation, each vertex (source) has a list of other vertices that it connects to (a list of targets). The left half of Figure 3.1(b) shows this representation for the graph in Figure 3.1(a). For instance, A's list consists of B and C; B's list has C, D, and E; and so on.

The ordering of vertices in this adjacency list does not affect the connectivity structure in terms of which vertex is connected to which other vertex. However, as we will see later, the ordering in the list may affect the running of some graph algorithms.

Adjacency Matrix. The second common representation of a graph is adjacency matrix. For a graph with n vertices, we use a square matrix of dimension $n \times n$. Every row corresponds to each source vertex. Every column corresponds to each target vertex. Each element has a value of 1 if there exists an edge between the source and the target vertices. Otherwise, the element has a value of 0. For example, the first row corresponds to A. Since A is connected to B and C, we see 1's in B and C's column of this row, and 0's in other columns.

Operations

A graph consists of a set of vertices and edges, and therefore it supports a number of operations for addition and removal of vertices and edges.

The `Graph` object has the following operations:

- `new` creates an empty graph
- `vertices` returns the list of vertices in this graph
- `addVertex(vertex)` inserts a new vertex into the graph's list of vertices
- `deleteVertex(vertex)` removes the specified vertex and its edges from the graph
- `addEdge(vertex1, vertex2)` inserts a new edge from `vertex1` to `vertex2`
- `deleteEdge(vertex1, vertex2)` removes the edge from `vertex1` to `vertex2`

The `Vertex` object has the following operations:

- `new(label)` creates a new vertex with the given label
- `adjList` returns the list of target vertices that this vertex has edges to

Connectivity and Paths

The **indegree** of a vertex is the number of incoming edges that the vertex has. Its **outdegree** is the number of outgoing edges it has. For example, in Figure 3.1(a), B has an indegree of 2 (corresponding to edges from A and D), and outdegree of 3 (corresponding to edges to C, D, E). We refer to the set of target vertices in a source vertex's adjacency list as its neighbors.

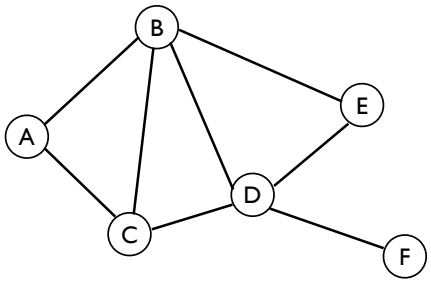
A **path** is a sequence of vertices such that for every pair consecutive vertices in the sequence, there exists an edge from the first vertex to the second vertex in the pair. In Figure 3.1(a), (A, B, E) is a path of length 2 (because it involves 2 edges). (A, C, D, B, E) is another path, with length 4. There could be multiple paths from a vertex to another vertex. For instance, both of these example paths begin at A and end at E.

A path that begins and ends with the same vertex is a **cycle**. Figure 3.1(a) contains cycles, such as (A, C, A), (A, B, C, A), (B, E, D, B), and several others.

Undirected vs. Directed

The discussion up to now has assumed that the graph models directed edges. An alternative graph model may have edges that are undirected. Figure 3.2(a) shows an example of such a graph. Notably, there is no "arrow" indicating direction. Such graphs are commonly used to represent relationships that are symmetrical in nature.

To illustrate, a social network graph may use directed edges if a vertex v_1 may indicate v_2 as a friend, but v_2 may not indicate v_1 as a friend. An example of this would be the Twitter follow network. Another social network graph may use undirected edges if both vertices have to agree before a friendship relationship is created. An example of this would be the Facebook friendship network.



(a) Graph

Adjacency List

A	B	C		
B	A	C	D	E
C	D	A	B	
D	B	E	F	C
E	D	B		
F	D			

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	1	1	1	0
C	1	1	0	1	0	0
D	0	1	1	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

(b) Representation

Figure 3.2: *Example of an undirected graph*

To represent undirected graphs, we can similarly use either an adjacency list or an adjacency matrix. The main difference is that every edge is represented twice, once in the first vertex's adjacency list or row in adjacency matrix, and another time in the second vertex's adjacency list or row in adjacency matrix. An alternative way of viewing an undirected graph is to view it as a directed graph, whereby every undirected edge corresponds to two directed edges in opposite directions. Figure 3.2(b) shows the representations for the undirected graph in Figure 3.2(a).

In subsequent discussions, unless otherwise specified, we will use directed graph by default. However, most of the concepts will also be transferable to undirected graphs.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include GraphLab
=>  Object
```

T1. Create a new graph `g`:

```
>>  g = Graph.new
=>  []
```

Currently, the graph is empty. It does not have any vertex yet.

T2. Make a new vertex with label 1 and insert it into the graph:

```
>>  v1 = Vertex.new(1)
=>  1

>>  g.addVertex(v1)
=>  [1 ]
```

Upon adding a new vertex, the new vertex will appear on the visualization canvas. The placement of the vertex on the canvas would be random, unless we specify the x and y coordinates of the new vertex, e.g., `Vertex.new(label, x-coord, y-coord)`. The canvas has a size of approximately 600 x 600, so you may specify any coordinates from 0 to 600.

T3. Let's create another vertex, and place it on the canvas at the coordinates (100, 500).

```
>>  v2 = Vertex.new(2, 100, 500)

>>  g.addVertex(v2)
```

T4. Create another six vertices and insert them into the graph. You may optionally specify the coordinates of each vertex to have a better visualization layout.

```
>>  v3 = Vertex.new(3)

>>  g.addVertex(v3)

>>  v4 = Vertex.new(4)

>>  g.addVertex(v4)

>>  v5 = Vertex.new(5)

>>  g.addVertex(v5)

>>  v6 = Vertex.new(6)

>>  g.addVertex(v6)

>>  v7 = Vertex.new(7)

>>  g.addVertex(v7)

>>  v8 = Vertex.new(8)

>>  g.addVertex(v8)
```

The graph now has eight vertices that are not connected to one another.

T5. We can remove an existing vertex using `deleteVertex` operation:

```
>>  g.deleteVertex(v8)
=>  [1 , 2 , 3 , 4 , 5 , 6 , 7 ]
```

T6. At any time, we can check the list of vertices of this graph:

```
>>  g.vertices
=>  [1 , 2 , 3 , 4 , 5 , 6 , 7 ]
```

T7. Let us now add an edge to this graph:

```
>>  g.addEdge(v1, v4)
=>  true
```

You can now see in the visualization window that there is a directed edge between the vertex with label 1 to the vertex with label 4.

T8. Let us now practise adding more edges to this graph:

```
>>  g.addEdge(v1, v7)
=>  true

>>  g.addEdge(v7, v4)
=>  true

>>  g.addEdge(v4, v5)
=>  true

>>  g.addEdge(v4, v2)
=>  true

>>  g.addEdge(v5, v2)
=>  true

>>  g.addEdge(v5, v6)
=>  true

>>  g.addEdge(v2, v6)
=>  true

>>  g.addEdge(v6, v3)
=>  true

>>  g.addEdge(v6, v7)
=>  true

>>  g.addEdge(v3, v7)
=>  true

>>  g.addEdge(v3, v1)
=>  true
```

T9. Sometimes we need to delete an edge, which can be done using `deleteEdge` operation.

```
>>  g.deleteEdge(v3, v1)
=>  true
```

Figure 3.3 shows how the graph we created above may look like. You may not get exactly the same layout, because the positioning of the vertices are random. However, you should still see the same number of vertices, with the same labels and the same connectivity.

T10. In this implementation, we use an adjacency list representation of the graph. We can check the list of other vertices that a vertex is connected to by calling the `adjList` operation.

```
>>  v1.adjList
=>  [4, 7]
```

This shows that from vertex 1, there are two directed edges, to vertex 4 and vertex 7 respectively.

T11. Check the adjacency lists of other vertices as well.

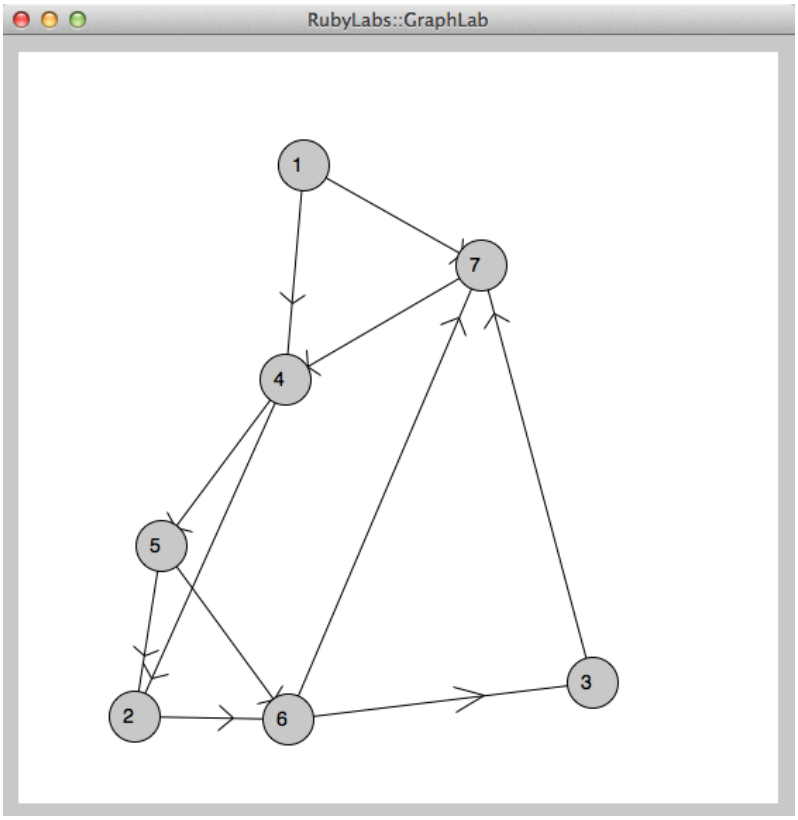


Figure 3.3: *Visualization of graph after the first tutorial project*

```
>>  v2.adjList
=>  [6]

>>  v3.adjList
=>  [7]

>>  v4.adjList
=>  [5, 2]

>>  v5.adjList
=>  [2, 6]

>>  v6.adjList
=>  [3, 7]

>>  v7.adjList
=>  [4]
```

T12. What would be the adjacency matrix representation of this graph?

```
def traversal(graph, vertex, option)
  graph.clearAll
  case option
  when :dfs
    dfs_traversal(vertex)
  when :bfs
    bfs_traversal(vertex)
  else
    return "Invalid option."
  end
  return nil
end
```

(a) `traversal`

```
def visit(vertex)
  vertex.visited = true
  p vertex
end
```

(b) `visit`

Figure 3.4: *Shared methods for traversing a graph*

3.3 Traversals of a Graph

Like a tree, a graph is a non-linear structure. What it means is that there could be multiple ways to "iterate" or to traverse a non-linear structure. Graph traversal is even more complex than tree. In a tree, every traversal begins with the root node, and every non-root node is "reachable" from the root node. In a graph, there is no single "root node". Instead, traversal may begin from any vertex in the graph.

There are two main ways to traverse a graph from any particular vertex: *depth-first search* and *breadth-first search*. In the following, we will explore how to write both traversal algorithms. First of all, in Figure 3.4(a), we show a common `traversal` method that takes in a `graph`, a `vertex` where we will begin the traversal from, as well as an `option`. At the beginning of each traversal, the line `graph.clearAll` clears or resets the list of vertices that we have visited. If the specified `option` is `:dfs`, we will call the depth-first search traversal method `dfs_traversal`. Alternatively, if the option is `:bfs`, we will call the breadth-first search traversal method `bfs_traversal`.

Depth-First Search (DFS)

In depth-first search, we prioritize going deep into the graph. Each time we encounter several choices of vertices to explore (e.g., v_1 and v_2), we will pick one vertex (e.g., v_1) and

```
def dfs_traversal(vertex)
  visit(vertex)
  for i in 0..(vertex.adjList.length-1)
    neighbor = vertex.adjList[i]
    if !neighbor.isVisited?
      dfs_traversal(neighbor)
    end
  end
end
```

Figure 3.5: *Recursive implementation of depth first search*

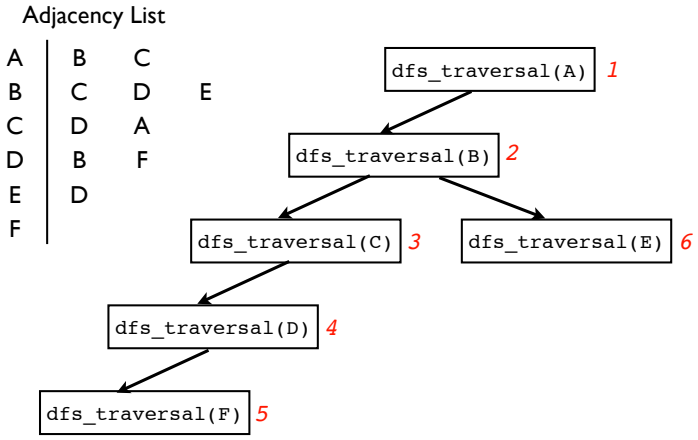


Figure 3.6: *Trace of depth first search for graph in Figure 3.1(a)*

keep exploring all other connected vertices, before moving on to the second vertex v_2, and so on.

Figure 3.5 illustrates a recursive method for depth-first search. Every time we encounter a vertex, we will visit this vertex. We will then iterate through each neighbor (according to the ordering in the vertex's adjacency list), and recursively traverses this neighbor (before moving on to the next neighbor).

The definition of "visit" here typically depends on each application scenario. It may involve a search, a comparison, or an operation with the visited vertex. In our context here, the method `visit` is illustrated in Figure 3.4(b), where we simply indicate that this vertex has been visited, and print out its label.

To illustrate the working of `dfs_traversal`, we will trace the execution of this recursive algorithm with the graph in Figure 3.1(a), beginning with the vertex A. This trace of recursive calls is shown in Figure 3.6. `dfs_traversal(A)` results in first visiting A (1st). It then leads to recursive calls to A's neighbor according to the ordering in A's adjacency list. We visit A's first neighbor B (2nd). This recursively goes on to B's neighbor C (3rd), and C's neighbor D (4th). We then visit D's neighbor F (5th), since D's other neighbor B has been visited before. By this time, D's neighbors are all visited. Same with C's neighbors.

```
def bfs_traversal(vertex)
  q = LinearDSLab::Queue.new
  visit(vertex)
  q.enqueue(vertex)
  while(!q.isEmpty?)
    v = q.dequeue
    for i in 0..(v.adjList.length-1)
      neighbor = v.adjList[i]
      if !neighbor.isVisited?
        visit(neighbor)
        q.enqueue(neighbor)
      end
    end
  end
end
```

Figure 3.7: *Queue-based implementation of breadth first search*

Adjacency List

A	B	C		
B	C	D	E	
C	D	A		
D	B	F		
E	D			
F				

step	dequeued	visited/enqueued	queue
0		A	[A]
1	A	B, C	[B, C]
2	B	D, E	[C, D, E]
3	C		[D, E]
4	D	F	[E, F]
5	E		[F]
6	F		[ ]

Figure 3.8: *Trace of breadth first search for graph in Figure 3.1(a)*

We then visit the remaining unvisited neighbor of B, which is E (6th). Finally, the recursion ends, resulting in order of traversal A, B, C, D, F, E.

Breadth-First Search (BFS)

Unlike depth-first search where we prioritize going deep into the graph, in breadth-first search we prioritize visiting vertices closer to the starting vertex. For instance, suppose we traverse Figure 3.1(a), starting from A. After visiting A, we will visit all vertices reachable from A through paths of length 1 (direct neighbors), followed by vertices reachable by paths of length 2 (neighbors of neighbors), and so on.

Figure 3.7 shows an implementation using a queue. In each loop of the iteration, we dequeue a vertex, and visit all its neighbors, and then enqueue them so that we can determine their neighbors to be visited later. Hence, the queue maintains the order in which vertices have been visited. Every dequeue returns the earliest vertex, whose neighbors are yet to be visited.

The trace of execution for Figure 3.1(a) starting from A is shown in Figure 3.8. Initially, the queue contains only the starting vertex A. In the first iteration, we dequeue A, and visit as well as enqueue vertices directly reachable from A, which are B and C. Next, we dequeue B and visit/enqueue its unvisited neighbors D and E. When dequeueing C, we realize that C's neighbors were already visited or are already in the queue. By now, we have visited all vertices reachable through path length of 2 from A. The next vertex visited is the only one with path length 3, which is F. The traversal order is A, B, C, D, E, F.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include GraphLab
=>  Object
```

T13. We will begin by traversing the graph constructed in the previous tutorial project (Figure 3.3) with depth first search, starting from vertex 1.

```
>>  g.dfs(v1)
 1
 4
 5
 2
 6
 3
 7
=>  nil
```

Watch the animation. Is it as you expected?

We begin from vertex 1 (1st), followed by visiting its first neighbor 4 (2nd). Next is 4's first neighbor 5 (3rd), which in turn visits its first neighbor 2 (4th). From 2, we go to 6 (5th), then 3 (6th), and finally 7 (7th). By now, all the vertices are already visited.

T14. Try depth-first search again but with other starting vertices. First, work out by hand what you expect to be the sequence. Then, run it on `irb` to verify your answer.

```
>>  g.dfs(v2)

>>  g.dfs(v3)

>>  g.dfs(v4)

>>  g.dfs(v5)

>>  g.dfs(v6)

>>  g.dfs(v7)
```

What do you observe? Does every depth first search always visit all the vertices?

T15. We now traverse the graph in Figure 3.3 using breadth first search, starting from vertex 1.

```
>>  g.bfs(v1)
 1
 4
 7
 5
 2
 6
 3
=>  nil
```

Note how the sequence is different from the previous depth first search traversal.

We first visit 1 (1st), and enqueue 1's neighbors 4 and 7. We dequeue and visit 4 (2nd), followed by enqueuing 4's neighbors 5, 2. Next, we visit 7 (3rd), but do not put anything in the queue since its neighbor 4 is already visited. We then visit 5 (4th), which enqueues its remaining neighbor 6. Afterwards, visit 2 (5th), but no new insertion to the queue, because 6 is already in the queue. After visiting 6 (6th), we enqueue 3. Finally, we visit 3 (7th).

T16. Try breadth-first search again but with other starting vertices. First, work out by hand what you expect to be the sequence. Then, run it on `irb` to verify your answer.

```
>>  g.bfs(v2)

>>  g.bfs(v3)

>>  g.bfs(v4)

>>  g.bfs(v5)

>>  g.bfs(v6)

>>  g.bfs(7)
```

What do you observe? Does every breadth first search always visit all the vertices?

T17. To better understand the algorithms for depth first search and breadth first search traversal, code your own Ruby methods. Put together the algorithms in Figure 3.4(a) to Figure 3.7 into a text file, and name the file "traversal.rb".

```
>>  load "traversal.rb"
=>  true
```

The traversal method takes in the graph, the starting vertex for traversal, as well as an option, which could be `:dfs` for depth first search, and `:bfs` for breadth first search.

```
>>  traversal(g, v1, :dfs)
 1
 4
 5
 2
 6
 3
 7
=>  nil
```

```
>>  traversal(g, v1, :bfs)
 1
 4
 7
 5
 2
 6
 3
=>  nil
```

Do they give you the same sequence of visits as the animation you have seen previously? Try the traversal starting from different vertices. Add and remove vertices, and see if the traversals are as expected.

3.4 Topological Sorting

Having looked at the basics of graph, and different methods of traversal, we are now ready to look at one of the graph-based applications.

Directed Acyclic Graph (DAG)

Earlier, we have defined a cycle as a path that begins and ends with the same vertex. If a directed graph has no cycle, we call it a *directed acyclic graph* or DAG. Figure 3.9(a) illustrates an example of a DAG, with adjacency list and matrix representations shown in Figure 3.9(b).

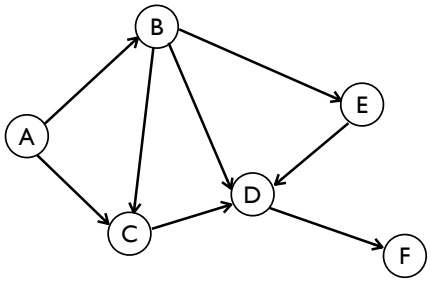
Such a graph is commonly used to represent dependency or precedence relationships. This is the case when to complete a task, we need to first complete another task. We can represent this with a DAG, with tasks as vertices, and edges connecting the preceding task to the following task. For example, if vertices are university courses, a DAG may have edges leading to a course from its prerequisites. Suppose that such a graph were to contain a cycle, e.g., to take a course c_1 you would first need to take c_2, but to take c_2 you would first need to take c_1, then it would not be possible to complete the courses in the cycle while still respecting the precedence relationship.

Topological Ordering

Topological ordering in a DAG is an ordering of *all* vertices in the DAG, $(v_1, v_2, \ldots, v_n)$, such that for any edge involving two vertices, from v_i to v_j, we have $i < j$. In other words, all the edges in this ordering "point forward".

In any DAG, there is at least one topological ordering. For the DAG in Figure 3.9(a), (A, B, E, C, D, F) is one topological ordering. For any edge in the graph, the source vertex appears earlier than the target vertex in the topological ordering. However, a DAG could also have more than one topological ordering. (A, B, C, E, D, F) is the second topological ordering for the DAG in Figure 3.9(a).

To come first in a topological ordering, a vertex cannot have any incoming edge in the graph. In any DAG, there is always *at least* one such vertex.



(a) Graph

Adjacency List

A	B	C		
B	C	D	E	
C	D			
D	F			
E	D			
F				

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	1	1	0
C	0	0	0	1	0	0
D	0	0	0	0	0	1
E	0	0	0	1	0	0
F	0	0	0	0	0	0

(b) Representation

Figure 3.9: *Example of a directed acyclic graph*

```
def topsort(graph)
  graph.clearAll
  s = Stack.new
  for i in 0..(graph.vertices.length-1)
    vi = graph.vertices[i]
    if !vi.isVisited?
      topsort_dfs(vi, s)
    end
  end
  return s
end
```

(a) `topsort`

```
def topsort_dfs(vertex, stack)
  visit(vertex)
  for i in 0..(vertex.adjList.length-1)
    neighbor = vertex.adjList[i]
    if !neighbor.isVisited?
      topsort_dfs(neighbor, stack)
    end
  end
  stack.push(vertex)
end
```

(b) `topsort_dfs`

Figure 3.10: *Topological Sort*

Topological Sorting

We will now explore an algorithm to find one of the topological orderings in a DAG. We begin with the observation that for any path that exists in the graph, the vertices that are part of that path must appear in the same sequential ordering as the path. However, there may be some other interleaving vertices as well.

The traversal method that explores a graph by following the sequential edges in a path is depth-first search. Note that when we call the recursive `dfs_traversal(v)` on a vertex v, we do not return from this call until we have recursively visited all the other vertices that v has a path to. This suggests that by the time we return from v's DFS traversal, we would already know which other vertices should appear *after* v in a topological ordering.

Therefore, one way to discover a topological ordering is to ensure that vertices whose recursive DFS terminates earlier will appear later in the topological ordering. The right data structure for this purpose is a stack, by pushing each vertex into a stack as we complete the recursive DFS on that vertex.

In Figure 3.10(a), we present an algorithm `topsort`, which takes as input a `graph` (which has to be a directed acyclic graph), and produces as output a stack. This stack contains the vertices of the graph. If we pop the vertices from the stack one by one, they will appear according to a topological ordering. This algorithm first resets the visiting record of the graph (`graph.clearAll`). It then creates a new stack to contain the topological ordering. It then iterates over each vertex in the graph, and runs a special type of DFS `topsort_dfs` (Figure 3.10(b)) on each vertex. This DFS traversal is identical to the previously defined recursive algorithm for depth-first search, except that it pushes a vertex into the stack when its recursive call completes.

Note that this algorithm will return only one of the valid topological orderings. For the same graph, depending on the ordering of vertices in each adjacency list, the sequence of DFS may vary slightly, potentially resulting in a different topological ordering. For example, for *B*'s adjacency list in Figure 3.9(b), `topsort` will return the topological ordering (*A*, *B*, *E*, *C*, *D*, *F*). However, if *B*'s adjacency list were to consist of *E*, *C*, and then *D*, the output topological ordering would be (*A*, *B*, *C*, *E*, *D*, *F*).

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include GraphLab
=>  Object
```

T18. In this tutorial, we will explore topological sort for the DAG in Figure 3.9(a). To do this, we first create the vertices A to F.

```
>> va = Vertex.new("A")
>> vb = Vertex.new("B")
>> vc = Vertex.new("C")
>> vd = Vertex.new("D")
>> ve = Vertex.new("E")
>> vf = Vertex.new("F")
```

T19. Create a new graph, and insert the vertices into this graph.

```
>> g2 = Graph.new
>> g2.addVertex(va)
>> g2.addVertex(vb)
>> g2.addVertex(vc)
>> g2.addVertex(vd)
>> g2.addVertex(ve)
>> g2.addVertex(vf)
```

T20. Let us now reproduce the connectivity in Figure 3.9(a).

```
>> g2.addEdge(va, vb)
>> g2.addEdge(va, vc)
>> g2.addEdge(vb, vc)
>> g2.addEdge(vb, vd)
>> g2.addEdge(vb, ve)
>> g2.addEdge(vc, vd)
>> g2.addEdge(vd, vf)
>> g2.addEdge(ve, vd)
```

Compare the visualization that you get now to Figure 3.9(a). The layout may be affected by the random placement of the vertices in the visualization window. However, the connectivity should still be the same as Figure 3.9(a).

T21. Let us now run the topological sort algorithm on this graph.

```
>>  topsort(g2)
 A
 B
 C
 D
 F
 E
=>  [A , B , E , C , D , F ]
```

After calling `topsort`, the method prints out the order of nodes visited at each line. In this case, the sequence of visits is A, B, C, D, F, E.

However, the final line prints out the stack that contains the topological sequence [A , B , E , C , D , F], which is not identical to the order of visits.

Is this a correct topological sequence?

T22. Let us now add a new node that has an outgoing edge to vertex A, and check the topological ordering.

```
>>  vh = Vertex.new("H")
=>  H

>>  g2.addVertex(vh)
=>  [A , B , C , D , E , F , H ]
```

Note that H comes last in the list of vertices of `g2`, because H is added last.

We now add an edge from H to A.

```
>>  g2.addEdge(vh, va)
=>  true
```

T23. Run the topological sort algorithm on this new graph.

```
>>  topsort(g2)
 A
 B
 C
 D
 F
 E
 H
=>  [H, A , B , E , C , D , F ]
```

Based on the sequence of visits, vertex H is visited last. This is because the first vertex in graph `g2` is A. The algorithm in Figure 3.10(a) runs depth first search from A, and visites the rest of the vertex, except for H, which is visited in the next depth first search.

However, the topological sequence [H, A , B , E , C , D , F] is correct because H is before A.

T24. Let us now add a new node that is expected to be last in topological sort. Such a vertex would have an incoming edge from F.

```
>>  vj = Vertex.new("J")
=>  J

>>  g2.addVertex(vj)
=>  [A , B , C , D , E , F , H , J ]

>>  g2.addEdge(vf, vj)
=>  true
```

T25. Run the topological sort algorithm on this new graph.

```
>>  topsort(g2)
 A
 B
 C
 D
 F
 J
 E
 H
=>  [H, A , B , E , C , D , F , J ]
```

Note that in the topological sequence, J comes after F, which is as expected.

T26. Add a few more vertices and edges on your own, and explore the topological sort algorithm.

3.5 Shortest Path

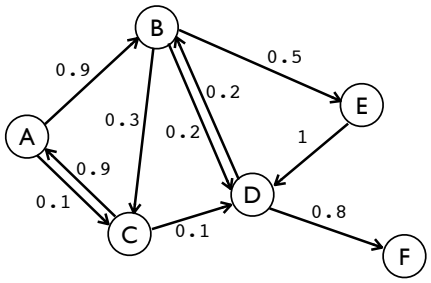
Previously, we have been looking at graphs where edges either exist or do not exist. Therefore, a graph's representation as an adjacency list or an adjacency matrix simply needs to reflect the existence of edges, where usually a binary value of 0 or 1 suffices. In addition, between two paths connecting the same source vertex and target vertex, we can only speak about the varying lengths of such paths (how many intermediate vertices exist in a path).

Weighted Graph

In some applications, not all edges have equal importance. For example, when we use a graph to represent an inter-city road network, where vertices are cities, and edges are roads connecting two cities, we may want to associate each edge with some value that reflects either the distance travelled, the time taken, or the cost of travel along the corresponding road. In this respect, the previous binary representation is no longer sufficient.

We now define a weighted graph as a graph where each edge is associated with a weight. As indicated in the previous example, this weight may reflect various kinds of values (e.g., distance, time, cost), but we assume the weight has a consistent meaning for all edges. Figure 3.11(a) illustrates an example of a weighted graph. The numbers appearing alongside edges are their respective weights. For example, the edge from A to C has a weight of 0.1, but the edge from C to A has a higher weight of 0.9.

As shown in Figure 3.11(b), a weighted graph could also be represented as either an adjacency list or an adjacency matrix, with some minor variation. In an adjacency list, for each source vertex, its list of target vertices now also indicate the weight of each edge to a target vertex. In an adjacency matrix, we replace the previously binary values with real values reflecting the weights. However, we no longer simply use 0 to reflect an edge that does not exist, because 0 itself is a possible valid weight. In Figure 3.11(b), we set the weight of non-existent edge as infinity, because in this specific example, we model a graph where lower weight indicates lower cost (e.g., to go through a non-existent road has infinity cost).



(a) Graph

Adjacency List

A	(B, 0.9) (C, 0.1)
B	(C, 0.3) (D, 0.2) (E, 0.5)
C	(D, 0.1) (A, 0.9)
D	(B, 0.2) (F, 0.8)
E	(D, 1)
F	

Adjacency Matrix

	A	B	C	D	E	F
A	∞	0.9	0.1	∞	∞	∞
B	∞	∞	0.3	0.2	0.5	∞
C	0.9	∞	∞	0.1	∞	∞
D	∞	0.2	∞	∞	∞	0.8
E	∞	∞	∞	1	∞	∞
F	∞	∞	∞	∞	∞	∞

(b) Representation

Figure 3.11: *Example of a directed, weighted graph*

Shortest Path

One of the advantages of modeling weight in a graph is the ability to associate different weights with paths in the graph. One way to compute the weight of a path is to simply sum the weights of individual edges in the path.

We can now speak about the "shortest" path, not simply in terms of which path has the smallest length, but also in terms of which path has the smallest weight. For example, in Figure 3.11(a), to go from A to B, we could go through either path (A, B), which has a length of only 1, and a weight of 0.9. Alternatively, we could use the path (A, C, D, B), which has a length of 3, but a weight of only $0.1 + 0.1 + 0.2 = 0.4$. This gives us a lot more flexibility in modeling graphs, as well as modeling what it means for a path to be short.

Algorithm for Finding Shortest Path (self-study)

Finding shortest paths connecting two vertices in a graph is an important problem with many applications. In an unweighted graph, finding shortest path is relatively simple, because by using breadth-first search traversal, we can find a way to go from the source vertex to the target vertex in as few hops as possible.

However, in a weighted graph, this simple heuristic alone is not sufficient because as we have seen previously, there could be a longer path with a smaller aggregate weight.

The most well-known algorithm for finding shortest paths in a weighted graph is *Dijkstra's shortest path algorithm*. There are many available references for this algorithm. The study of this algorithm is left to the students as self-study.

3.6 Summary

A graph data structure allows us to represent relationships in real-world networks. The basic principle of a graph is to related vertices with edges. There could be several variants of graph data structures, depending on the properties of these edges. For instance, whether edges are directed or undirected, as well as whether edges are binary or weighted.

To traverse a graph, we use two main strategies. Depth-first search or DFS prioritizes following edges in paths. Breadth-first search or BFS prioritizes visiting vertices closer to the starting vertex.

If a directed graph does not contain any cycle, we call it a directed acyclic graph or DAG. DAG is commonly used to represent precedence or dependency relationships. In a DAG, there could be one or more topological orderings. In any such ordering, an edge always points forward, with the source vertex appearing earlier in the ordering than the target vertex of each edge.

If a graph (directed or undirected) has weighted edges, a path in the graph can be associated with a weight as well, which would be some form of aggregation (e.g., summation) of the component edge weights. This is a common application in shortest path problems, where we may need to find a sequence of vertices that can be visited with the lowest cost.