CSci 4041: Algorithms and Data Structures
Graphs and their Representations

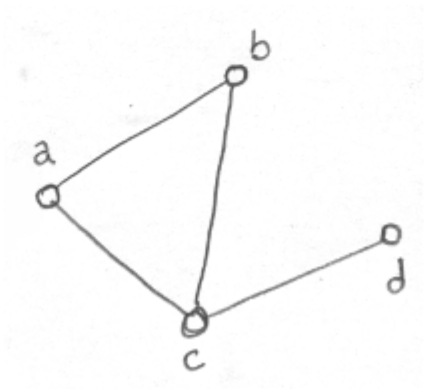
Last revision April 9, 2020

Here's the agenda for this lecture.

- What's a graph?
- Why I never say the word *vertices.*
- Why you shouldn't say it either.
- Kinds of graphs.
  - Directed and Undirected.
  - Weighted.
  - Cyclic and Acyclic.
  - Graphs with self-loops.
- Graph representations.
  - Nodes and pointers.
  - Adjacency structures.
  - Adjacency matrixes.
- Choosing a representation.

Chapters 22–25 of Cormen talk about *graphs* and algorithms for graphs, which is what the rest of this course will be about. You may also want to read Appendix B.4 in Cormen for more detail about what I'm discussing here. Graphs appear all over computer science, because many important problems can be stated in terms of them. Also, graph procedures are fun (for some definition of that word), most are easy to understand, and they provide good examples of principles for algorithm design.

What's a graph? Informally, we can think of a graph as a *network.* It consists of *vertexes* that are connected to each other by *edges.* Vertexes are also sometimes called *nodes,* and edges are also sometimes called *links.* We often draw graphs as diagrams that look like this.



The graph G_1.

Each circle is a vertex, and each line that connects two vertexes is an edge. This graph has four vertexes, labeled *a, b, c,* and *d.* It also has four edges: one that connects vertexes *a* and *b,* one that connects *a* and *c,* one that connects *b* and *c,* and one that connects *c* and *d.* Two vertexes are not connected if there is no edge between them.

Formally, we can define a graph G as as a 2-tuple $G = \langle V, E \rangle$. Here V is a set of vertexes, and E is a set of edges. Each edge in E is an ordered pair (v_1, v_2) where $v_1 \in V$ and $v_2 \in V$. In other words, $E \subseteq V \times V$, where the operator '×' denotes the cross product of sets. (We're using things from set theory that you saw in a discrete math course like CSci 2011 and forgot, because you thought you'd never need them again.) For example, the graph G_1 shown above could be represented like this.

$$G_1 = \langle \{ a, b, c, d \}, \{ (a, b), (a, c), (b, c), (c, d) \} \rangle$$

In G_1, if a vertex like *a* is connected by an edge to another vertex like *b*, then *b* is also connected to *a.* This is represented by only one ordered pair (*a*, *b*) in *E.*

A graph with as many edges as possible would have $|V|^2$ edges, where $|V|$ is the number of elements in *V.* We say that a graph is *sparse* if $|E|$ is much less than $|V|^2$, and we say that a graph is *dense* if $|E|$ is about the same as $|V|^2$. The phrases *much less than* and *about the same as* aren't well defined, so please don't ask for their formal definitions, because there aren't any. However, we can still usually tell whether a graph is sparse or dense. We might need to do that because some things (algorithms and representations) work better for sparse graphs, and some things work better for dense ones.

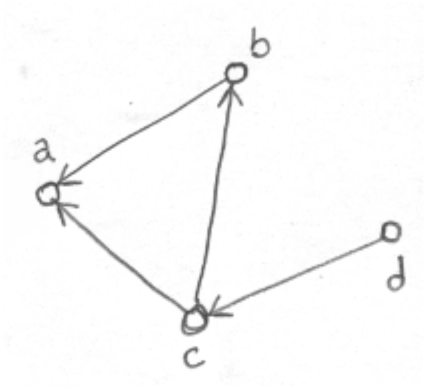
There's a lot more graph terminology. I'll discuss some of it here, and the rest as needed for specific graph algorithms, later in the course.

Why I won't say the word *vertices.* In previous paragraphs, I used the word *vertexes* as the plural of the word *vertex.* I didn't use the word *vertices,* even though either one is correct. Why?

I didn't say *vertices* because when some people hear that word, they think its singular is *vertisee,* by what linguists call *back-formation.* This happens even though no one thinks the singular of *appendices* is *appendisee,* or that the singular of *matrices* is *matrisee.* There is no such word as *vertisee!* If you say *vertisee,* then you'll sound like a fool—and none of my students are fools. To keep you from saying *vertisee,* or even thinking about it, I'll always use *matrixes* as the plural of *matrix,* and *vertexes* as the plural of *vertex.*

Directed and undirected graphs. The graph G_1 shown above is an *undirected* graph. In an undirected graph, for all vertexes *u* and *v,* if *u* is connected to *v* by an edge, then *v* is also connected to *u* by that same edge. Undirected graphs are drawn with edges as simple lines.

We can also have *directed* graphs. In a directed graph, for all vertexes *u* and *v,* if *u* is connected to *v* by an edge, then *v* is not connected to *u* by that same edge. (But *v* might still be connected to *u* by a different edge.) Directed graphs are drawn with edges as arrows. For example, here's another graph G_2 that looks something like G_1, but it's a directed graph.



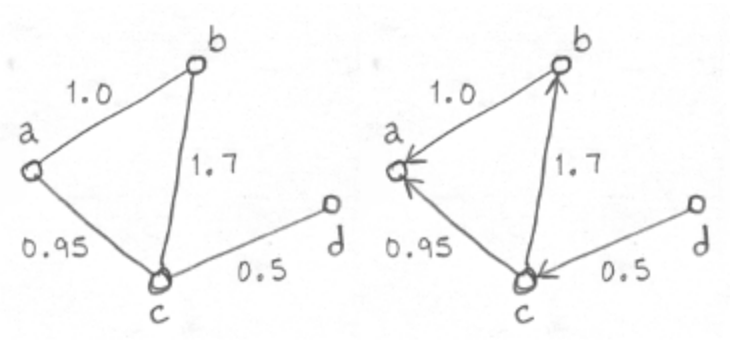
The graph G_2.

The formal definition $G = \langle V, E \rangle$, is a little different for a directed graph than for an undirected graph. The set of vertexes *V* remains the same, but the set of edges *E* now contains a pair (v_1, v_2) only if there is an edge (an arrow) from v_1 to v_2, where $v_1 \in V$ and $v_2 \in V$. That means the graph G_2 is represented like this.

$$G_2 = \langle \{ a, b, c, d \}, \{ (b, a), (c, a), (c, b), (d, c) \} \rangle$$

One thing we might want to do with a graph is follow a *path* through it. To follow a path, we start with a vertex, and move from one vertex to another by following the edges connecting them. In a directed graph, we can follow edges only in the directions of their arrows. For example, in G_2, we can move from *b* to *a* but not from *a* to *b.* In an undirected graph, we can follow edges in either direction. For example, in G_1, we can move from *a* to *b* and from *b* to *a.*

Weighted graphs. In *weighted graphs,* edges have *weights* that are real numbers. A weighted graph can be either directed or undirected. For example, G_3 and G_4 show weighted versions of the graphs G_1 and G_2. I made up their weights at random.

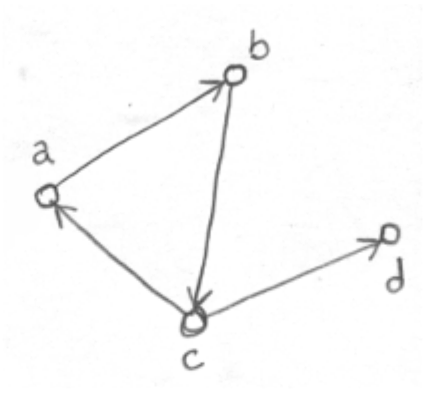


The graphs G_3 and G_4.

Suppose we want to follow a path through a weighted graph. Then we may want to find a path for which the sum of its edge weights is at a minimum. For example, we may interpret an edge's weight as the *cost* of moving from one vertex to another. Then we may want to find a path with a minimum cost.

Some algorithms for weighted graphs require that no edge has a weight of 0. Some other algorithms require that no edge has a negative weight, or that edges with negative weights can't appear in certain ways. We'll see examples of those algorithms later in the course.

Cyclic and acyclic graphs. Suppose we have a directed graph. The graph has a *cycle* if we can start at a vertex *v,* and follow one or edges in the direction of the arrows to revisit *v* again. For example, the graph G_5 has a cycle involving vertexes *a, b,* and *c.*

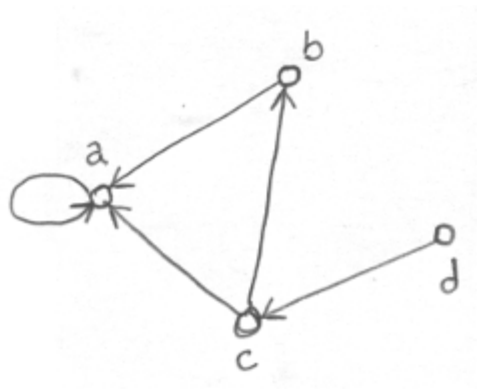


The graph G_5.

By convention, we say that an undirected graph can't have cycles, even though we might be able to start at one of its vertexes and follow a path back to that vertex again.

A directed graph can have zero or more cycles. If it has at least one cycle, then it's called a *cyclic* graph. If it has no cycles, then it's called an *acyclic* graph. You've seen acyclic and cyclic graphs before in whatever data structures course you took before registering for CSCI 4041, although maybe you didn't know it. A singly-linked linear linked list is a kind of acyclic graph, and so is a tree. A doubly-linked list, or a doubly-linked circular list, is a kind of cyclic graph.

Graphs with self-loops. Directed graphs can have *self-loops,* which are edges that start and end at the same vertex. By convention, undirected graphs aren't allowed to have self-loops. For example, the directed graph G_6 has a self-loop from vertex *a* back to vertex *a.*

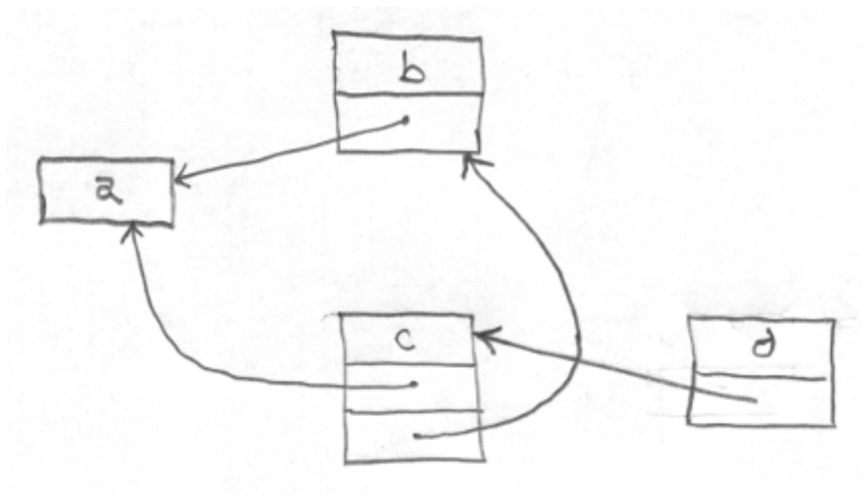


The graph G_6.

When we represent a directed graph as $G = \langle V, E \rangle$, a self-loop is represented as an ordered pair like $(v, v) \in E$, where $v \in V$. Some graph algorithms can't correctly handle graphs with self-loops.

Nodes and pointers. Since we're computer scientists (or at least I am), we'd like to represent graphs as data structures, so we can write programs that work with them. In a data structures course, you saw that we can represent lists and trees as *nodes,* whose *pointers* reference other nodes.

We could represent a graph with nodes and pointers in the same way, with the nodes as vertexes and the pointers as edges. For example, here's a data structure that represents the directed graph G_2. Each node has a slot that tells its label. It also has zero or more pointers to other nodes. The node for *a* has no pointers, the nodes for *b* and *d* have one pointer each, and the node for *c* has two pointers. The order in which the pointers appear within the node isn't important.



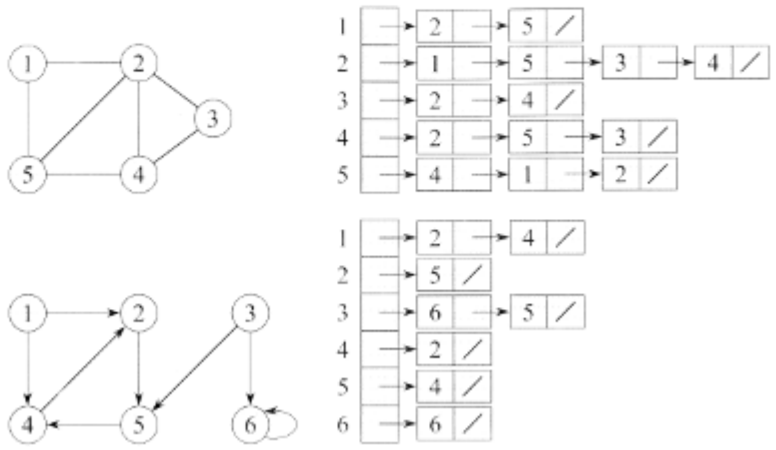
Nodes and pointers for G_1.

An undirected graph has a similar representation, but it needs twice as many pointers: two for each edge. If a node *p* has a pointer to another node *q,* then *q* also has a pointer to *p,* because we can move from one vertex to another along an undirected edge in either direction.

How efficient is this representation in terms of memory space? Since we have one node for each vertex, we need $|V|$ nodes. In a directed graph, we also need $|E|$ pointers, one for each edge. In an undirected graph, we need $2|E|$ pointers, two for each edge. We therefore need $\Theta(|V| + |E|)$ space.

One difficulty with this representation is how to access the graph. If we access it only through a pointer to the node *a,* then there will be no pointers to the nodes *b, c,* and *d,* so they might be garbage collected. Similarly, if we access it only through a pointer to the node for *c,* then there would be no pointers to the node *d,* and it might be garbage collected. Another difficulty is that the nodes have different sizes. We'd need a way to allocate nodes with different numbers of pointers in them, and we'd need a way to tell how many pointers are in each node.

Adjacency structures. Perhaps because of these difficulties, Cormen doesn't use the simple nodes-and-pointers representation for graphs that we just discussed. Instead, one way he represents a graph is by using what he calls an *adjacency list.* However, it isn't a list at all, but an array whose elements are lists. Since we should call things by their right names, I'll call it an *adjacency structure* instead. Adjacency structures for directed and undirected graphs are shown on top of page 590, and here.



Adjacency structures.

An adjacency structure represents vertexes as integers 1, 2 ..., |*V*|. They're indexes in the adjacency structure's array. Now suppose that *G* is a graph. Cormen writes *G.Adj* to mean the array in *G*'s adjacency structure. If *u* is a integer representing a vertex, then *G.Adj*[*u*] is the linear linked list that holds the vertexes to which *u* is connected.

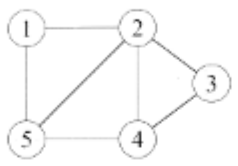
For example, in the undirected graph shown here (without arrows), vertex 1 is is connected to the vertexes 2 and 5. That means the linked list at index 1 has two nodes, one containing 2 and the other containing 5. Similarly, in the directed graph shown here (with arrows) vertex 3 is connected to the vertexes 5 and 6. That means the linked list at index 3 has two nodes, one containing 5 and the other containing 6. Also, vertex 6 has a self-edge, so the linked list at index 6 has one node containing 6.

We can also use an adjacency structure to represent a weighted graph, although we don't show that here. Since each node in the adjacency structure represents an edge, we just add an extra real-valued slot to that node, which holds the edge's weight.

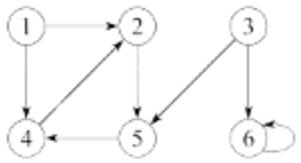
How efficient is an adjacency structure? If *G* is a directed graph, then the sum of the lengths of the adjacency lists is |*E*|, since each edge (*u, v*) is represented by having *v* appear in the linked list *G.Adj*[*u*]. If *G* is an undirected graph, then the sum of the lengths of the lists is 2|*E*|, since each edge (*u, v*) is represented by having *v* appear in the linked list *G.Adj*[*u*], and by having *u* appear in the linked list *G.Adj*[*v*]. As a result, for either kind of graph, the adjacency list representation requires Θ(|*E*| + |*V*|) memory space.

Adjacency lists have an unfortunate property. If we want to test if a graph *G* contains an edge (*u, v*), then we must search the list *G.Adj*[*u*] for a node containing *v.* That requires linear search, which is potentially slow. We might be able go faster by using data structures other than linked lists. For example, we might use binary search trees or hash tables instead.

Adjacency matrixes. Cormen also represents a graph using an *adjacency matrix.* Suppose that a graph *G* has vertexes represented by the integers 1, 2, ..., |*V*|. Then an adjacency matrix is a |*V*| × |*V*| array whose elements are 0 or 1. Cormen writes *G.M* to mean *G*'s adjacency matrix. If *G* has an edge (*u, v*), then *G.M*[*u, v*] = 1. If *G* has no such edge, then *G.M*[*u, v*] = 0. Adjacency matrixes for directed and undirected graphs are shown on top of page 590, and here.



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Adjacency matrixes.

For example, in the undirected graph (without the arrows), there's an edge from vertex 2 to vertex 5, so *G.M*[2, 5] = 1. There's no edge from vertex 1 to vertex 3, however, so *G.M*[1, 3] = 0. In the directed graph (with the arrows), there's an edge from vertex 1 to vertex 2, so *G.M*[1, 2] = 1. However, there's no edge from vertex 2 to vertex 1, so *G.M*[2, 1] = 0.

We can also use an adjacency matrix to represent a weighted graph. Suppose the edge (*u, v*) has weight *w,* which we'll assume is not 0. Then *G.M*[*u, v*] = *w.* As before, if (*u, v*) is not an edge in *G*, then *G.M*[*u, v*] = 0. If we allow edges with zero weights, then we must figure out another way to represent missing edges. We might be able to represent them by using a special value like NIL instead of a weight.

How efficient is an adjacency matrix? Suppose we have $|V|$ vertexes. If *G* is a directed graph, then its adjacency matrix *G.M* must have $|V|^2$ elements. However, if *G* is an undirected graph, then we don't need that many. If there's an edge between vertexes *u* and *v,* then there's also effectively an edge from *v* to *u,* so *G.M* needs elements only above (or below) its diagonal. In either case, we need $\Theta(|V|^2)$ memory space.

Choosing a representation. Which representation, an adjacency structure or an adjacency matrix, should we use to represent a graph? If the graph is sparse, then an adjacency structure is probably best, because it uses memory only to represent the few edges that actually exist. If the graph is dense, then an adjacency matrix is probably best, because a large number of edges must be represented, and a matrix can represent them more efficiently than many lists of nodes.

However, if execution time is important, then we may want to use an adjacency matrix even for a sparse graph. This is because a matrix is represented as some kind of array, and in most programming languages, we can access array elements in *O*(1) time. If we use an adjacency structure instead, then we may have use linear search on lists of nodes, which is slower.