Graphs Introduction and Depth-first algorithm

Carol Zander

Introduction to graphs

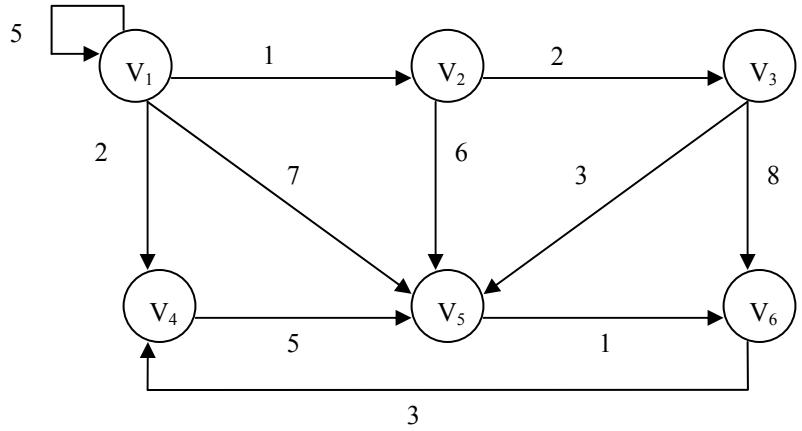
Graphs are extremely common in computer science applications because graphs are common in the physical world. Everywhere you look, you see a graph. Intuitively, a graph is a set of locations and edges connecting them. A simple example would be cities on a map that are connected by roads. Or cities connected by airplane routes. Another example would be computers in a local network that are connected to each other directly. Constellations of stars (among many other applications) can be also represented this way. Relationships can be represented as graphs. Section 9.1 has many good graph examples.

Graphs can be viewed in three ways (trees, too, since they are special kind of graph):

1. A mathematical construction – this is how we will define them
2. An abstract data type – this is how we will think about interfacing with them
3. A data structure – this is how we will implement them

The mathematical construction gives the definition of a graph: graph **G = (V, E)** consists of **a set of vertices V (often called nodes)** and **a set of edges E** (sometimes called arcs) that connect the edges. Each edge is a pair (u, v), such that $u, v \in V$. Every tree is a graph, but not vice versa.

There are two types of graphs, **directed** and **undirected**. In a directed graph, the edges are **ordered pairs,** for example (u,v), indicating that a path exists from u to v (but not vice versa, unless there is another edge.) For the edge, (u,v), **v is said to be adjacent to u**, but not the other way, i.e., u is not adjacent to v. In an undirected graph, an edge (u,v) would be equivalent to (v,u). Labeled or weighted graphs have a number (weight) associated with each edge. The weight can represent a distance or cost or some meaningful value associated with the nodes. Here is an example of a weighted, directed graph.



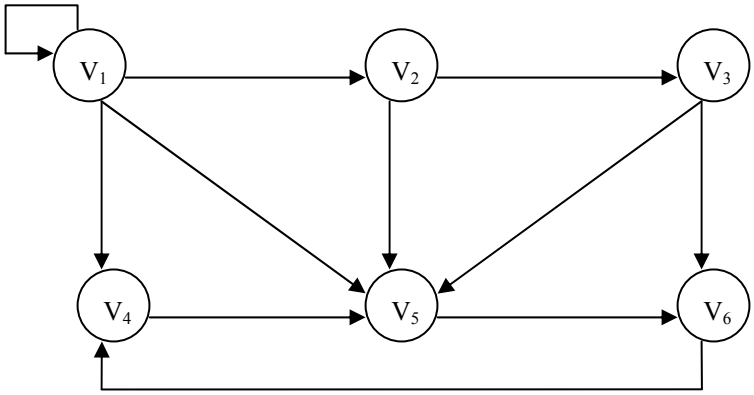
A **path** between two nodes is a sequence of edges with which you can travel from one node to the other. The weight of the path is the sum of the weights of the edges. A common operation is to find the minimum weight (or distance) path between two nodes. For an unweighted graph, this would simply be the shortest path (that is, the fewest edges required). In a simple path, the nodes are unique except possibly for the first and last node. In the above graph, v_1 v_1 v_2 v_5 is a path, but not a simple path, whereas v_1 v_2 v_5 is a simple path. The **length of a path** is the number of edges.

Any path starting and ending at the same vertex (and including at least one edge) is called a **cycle.** In the above graph, v_4 v_5 v_6 v_4 is a cycle. An **acyclic graph** has no cycles. A **connected graph** has a path from every vertex to every other vertex. (These are usually undirected graphs, but not necessarily.) A **completely connected graph** has an edge between every pair of vertices.

We have already seen one important problem that can be framed in terms of a graph: the **traveling salesperson problem.** Usually undirected graphs are used. Recall that the idea is to represent all of the cities that a traveling salesperson must visit as nodes in the graph, with edges between them representing the distance for the salesperson to travel between the cities. The problem is to determine a route starting in some city that travels to every other city once and returns to the start with the minimum travel cost. No algorithm has been found to solve this problem that is not exponential in the number of cities (nodes in the graph.) But, no one has proved that there isn't such a solution either.

Graph representations

We want to be able to represent a graph as a data structure in a computer. As with most of the data structures you have seen, there are **two common representations**, one uses just an **array (often static allocation),** and one uses **linked lists (dynamic allocation, array of lists, or lists of lists)**. The simplest representation is an **adjacency matrix** which stores the adjacencies in a graph. If there are *n* vertices in the graph, then the adjacency matrix is an *n* x *n* array of values between the nodes. In an undirected graph, typically zero means there is no edge; a one means there is an edge. If the graph above had no labels, the adjacency matrix would be as follows.

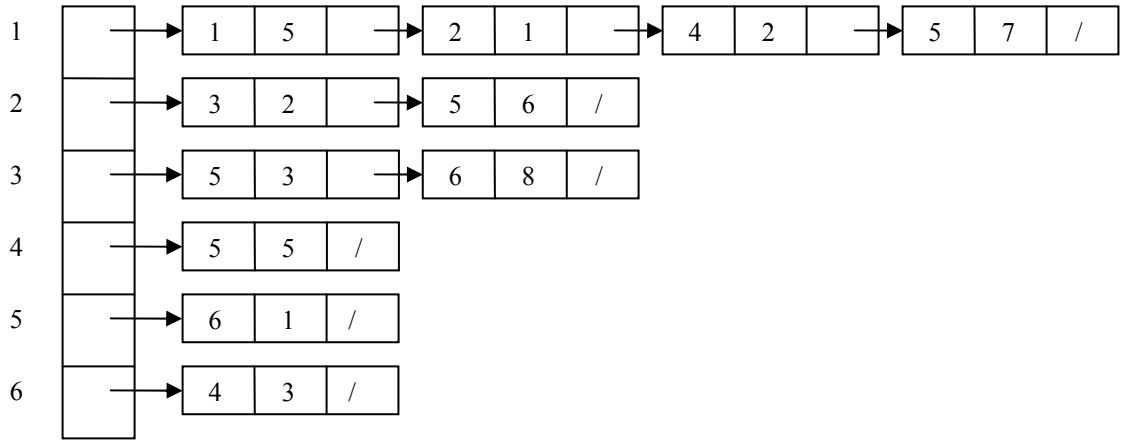


	1	2	3	4	5	6
1	1	1	0	1	1	0
2	0	0	1	0	1	0
3	0	0	0	0	1	1
4	0	0	0	0	1	0
5	0	0	0	0	0	1
6	0	0	0	1	0	0

If the graph has weights, then when an edge exists, the weight is stored. And when there is no edge, the value stored is infinity (or a very large number, in practice). The label for the edges is called weight, distance, cost, or whatever is appropriate. Here is the adjacency matrix for the first graph above that has weights.

	1	2	3	4	5	6
1	5	1	∞	2	7	∞
2	∞	∞	2	∞	6	∞
3	∞	∞	∞	∞	3	8
4	∞	∞	∞	∞	5	∞
5	∞	∞	∞	∞	∞	1
6	∞	∞	∞	3	∞	∞

The other common representation is the adjacency list. In this case, there is a (one-dimensional) array of linked-lists that store, for each node, the set of nodes that are adjacent to it and the weights of the edges. This can also be implemented as a linked list of linked lists. Here is the array of lists version.



Which representation is better?

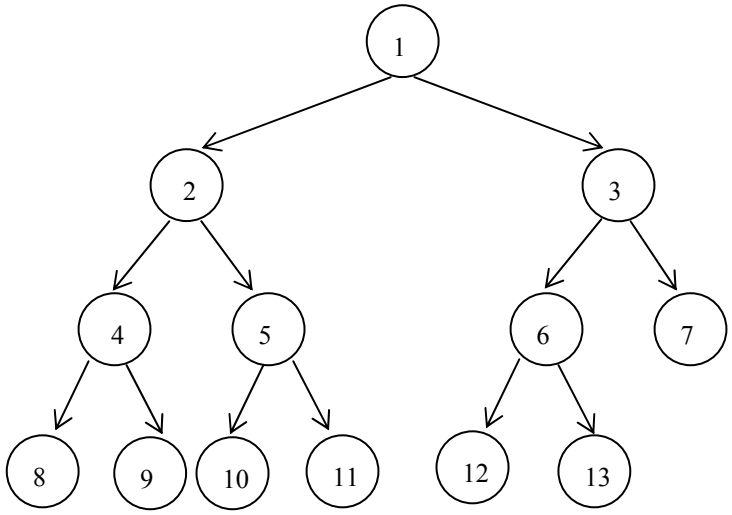
The adjacency matrix allows direct access, but can take more space (and time) for sparse graphs.
The adjacency list is easier to manage when vertices are added to or removed from the graph.

We could also store each edge list in a binary search tree. However, it is common to traverse all of the edges starting at a vertex, so the binary search tree would help only with insert and remove operations (which are less common).

Graph algorithms

We will consider several graph algorithms. We start with graph traversals. In a linked list, there is only one sensible way to traverse the list. In a tree, there are several common ways to traverse the data structure: inorder, preorder, and postorder. In a graph, a **depth-first** traversal and **breadth-first** traversal are the common traversals.

In a depth-first you keep visiting adjacent nodes until you can't go any further; in other words, you go deeper and deeper into the graph and then back up. In a binary tree, a depth-first traversal (often referred to as depth-first search) is a preorder traversal. In a breadth-first traversal, the breadth at a level is traversed before going deeper. There is no common breadth-first traversal in a binary tree. It would mean you would examine all the children before you examined the children's children. Consider the following tree:



A preorder (or depth-first traversal) is 1 2 4 8 9 5 10 11 3 6 12 13 7 .

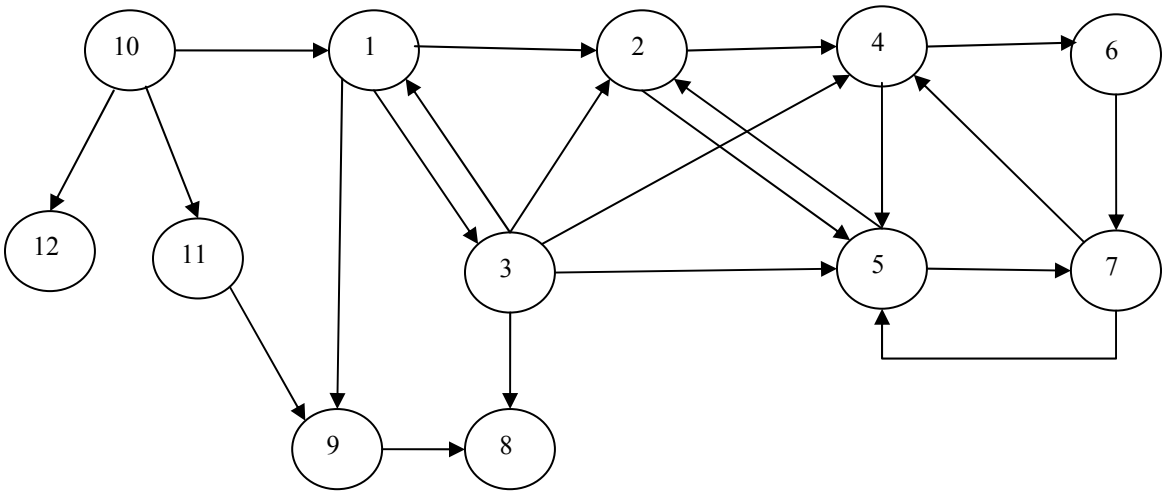
A breadth-first traversal is 1 2 3 4 5 6 7 8 9 10 11 12 13 .

Depth-first search

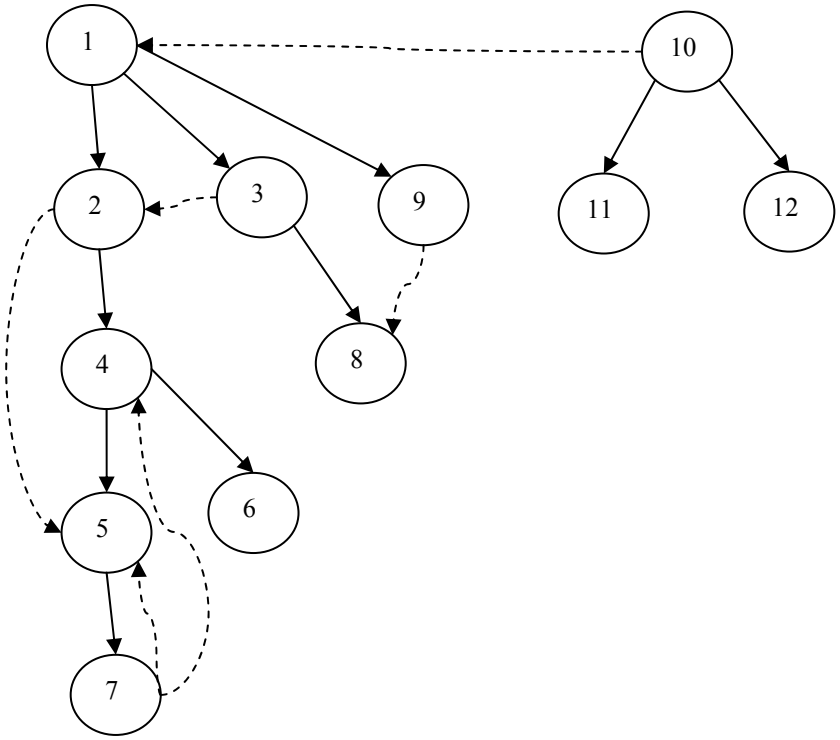
The depth-first algorithm is a generalization of a preorder traversal, visiting adjacent nodes (children) then visiting their adjacent nodes, and so on. When we visit each vertex, we have to decide in what order to visit the adjacent vertex. We will assume that there is some underlying ordering implied by the node number (or index) and use this order to break ties. The graph would be stored as either an adjacency matrix or an adjacency list so there would be a natural node to choose. In my examples, the numbers represent this natural order. So we should always start at vertex 1 (or 0 depending on where we start counting) and consider a smaller numbered node before a larger numbered node.

```
void depthFirstSearch(. . .) {
      // mark all the vertices as not visited

      for v = 1 to n {
            if (v is not visited)
                  dfs(v);
      }
}

void dfs(v ...){
      // mark v as visited
      // output v (or do whatever with v), display gives depth-first order
      for each vertex w adjacent to v {
            if (w is not visited) {
                  dfs(w);
      }
}
```

Here is an example (unconnected graph):



As you do a depth-first traversal, redraw the graph, essentially showing an execution tree. When we start at one, this gives a depth-first spanning tree. The whole thing is called a **depth-first spanning forest**.



The depth-first ordering is 1 2 4 5 7 6 3 8 9 10 11 12 .

In the depth-first spanning forest, the edges representing the traversal are called **tree edges or primary edges**. Edges in the original graph drawing that now go down or forward are called **forward edges.** The edge (2,5) is an example of a forward edge. Edges now going up or back are called **back edges**. The edges (7,4) and (7,5) are two examples of back edges. Edges that cross from one branch to another are called **cross edges.** Three examples of cross edges are (3,2), (9,8), and (10,1). If all the forward, back, and cross edges were added, it would be the original graph, just redrawn, giving it *depth*.