Chapter 2
Manipulating Network Data

2.1 Introduction

We have seen that the term ‘network,’ broadly speaking, refers to a collection of elements and their inter-relations. The mathematical concept of a graph lends precision to this notion. We will introduce the basic elements of graphs—both undirected and directed—in Sect. 2.2 and discuss how to generate network graphs, both ‘by hand’ and from network data of various forms.

As a representation of a complex system, a graph alone (i.e., as merely a collection of vertices and edges) is often insufficient. Rather, there may be additional information important to the application at hand, in the form of variables that can be indexed by the vertices (e.g., gender of members of a social network) or the edges (e.g., average time required to traverse a link in a transportation network). Alternatively, at a coarser level of granularity, it may be convenient to associate vertices or edges with groups (e.g., all proteins in a protein–protein interaction network that are involved with a certain type of signaling event in a cell). Indeed, we can imagine potentially equipping vertices and edges with several variables of interest. Doing so corresponds to the notion of decorating a network graph, which is discussed in Sect. 2.3.

Finally, in using graphs to represent network data, a certain level of familiarity with basic graph theoretic concepts, as well as an ability to assess certain basic properties of graphs, is essential. We therefore devote Sect. 2.4 to a brief overview of such concepts and properties, including a quick look at a handful of important special classes of graphs.

For creating, decorating, and assessing basic properties of network graphs, **igraph** is particularly useful.[1] A library and R package for network analysis, **igraph** contains a set of data types and functions for (relatively!) straightforward implementation and rapid prototyping of graph algorithms, and allows for the fast handling of

[1] Alternatively, there is within the **network** and **sna** packages, found in the **statnet** suite, a similarly rich set of tools for the manipulation and characterization of network graphs. These packages share nontrivial overlap with **igraph**.

large graphs (e.g., on the order of millions of vertices and edges). As such, its use will figure heavily in this and the following two chapters (i.e., where the emphasis is on descriptive methods). The fact that **igraph** was developed as a research tool and that its focus originally was to be able to handle large graphs efficiently, means that its learning curve used to be somewhat steep. Recent versions do not necessarily flatten the learning curve, but are nevertheless friendlier to the user, once she has mastered the basics.

2.2 Creating Network Graphs

2.2.1 Undirected and Directed Graphs

Formally, a *graph* $G = (V, E)$ is a mathematical structure consisting of a set V of *vertices* (also commonly called *nodes*) and a set E of *edges* (also commonly called *links*), where elements of E are unordered pairs $\{u, v\}$ of distinct vertices $u, v \in V$. The number of vertices $N_v = |V|$ and the number of edges $N_e = |E|$ are sometimes called the *order* and *size* of the graph G, respectively. Often, and without loss of generality,[2] we will label the vertices simply with the integers $1, \ldots, N_v$, and the edges, analogously.

In **igraph** there is an 'igraph' class for graphs.[3] In this section, we will see a number of ways to create an object of the igraph class in R, and various ways to extract and summarize the information in that object.

For small, toy graphs, the function `graph.formula` can be used. For example,

```
#2.1 > library(igraph)
     > g <- graph.formula(1-2, 1-3, 2-3, 2-4, 3-5, 4-5, 4-6,
     +                    4-7, 5-6, 6-7)
```

creates a graph object `g` with $N_v = 7$ vertices

```
#2.2 > V(g)
     Vertex sequence:
     [1] "1" "2" "3" "4" "5" "6" "7"
```

and $N_e = 10$ edges

```
#2.3 > E(g)
     Edge sequence:

     [1]  2 -- 1
     [2]  3 -- 1
     [3]  3 -- 2
     [4]  4 -- 2
```

[2] Technically, a graph G is unique only up to relabellings of its vertices and edges that leave the structure unchanged. Two graphs that are equivalent in this sense are called *isomorphic*.

[3] The exact representation of 'igraph' objects is not visible for the user and is subject to change.

```
[5]   5 -- 3
[6]   5 -- 4
[7]   6 -- 4
[8]   7 -- 4
[9]   6 -- 5
[10]  7 -- 6
```

This same information, in a slightly more compressed format, is recovered easily using the relevant structure command.

```
#2.4 > str(g)
     IGRAPH UN-- 7 10 --
     + attr: name (v/c)
     + edges (vertex names):
     1 -- 2, 3
     2 -- 1, 3, 4
     3 -- 1, 2, 5
     4 -- 2, 5, 6, 7
     5 -- 3, 4, 6
     6 -- 4, 5, 7
     7 -- 4, 6
```

A visual representation of this graph, generated simply through the command[4]

```
#2.5 > plot(g)
```

is shown in Fig. 2.1, on the left.

The character U seen accompanying the summary of g above indicates that our graph is *undirected*, in that there is no ordering in the vertices defining an edge. A graph G for which each edge in E has an ordering to its vertices (i.e., so that (u,v) is distinct from (u,v), for $u,v \in V$) is called a *directed graph* or *digraph.* Such edges are called *directed edges* or *arcs*, with the direction of an arc (u,v) read from left to right, from the *tail* u to the *head* v. Note that digraphs may have two arcs between a pair of vertices, with the vertices playing opposite roles of head and tail for the respective arcs. In this case, the two arcs are said to be *mutual*.

Directed edges in graph.formula are indicated using a minus/plus convention. In Fig. 2.1, on the right, is shown an example of a digraph consisting of three vertices, with two directed edges and one mutual edge.

```
#2.6 > dg <- graph.formula(1-+2, 1-+3, 2++3)
     > plot(dg)
```

We note that in defining both of the graphs above we have used the standard convention of labeling vertices with the numbers 1 through N_v, which is also the default in **igraph**. In practice, however, we may already have natural labels, such as the names of people in a social network, or of genes in a gene regulatory network. Such labels can be used instead of the default choice by generating the graph with them explicitly.

[4] This is the most basic visualization. We will explore the topic of visualization on its own in more depth in Chap. 3.

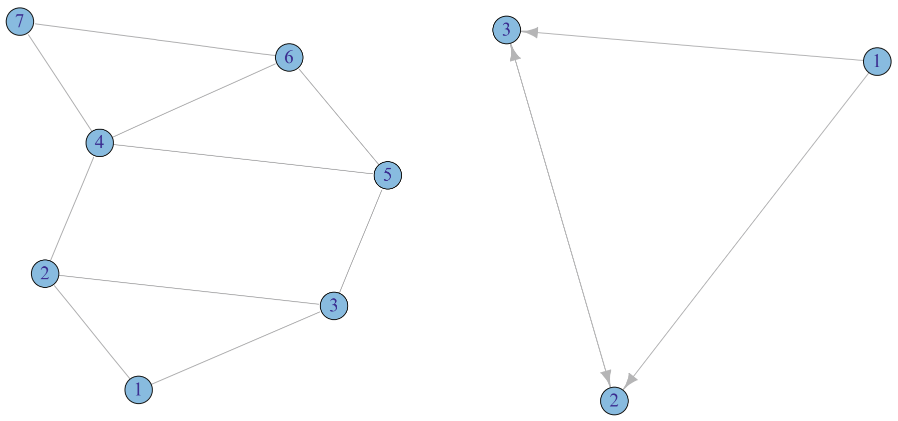

Fig. 2.1 *Left*: an undirected graph. *Right*: a directed graph

```
#2.7 > dg <- graph.formula(Sam-+Mary, Sam-+Tom, Mary++Tom)
     > str(dg)
     IGRAPH DN-- 3 4 --
     + attr: name (v/c)
     + edges (vertex names):
     [1] Sam ->Mary Sam ->Tom  Mary->Tom  Tom ->Mary
```

Alternatively, vertex labels can be changed from the default after initially creating the graph, by modifying the `name` vertex attribute of the graph object.

```
#2.8 > V(dg)$name <- c("Sam", "Mary", "Tom")
```

2.2.2 Representations for Graphs

Realistically, we do not usually expect to enter a graph by hand, since most networks encountered in practice have at least tens of vertices and edges, if not tens of thousands (or even millions!). Rather, information for constructing a network graph typically will be stored in a data file. At the most elementary level, there are three basic formats: adjacency lists, edge lists, and adjacency matrices.

An *adjacency list* representation of a graph G is simply an array of size N_v, ordered with respect to the ordering of the vertices in V, each element of which is a list, where the ith list contains the set of all vertices j for which there is an edge from i to j. This is the representation usually used by **igraph**, evident in printing the output from the structure function `str` in the examples above.

An *edge list* is a simple two-column list of all vertex pairs that are joined by an edge. In **igraph**, edge lists are used, for example, when printing the edge set E.

```
#2.9 > E(dg)
     Edge sequence:

```

```
[1] Sam  -> Mary
[2] Sam  -> Tom
[3] Mary -> Tom
[4] Tom  -> Mary
```

The function `get.edgelist` returns an edge list as a two-column R matrix.

Finally, graphs can also be stored in matrix form. The $N_v \times N_v$ *adjacency matrix* for a graph $G = (V, E)$, say $\mathbf{A}$, is defined so that

$$A_{ij} = \begin{cases} 1, & \text{if } \{i,j\} \in E \ , \\ 0, & \text{otherwise} \ . \end{cases} \tag{2.1}$$

In words, $\mathbf{A}$ is non-zero for entries whose row-column indices (i, j) correspond to vertices in G joined by an edge, from i to j, and zero, for those that are not. The matrix A will be symmetric for undirected graphs.

```
#2.10
> get.adjacency(g)
7 x 7 sparse Matrix of class "dgCMatrix"
  1 2 3 4 5 6 7
1 . 1 1 . . . .
2 1 . 1 1 . . .
3 1 1 . . 1 . .
4 . 1 . . 1 1 1
5 . . 1 1 . 1 .
6 . . . 1 1 . 1
7 . . . 1 . 1 .
```

This last choice of representation is often a natural one, given that matrices are fundamental data objects in most programming and software environments and that network graphs frequently are encoded in statistical models through their adjacency matrices. However, their use with the type of large, sparse networks commonly encountered in practice can be inefficient, unless coupled with the use of sparse matrix tools.

In **igraph**, network data already loaded into R in these specific formats can be used to generate graphs using the functions `graph.adjlist`, `graph.edgelist`, and `graph.adjacency`, respectively. For data stored in a file, the function `read.graph` can be used. In fact, this latter function not only supports the three formats discussed above, but also a number of other formats (e.g., such as GraphML, Pajek, etc.). Conversely, the function `write.graph` can be used to save graphs in various formats.

2.2.3 Operations on Graphs

The graph(s) that we are able to load into R may not be the graph that we ultimately want. Various operations on the graph(s) we have available may be necessary, including extracting part of a graph, deleting vertices, adding edges, or even combining multiple graphs.

The notion of a 'part' of a graph is captured through the concept of a subgraph. A graph $H = (V_H, E_H)$ is a *subgraph* of another graph $G = (V_G, E_G)$ if $V_H \subseteq V_G$ and $E_H \subseteq E_G$. Often we are interested in an *induced subgraph* of a graph G, i.e., a subgraph $G' = (V', E')$, where $V' \subseteq V$ is a prespecified subset of vertices and $E' \subseteq E$ is the collection of edges to be found in G among that subset of vertices. For example, consider the subgraph of g induced by the first five vertices.

```
#2.11 > h <- induced.subgraph(g, 1:5)
      > str(h)
      IGRAPH UN-- 5 6 --
      + attr: name (v/c)
      + edges (vertex names):
      [1] 1--2 1--3 2--3 2--4 3--5 4--5
```

The inclusion or exclusion of vertices or edges in a graph $G = (V, E)$ can be conceived of as the application of addition or subtraction operators, respectively, to the sets V and E. For example, the subgraph h generated just above could also have been created from g by removing the vertices 6 and 7.

```
#2.12 > h <- g - vertices(c(6,7))
```

Similarly, g can be recovered from h by first adding these two vertices back in, and then, adding the appropriate edges.

```
#2.13 > h <- h + vertices(c(6,7))
      > g <- h + edges(c(4,6),c(4,7),c(5,6),c(6,7))
```

Finally, the basic set-theoretic concepts of union, disjoint union, intersection, difference, and complement all extend in a natural fashion to graphs. For example, the union of two graphs, say H_1 and H_2, is a graph G in which vertices and edges are included if and only if they are included in at least one of H_1 or H_2. For example, our toy graph g may be created through the union of the (induced) subgraph h defined above and a second appropriately defined subgraph.

```
#2.14 > h1 <- h
      > h2 <- graph.formula(4-6, 4-7, 5-6, 6-7)
      > g <- graph.union(h1,h2)
```

2.3 Decorating Network Graphs

2.3.1 Vertex, Edge, and Graph Attributes

At the heart of a network-based representation of data from a complex system will be a graph. But frequently there are other relevant data to be had as well. From a network-centric perspective, these other data can be thought of as *attributes*, i.e., values associated with the corresponding network graph. Equipping a graph with such attributes is referred to as *decorating* the graph. Typically, the vertices or edges of a graph (or both) are decorated with attributes, although the graph as a whole may be decorated as well. In **igraph**, the elements of graph objects (i.e., particularly the vertex and edge sequences, and subsets thereof) may be equipped with attributes simply by using the `'$'` operator.

Vertex attributes are variables indexed by vertices, and may be of discrete or continuous type. Instances of the former type include the gender of actors in a social network, the infection status of computers in an Internet network in the midst of an on-line virus (e.g., a worm), and a list of biological pathways in which a protein in a protein–protein interaction network is known to participate, while an example of the latter type is the voltage potential levels in the brain measured at electrodes in an electrocorticogram (ECoG) grid. For example, recall that the names of the three actors in our toy digraph are

```
#2.15 > V(dg)$name
      [1] "Sam"  "Mary" "Tom"
```

Their gender is added to `dg` as

```
#2.16 > V(dg)$gender <- c("M","F","M")
```

Note that the notion of vertex attributes also may be used advantageously to equip vertices with properties during the course of an analysis, either as input to or output from calculations within R. For example, this might mean associating the color red with our vertices

```
#2.17 > V(g)$color <- "red"
```

to be used in plotting the graph (see Chap. 3). Or it might mean saving the values of some vertex characteristic we have computed, such as the types of vertex centrality measures to be introduced in Chap. 4.

Edge attributes similarly are values of variables indexed by adjacent vertex pairs and, as with vertex attributes, they may be of both discrete or continuous type. Examples of discrete edge attributes include whether one gene regulates another in an inhibitory or excitatory fashion, or whether two countries have a friendly or antagonistic political relationship. Continuous edge attributes, on the other hand, often represent some measure of the strength of relationship between vertex pairs. For example, we might equip each edge in a network of email exchanges (with vertices representing email addresses) by the rate at which emails were exchanged over a given period of time. Or we might define an attribute on edges between adjacent stations in a subway network (e.g., the Paris metro) to represent the average time necessary during a given hour of the day for trains to run from one to station to the next.

Often edge attributes can be thought of usefully, for the purposes of various analyses, as weights. Edge weights generally are non-negative, by convention, and often are scaled to fall between zero and one. A graph for which the edges are equipped with weights is referred to as a *weighted graph*.[5]

[5] More generally, a weighted graph can be defined as a pair (V,E), where V is a set of vertices, as before, but the elements in E are now non-negative numbers, with one such number for each vertex pair. Analogously, the adjacency matrix $\mathbf{A}$ for a weighted graph is defined such that the entry A_{ij} is equal to the corresponding weight for the vertex pair i and j.

```
#2.18 > is.weighted(g)
      [1] FALSE
      > wg <- g
      > E(wg)$weight <- runif(ecount(wg))
      > is.weighted(wg)
      [1] TRUE
```

As with vertex attributes, edge attributes may also be used to equip edges with properties to be used in calls to other R functions, such as the `plot` function.

In principle, a graph itself may be decorated with an attribute, and indeed, it is possible to equip graph objects with attributes in **igraph**. The most natural use of this feature arguably is to equip a graph with relevant background information, such as a name

```
#2.19 > g$name <- "Toy Graph"
```

or a seminal data source.

2.3.2 Using Data Frames

Just as network graphs typically are not entered by hand for graphs of any nontrivial magnitude, but rather are encoded in data frames and files, so too attributes tend to be similarly encoded. For example, in R, a network graph and all vertex and edge attributes can be conveniently represented using two data frames, one with vertex information, and the other, with edge information. Under this approach, the first column of the vertex data frame contains the vertex names (i.e., either the default numerical labels or symbolic), while each of the other columns contain the values of a given vertex attribute. Similarly, the first two columns of the edge data frame contain an edge list defining the graph, while each of the other columns contain the values of a given edge attribute.

Consider, for example, the lawyer data set of Lazega [98], introduced in Chap. 1. Collecting the information on collaborative working relationships, in the form of an edge list, in the data frame `elist.lazega`, and the various vertex attribute variables, in the data frame `v.attr.lazega`, they may be combined into a single graph object in **igraph** as

```
#2.20 > library(sand)
      > g.lazega <- graph.data.frame(elist.lazega,
      +                              directed="FALSE",
      +                              vertices=v.attr.lazega)
      > g.lazega$name <- "Lazega Lawyers"
```

Our full set of network information on these

```
#2.21 > vcount(g.lazega)
      [1] 36
```

lawyers now consists of the

```
#2.22 > ecount(g.lazega)
      [1] 115
```

pairs that declared they work together, along with the eight vertex attributes

```
#2.23 > list.vertex.attributes(g.lazega)
      [1] "name"       "Seniority" "Status"     "Gender"
      [5] "Office"     "Years"     "Age"        "Practice"
      [9] "School"
```

(in addition to the vertex name).

We will see a variety of ways in the chapters that follow to characterize and model these network data and others like them.

2.4 Talking About Graphs

2.4.1 Basic Graph Concepts

With the adoption of a graph-based framework for representing relational data in network analysis we inherit a rich vocabulary for discussing various important concepts related to graphs. We briefly review and demonstrate some of these here, as they are necessary for doing even the most basic of network analyses.

As defined at the start of this chapter, a graph has no edges for which both ends connect to a single vertex (called *loops*) and no pairs of vertices with more than one edge between them (called *multi-edges*). An object with either of these properties is called a *multi-graph*.[6] A graph that is not a multi-graph is called a *simple* graph, and its edges are referred to as *proper* edges.

It is straightforward to determine whether or not a graph is simple. Our toy graph `g` is simple.

```
#2.24 > is.simple(g)
      [1] TRUE
```

But duplicating the edge between vertices 2 and 3, for instance, yields a multi-graph.

```
#2.25 > mg <- g + edge(2,3)
      > str(mg)
      IGRAPH UN-- 7 11 -- Toy Graph
      + attr: name (g/c), name (v/c), color (v/c)
      + edges (vertex names):
      1 -- 2, 3
      2 -- 1, 3, 3, 4
      3 -- 1, 2, 2, 5
      4 -- 2, 5, 6, 7
```

[6] In fact, the **igraph** data model is more general than described above, and allows for multi-graphs, with multiple edges between the same pair of vertices and edges from a vertex to itself.

```
5 -- 3, 4, 6
6 -- 4, 5, 7
7 -- 4, 6
> is.simple(mg)
[1] FALSE
```

Checking whether or not a network graph is simple is a somewhat trivial but nevertheless important preliminary step in doing a typical network analysis, as many models and methods assume the input graph to be simple or behave differently if it is not.

Note that it is straightforward, and indeed not uncommon in practice, to transform a multi-graph into a weighted graph, wherein each resulting proper edge is equipped with a weight equal to the multiplicity of that edge in the original multi-graph. For example, converting our toy multi-graph `mg` to a weighted graph results in a simple graph,

```
#2.26
> E(mg)$weight <- 1
> wg2 <- simplify(mg)
> is.simple(wg2)
[1] TRUE
```

the edges which match our initial toy graph `g`,

```
#2.27
> str(wg2)
IGRAPH UNW- 7 10 -- Toy Graph
+ attr: name (g/c), name (v/c), color (v/c),
  weight (e/n)
+ edges (vertex names):
1 -- 2, 3
2 -- 1, 3, 4
3 -- 1, 2, 5
4 -- 2, 5, 6, 7
5 -- 3, 4, 6
6 -- 4, 5, 7
7 -- 4, 6
```

but for which the third edge (i.e., connecting vertices 2 and 3) has a weight of 2.

```
#2.28
> E(wg2)$weight
 [1] 1 1 2 1 1 1 1 1 1 1
```

Moving beyond such basic concerns regarding the nature of the edges in a graph, it is necessary to have a language for discussing the connectivity of a graph. The most basic notion of connectivity is that of adjacency. Two vertices $u, v \in V$ are said to be *adjacent* if joined by an edge in E. Such vertices are also referred to as *neighbors*. For example, the three neighbors of vertex 5 in our toy graph `g` are

```
#2.29
> neighbors(g, 5)
[1] 3 4 6
```

Similarly, two edges $e_1, e_2 \in E$ are adjacent if joined by a common endpoint in V. A vertex $v \in V$ is *incident* on an edge $e \in E$ if v is an endpoint of e. From this follows the notion of the *degree* of a vertex v, say d_v, defined as the number of edges incident on v.

#2.30
```
> degree(g)
1 2 3 4 5 6 7
2 3 3 4 3 3 2
```

For digraphs, vertex degree is replaced by *in-degree* (i.e., d_v^{in}) and *out-degree* (i.e., d_v^{out}), which count the number of edges pointing in towards and out from a vertex, respectively.

#2.31
```
> degree(dg, mode="in")
 Sam Mary  Tom
   0    2    2
> degree(dg, mode="out")
 Sam Mary  Tom
   2    1    1
```

It is also useful to be able to discuss the concept of movement about a graph. For example, a *walk* on a graph G, from v_0 to v_l, is an alternating sequence $\{v_0, e_1, v_1, e_2, \ldots, v_{l-1}, e_l, v_l\}$, where the endpoints of e_i are $\{v_{i-1}, v_i\}$. The *length* of this walk is said to be l. Refinements of a walk include *trails*, which are walks without repeated edges, and *paths*, which are trails without repeated vertices. A trail for which the beginning and ending vertices are the same is called a *circuit*. Similarly, a walk of length at least three, for which the beginning and ending vertices are the same, but for which all other vertices are distinct from each other, is called a *cycle*. Graphs containing no cycles are called *acyclic*. In a digraph, these notions generalize naturally. For example, a *directed walk* from v_0 to v_l proceeds from tail to head along arcs between v_0 and v_l.

A vertex v in a graph G is said to be *reachable* from another vertex u if there exists a walk from u to v. The graph G is said to be *connected* if every vertex is reachable from every other. A *component* of a graph is a maximally connected subgraph. That is, it is a connected subgraph of G for which the addition of any other remaining vertex in V would ruin the property of connectivity. The toy graph `g`, for example, is connected

#2.32
```
> is.connected(g)
[1] TRUE
```

and therefore consists of only a single component

#2.33
```
> clusters(g)
$membership
[1] 1 1 1 1 1 1 1

$csize
[1] 7

$no
[1] 1
```

For a digraph, there are two variations of the concept of connectedness. A digraph G is *weakly connected* if its underlying graph (i.e., the result of stripping away the labels ‘tail’ and ‘head’ from G) is connected. It is called *strongly connected* if every vertex v is reachable from every u by a directed walk. The toy graph `dg`, for example, is weakly connected but not strongly connected.

```
#2.34 > is.connected(dg, mode="weak")
      [1] TRUE
      > is.connected(dg, mode="strong")
      [1] FALSE
```

A common notion of *distance* between vertices on a graph is defined as the length of the shortest path(s) between the vertices (which we set equal to infinity if no such path exists). This distance is often referred to as *geodesic distance*, with ‘geodesic’ being another name for shortest paths. The value of the longest distance in a graph is called the *diameter* of the graph. Our toy graph `g` has diameter

```
#2.35 > diameter(g, weights=NA)
      [1] 3
```

Ultimately, the concepts above are only the most basic of graph-theoretic quantities. There are a wide variety of queries one might make about graphs and quantities to calculate as a part of doing descriptive network analysis. We cover more of these in Chap. 4.

2.4.2 Special Types of Graphs

Graphs come in all ‘shapes and sizes,’ as it were, but there are a number of families of graphs that are commonly encountered in practice. We illustrate this notion with the examples of four such families shown in Fig. 2.2.

```
#2.36 > g.full <- graph.full(7)
      > g.ring <- graph.ring(7)
      > g.tree <- graph.tree(7, children=2, mode="undirected")
      > g.star <- graph.star(7, mode="undirected")
      > par(mfrow=c(2, 2))
      > plot(g.full)
      > plot(g.ring)
      > plot(g.tree)
      > plot(g.star)
```

A *complete* graph is a graph where every vertex is joined to every other vertex by an edge. This concept is perhaps most useful in practice through its role in defining a *clique*, which is a complete subgraph. Shown in Fig. 2.2 is a complete graph of order $N_v = 7$, meaning that each vertex is connected to all of the other six vertices.

A *regular* graph is a graph in which every vertex has the same degree. A regular graph with common degree d is called *d-regular.* An example of a 2-regular graph is the ring shown in Fig. 2.2. The standard (infinite) lattice, such as is associated visually with a checker board, is an example of a 4-regular graph.

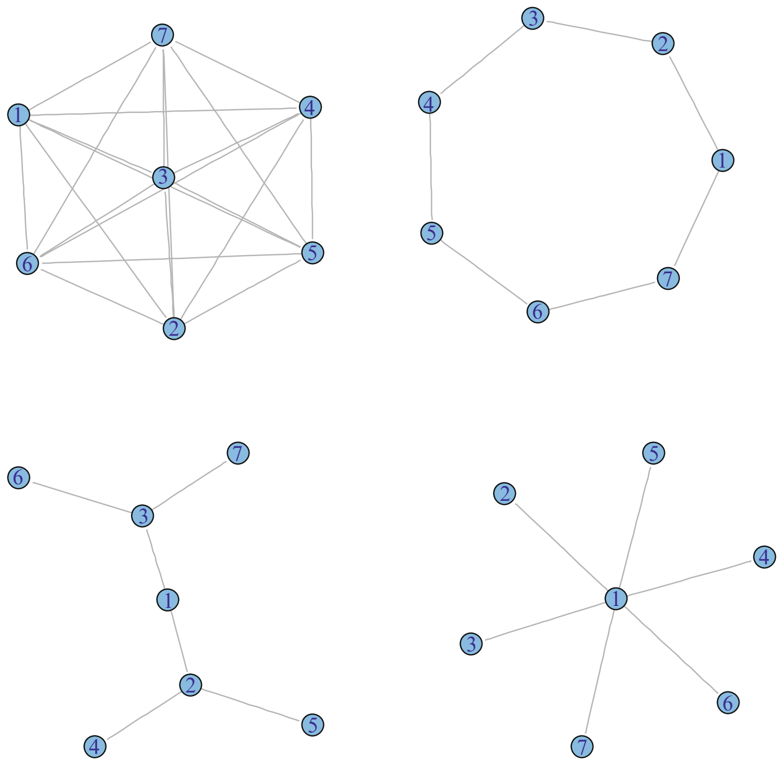

Fig. 2.2 Examples of graphs from four families. Complete (*top left*); ring (*top right*); tree (*bottom left*); and star (*bottom right*)

A connected graph with no cycles is called a *tree*. The disjoint union of such graphs is called a *forest*. Trees are of fundamental importance in the analysis of networks. They serve, for example, as a key data structure in the efficient design of many computational algorithms. A digraph whose underlying graph is a tree is called a *directed tree*. Often such trees have associated with them a special vertex called a *root*, which is distinguished by being the only vertex from which there is a directed path to every other vertex in the graph. Such a graph is called a *rooted tree*. A vertex preceding another vertex on a path from the root is called an *ancestor*, while a vertex following another vertex is called a *descendant*. Immediate ancestors are called *parents*, and immediate descendants, *children*. A vertex without any children is called a *leaf*. The distance from the root to the farthest leaf is called the *depth* of the tree.

Given a rooted tree of this sort, it is not uncommon to represent it diagrammatically without any indication of its directedness, as this is to be understood from the definition of the root. Such a representation of a tree is shown in Fig. 2.2. Treating vertex 1 as the root, this is a tree of depth 2, wherein each vertex (excluding the leafs) is the ancestor of two descendants.

A *k-star* is a special case of a tree, consisting only of one root and k leaves. Such graphs are useful for conceptualizing a vertex and its immediate neighbors (ignoring any connectivity among the neighbors). A representation of a 6-star is given in Fig. 2.2.

An important generalization of the concept of a tree is that of a *directed acyclic graph* (i.e., the DAG). A DAG, as its name implies, is a graph that is directed and that has no directed cycles. However, unlike a directed tree, its underlying graph is not necessarily a tree, in that replacing the arcs with undirected edges may leave a graph that contains cycles. Our toy graph `dg`, for example, is directed but not a DAG

```
#2.37
> is.dag(dg)
[1] FALSE
```

since it contains a mutual edge, hence a 2-cycle. Nevertheless, it is often possible to still design efficient computational algorithms on DAGs that take advantage of this near-tree-like structure.

Lastly, a *bipartite* graph is a graph $G = (V,E)$ such that the vertex set V may be partitioned into two disjoint sets, say V_1 and V_2, and each edge in E has one endpoint in V_1 and the other in V_2. Such graphs typically are used to represent 'membership' networks, for example, with 'members' denoted by vertices in V_1, and the corresponding 'organizations', by vertices in V_2. For example, they are popular in studying the relationship between actors and movies, where actors and movies play the roles of members and organizations, respectively.

```
#2.38
> g.bip <- graph.formula(actor1:actor2:actor3,
+   movie1:movie2, actor1:actor2 - movie1,
+   actor2:actor3 - movie2)
> V(g.bip)$type <- grepl("^movie", V(g.bip)$name)
> str(g.bip, v=T)
IGRAPH UN-B 5 4 --
+ attr: name (v/c), type (v/l)
+ vertex attributes:
      name  type
[1] actor1 FALSE
[2] actor2 FALSE
[3] actor3 FALSE
[4] movie1  TRUE
[5] movie2  TRUE
+ edges (vertex names):
[1] actor1--movie1 actor2--movie1 actor2--movie2
[4] actor3--movie2
```

A visualization of `g.bip` is shown[7] in Fig. 2.3.

It is not uncommon to accompany a bipartite graph with at least one of two possible induced graphs. Specifically, a graph $G_1 = (V_1,E_1)$ may be defined on the vertex set V_1 by assigning an edge to any pair of vertices that both have edges in E to at least one common vertex in V_2. Similarly, a graph G_2 may be defined on V_2.

[7] The R code for generating this visualization is provided in Chap. 3.

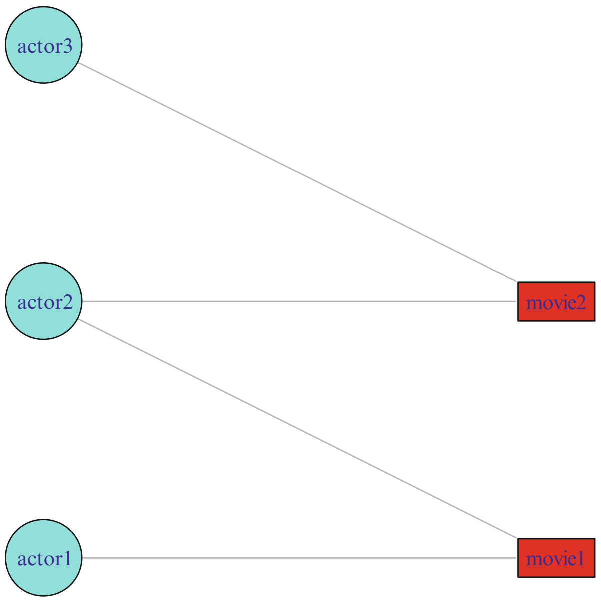

Fig. 2.3 A bipartite network

Each of these graphs is called a *projection* onto its corresponding vertex subset. For example, the projection of the actor-movie network `g.bip` onto its two vertex subsets yields

```
#2.39
> proj <- bipartite.projection(g.bip)
> str(proj[[1]])
IGRAPH UNW- 3 2 --
+ attr: name (v/c), weight (e/n)
+ edges (vertex names):
[1] actor1--actor2 actor2--actor3
> str(proj[[2]])
IGRAPH UNW- 2 1 --
+ attr: name (v/c), weight (e/n)
+ edges (vertex names):
[1] movie1--movie2
```

Within the actor network, `actor2` is adjacent to both `actor1` and `actor3`, as the former actor was in movies with each of the latter actors, although these latter were not themselves in any movies together, and hence do not share an edge. The movie network consists simply of a single edge defined by `movie1` and `movie2`, since these movies had actors in common.

2.5 Additional Reading

A more thorough introduction to the topic of graph theory may be found in any of a number of introductory textbooks, such as those by Bollobás [15], Diestel [47], or Gross and Yellen [67]. Details on graph data structures and algorithms are in many computer science algorithms texts. See the text by Cormen, Leiserson, Rivest, and Stein [35], for example.