Learning Outcomes

1. Store and clean network and attribute data in a spreadsheet
2. Import network and attribute data in R, so they can be used by R packages for social network analysis
3. Apply elementary transformations to network data and reconfigure attribute data inside R
4. Export data and results for use in other programs and documents

5.1 Introduction

In this chapter we discuss how to format network and attribute data in a spreadsheet editor, such as Microsoft Excel, so they can subsequently be imported into the free software environment R, and so they are suitable for social network analysis with R. While the core R program contains functions that can perform basic management and analysis of data, such as matrix multiplication or regression analysis, additional functions written by users and distributed in the form of add-on R packages are needed for network analysis. Functions tailored for network data can be found in specific R packages such as *xUCINET*, *igraph*, *statnet* and *RSiena*, which require data to be stored in a specific way; and this means that we will need to perform a series of steps and checks to be able to analyze our data. Figure 5.1 provides an overview of the main steps discussed in this chapter.

We start the chapter with a basic introduction to R. We then move on to discuss the various formats that are available for storing network data using a spreadsheet editor (as comma separated value (CSV) files), including suggestions for checking and cleaning the stored data. We subsequently discuss how to import data into R as a *matrix* object, as well as a way

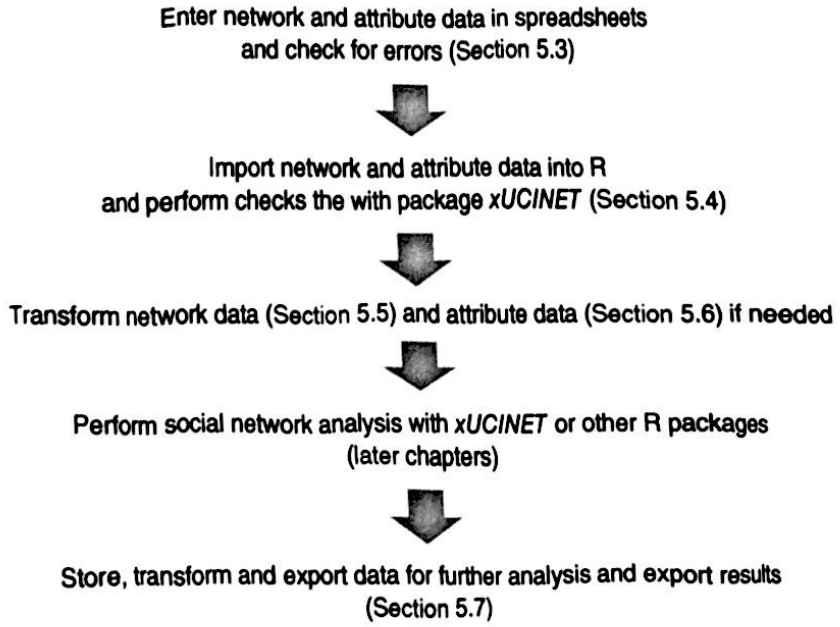

Figure 5.1 Overview of steps discussed in this chapter.

to merge and store all network and attribute data into one single object which can then be used for specific network packages in R, such as *xUCINET*, *statnet* or *RSiena*. Once the data are read in, the section on data transformation focuses on how common adjustments can be made to network data, such as dichotomizing, symmetrizing and aggregating network data, as well as discussing how to deal with missing data. This is followed by a discussion on possible transformations of attribute data. At the end of this chapter, we also show how to export network data and results for use in other programs and documents.

5.2 The R program

Before explaining how to prepare and import network data into R, for those unfamiliar with the R program, we briefly discuss some basic information about the program. The R program is a software environment that can be freely downloaded from its website. However, we will actually make use of RStudio, which is a popular graphical user interface to run R and allows more flexibility than the simple version of the R program. You can download RStudio from https://rstudio.com/products/rstudio/download/.

R is a widely used programming language that allows the user to manipulate data and perform a wide variety of (statistical) analyses. It is not specific to social network analysis, and in fact, to perform social network analysis, we need to download extra scripts, which have been collected into packages (such as the packages *xUCINET*, *igraph*, *statnet* and *RSiena*). We will explain later how to download these.

At this point we focus on R itself. Figure 5.2 shows a screenshot of RStudio. The command prompt > in the left window (the console window) is where you can specify what you want to do. To understand the R programming language, it is important to know that R has two important types of components – objects and functions – which we discuss in turn.

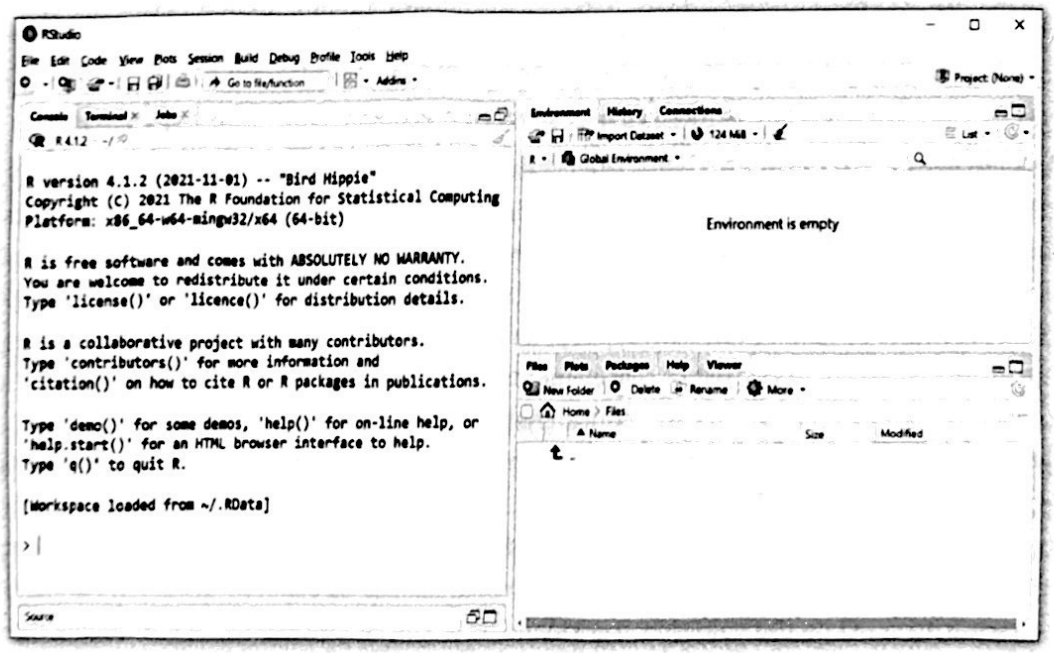

Figure 5.2 Screenshot of RStudio, a trademark of RStudio, PBC.

5.2.1 Objects

Objects are ways of (temporarily) storing information inside the workspace of R so that these data can be used in analyses and/or can be exported to a file. Objects can represent individual numbers, variables such as respondents' ages, whole matrices, and many other things. You can think of these objects as boxes with a name. To create or change an object, we use the assignment operator `<-`. It creates or changes the object on the left of the assignment operator, based on whatever is on the right of the operator. For example, if you type `objectA<-3.1` at the prompt and press the `Enter` key (also called the return key, and may be indicated on your keyboard as '↵'), the value 3.1 is now stored in the object called `objectA` inside R. You can view the contents of `objectA` by simply typing its name and pressing `Enter`. You will also notice that in the top right-hand window in R (under 'Environment'), the object will have appeared, as will any other objects you subsequently add. Note that the object is temporarily stored inside the current R session, but you will need to save it if you want to use it after you close RStudio.

At this stage we introduce a more convenient way of entering R commands, so that you can collect them together into scripts for future use. If, in RStudio, you go to **`File>New File>R Script`**, you will notice that a new window appears, which you can use as a draft place to write script and from where you can run specific commands by putting the cursor on that line of code and simultaneously pressing `Control` and `Enter`. For example, if you write `objectA*2` in the new window and press `Control+Enter` you will see that it appears in the console window and that it returned the answer 6.2, that is, the value for `objectA` - stored in the previous step - multiplied by 2. The advantage of this approach is that you can work in the R Script window and save this script for later use.

Returning to dealing with objects, note that objects can also hold strings (character) or Boolean values (i.e., TRUE, FALSE). For example, `objectB<-"I am an R object"` will save the statement in the object `objectB`. Note that objects are case sensitive, and so **`objectB`** is not the same as **`ObjectB`** or **`OBJECTB`**. Names of objects should not include any special characters (such as '*', '$', '-', '?', '&', '^', '%', '#', '/', '\', or any type of brackets) and also cannot have spaces. Only '.' and '_' are permitted. For example, `data.new`, `data_new` and `DataNew` are three common ways to create easily understandable names for objects. Objects should start with a letter. Finally, you should also avoid any names (such as 'sum') that are reserved for R functions (see below).

An object can also hold more than a single value (i.e., it can be a box with a series of compartments or cells - called elements), such as a vector with a series of elements, a two-dimensional matrix, a three-dimensional array, an object with a list of elements, or a data frame. Each of these data types can hold numeric values, characters (strings) or Boolean values, although there are limits in mixing different types of data. When typing `objectC<-c(3, 4, 1, 1, 2)`, the vector created will consist of five elements with respectively values 3, 4, 1, 1 and 2. The 'c' in this line of code stands for 'concatenate' and is needed to indicate that you want a set of elements to be saved in the same object in that order. You can check this by asking for the object `objectC`. You can also ask for elements of a vector using square brackets; for example, `objectC[2]` means that you want to retrieve the second element (value 4) from the object. Or you can ask for a range of elements, such as `objectC[2:4]`, which means you want to retrieve the values in the second to fourth cells of the object (where ':' means 'from ... to ...'). Be aware that any object can be overwritten, so that if you subsequently run `objectA<-c("OneMode","Directed")`, the object is now replaced with a set of two non-numeric elements (a vector of length 2), and the old value will be gone.

5.2.2 Functions

Functions are sections of code – whether written by the R Foundation or by others, including the user – for doing a set of tasks, such as measuring the centrality of each node in a network. We have already encountered one standard function: `c(…)`. Many more-specialized functions have been written by contributors and compiled in packages. These can be downloaded as needed.

A function is identified by a string name followed by parentheses, where the function performs an operation based on the arguments specified inside '(...)'. For example, if you type `abs(-2.3)`, this will return the absolute value of -2.3, which is 2.3. The operation can be performed on multiple elements at the same time (when organized in a vector or matrix). For example, `abs(c(2,-4,-5))` will result in the set of values 2, 4 and 5. A function might have extra arguments. For example, `mean(c(2,4,NA,5,NA))` will calculate the average of the five values. In this case, however, it will give a missing value (which in R is represented as 'NA' – 'not available'), since some of the values in the vector are missing (i.e., 'NA'). However, for the function `mean()` you can add an extra argument to ignore missing values '`na.rm=TRUE`', and hence ask to calculate the average based on the non-missing values only. When using `mean(c(2,4,NA,5,NA), na.rm=TRUE)`, the missing values are ignored, and the result will be 3.667. Using the earlier defined vector, you can also calculate the average of a vector by using the command `mean(objectC)`, or you can multiply each element of the vector by 2, as in `objectC*2`. You can also save this output in a new object `average_objectC<-mean(objectC)`.

Note that R is case sensitive when it comes to functions just as it is with objects, and so `mean()` is different from `Mean()` or `MEAN()`; the last two will actually give an error, as no such functions exist. Finally, it is important to know that there are also a number of arithmetic operators (such as '*' and '^') and logical operators (such as '==', '>=' and '!='). More information on R functions and operations is provided on the website and via the exercises.

5.2.3 Different classes of objects: *matrix*, *data.frame* and *list* objects

Now that we have introduced the notion of functions, we return to ways to store data. Objects in R belong to a specific 'class', which defines how data are organized in objects and what you can do with the object. We discuss three classes of objects here: the *matrix* class, the *data.frame* class and the *list* class.

Matrix class

Matrix class objects can be used to store any kind of matrix data, including adjacency matrices for networks. Using the function `matrix()`, `objectD<-matrix (c(-999,2,1,6,-999,5,4,2,-999),3,3)` creates a *matrix* class object with dimensions 3 (rows) by 3 (columns), as illustrated in Figure 5.3 (left). In the command, the first value (-999) is stored in the cell in the first row and first column, the second value (2) is placed in the cell in the second row and first column, the third value (1) is placed in the third row and first column, the fourth value (6) is placed in the cell in the first row and second column, and so on. Because it has been identified as a *matrix* class object in R, we will be able to perform matrix operations, such as matrix multiplication (`%*%`), with the object, which we cannot do

with other types of objects, such as *data.frame* class objects. To see the content of an object, you could call the object. However, for larger matrices it might be better to use the function `View()`, which will open a spreadsheet in RStudio. You can also ask to select a subset of the matrix by identifying the specific rows and columns. For example, `objectD[2,2:3]` will give the values for the cells in the second row, second and third column (values -999 and 2). Note that the first number in the squared brackets after the matrix refers to the row(s), while the second refers to the column(s). Since we identified a single row, the result is a vector rather than a matrix.

```
> objectD
     [,1] [,2] [,3]
[1,] -999    6    4
[2,]    2 -999    2
[3,]    1    5 -999
```

```
> objectD
      John  Mary  Leo
John  -999     6    4
Mary     2  -999    2
Leo      1     5 -999
```

```
> objectD
      John  Mary  Leo
John    NA     6    4
Mary     2    NA    2
Leo      1     5   NA
```

Figure 5.3 Illustrations of a *matrix* object without row and column names (left), with row and column names (middle) and with missing data (–999) changed to NA (right).

When dealing with network data, we might want to change the row names of the matrix to reflect the nodes' names using the function `rownames(objectD)<-c("John","Mary","Leo")` and the column names using the function `colnames(objectD)<-c("John","Mary","Leo")`. The result of this change is presented in Figure 5.3 (middle). Note that this does not change the way you can retrieve elements from a matrix, which still requires square brackets with row and column numbers. Note also that in this example we used the value –999 to indicate missing values on the diagonal. We could also recode the missing values (–999) as 'NA', the default indicator for missing data in R using the function `objectD[objectD == -999]<-NA`, as in Figure 5.3 (right). Doing this identifies a subset of elements in **objectD** (those with value –999) and assigns a value 'NA' to these cells. Hence, it is a simple way to replace those cells that contain a specific value or string with another value or string. Standard matrix manipulation can be applied to *matrix* objects, as long as they contain only numerical values or logical values 'TRUE' and 'FALSE'. For example, we could ask for the mean of the values `mean(objectD, na.rm=TRUE)`, which results in a value of 3.33. However, when we create a matrix where at least one element is a character, then all elements will be transformed into characters, as is the case with `objectE<-matrix(c("man", "woman","man",24,31,27),3,2)`. As the resulting output in Figure 5.4 (left) shows, all elements are put in between quotes. This is because all elements of a matrix need to be of the same type, and since it is not always possible to transform string elements into numeric values, numeric values are changed into string values. We can no longer obtain a mean for this matrix, or part of it, even if we only focus on the second column, as in `mean(objectE[,2])` which will result in a warning and 'NA'. When some columns contain strings while others contain numeric values, a *data.frame* is more appropriate.

Data.frame class

A *data.frame* object is like a dataset in a statistical package. The columns represent variables, and the rows are cases. To obtain a *data.frame* object corresponding to the matrix in Figure 5.4 (left

we use the following command: `objectF<-data.frame(Gender=c("man","woman","man"), Age=c(24,31,27))`. Considering the outcome in Figure 5.4 (middle), the data seem to be saved in a similar way to a matrix, except for the quotes. However, in a *data.frame* object the columns are considered a separate variable, which are labeled as `Gender` and `Age`, and the quotes have disappeared. To extract a single variable from a *data.frame*, such as the gender of each case, we can simply use '$': `objectF$Gender`. In a *data.frame* some variables might be strings, while others might be numeric or logical, and calculations can be performed on the numerical variables, such as `mean(objectF$Age)`.

```
> objectE
     [,1]    [,2]
[1,] "man"   "24"
[2,] "woman" "31"
[3,] "man"   "27"
```

```
> objectF
  Gender Age
1    man  24
2  woman  31
3    man  27
```

```
> objectF
      Gender Age
John     man  24
Mary   woman  31
Leo      man  27
```

Figure 5.4 Illustrations of a *matrix* object with character and numeric elements (left), and its corresponding *data.frame* object, without row names (middle) and with row names (right).

We could further improve the presentation of a *data.frame* by changing the row names to the names of the people about whom we collected such information `rownames(objectF)<-c("John","Mary","Leo")`. The result is presented in Figure 5.4 (right).

List class

The most flexible of the three types of objects is the *list* object. To create a *list* object, we can use the command `objectG<-list(objectA,objectD,objectF)`, as in Figure 5.5 (left). The different objects are now stored as separate elements [[1]], [[2]], ... and we can extract an element of a list by using `objectG[[3]]`, and use `objectG[[3]]$Gender` to extract the first variable from the *data.frame* (third element of `objectG`). This is an example of a way to store not only a network, but also additional information and attribute data in one single *list* object.

```
> objectG
[[1]]
[1] "OneMode"  "Directed"

[[2]]
     John  Mary  Leo
John   NA     6    4
Mary    2    NA    2
Leo     1     5   NA

[[3]]
     Gender Age
John    man  24
Mary  woman  31
Leo     man  27
```

```
> objectG
$Info
[1] "OneMode"  "Directed"

$Network
     John  Mary  Leo
John   NA     6    4
Mary    2    NA    2
Leo     1     5   NA

$Attributes
     Gender Age
John    man  24
Mary  woman  31
Leo     man  27
```

Figure 5.5 Illustrations of a *list* object, without and with element names.

The numeric way of naming elements of the list is not always convenient. Instead, we can also add names to the list elements by using the command `names(objectG)<-c("Info","Network","Attributes")`, which generates the object with unique names for the elements of the list, as in Figure 5.5 (right). As a result, instead of element `[[3]]` we can now call this using the name `Attributes`. To retrieve the `Gender` variable again, we could use `objectG$Attributes$Gender`.

Practice 5.1 Go to the website and check out

1. The results for all the scripts we have run so far in this chapter
2. Some extra exercises with additional R functions and operations
3. Examples of how to create *matrix*, *data.frame*, and *list* objects, and simple functions that could be useful

5.3 Data storage

One of the most important steps in any network analysis is formatting the data so that they can be imported into a software program such as R. Regardless of how data are obtained, they eventually take the form of an electronic file, such as a spreadsheet, database, or text file. For large datasets, it might be useful to use a database system such as Microsoft Access or MySQL, but in most cases, we recommend using CSV files. These are easily and reliably saved and loaded by both R and spreadsheet editors, such as Microsoft Excel, which provide a convenient interface for transforming the data.

It is generally advisable to store each network as a separate CSV file (or at least to store them in different sheets of a spreadsheet file), and to have an additional file for attribute data. Regarding the naming of files, it is good practice to add the name of the project before the name of the networks/attribute file, so there is no confusion later on about which files are part of which data project (e.g., when you end up collecting the friendship network for multiple projects). If the dataset comes from a specific source, e.g., from a specific author or organization, the name of this source could be added in front of the network name, so that we get, for example, 'Padgett_FlorentineFamilies_Business_AM.csv', where 'Business' is the network, 'FlorentineFamilies' is the project, and 'Padgett' is the source. In addition 'AM' might be added to indicate that the file contains an adjacency matrix (see below).

In general, network data tend to be organized in one of three common formats within a file: the adjacency matrix, the edgelist, or the nodelist format. In addition, attribute data have their own specific format.

5.3.1 Adjacency matrix format

The adjacency matrix format is a conceptually straightforward data format. Figures 5.6 and 5.7 show screenshots of two networks in adjacency matrix format, which were held in two

different spreadsheet files. Note that the first row and column are used for the node labels. As is evident from the labels in both networks, they record different network relations ('flatmates' and 'liking') among the same six people. They are both one-mode datasets, as the row and column entities are the same. The first network is binary and undirected and captures whether two people shared an apartment at some point. The second network is valued (weighted) and directed and indicates the extent to which each person claims to be a friend with the other person (0 = not a friend, 1 = friend, 2 = good friend, and 3 = dating). As noted before, the convention in social network analysis is that the row node 'sends to' the column node. This means that in the liking network shown in Figure 5.7, Chandler indicates he is a good friend of Joey (value 2), while Joey says he is only a friend of Chandler (value 1). By convention, we generally also ignore self-ties and therefore will use zeros down the main diagonal.

	A	B	C	D	E	F	G	H
1	ID	Chandler	Joey	Monica	Phoebe	Rachel	Ross	
2	Chandler	0	1	1	0	0	0	
3	Joey	1	0	0	0	1	0	
4	Monica	1	0	0	1	1	0	
5	Phoebe	0	0	1	0	1	0	
6	Rachel	0	1	1	1	0	1	
7	Ross	0	0	0	0	1	0	
8								

Figure 5.6 Screenshot of a binary and undirected one-mode network of six people, capturing which ones had ever been flatmates (1 = yes), which was stored as an adjacency matrix.

	A	B	C	D	E	F	G	H
1	ID	Chandler	Joey	Monica	Phoebe	Rachel	Ross	
2	Chandler	0	2	3	1	0	0	
3	Joey	1	0	1	1	2	0	
4	Monica	3	0	0	0	1	2	
5	Phoebe	1	0	0	0	0	0	
6	Rachel	1	2	1	1	0	3	
7	Ross	1	0	2	0	3	0	
8								

Figure 5.7 Screenshot of a valued and directed 'liking' network of six people (0 = not a friend, 1 = friend, 2 = good friend, 3 = dating), which was stored as an adjacency matrix.

This type of format is also applicable to two-mode data, in which case the column names (labels on the top) and the row names (labels on the left) will not be the same, and the adjacency matrix is typically not square. Figure 5.8 shows a two-mode network that captures which types of food the six people like.

	A	B	C	D	E	F	G	H
1	ID	Asian	Greek	Indian	Italian	Mexican		
2	Chandler	0	0	0	1	1		
3	Joey	0	0	0	1	0		
4	Monica	0	1	0	0	0		
5	Phoebe	1	0	1	0	1		
6	Rachel	0	0	0	0	1		
7	Ross	0	1	0	0	0		
8								

Figure 5.8 Screenshot of a binary two-mode network capturing the favorite types of food of six people, which was stored as an adjacency matrix.

Checking your adjacency matrix for errors

Before importing the CSV dataset into R, it is advisable to examine it in some detail. There are a number of potential problems that can arise, and if these are detected early, you can avoid wasting time performing analyses on the flawed data. An overview of potential issues for data in adjacency matrix format is given in Table 5.1.

Table 5.1 Checklist of potential issues with the adjacency matrix format.

	Adjacency matrix
Format	If one-mode data were collected, is the adjacency matrix a square matrix?
Missing values	Are missing nodes included in the adjacency matrix, and are they clearly distinguishable from nodes that are isolates? Are missing values correctly identified (e.g., as '-999' or 'NA')? Do not use empty cells to indicate missing values.
Binary	Are there values other than 0, 1, and potentially the missing-value indicator? Do all values make sense?
Directed	If undirected, is the matrix symmetrical?
Diagonal	For one-mode data, are self/nominations relevant, and is this reflected in the diagonal?
Labels	Are the names used unique, and do they avoid any special characters? For one-mode networks, are the names in the rows and columns the same?

Missing network information could be caused by non-response or an error in entering or copying the data. The non-response issue is a serious problem, and we will return later to discuss how to deal with the presence of missing data. However, it is important to make sure that missing data are the result of non-response and not a mistake in data input (or simply someone who did not nominate any others because they are an isolate). If missing data are the result of non-response, you need to include them in the dataset and to identify them as such (e.g., by using '–999' or 'NA') in their respective cells. Avoid empty

cells in the matrix format, as they might be skipped when the data are read in. In your spreadsheet editor, you can easily replace all blank cells with 'NA' or a unique code (such as '–999') before importing the dataset into R.

In addition, it is always worth thinking about whether the data should logically have certain characteristics, and then checking that those are in fact present. For example, if your matrix data are supposed to be binary, finding a value other than 0 or 1 is a red flag. Similarly, many relations, such as 'had lunch with' or 'talked with', are supposed to be symmetric, so you should make sure that this is true. If it is not, you need to determine whether this is due to some mistake in the process of entering the data, whether it simply reflects what the respondents said (due to recall problems or problems with the way the questionnaire was worded), or whether your network really is symmetrical in nature. For one-mode networks, you should also reflect on whether self-nominations are relevant, in which case attention needs to be paid to the values on the diagonal.

A third issue relates to the labels (names) used for nodes. It is possible not to use any labels at all when storing your data. However, in general, it is advisable to include names for nodes, especially when you have multiple network relations and/or attribute files, so that you can check that the nodes are in the same order across all of the matrices, as well as in the attribute file. For example, if you compare the two matrices in Figures 5.6 and 5.7, you will notice that the order of names is the same, as is the order of the row names in the two-mode network in Figure 5.8. This also implies that if a node is missing in one of the networks, you should position it in the same place and add 'NA's to the data.

When using numeric labels for nodes, it is generally advisable to add a letter, for example *a1*, *a2*, ..., in order not to confuse the labels with data values. If you do add the names for nodes, it is generally not advisable to use long names, as this can make network diagrams and data displays difficult to read. You can also keep a spreadsheet that pairs short names with the original long labels, so that this information is not lost. It is also not advisable to use special characters, as some programs may misinterpret them, particularly when importing or exporting data. A few special characters, such as '_', are generally safe. In some cases, spaces inside a label like 'Chandler Bing' can be a problem and might end up being read in as two different cells. Instead, it is safer to use a single character string such as 'Chandler_Bing'. When using labels, make sure you always spell the names *exactly the same* way in all data files. Remember that R is case-sensitive and will regard 'Chandler' as a different label from 'chandler' or 'CHANDLER'. Therefore, be sure to use the same upper/lower cases each time. For one-mode networks, the names in rows and columns should also be the same and be in the same order.

Another thing to make sure of, particularly in larger studies, is that each node is given a unique name. Hence, one kind of problem to look for is repeated names. This might mean that the same actor has been accidentally entered multiple times in different rows (and/or columns) or has answered the survey twice. This can be more difficult to detect when there are slight differences in how the same node's name was typed. However, it could also be that two different nodes were given the same names (e.g., two people named John), which you could easily correct by adding a number, such as 'John1' and 'John2'. As discussed in the next section, when reading in network data in R with *xUCINET*, the software will check

many of the issues discussed above, provide a warning, and automatically add names or make them unique if needed.

Adjacency matrices are most useful when the network is small and dense. If the network is very sparse, the adjacency matrix will consist mostly of zeros, in which case it is more economical to use a format where non-ties are not entered, such as in the edgelist format. Many valued datasets, such as one giving the physical distances between pairs of nodes, are completely dense, as there is a value for every pair of nodes. For these kinds of data, matrix formats are the best choice.

5.3.2 Edgelist format

The edgelist format consists of a set of rows in which each row represents a tie in the network (see Figures 5.9 to 5.11). When network data are stored in databases, they are frequently organized in this way (recall the IMDb example in Chapter 4), as this allows the table fields (columns) to be the same for every dataset. For directed networks, the first column will indicate the sender, while the second column indicates the receiver (see Figure 5.9). Optionally, a third column can be included which gives the strength of the tie. The screenshot of the spreadsheet in Figure 5.10 represents the directed 'liking' network (the same data as stored in the adjacency matrix in Figure 5.7). Since the 'liking' network is valued, this edgelist has a third column for the strength of the tie.

Note that in a directed network, if nodes *i* and *j* both choose each other, we must enter the pair twice: first as (*i*, *j*) and then as (*j*, *i*). For undirected networks, we only need to enter the pair once. The undirected 'flatmates' network (Figure 5.9) illustrates this. For example, the fact that Chandler was flatmates with Joey also implies that Joey was flatmates with Chandler. Figure 5.11 also shows the edgelist format for the two-mode network capturing food preferences. For a two-mode network, it is important that the first column represents one mode (persons) and the second column represents the other mode (foods).

	A	B	C	D
1	Chandler	Joey		
2	Chandler	Monica		
3	Joey	Rachel		
4	Monica	Phoebe		
5	Monica	Rachel		
6	Phoebe	Rachel		
7	Rachel	Ross		

Figure 5.9 Screenshot of the binary and undirected one-mode 'flatmates' network stored as an edgelist.

sender receiver strength

	A	B	C	D
1	Chandler	Joey	2	
2	Chandler	Monica	3	
3	Chandler	Phoebe	1	
4	Joey	Chandler	1	
5	Joey	Monica	1	
6	Joey	Phoebe	1	
7	Joey	Rachel	2	
8	Monica	Chandler	3	
9	Monica	Rachel	1	
10	Monica	Ross	2	
11	Phoebe	Chandler	1	
12	Rachel	Chandler	1	
13	Rachel	Joey	2	
14	Rachel	Monica	1	
15	Rachel	Phoebe	1	
16	Rachel	Ross	3	
17	Ross	Chandler	1	
18	Ross	Monica	2	
19	Ross	Rachel	3	
20				
21				
22				
23				
24				
25				

Figure 5.10 Screenshot of the valued and directed one-mode 'liking' network stored as an edgelist.

Person Food.

	A	B	C	D
1	Chandler	Italian		
2	Chandler	Mexican		
3	Joey	Italian		
4	Monica	Greek		
5	Phoebe	Asian		
6	Phoebe	Indian		
7	Phoebe	Mexican		
8	Rachel	Mexican		
9	Ross	Greek		
10				
11				

Figure 5.11 Screenshot of the binary two-mode 'favorite food' network stored as an edgelist.

Checking your edgelist for errors

Table 5.2 provides an overview of aspects to check. In any list format, the node identifiers can be numbers instead of names. Using numbers can be more economical than repeatedly typing in long names, and the labels for the nodes can be added later. If using names, it is important to ensure that they are short, unique, and written consistently, and contain no special characters. One should also take care that names are not misspelled, as this will be seen as two different nodes.

Table 5.2 Checklist of potential issues with the edgelist format.

	Edgelist
Format	The dataset must contain two or three columns.
Isolates	Are isolates excluded in the edgelist? If so, you might need to add them later.
Missing values	Do you have a third column where missing ties are indicated as 'NA' (or '-999')?
Binary	Is there a third column with values?
Directed	If undirected, do the data consist of unordered pairs (where, if (i, j) is included, (j, i) is not), or ordered pairs (in which both (i, j) and (j, i) are always included)?
Labels	Are the names used unique and short, and do they avoid any special characters? Are there any misspellings of names? Are there multiple instances of the same ordered pairs of nodes in the edgelist, which might indicate non-unique names? For two-mode data, are the names in the first column (one mode) distinct from those in the second column (the second mode)?

Another particular kind of problem to look for in this respect is multiple instances of the same ordered pairs of nodes in the list. This might indicate an error – for example, because multiple nodes have the same name, which is not allowed. Alternatively, the presence of multiple pairs might indicate tie strength. When the same pair of nodes is entered in different rows of the list, the program might consider this as multiple instances of a tie and might end up counting up the number of times each pair occurs. For example, when we have a list of instances of emails being sent from one person to another, we could be interested in the frequency with which i sends an email to j.

Another thing to remember when using the edgelist format is that isolates and missing nodes will not automatically appear in the data, since the data only record ties. One approach for one-mode networks is to add isolates to the raw data file by giving them a tie to themselves with value 0. The advantage of this is that there is a record of the node's existence in the very rawest form of the data. Alternatively, isolates can be inserted when importing the data to R. In addition, if you have any missing data, you will need to add rows for the dyads with missing values and put an 'NA' in the third column to distinguish an unknown tie from a definite non-tie.

Finally, as mentioned before for the adjacency matrix, you might want to check if the network is truly undirected. For ties in an undirected network, you can either enter each ordered pair just once in the edgelist and indicate later that it is undirected (when you read it in),

or you can include all ordered pairs in both directions. However, if some ordered pairs of nodes are present in both directions but others are not, this would generally indicate that the network is directed.

5.3.3 Nodelist format

The nodelist format is the most economical of the three formats discussed here. Figure 5.12 shows a screenshot of a spreadsheet with data in a nodelist format representing the 'flatmates' network. There are n rows of data, one for each node. The first name in each row gives the node that is 'sending' a tie – the ego. The names that follow in the same row are the nodes receiving each tie – the alters. Hence, the first row with 'Chandler', 'Joey', and 'Monica' states that there is a tie from 'Chandler' to 'Joey', and also a tie from 'Chandler' to 'Monica'. Note that if it is known that the network is undirected (as the flatmates relation is), it is unnecessary to add Chandler to Joey's and Monica's rows. Since this is an undirected network, it is understood that a tie exists from 'Joey' to 'Chandler', and from 'Monica' to 'Chandler', and we therefore do not have to record these separately. Note that a nodelist can accommodate isolates: we simply enter the isolate in the first column, with no additional columns filled in.

Nodelists are usually used for binary data. But if tie strengths are available (or missing data exist), they can be handled by saving the values in a second file that corresponds to the positions in the first network. For example, in Figure 5.13 we find the nodelist for the directed network capturing liking (as originally introduced in Figure 5.7). The nodelist in this spreadsheet only captures which alters each ego has ties with. In a second sheet (Figure 5.14), we find the corresponding values for each of the ties. Hence, Chandler likes Joey, Monica and Phoebe, and the respective values for the ties are 2, 3 and 1. Notice that even if liking nominations were reciprocated (i.e., i nominates j with a certain value and j nominates i with a certain value), the tie strengths need not be symmetric. For example, the tie from Chandler to Joey has value 2, but the value for the tie from Joey to Chandler is only 1.

Finally, Figure 5.15 shows the spreadsheet for a two-mode network in a nodelist format, where the first column represents one mode (e.g., the people) and the subsequent columns the links with the second mode (e.g., their choices of favorite food).

	A	B	C	D	E	F	G	H
1	Chandler	Joey	Monica					
2	Joey	Rachel						
3	Monica	Phoebe	Rachel					
4	Phoebe	Rachel						
5	Rachel	Ross						
6								

Figure 5.12 Screenshot of the binary and undirected one-mode 'flatmates' network stored as a nodelist.

	A	B	C	D	E	F	G	H
1	Chandler	Joey	Monica	Phoebe				
2	Joey	Chandler	Monica	Phoebe	Rachel			
3	Monica	Chandler	Rachel	Ross				
4	Phoebe	Chandler						
5	Rachel	Chandler	Joey	Monica	Phoebe	Ross		
6	Ross	Chandler	Monica	Rachel				
7								

Figure 5.13 Screenshot of the directed one-mode 'liking' network stored as a nodelist.

	A	B	C	D	E	F	G	H
1	Chandler	2	3	1				
2	Joey	1	1	1	2			
3	Monica	3	1	2				
4	Phoebe	1						
5	Rachel	1	2	1	1	3		
6	Ross	1	2	3				
7								

Figure 5.14 Screenshot of the tie strengths for the directed one-mode 'liking' network corresponding to the nodelist in Figure 5.13, with values 1 = friend, 2 = good friend, and 3 = dating.

	A	B	C	D	E	F	G	H
1	Chandler	Italian	Mexican					
2	Joey	Italian						
3	Monica	Greek						
4	Phoebe	Asian	Indian	Mexican				
5	Rachel	Mexican						
6	Ross	Greek						
8								

Figure 5.15 Screenshot of the binary two-mode 'favorite food' network stored as a nodelist.

Checking your nodelist for errors

Table 5.3 provides an overview of aspects to check. As with edgelists, it is important to ensure that the names are unique and written consistently and do not contain special characters. In the case of a one-mode network, each node occupies a separate row in the nodelist, and its name needs to be provided in the first column of that row. This means that each of

the names in the first column should be unique and that it should also include isolates and non-respondents. It also implies that all nodes that are nominations should also appear as nodes in the first column. If some nodes, who are nominated by others, do not appear in the first column, it might be because they were non-respondents. Or it could be a data entry error, for example, because of some issue with how the node's name was written. Missing data will need to be recorded in the second dataset, which holds the values for the ties, and therefore, for binary networks this requires adding a second CSV file, which specifies whether the tie is present ('1') or missing ('NA' or '-999').

Table 5.3 Checklist of potential issues with nodelist-formatted files.

	Nodelist
Format	The dataset will consist of two or more columns, although not all cells need to contain values.
Isolates	Are isolates included in the nodelist as a separate row?
Missing values	If you have missing data, you need to add these in a second dataset which has the same positions as the network dataset.
Binary	If you have values, they need to be added in a second dataset which has the same positions as the network dataset.
Directed	If undirected, either none of the pairs are reported in both directions (only one of both ordered pairs reported), or all are reported in both directions (both ordered pairs reported).
Labels	Are the names used unique and short, and do they avoid any special characters? Are there any misspellings of names? Are there multiple instances of the same ordered pairs of nodes in the nodelist, which might indicate non-unique names?

For two-mode networks it is important to ensure the first column contains one mode of nodes, and the subsequent columns only contain nominations of the other mode, and therefore in the two-mode case all names in the first column should be distinct from the names in subsequent columns. Isolates for the second mode should be added separately, although this is unlikely to take place as one mode normally nominates the other, and isolates of the second mode are generally not part of the analysis. If a one-mode network is assumed to be undirected, we might also want to check if there are some nominations from i to j that are also found in the reverse order, while others are not, as this might indicate that this is really a directed network.

5.3.4 Attribute data

Irrespective of the network format being used, the format for the attribute data will always be the same, namely, a matrix format where each column in the matrix is an attribute. For example, Figure 5.16 shows a screenshot of a spreadsheet with three attributes pertaining to the six

people in the earlier network. While the data can be either character string or numerical, it is usually less trouble if the attributes are numeric, in which case a codebook must be kept in order to know whether a '1' for 'Gender' means male, female, or something else.

In most cases, the first column should contain the names of the nodes, as these are needed to match the nodes in the network with the attribute file. Exceptions could be when a matrix format or nodelist is used as long as the order of the nodes is exactly the same in the attribute file and the matrix or nodelist.

	A	B	C	D	E	F	G
1	ID	Gender	Siblings	Occupation			
2	Chandler	Male	0	Data_analyst			
3	Joey	Male	7	Actor			
4	Monica	Female	1	Chef			
5	Phoebe	Female	1.5	Massage_therapist			
6	Rachel	Female	2	Waitress			
7	Ross	Male	1	Paleontologist			
8							

Figure 5.16 Screenshot of node attribute data.

Practice 5.2 Go to the website and check out

1 How to check a network dataset stored in adjacency matrix format and a corresponding attribute file for mistakes
2 How to check a network containing an edgelist for mistakes

5.4 Importing and storing data in R

Once the network dataset(s) have been saved as CSV file(s) in either adjacency matrix, edge list or nodelist format (and potential data-cleaning issues have been dealt with), we next need to import these as objects into R so we can perform network analysis on them using the specialized functions from the social network packages. As we discussed earlier in this chapter, data can be organized in different ways in R objects and this is reflected in the class of the object (e.g., *matrix*, *data.frame*, or *list*). The class of the object also defines what you can subsequently do with an object, that is, what specific R functions can be used on such data. An obvious solution would be to store network data as an adjacency matrix in an R object of class *matrix*. For most small to medium-sized networks it is easy to visually inspect a network in R when it is stored as a matrix object.

It is important to note, however, that some R packages for network analysis have their own unique way of storing a network, and therefore require network data to be organized

in their own tailored class in order for their functions to work on the data. For example, the *igraph* package requires network data to be organized as *igraph* objects. Moreover, while the *statnet* package can have *matrix* objects as input, it often uses objects of its own *network* class. Figure 5.17 provides an overview of the type of network formats and the ways they can be stored as R objects.

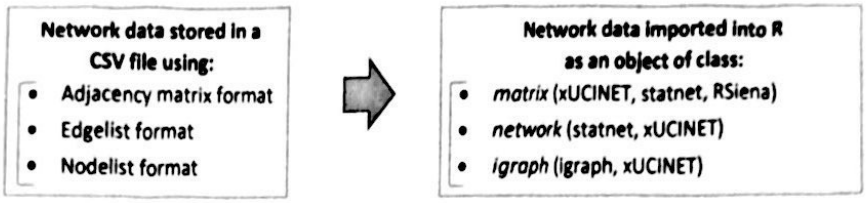

Figure 5.17 Overview of potential formats for storing single network data in a spreadsheet, as well as different classes of network objects in R needed for specific packages.

In this book we will focus on importing single networks as *matrix* class objects since such data are easy to inspect and the functions we will use in this book are all part of the *xUCINET*, *RSiena* or *statnet* packages, which are able to directly use network data stored as *matrix* objects (containing the network in an adjacency matrix format).

5.4.1 Importing network and attribute data as part of a data project

We could import each network and attribute file as separate R objects. However, because many projects involve multiple network relations and node attributes (potentially even including multiple modes), we recommend instead storing all network and attribute data for a specific project into a single object, so you can retrieve these later on in one go. This has the added benefit that it allows for extra checks on these data regarding consistency across networks (e.g., whether different networks and attribute data are consistent regarding the size and ordering of nodes), as well as allowing us to store extra information about the project and the different networks as part of the object. In practice this means that the multiple network and attribute datasets are compiled into a single 'meta'-object, that is, each of these different series of data become elements of a *list* object that contains all information about a single research project (similar to Figure 5.5).

Figure 5.18 provides a concrete example of how the overall structure of such a *list* object for a specific study is organized. In our example, the study was called 'CentralNYC'. The object for the CentralNYC project incorporates data about two one-mode networks (`$Flatmates` and `$Liking`), one two-mode network (`$Food`), and attribute data for the first mode (`$Attributes$A`). Inside this object, network data can be of class *matrix*, *network* or *igraph*, while attribute data are of class *data.frame*.

Besides network data and attribute data, additional information about each of the networks, as well as information about the project as a whole, is stored in additional elements. For example,

CentralNYC

$ProjectInfo

"This is a study of six people in a big city."

$Attributes$A

	Gender	Siblings	Occupation
Chandler	Male	0.0	Data_analyst
Joey	Male	7.0	Actor
Monica	Female	1.0	Chef
Phoebe	Female	1.5	Massage_therapist
Rachel	Female	2.0	Waitress
Ross	Male	1.0	Paleontologist

$NetworkInfo

	Name	Mode	Directed	Values	Loops
1	Flatmates	A	FALSE	Binary	FALSE
2	Liking	A	TRUE	Ordinal	FALSE
3	Food	AB	FALSE	Binary	FALSE

$Flatmates

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	1	1	0	0	0
Joey	1	0	0	0	1	0
Monica	1	0	0	1	1	0
Phoebe	0	0	1	0	1	0
Rachel	0	1	1	1	0	1
Ross	0	0	0	0	1	0

$Liking

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	2	3	1	0	0
Joey	1	0	1	1	2	0
Monica	3	0	0	0	1	2
Phoebe	1	0	0	0	0	0
Rachel	1	2	1	1	0	3
Ross	1	0	2	0	3	0

$Food

	Asian	Greek	Indian	Italian	Mexican
Chandler	0	0	0	1	1
Joey	0	0	0	1	0
Monica	0	1	0	0	0
Phoebe	1	0	1	0	1
Rachel	0	0	0	0	1
Ross	0	1	0	0	0

Figure 5.18 The CentralNYC data project as a *list* object, with network data stored as *matrix* objects.

we will know for a specific network whether to treat it as a directed or undirected netwo what type of values the network represents (e.g., binary, ordinal or nominal), and whether t networks are related to the same sets of nodes, or to different sets (e.g., when considering two-mode network).

There are a number of alternative functions that can be used to import network data into R. Here we will use the functions **`xCreateDataProject()`** and **`xAddToDataProject()`**, which are part of the *xUCINET* package. These functions are able to import network data which were saved in any of the three data formats discussed before (i.e., adjacency matrix, edgelist or nodelist format) and allow the network data to be organized into objects of any of the three classes (*matrix*, *network* or *igraph*) so that many of the major social network packages can be used (Figure 5.17). They also offer checks and warnings about the data when they get imported.

The easiest way to create a project is to import the network and attribute data in R one at a time. First, we need to set up the project object using **`xCreateDataProject()`** which will import a first network, and subsequently add additional network using **`xAddToDataProject()`**, specifying every time the specific information about the dataset that is being added to the existing project. By using a stepwise approach for compiling a project, it will be easier to figure out which (if any) of the files contains errors (e.g., whether it has the wrong size or whether the nodes are in the wrong order). In Practice 5.3, we discuss ways to import a single network into such a project object using the function **`xCreateDataProject()`**, and illustrate how to add additional network and attribute data to the overall project with the function **`xAddToDataProject()`**.

To use the functions **`xCreateDataProject()`** and **`xAddToDataProject()`** (and the functions later in the book) we will first need to download and 'activate' the R package *xUCINET*, which these (and other) functions are part of. To install the package in RStudio, you can go to **`Tools>Install Packages`** and search for the package *xUCINET*. Alternatively, you can use the command `install.packages("xUCINET")`. Once installed, the package also needs to be activated. To do so, type `library(xUCINET)`. You can now use the functions from the *xUCINET* package. (This last part must be done every time you start up RStudio.) Specific details on these aspects are given on the book's website.

Practice 5.3 Go to the website and check out

1. How to create a project using **`xCreateDataProject()`**
2. How to import different types of network data, as well as attribute data, into R using **`xAddToDataProject()`**

5.5 Data transformation for network data

Once the data have been imported in R, you might want to perform some basic transformations on them and potentially add these to the project. Here we discuss a small sample of the myriad of transformations that are often applied to data in the course of an analysis. These include dichotomizing, combining relations, transposing matrices, symmetrizing, imputing missing values, combining nodes, extracting subgraphs, and many more. We will be using the *matrix* format to illustrate the transformations in this section, although these functions can also be applied to network data organized in objects of class *network* or *igraph*.

5.5.1 Dichotomizing

One common data transformation is dichotomizing, which refers to converting valued data into binary data. Typically, we take a valued adjacency matrix and set all cells with a value greater than (or less than, or exactly equal to) a certain threshold (also called a cut-off value) to 1, and set all the remaining cells to 0. For example, Matrix 5.2 is the result of dichotomizing Matrix 5.1 using the rule that any tie strength greater than 1 should be considered a tie. In contrast, Matrix 5.3 is based on a dichotomization using a cut-off of greater than 0.

Matrix 5.1

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	2	3	1	0	0
Joey	1	0	1	1	2	0
Monica	3	0	0	0	1	2
Phoebe	1	0	0	0	0	0
Rachel	1	2	1	1	0	3
Ross	1	0	2	0	3	0

Matrix 5.2

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	1	1	0	0	0
Joey	0	0	0	0	1	0
Monica	1	0	0	0	0	1
Phoebe	0	0	0	0	0	0
Rachel	0	1	0	0	0	1
Ross	0	0	1	0	1	0

Matrix 5.3

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	1	1	1	0	0
Joey	1	0	1	1	1	0
Monica	1	0	0	0	1	1
Phoebe	1	0	0	0	0	0
Rachel	1	1	1	1	0	1
Ross	1	0	1	0	1	0

Matrices 5.1–5.3 Original valued data for the 'liking' network in Figure 5.7 (Matrix 5.1), and two different dichotomizations when using greater than 1 (Matrix 5.2) and when using greater than 0 (Matrix 5.3).

One reason why we might dichotomize data rather than use valued data is because different research questions call for different kinds of ties. For example, we might have measured ties using a scale like this: 0 = 'not a friend', 1 = 'friend', 2 = 'good friend', 3 = 'dating'. When interested in the chance of a person hearing some random bit of information from another person, perhaps we would dichotomize at anything greater than 0. But with respect to estimating a person's feeling of having emotional support, we might only consider good friendship ties, and dichotomize at greater than 1, or even greater than 2.

In other cases we might decide to dichotomize valued relations because some methods, especially graph-theoretic methods, are only applicable to binary data. Also, dichotomizing with a high cut-off can serve to reduce the proportion of dyads with value 1 in the network (i.e., the density), and so we might want to find a cut-off which creates a network where, for example, 30% of dyads have a tie.

When dichotomizing for these very practical reasons, it is usually advisable to dichotomize at different levels and run the analyses on each of the resulting datasets. This approach retains the richness of the data and can reveal insights into the network structure that would not be easy to deduce from techniques designed to deal with valued data directly, as many effects, such as Granovetter's forbidden triad principle (Granovetter, 1973), are not necessarily linear. This approach also gives you an idea of the extent to which your findings are robust across

different thresholds of ties. Unless theoretically motivated, you do not want results that hinge on a particular, perhaps arbitrary, choice of dichotomization threshold.

An alternative approach is to decide on the cut-off point based on maximizing the information retained when dichotomizing the matrix. To decide this, we could dichotomize at different levels, and then choose that cut-off point which, for example, results in the maximum correlation with the original valued data (Borgatti and Quintane, 2018). The function `xDichotomize()` provides a simple way to dichotomize a network, as well as a number of options to decide at which point to dichotomize.

Practice 5.4 Go to the website and check out

1. How to use `xDichotomize()` to find the cut-off for the advice-giving network from Burkhardt and Brass (1990) so that approximately 40% of ties are 1
2. How to use `xDichotomize()` to find the cut-off point for the advice-giving network from Burkhardt and Brass (1990) which maximizes the information retained using the Borgatti-Quintane approach

5.5.2 Combining relations

As noted before, most network studies collect multiple relations on the same set of nodes. For some analyses, however, we might want to combine some of these separate relations into one. For example, we might take three separate network questions, such as 'Who do you attend sports events with?', 'Who do you go to the theater with?', and 'Who do you go out to dinner with?' and combine them into a more general, analytically defined relation, such as 'who socializes with whom'.

Considering our example, we might combine the newly created dichotomized 'liking' network (Matrix 5.3) and the original 'flatmates' network (Figure 5.3) to create a new network.

Matrix 5.4

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	2	2	0	0	0
Joey	1	0	0	0	2	0
Monica	2	0	0	1	1	1
Phoebe	0	0	1	0	1	0
Rachel	0	2	1	1	0	2
Ross	0	0	1	0	2	0

Matrix 5.5

	Chandler	Joey	Monica	Phoebe	Rachel	Ross
Chandler	0	1	1	0	0	0
Joey	0	0	0	0	1	0
Monica	1	0	0	0	0	0
Phoebe	0	0	0	0	0	0
Rachel	0	1	0	0	0	1
Ross	0	0	0	0	1	0

Matrices 5.4–5.5 Two different ways of combining the dichotomized 'liking' network relation (in Figure 5.7) and the 'flatmates' relation (in Figure 5.6), taking the sum of both matrices (Matrix 5.4), and using the minimum of both (Matrix 5.5).

Depending on the aim, we might combine both network relations in a number of ways. First, we could simply use the sum of both networks to get a strength of tie (perhaps weighting one more than the other). We could use the raw numbers as an indication of the strength of the positive tie (see Matrix 5.4). This can be achieved by simply summing the corresponding cells in both matrices. Other options could be that we dichotomize this new matrix so that the presence of either tie (being flatmates *or* liking each other) constitutes a tie in the new relation, or alternatively identifying something as a tie if two people share a flat *and* at the same time like each other (see Matrix 5.5).

Alternatively, we can take an empirical approach and try to discover which relations are highly correlated. We can compute Pearson correlations between all pairs of adjacency matrices, and then inspect the correlation matrix or use factor analysis to identify their dimensions. The function **`xCombineTies()`** provides a range of possibilities for combining networks.

5.5.3 Transposing

To transpose a matrix is to interchange the sender with the receiver (see Matrices 5.6 and 5.7). When applied to a non-symmetric adjacency matrix, this has the effect of reversing the direction of all the arcs (see Figure 5.19). This can be helpful in maintaining a consistent interpretation of the ties in a network. When we construct adjacency matrices from surveys, we generally do it in such a way that the rows correspond to respondents (egos) and the columns correspond to the people mentioned by ego (alters). However, it is also convenient to think of the row node as sending to the column node, as in sending information or resources. These two conventions can be in conflict. For example, suppose the survey asks, 'Who do you seek advice from?' A '1' in cell (i, j) means that person i says they seek advice from person j. But in which direction does advice go? Advice is flowing from person j to person i. In this case, it might be useful to transpose the matrix and think of it as 'Who gives advice to whom?'. A similar situation occurs with food webs, where we have data on which species eat which other species. Ecologists like to reverse the direction of the arrows because they think in terms of the direction of energy flow through the ecosystem (e.g., carbon flowing from the prey to the predator). To transpose a network, the function **`xTranspose()`** can be applied.

Matrix 5.6

	a	b	c	d	e	f
a	0	1	1	1	1	1
b	0	0	1	0	0	0
c	0	1	0	0	0	0
d	0	0	1	0	1	0
e	0	0	0	0	0	0
f	1	0	0	0	0	0

Matrix 5.7

	a	b	c	d	e	f
a	0	0	0	0	0	1
b	1	0	1	0	0	0
c	1	1	0	1	0	0
d	1	0	0	0	0	0
e	1	0	0	1	0	0
f	1	0	0	0	0	0

Matrices 5.6–5.7 Two different ways of capturing the same relation. Network capturing 'Who do you seek advice from?' (Matrix 5.6), and its transpose capturing 'Who asks you for advice?' (Matrix 5.7).

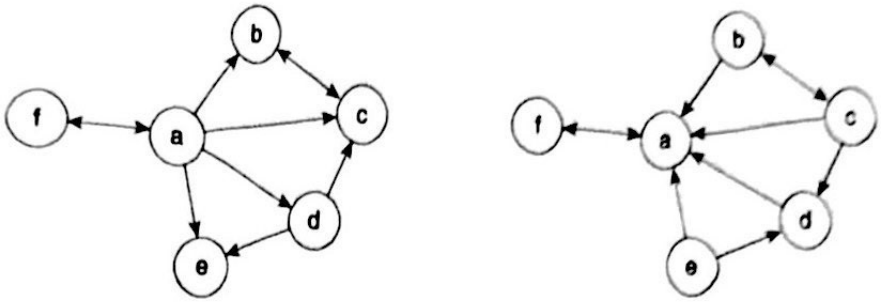

Figure 5.19 Illustration of how transposing an adjacency matrix is equivalent to reversing the direction of the arrows: graphical representations of Matrices 5.6 and 5.7.

5.5.4 Symmetrizing

Symmetrizing refers to creating a new dataset in which all ties are reciprocated (and hence regarded as undirected). There are many reasons to symmetrize data. One very practical reason is that some analytical techniques, such as multidimensional scaling, assume symmetric data. In other cases, symmetrizing is part of data cleaning. For example, when we ask respondents to name the people they communicate with using an open-ended network question, we often find unintended asymmetry because respondents simply forget to mention people, as noted in the previous chapter. In these cases, we often create a new, symmetric adjacency, using the rule that if either person mentioned the other, then there is a tie. We call this the 'OR', or 'union', rule. Alternatively, if we suspect name-dropping, we might adopt a stricter rule, namely that only if both people mention each other will we consider it a tie. This is called the 'AND', or 'intersection', rule. Obviously, the union rule creates networks denser than the original, while the intersection rule makes them sparser.

In other cases, we symmetrize in order to study an underlying symmetric relationship that is not quite the same thing as the observed tie. For example, suppose we have asked respondents who they receive advice from. In seeking advice, we know that an actor reveals the problem they are trying to solve. In this sense, there is an exchange of information. It may be that this exchange of information is the social relation we are really interested in studying, perhaps because we see it as a proxy for a certain level of collaboration or intimacy. In this case, we symmetrize using the rule that if either person gives advice to the other, we say there is an exchange tie.

From the point of view of a matrix representing a network, when we symmetrize we are comparing an (i, j)th entry with the corresponding (j, i)th entry and, if needed, making them the same. The union rule corresponds to taking the larger of the two entries. The intersection rule takes the smaller of the two. Many other options are possible as well. For valued data, we might consider taking the average of the two entries. For example, if i estimates having had lunch with j eight times in a month, but j estimates having lunched with i ten times, we can view these as two measurements of the same underlying quantity, and use the average as the best available estimate of that quantity. The function **`xSymmetrize()`** allows a range of options to symmetrize network data, including taking the minimum, maximum, sum or average.

Practice 5.5 Go to the website and check out

1. How to transpose a matrix using the function **`xTranspose()`**
2. How to symmetrize a matrix using the function **`xSymmetrize()`**
3. How to correlate two networks
4. How to combine two networks using the minimum and sum with the function **`xCombineTies()`**

5.5.5 Dealing with missing data

Missing data can be a problem in full network research designs. The most common kind of missing data is where a respondent has chosen not to fill out the survey. This creates a row of missing values in the network adjacency matrix. Some researchers might be tempted to deal with missing values as non-ties, which is simply incorrect. Another obvious solution is to eliminate that node from the analysis altogether (deleting both their row and their corresponding column in the adjacency matrix). The trouble with this is that it assumes that the missingness can simply be ignored, which is not the case. For example, if the missing node is the most important in the network – and it is not unusual for important people to be unwilling to fill out a survey – the picture we get will be very different from what we should have gotten. Moreover, since the other nodes provided responses about their relations towards that node, removing them means ignoring available data. It would seem worthwhile, then, to search for ways to retain the problematic node.

In the case of symmetric or undirected relations, a simple cure is to fill in any missing rows with the data found in the corresponding column. The assumption is that, if the respondent had been able to answer, they would have listed all the actors that mentioned them. We might also try to use this in the case of a directed network, provided the network relation is highly reciprocal (e.g., 'Who do you discuss important personal matters with?'). In this case, the results may not be exactly right, but they will probably be more accurate than treating the missing values as zeros. Another rule that can be used to impute a value for cell (i, j) is to ask, 'What do most respondents say about j'? If most people chose j as someone whom they seek advice from, it is reasonable to guess that i would have as well. Even better, we can look at i's pattern of incoming ties, and find the two or three other people with similar patterns of incoming ties, and choose whatever they said about j. These and other rules are outlined in Žnidaršič et al. (2012). Butts (2003) presents a Bayesian approach intended for cognitive social structure data. For general matrices, Candès and Recht (2012) present methods for recovering missing cells. Some of these methods are available in the function **`xImputeMissingData()`**.

As with missing data in classic data, we will always need to make some assumptions about the missing data, as we simply did not observe the data and cannot be sure about their real value. In general, it is a good idea first of all to check if missing nodes are similar to non-missing nodes in their incoming ties, or any other information that might be available, and second, to run the analyses separately using different ways of handling missing values (e.g., using multiple imputations), to see if the results are robust. If they are not, there is the danger that any findings are an artifact of the method used to handle missing values. In the end, taking significant steps to avoid having missing data in the first place is always best. For more sophisticated ways of imputing missing values and other ways of dealing with missing data, the reader is advised to consult the relevant literature (e.g., Koskinen et al., 2010; Krause et al., 2020).

Practice 5.6 Go to the website and check out

1 How to impute missing data in a network using the function **`xImputeMissingData()`**

5.5.6 Normalization

Sometimes we want to re-express, standardize or normalize network data to ensure we are making fair comparisons across rows, columns or entire matrices. In Chapter 4 we discussed the concern that in the use of rating scales for collecting strength-of-tie data there could be a problem due to respondents' use and interpretations of the scales. Some respondents may have more readily used the high end of the scale while others used the lower end for reporting on essentially the same tie strength. In another example, if respondents are asked to assess the physical distance from their homes to the homes of all other actors in the network, it may be that some respondents tend to overestimate distances, while others might underestimate them. In both of these cases, it might make sense to reduce each row to a common denominator in order to make the data comparable. One way to get around these different scale issues is to normalize the data. A classic normalization procedure is to compute *z*-scores for each row so that each has a mean of 0 and a standard deviation of 1 (excluding the main diagonal). Another approach is to divide each row by the maximum value. Both of these procedures assume that the rows correspond to respondents (in effect, measuring devices) that need to be made comparable. There is a range of procedures: one can normalize with respect to means, marginals, standard deviations, means and standard deviations together, Euclidean norms, and maximums. Each type of normalization can be performed on each row separately, on each column separately, on each row *and* each column (iteratively), and on the matrix as a whole. The function **`xNormalize()`** provides a range of normalizations.

Practice 5.7 Go to the website and check out

1 Different ways to normalize a network using the function **`xNormalize()`**

5.5.7 Geodesic distance

A frequently used transformation of an adjacency matrix, which goes beyond a simple transformation of the matrix, involves calculating the geodesic distance between nodes and storing them as a matrix. We have already defined the geodesic distance between two nodes as the length of the shortest path between them (see Chapter 2). To calculate the geodesic distances between every pair of nodes, we can use the **`xGeodesicDistance()`** function, which takes the adjacency matrix as an input. A simple example of such a transformation was provided for Borgatti's Camp92 dataset in Chapter 2. In this example the adjacency matrix for the top 3 interaction partners in week 3 of the workshop can be found in Matrix 2.1, and the matrix containing the geodesic distances is presented in Matrix 2.2. We note that in R the default setting for nodes that can never reach each other is 'Inf' (for positive infinity).

Practice 5.8 Go to the website and check out

1. How to use the **xGeodesicDistance()** function to calculate distances between nodes for the top 3 interaction partners in week 3 of a workshop (Borgatti's Camp92) as depicted in Matrix 2.2

5.5.8 Combining nodes

Sometimes we collect data at an individual level but want to analyze them at a higher level. For example, we collect data on who collaborates with whom in an organization, but what we are really interested in is the pattern of ties between departments (see Matrix 5.8). As a result, we want to aggregate the nodes into departments such that a tie between any two nodes becomes a tie between their departments. The interdepartmental ties could be defined as a simple count of the individual-level ties, or we could normalize the count to account for

	A07	A06	A08	A12	A17	A21	A03	A05	A09	A13	A14	A15	A19	A20	A10	A11	A18	A01	A02	A04	A16	DEPT
A07	0	0	1	1	1	1	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0	0
A06	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	0	0	0	1	1	0	1
A08	0	0	0	0	0	1	1	1	1	0	0	1	0	1	1	0	1	1	0	1	0	1
A12	1	0	0	0	0	1	1	0	1	0	0	1	0	1	0	0	0	0	0	1	0	1
A17	1	0	0	0	0	1	1	1	1	0	0	1	0	1	1	0	0	0	0	1	0	1
A21	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1
A03	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	2
A05	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	2
A09	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	2
A13	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	0	2
A14	1	0	0	0	0	1	1	1	1	1	0	1	1	1	0	0	1	0	0	0	0	2
A15	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	2
A19	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	0	2
A20	0	0	0	0	0	1	1	1	0	0	0	1	1	0	1	0	1	0	0	1	0	2
A10	0	0	1	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	1	3
A11	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	0	1	0	0	1	0	3
A18	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	3
A01	0	0	0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	0	0	1	1	4
A02	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	4
A04	0	0	1	0	1	1	1	0	0	0	0	1	0	0	1	0	1	1	0	0	0	4
A16	0	0	0	0	0	0	0	1	1	0	0	1	0	1	1	0	1	1	0	1	0	4

Matrix 5.8 Advice ties among 21 high-tech employees (Krackhardt, 1987), reordered according to department. The department each employee belongs to is provided on the right-hand side.

the number of people in each department. One normalization is to divide the count of ties between department A and department B by the maximum number possible, which is simply the size of department A multiplied by the size of department B.[1] These are called densities, and the valued adjacency matrix thus created is called a density table. Matrix 5.8 shows the advice network for Krackhardt's high-tech managers ordered by department (with 0 indicating the top manager). Aggregating these cells by department, we get the values in Matrix 5.9. We can get the density between each group (Matrix 5.11) by dividing the total number of observed ties between the respective departments (Matrix 5.9) by the maximum possible number of ties (Matrix 5.10). These results (Matrix 5.11) show that managers in Department 3 ask quite a bit of advice from Department 2 (0.79) and from colleagues inside their own department (0.67), but not from Department 1 (0.27). Department 1 also asks quite a bit of advice from Department 2 (0.63) and from the top manager (0.80), but very limited advice from Department 3 (0.27).

Matrix 5.9

	D0	D1	D2	D3	D4
D0	-	4	6	2	1
D1	4	9	25	4	9
D2	1	3	25	14	1
D3	2	4	19	4	6
D4	1	7	21	10	8

Matrix 5.10

	D0	D1	D2	D3	D4
D0	-	5	8	3	4
D1	5	20	40	15	20
D2	8	40	56	24	32
D3	3	15	24	6	12
D4	4	20	32	12	12

Matrix 5.11

	D0	D1	D2	D3	D4
D0	-	0.80	0.75	0.67	0.25
D1	0.80	0.45	0.63	0.27	0.45
D2	0.13	0.08	0.45	0.58	0.03
D3	0.67	0.27	0.79	0.67	0.50
D4	0.25	0.35	0.66	0.83	0.67

Matrices 5.9–5.11 Ties between departments for the advice network among Krackhardt's 21 high-tech employees. Total number of observed ties between each department (Matrix 5.9), maximum number of possible ties between departments (Matrix 5.10) and density between departments (Matrix 5.11).

A different approach is to divide the values in each row by the sum of outgoing ties by all members of that department. The resulting matrix gives the proportion of a department's ties that are going to each department (including itself). This tells you how the department members are allocating their 'relational energy' across departments. Another alternative is to divide the number of ties between departments by the expected value, given a model of independence – that is, a model in which nodes make ties without regard for what department they are in.

Regardless of the choice of method to combine nodes, the valued matrix obtained can also be dichotomized. The result is a new network (in which the nodes are departments) which can be analyzed in all the usual ways, such as running a subgroup analysis to find out which departments cluster together. Figure 5.20 shows the network of ties based on a dichotomized version of Matrix 5.11. To aggregate nodes, the function `xDensity()` can be used.

[1] When A and B refer to the same department and the network is non-reflexive, the maximum number of ties is $n(n-1)$, where n is the number of nodes in that department.

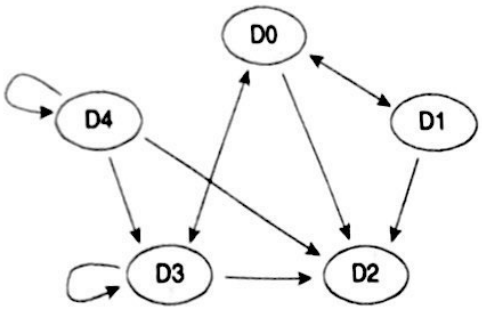

Figure 5.20 Ties between departments based on Matrix 5.11 using dichotomization (density greater than 0.6).

Practice 5.9 Go to the website and check out

1 How to aggregate the advice network among high-tech managers based on the nodal attribute, department, using the function **`xDensity()`**

5.6 Converting attributes to matrices

In network analysis we sometimes need to transform node attribute data into network data. In general, what we are doing is changing a node quality into a relational quality, so that instead of height we have 'is taller than', and instead of gender we have 'is same gender as'. Often, the purpose is to use the 'matricized' attribute as a dyadic predictor of ties in a quadratic assignment procedure regression or exponential random graph model (Chapters 14 and 15). For example, suppose we have a directed relation such as 'liking' and we want to predict who likes whom. One idea is that people like people who are similar to themselves on socially significant attributes such as race, gender, age and status. For race and gender, we can construct 'same race as' and 'same gender as' matrices, as shown in Figure 5.21. For age, we might use absolute difference in age, as shown in Figure 5.22. For status, we might anticipate that low-status individuals have a bias for higher-status individuals and against lower-status individuals. In this case, we would probably want to use the simple difference in status, as shown in Figure 5.23. Alternatively, we might argue that the probability of liking someone increases with their status, even if that status is still lower than one's own. In that case, we create a matrix in which the values in each column equal the status of the node corresponding to that column. Effectively, it is like writing the status vector as a row vector and

Actor	Gender
a	1
b	2
c	2
d	1
e	1
f	2

	a	*b*	*c*	*d*	*e*	*f*
a	1	0	0	1	1	0
b	0	1	1	0	0	1
c	0	1	1	0	0	1
d	1	0	0	1	1	0
e	1	0	0	1	1	0
f	0	1	1	0	0	1

Figure 5.21 Converting gender into a 'same gender as' matrix.

Actor	Age
a	14
b	67
c	34
d	33
e	56
f	45

	a	*b*	*c*	*d*	*e*	*f*
a	0	53	20	19	42	31
b	53	0	33	34	11	22
c	20	33	0	1	22	11
d	19	34	1	0	23	12
e	42	11	22	23	0	11
f	31	22	11	12	11	0

Figure 5.22 Converting age into an 'absolute difference in age' matrix.

Actor	Status
a	6
b	10
c	3
d	5
e	9
f	4

	a	*b*	*c*	*d*	*e*	*f*
a	0	−4	3	1	−3	2
b	4	0	7	5	1	6
c	−3	−7	0	−2	−6	−1
d	−1	−5	2	0	−4	1
e	3	−1	6	4	0	5
f	−2	−6	1	−1	−5	0

Figure 5.23 Converting status into a 'relative status difference' matrix.

Actor	Status
a	6
b	10
c	3
d	5
e	9
f	4

	a	*b*	*c*	*d*	*e*	*f*
a	6	10	3	5	9	4
b	6	10	3	5	9	4
c	6	10	3	5	9	4
d	6	10	3	5	9	4
e	6	10	3	5	9	4
f	6	10	3	5	9	4

Figure 5.24 Converting status into a 'status of alter' matrix.

then copying it $n-1$ times to form a matrix. This is shown in Figure 5.24. All of these conversions can be done in R using the `xAttributeToMatrix()` function.

Practice 5.10 Go to the website and check out

1. How to create a network that represents the same gender based on the attribute file using the function **`xAttributeToMatrix()`**
2. How to transform the attribute age into an absolute difference-in-age network
3. Some other ways to create network data from attributes

5.7 Storing, transforming and exporting network data and results

Network analysis often consists of a series of transformations where the output of one analysis becomes the input to another. In analyzing social network data, for example, we might start with a set of files with data, convert them to CSV files, import them as a project in R, dichotomize them, symmetrize the dichotomized matrices, then run centrality on each of these networks, factor-analyze the centrality measures, and regress some outcome variable on the resultant principal factor. Anywhere along the line, we might want to store data, convert them to another format to apply other R packages on these data, output data to other software packages outside R, or create a table or figure for a presentation or paper. We discuss these in turn.

5.7.1 Storing data inside R

First of all, we might want to save one or more R objects as an internal R file, so we (or another person) can use them again the next time we open R, rather than every time having to read them in from the original CSV files. To do so we can use the function **`save()`**. The files will be saved in the defined directory, and can subsequently be retrieved in the next R session by simply using the function **`load()`**.

5.7.2 Storing data as a CSV

We might also want to export the data so they can be used by other programs (e.g., UCINET, Pajek, SPSS or SAS). As with data import, for exporting data we usually recommend using a spreadsheet editor as an intermediary that can communicate with many kinds of software. While it is possible to cut and paste the output from R into a spreadsheet, it is a cumbersome and needlessly complicated way to transfer data. A better approach is to save the network or attribute data file that was created as a CSV and then open it in your spreadsheet editor or import it in the other program directly. This can be done using the function **`write.csv()`**. For example, we can save the earlier defined matrix that combines both the 'liking' and 'flatmates' matrix (as a network called 'Strong').

5.7.3 Transforming network data into *igraph* or *network* objects

In most of this book we will rely on *xUCINET* to perform our analysis, and therefore the objects can be any of the three classes. However, for some specific analysis, you might want to use other specialized R packages, such as *igraph* or *statnet*. Different packages in R are useful for

different types of analysis. For example, you might want to use specific packages to visualize your data, to perform statistical analysis such as exponential random graph or stochastic actor-oriented modeling (see Chapter 15), or to perform blockmodeling (Chapter 12). In Practice 5.11 we discuss ways to convert network data into a specific class, such as a *matrix*, *network* (for *statnet*) or *igraph* object (for *igraph*).

Practice 5.11 Go to the website and check out

1 How to save an object as a .RData file using the **`save()`** function
2 How to save an object as a CSV file using the **`write.csv()`** function
3 How to transform a *matrix* or *network* object into an *igraph* object so it can be used by the R package *igraph*

5.8 Summary

This chapter introduced ways to store network and attribute data in spreadsheets using CSV files for each network and attribute dataset, and to subsequently import such data in R. The advantage of a CSV file over a simple space or tab-delimited text file is that it can easily be opened again in a spreadsheet editor, such as Excel. Network data need to be in specific formats in order to efficiently enter them into computer programs. Some networks are sparse, in which case nodelist or edgelist formats are the most efficient ways to enter data in a spreadsheet. We might also hold attribute information on the nodes of a network. This information is commonly in the form of full matrices or vectors and can be imported using a classic spreadsheet format. Once imported into a spreadsheet editor, data should be examined and cleaned, if necessary, to make sure they are accurate. The resulting CSV files can be imported into R as a *matrix*, *network* or *igraph* object. Since a project often contains multiple network relations and attribute data, we explained how to read in multiple networks and attributes and how to combine these into a single list/project object. Once imported into R, various transformations such as symmetrization, dichotomization and transposition can be applied to the network data, which can be added as additional elements to the list object. Note that if the transformations have fundamentally changed the nature of the relation, this must be taken into account when interpreting the output obtained from any analysis. For example, if an advice network has been symmetrized via the maximum method, we can no longer interpret x_{ij} as indicating that node i gives advice to node j. However, we can reasonably regard x_{ij} as indicating that i and j are involved in an exchange of information. Finally, the matrix objects or specific output can be saved either as R-type data (.RData) or in other formats such as CSV files so they can be retrieved by other programs or at later stages by the same users. Figure 5.1 summarizes these different steps.

5.9 Problems and exercises

1 Below are social network data collected from members of a book club at a university. The students were asked to rate 'how much they interacted online' with the other students over the last two weeks on a five-point Likert scale with '1' being 'very little' and '5' being 'a great deal'.

Table 5.4 Hypothetical network dataset.

Mark	Gene	5
Mark	Silvia	4
Mark	Sarah	4
Mark	Tim	2
Gene	Mark	2
Gene	Silvia	2
Gene	Sarah	3
Gene	Tim	4
Silvia	Mark	2
Silvia	Gene	2
Silvia	Sarah	4
Silvia	Tim	4
Sarah	Mark	2
Sarah	Gene	1
Sarah	Silvia	2
Sarah	Tim	4
Tim	Mark	1
Tim	Gene	3
Tim	Silvia	2
Tim	Sarah	4

a Enter the data in a spreadsheet using the adjacency matrix format and save the result as a CSV file.
b Import the data into R as an *xUCINET* dataproject object.
c Symmetrize the data using the minimum and store as an extra element of the dataproject.
d Dichotomize the data using greater than 2 and store as a new file.
e Save the data (as a .RData file), and also export it as a CSV file.
f What would have happened if we had first dichotomized the data using the 'greater than 2' rule, and subsequently symmetrized the data using the minimum?

2 Demographic and attribute data were also collected (see below). Data include age, gender and attitude towards premarital sex on a Likert scale from 1 to 5 (with '1' being 'completely against' and '5' being 'completely in favor').

Table 5.5 Hypothetical attribute dataset.

	Age	Gender (1=male, 2=female)	Attitude
Mark	18	1	4
Gene	25	1	2
Silvia	19	2	2
Sarah	21	2	5
Tim	22	1	1

a Enter the data in a matrix format in a spreadsheet.
b Add the attribute file to the *xUCINET* project dataset that contains the networks from Problem 1.
c Convert the *Attitude* variable into a network using the absolute difference and store as a new matrix in the dataproject. What do the values represent?
d Convert the *Gender* variable into a network using the difference and add it to the dataproject. What does this result in?

3 In addition to the one-mode social network data above, two-mode data were also collected on which university events each had attended over the last six months. Here are the data:

- Mark went to Hawking's Lecture
- Gene went to Hawking's Lecture and Spring Concert
- Silvia went to Hawking's Lecture, Halloween Event, and Fall Dance
- Sarah went to Hawking's Lecture, Halloween Event, Spring Concert, and Fall Dance
- Tim went to Halloween Event, Spring Concert, and Fall Dance

a Format the data as a spreadsheet file using a 'nodelist' format, and save as a CSV file.
b Add the network to the dataproject (from Problems 1 and 2) as a *matrix* object, and check the output when calling the resulting object.

4 Finally, the students were asked who talks to whom about school-related matters.

- Mark reports he talks to Gene, Silvia, and Sarah
- Gene reports he talks to Tim
- Silvia reports she talks to Tim and Sarah
- Sarah reports she talks to Tim
- Tim reports he talks to Sarah

a Enter the data as a spreadsheet file using the edgelist format. Import the CSV file into R.
b Symmetrize the network using the 'OR' approach.

5 Imagine you would want to combine the dichotomized online interaction network (Problem 1) and the face-to-face talking network (Problem 4) into one matrix.

a How would you go about making sure you are able to distinguish between ties where (a) members only talk face-to-face, but do not interact online, (b) members only interact online, but not face-to-face, (c) members do both, and (d) members do neither? Use R to solve this.
b Given the type of data, could you consider a different approach to combine the two networks? Reflect on your answer.

6 Sometimes it might be necessary to reverse the direction of ties for conceptual or other reasons. Being antagonistic towards someone is one example. It captures whether ego has antagonistic behavior towards alter, but we might want to reverse the direction of the tie. Open the Hawthorne bank wiring room dataset and for the antagonistic network transpose the resulting matrix. Conceptually, what might the direction of the ties now represent?

7 Suppose you asked each child in a school with 783 pupils to name their three best friends. How would you store the data? What if you were to ask them to rank their three choices from closest (1) to least close (3)? How would you store the data?