

Analyzing biological data using R: methods for graphs and networks.

Nolwenn Le Meur^{1,2,*}, Robert Gentleman²

* corresponding author

¹ IRSET EA SeRAIC 4427 - IRISA - Equipe Symbiose, Université de Rennes I Campus de Beaulieu, 35042 Rennes cedex, France. email:nlemeur@irisa.fr

² Fred Hutchinson Cancer Center Research, Program in Computational Biology, Fred Hutchinson Cancer Center Research, Program in Computational Biology, M2-B876, P.O. Box 19024, Seattle, WA 98109, USA. email:rgentlem@fhcrc.org

Email:

*Corresponding author

Abstract

R is a powerful language and widely used software tool for the analysis and visualization of data. Its core capabilities can be extended through many different add-on packages. Among the many packages are some which offer a broad range of facilities for analyzing statistical properties of graphs. This chapter will provide a practical tutorial covering the use of R methods for graphs and networks to examine biological data and analyze their topological and statistical properties.

keywords

graph, random graph, network, statistic, systems biology

1 Introduction

1.1 Context

Recent advances in molecular biology and associated technologies have expanded our opportunities to explore and understand the complexity of biological systems at different levels; from DNA to proteins to signaling networks and beyond. To integrate these data, to build and explore biological networks, we now need computational tools, efficient flexible data structures and statistical methods designed with the particular challenges of systems biology and the analysis of high-throughput data in mind.

For the past 10 years, the Bioconductor community has worked on developing such infrastructure. The tools come in the form of different packages for R [1], a free software environment for statistical computing and graphics. Each package provides software, or data, to carry out a specific set of tasks. Developing workflows that rely on multiple packages is an effective strategy for the rapid development of tools for analyzing biological data. Packages are being developed to address the specific needs of systems biology approaches, such as network modeling, simulation, formal verification and graph visualization [2]. For instance, Castelo and Roverato developed a R package providing reverse engineering procedures for molecular network discovery from microarray expression data [3]. Similarly, using graph theoretical concepts, Le Meur and Gentleman [4] identified multi-protein complexes and pairs of multi-protein complexes that share an unusually high number of synthetic genetic interactions. Network data computed and analyzed in R can directly be visualized within R using for instance the **graph** and **Rgraphviz** Bioconductor libraries that will be discussed in this chapter, or **igraph**, an interesting and valuable R library [5]. Additionally, the network data generated by R can easily be exported to other visualization tools. For example, Cytoscape [6] can be seen as an analyst-friendly and interactive alternative. Among many features, Cytoscape notably allows zooming in and out or using *bird's eye view* for browsing large graphs. It also offers classical network analysis such as selecting subnetworks based on graph properties (*e.g.*, node or edge annotation).

In this chapter, we review methods available in R for working with graphs. We discuss how to retrieve and integrate biological data to build graphs of gene networks. We also consider some of the techniques available to explore and analyze the topological and statistical properties of these graphs. This chapter was written using the Sweave system [7], and all the code used to produce the outputs, graphics, etc. can be retrieved from our on-line supplements, located at <http://www.bioconductor.org/pub/MiMB>.

1.2 Case Studies

Case Study 1: *Bacillus subtilis* protein-protein interaction data.

Bacillus subtilis known as the hay bacillus or grass bacillus, is a Gram-positive bacterium commonly found in soil. It may contaminate food but rarely causes food poisoning. It is not considered a human pathogen. *B. subtilis* and the enzymes it produces have many applications such as immuno-stimulatory agents in alternative medicine or as additives in laundry detergents. In this chapter, we will explore some of the protein-protein interaction (PPI) data available for *B. subtilis*.

Case Study 2: *Response of Escherichia coli* during an oxygen shift.

Escherichia coli is a Gram-negative bacterium that is commonly found in the lower intestine of warm-blooded organisms. Most *E. coli* strains are harmless, but some, such as serotype O157:H7, can cause serious food poisoning in humans. *E. coli* can grow in aerobic as well as in anaerobic conditions (*e.g.*, the

intestine). In this case study, we will explore gene expression results of various knock-out (KO) mutants under aerobic conditions to characterize elements of the transcriptional network.

2 Material

We assume that the reader is acquainted with R and only give a quick overview on the installation procedures and help functionality. For more details about R and its programming language, we refer the reader to the many manuals and courses freely available on the R and Bioconductor websites (<http://www.r-project.org>; <http://bioconductor.org/>). In addition there are many good books on R and Bioconductor such as [8, 9, 10, 11].

The Graphviz package (<http://www.graphviz.org/>) and the Boost graph library (<http://www.boost.org/>) are additional tools required to run graph examples of this chapter. Their installation might not be trivial; it depends on the operating system. One should be especially careful to the location of those libraries and the definition of their path in the environmental variables. For details, we refer the reader to the project websites or the Bioconductor packages that depend on them, namely **Rgraphviz** and **RBGL**.

2.1 R and Bioconductor

R is freely available and runs on a wide variety of commonly used operating systems. To install R, download the most recent version of R from <http://www.r-project.org/>. Detailed instructions of installation are available in the R FAQ and the R Installation and Administration Manual on the R project website. The base installation of R consists of the programming language itself together with a number of packages that are mainly oriented to statistical analyses. R can be extended and its capabilities expanded through the use of add-on packages that are notably available through the R and Bioconductor websites. Bioconductor is comprised of several hundred R packages that provide solutions to a large number of bioinformatic and computational biology problems. The scope of Bioconductor packages is broad covering many aspects of the analysis of microarrays but also including tools for genomic annotation, high-throughput genotyping, cell-based assays (*e.g.*, RNAi, cytometry), handling graphs, and more recently high-throughput sequencing.

2.2 Using R and Bioconductor packages

Starting R on Windows or OS X is usually accomplished by double-clicking on the R application icon. On UNIX-like systems users typically type R at a shell prompt. Because there are many complex dependencies between packages and the version of R being used the recommended way to install Bioconductor packages is to use the `biocLite.R` installation script in a R session:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite()
```

To find, download, and install add-on packages, we recommend that you use built-in capabilities, either in the GUI or through the command line. On Windows and OS X packages can be installed *via* a menu. You can also use the R function `install.packages` or the `biocLite` script to download a specified list of packages and their dependencies. There is a difference between installing and using a package. Typically you install a package once, using the tools described above and use it multiple times. For example we could install the **graph** and **Rgraphviz** packages using the code below.

```
> biocLite(c("graph", "Rgraphviz"))
```

To use those packages you must load them using the `library` function:

```
> library("graph")
> library("Rgraphviz")
```

2.3 R Documentation

To get started it is important to know where you can find documentation such as help for functions, classes, and concepts. The `?` operator followed by the name of a function will display the manual page of this function. `class?foo` will show the manual page for the `foo` class and most often the available methods for this class. The function `apropos` can be used to find objects in the search path with names that partially match the character string given as an argument to `apropos`. The function `help.search` will search the help system for documentation matching a given character string, and again partial matching will be used. You should compare the output for the three commands below on your system.

```
> class?graphNEL
> apropos("graphNEL")
> help.search("graphNEL")
```

In addition, each Bioconductor package contains at least one *vignette*, which is a document that provides a task-oriented description of the functionality provided by the package. Vignettes should contain executable examples and are intended to be used interactively.

2.4 R data structures

The basic R data structure is the atomic vector. It contains an indexed set of values that are all of the same type: logical, numeric, complex or character. The numeric type can be further broken down into integer, single or double types. In the code below we create a vector named `x`, and while it may seem that we have mixed integer and real types, the integers have been coerced to their real representation, which can be seen in the output of the call to `typeof`.

```
> x <- c(10, 5, 6, 1.5, 9)
> typeof(x)

[1] "double"
```

Additional data structures are lists, arrays and data frames. An **array** is a vector with a dimension attribute and matrices are a special variant where there are exactly two dimensions. Arrays are like atomic vectors in that all elements must be of the same basic type. A **list** is an object consisting of an ordered collection of objects. It is different from atomic vectors because its elements can be of any type, and for example a list could contain a numeric vector, a logical value, and a matrix. A **data.frame** is much like a matrix in that it is two dimensional. Data frames are widely used for holding sample level data since that is typically rectangular (some variables for each sample or individual) but the data types (columns) can be of mixed types (*e.g.*, age, gender, and so on).

In the code below we create a matrix, `y`, of integers with 2 rows and 5 columns. We also create a data.frame with two columns, one contains numeric values and the other contains character values.

```
> y <- c(1:10)
> dim(y) <- c(2, 5)
> age <- c(33, 44, 55)
> gender <- c("M", "F", "F")
> myDF <- data.frame(age, gender)
```

While creating the data.frame with the default parameters, we allow the conversion of the character vector to factors. Factors are used to encode vectors as a category (or enumerated type). Levels are then an optional vectors of the values that in our case might be taken by `gender`. The default is the unique set of values taken by `as.character(gender)`, sorted into increasing order of `gender`.

```
> is.factor(myDF$gender)

[1] TRUE
```

```
> levels(myDF$gender)

[1] "F" "M"
```

Subsets of the elements of a vector may be selected using the square bracket operator. If the object being subscripted has multiple dimensions, then each dimension can be subscripted independently with arguments separated by commas. If a subscript is not supplied (see the examples for `y` and `myDF` in the code below) then it is presumed that all elements are selected. In the code below we select the first 2 elements of the vector `x`, then the first two columns of the matrix `y`, and finally the first row (patient) of the data.frame `myDF`.

```
> x[1:2]
> y[,1:2]
> myDF[1,]
```

R also supports a number of paradigms for object oriented programming. Object-oriented programming provides a coherent way to describe and manipulate rich data structures. For instance in the **Biobase** package, the *ExpressionSet* class, designed to standardize data structures to represent DNA microarray data, has become the basic data structure for most gene expression data analysis packages. Similarly, recent developments propose classes and basic methods to support many other types of high-throughput data such as cell-based assays [12, 13], genotyping, or DNA sequencing [14].

More generally, we distinguish packages that are mainly infrastructure (sets of tools that can be used to create other pieces of software) and packages that are designed to provide an end-user application. For instance, the **graph** and **Rgraphviz** packages are infrastructure, while the **GStats** and **apComplex** are specific applications built on that infrastructure. In addition there is a large set of packages that provide annotation for whole organisms and a number of real data sets are also available as self-contained packages suitable for method development and testing.

In the following sections, we will make use of the packages listed in Table 1. We will focus on data structures suited to graphs and demonstrate their usefulness to manipulate and represent biological networks.

3 Methods

3.1 Working with graphs in R

3.1.1 Graph types

There are many equivalent ways to represent a graph. One is as a pair of sets, $G = (V, E)$ where V is the set of nodes (or vertices) and E is the set of edges.

An edge is used to represent a particular type of relationship (interaction) between two nodes. There are many different types of graphs and their definitions primarily depend on their edge and node properties. For instance, a graph can be undirected or directed. For an undirected graph the relationship determined by the edges is symmetric, and hence does not have direction assigned to its edges like **g1** in Figure 1. For a directed graph (or digraph) the relationship is asymmetric, and node A can have an edge to node B, but B need not have an edge to A. These relationships are thought of as being directed and are often represented by edges with arrowheads indicating the direction, see panel a) of Figure 2 for an example. A directed graph with no directed cycles is called a *directed acyclic graph*, or DAG.

The *degree* of a node is the number of edges that are incident on that node. If the graph is directed then we can consider in-degree and out-degree separately.

In data analysis, directed graphs are useful for representing interactions, or experiments, that are asymmetric such as protein interaction experiments where one protein is the *bait* and other proteins are *prey*. In many cases the proteins need to be modified to perform the assay and it is not uncommon to observe that protein A interacts with B, when A is the bait, but B fails to interact with A when B is the bait. Another situation where directed graphs may be helpful is in modeling the relationship between transcription factors (TFs) and their targets, we explore this use in our second case study below.

Special types of graphs can be used and developed to answer specific biological questions. For instance DAGs are used by the Gene Ontology consortium [15] to store and explore gene annotation data. One useful type of graph is one that represents the output of a clustering algorithm. We call such graphs *cluster graphs* and they have the property that nodes are separated into groups or clusters and within a cluster all nodes are connected (a complete graph) but between clusters there are no edges. A network is a type of graph that can be used to represent flow. In particular a network is a directed graph with a vertex, called the source, whose in-degree is 0, a vertex, called the sink, whose out-degree is 0, and there can be a numerical capacity associated with each edge in a network.

3.1.2 Building and exploring graphs

There are many ways to computationally represent graphs. Some data structure are designed for efficiency of different types of access or manipulation while others are efficient for handling very large, but sparse, graphs. The **graph** package provides different representations such as matrix representations and a node and an edge list representation. A number of methods are available to coerce graphs between the different representations. There are other packages, such as **hypergraph** that handle even more specialized operations or types of graphs.

The **graph** package offers 3 basic strategies for building random graphs. The **randomEGraph** function generates a random edge graph. In this graph edges are either randomly generated according to a specified probability, or the user can specify a number of edges that are then randomly assigned. **randomNodeGraph**

builds a random graph with a pre-specified node degree distribution. And for the `randomGraph` function the user provides both the node labels and a factor with some fixed number of levels. Each node is randomly assigned levels of the factor and edges are created between nodes that share the same levels of the factor. As an example, we create a random graph `g1` using the `randomEGraph` function from the **graph** package. `g1` is an undirected graph composed of 15 nodes (labeled by the first 15 letters of the alphabet) and 20 edges.

```
> set.seed(123)
> g1 <- randomEGraph(LETTERS[1:15], edges= 20)
> class(g1)

[1] "graphNEL"
attr(,"package")
[1] "graph"

> g1

A graphNEL graph with undirected edges
Number of Nodes = 15
Number of Edges = 20
```

The **graph** package also offers a collection of functions and methods to convert various forms of matrices into graph objects (*e.g.*, `ftM2graphNEL`, `aM2bpG`).

Other graph-related packages provide methods for computing graph properties. Some packages are aimed at specific areas of application such as the **ppiStats** package which is aimed at PPI data and the **rsbml** package which provides support for SBML modeling (Systems Biology Markup Language <http://www.sbml.org>). The **RBGL** package is an interface from **R** to the Boost Graph Library [16]. It is the primary source of algorithms to explore and analyze graphs. For instance, it contains methods to search for shortest paths, connected components, and cliques. As an example, if a graph represents a metabolic network, we might be interested in computing the shortest-path to generate a product downstream of a cascade of interlacing reactions. Alternatively, for a PPI network or for a gene expression network we might want to retrieve the connected components and cliques.

In the code below, we determine the connected components in the random graph `g1` using the `connectedComp` function. Then once we know which nodes are connected we ask about paths in the graph, from one node to another.

```
> library("RBGL")
> s1 <- connectedComp(g1)
> s1
```



```

$`1`
[1] "A" "B" "C" "D" "E" "G" "H" "I" "J" "K" "L" "M" "N"
[14] "O"

$`2`
[1] "F"

> sp.between(g1, "A", "O")

$`A:O`
$`A:O`$length
[1] 2

$`A:O`$path_detail
[1] "A" "D" "O"

$`A:O`$length_detail
$`A:O`$length_detail[[1]]
A--D D--O
  1   1

```

We can extract the large connected component using the `subGraph` function and then perform other computations using just that subgraph. In the code below we look for a solution to the minimum cut problem. In graph theory, a cut is a partition of the vertices of a graph into two non-empty but disjoint subsets by cutting the minimum number of edges (for more details see the `minCut` man page).

```

> gsub <- subGraph(s1[[1]], g1)
> minCut(gsub)

$mincut
[1] 1

$S
[1] "B"

$`V-S`
[1] "A" "C" "D" "E" "G" "H" "I" "J" "K" "L" "M" "N" "O"

```

3.2 Graph layout

In Bioconductor, we have separated the process of drawing graphs into two distinct phases. The first is *layout* which involves determining the locations of

the nodes and edges, typically in a two dimensional space. This can then be followed by the process of rendering the graph, which is the actual plotting of the graph. There are two reasons for separating these tasks. First, we would like to use R's graphics capabilities for rendering, so that graphs can easily be intermingled with other plots and so that the full range of R's advanced graphics can be used. Second, we often want to plot graphs based on the same set of nodes (and sometimes edges) multiple times. Being able to control the location of the nodes and edges helps us achieve that goal. The main graph layout algorithms are provided by the **Rgraphviz** package, which is based on the *Graphviz project* [17] and others are available in the **RBGL** package.

A typical workflow of a graph layout is to pass a graph object to the layout function (`layoutGraph`), which returns another graph object containing all the necessary information for subsequent rendering. Figure 1(a) shows the results of the rendering function (with default parameters) used below to represent our random graph `g1`.

```
> g1 <- layoutGraph(g1)
> renderGraph(g1)
```

There is a hierarchy to set rendering parameters for a graph. The levels of this hierarchy are the session, the rendering operations, and the individual nodes and edges. At the session level, defaults parameters are used if not set somewhere further down the hierarchy. For single rendering operation, *i.e.*, a single call to `renderGraph`, defaults can be set using its `graph.pars` argument.

```
> renderGraph(g1,
+             graph.pars=list(edges= list(col= "lightblue",
+                                       lty= 1)))
```

All parameters set using `renderGraph`'s `graph.pars` argument are transient, whereas setting session-wide parameters, using the function `graph.par` will affect all subsequent rendering operations. Figure 1(b) is the result of modifying the `graph.par` as follow:

```
> graph.par(list(nodes = list(fill= "lightgray",
+                             textCol= "blue", cex= 1),
+                 edges=list(col= "red", lty= 2)))
> renderGraph(g1)
```

Parameters for individual nodes or edges in a graph object can also be set using the `nodeRenderInfo` and `edgeRenderInfo` functions. These changes will

be retained in all subsequent layout or rendering operations of that particular graph.

```
> nodeRenderInfo(g1) <- list(fill= c(A= "lightyellow",  
+                                   B= "lightblue"))  
> renderGraph(g1)
```

There are many options for layout type, node color, node shape, edge representation etc., and we refer the reader to the vignette of the **Rgraphviz** package for further details.

4 Case studies

4.1 Case study 1: *B. subtilis* protein-protein interaction data.

In this case study we are interested in *B. subtilis* PPI data. We will focus on physical binary interactions between protein pairs obtained via a yeast 2-hybrid (Y2H) assay [18] and complex co-membership between sets of proteins detected via Affinity Purification - Mass Spectrometry (AP-MS) [19]).

It is worth briefly commenting on the data that arise from these two assays. For Y2H both bait and prey proteins need to be modified. One of the two is modified to contain a DNA binding domain, while the other is modified to contain an activation domain. A typical experiment involves a generally smaller set of bait proteins crossed with a generally larger set of prey. The resulting data are usually processed by the experimenter to be binary and usually only positive results are reported. Thus the data generally consist of a set of bait-prey pairs. For these experiments both the set of baits that were used and the set of prey they were targeted to are known, but are seldom reported. For AP-MS like experiments only the bait is modified and hence the potential set of prey is very large. Each *pull-down* provides data on the set of interactors with the bait protein (the protein that was pulled-down). For these experiments the set of baits is known, but the set of prey is typically the set of expressed proteins in the tissue being assayed.

Curated, PSI-MI 2.5 XML, *B. subtilis* PPI data are available from IntAct (<http://www.ebi.ac.uk/intact/>). The PSI-MI 2.5 XML schema is a recent initiative by the *Protein Standardization Initiative* to formally standardize the way biologists should report molecular interaction data. In Bioconductor, the **RpsiXML** package serves as a programmatic interface between the R statistical environment and the PSI-MI 2.5 XML files. It implements the PSI-MI 2.5 standard and currently supports 8 databases (unfortunately each database has its own implementation of the PSI-MI 2.5 standard). The code chunk below demonstrates how to download and parse the *B. subtilis* PPI data using **RpsiXML**. First, the URL is specified, then that is used to open the connection and download the text and finally that text is parsed to extract the different components.

```

> library("RpsiXML")
> site <- "ftp://ftp.ebi.ac.uk/pub/databases/intact/current/psi25"
> burl <- paste(site, "species/bacsu_small.xml", sep="/")
> bfile <- paste(readLines(url(burl)), collapse= "")
> bacsu <- parsePsimi25Interaction(bfile,
+                                INTACT.PSIMI25,
+                                verbose= FALSE)

```

The R object `bacsu` is of class `psimi25InteractionEntry`. You can access its attributes using the associated methods like `interactions`, that lists each interaction, or `organismName`, that returns the name of the organism (see man page for the complete list of methods).

To explore the data, we might first want to understand what types of interactions are present and in what quantities, as well as how many different experiments are represented. The interaction types refer to the methods used to detect the interactions as stored in the IntAct database like 'Y2H', representing binary physical interactions, or 'pull-down', representing complex co-membership (See the IntAct annotation rules for more details <http://www.ebi.ac.uk/~intact/site/doc/IntActAnnotationRules.pdf>). The code below shows how to tabulate that information.

```

> ii = interactions(bacsu)
> table(sapply(ii, function(x) x@interactionType))

```

2 hybrid	adenylate cyclase
41	1
affinity chrom	anti bait coip
7	1
anti tag coip	confocal microscopy
3	1
crosslink	electron microscopy
1	4
emsa	fluorescence imaging
4	3
fret	light scattering
1	2
protein kinase assay	pull down
5	12
tap	transcription compl
4	3
x-ray diffraction	
4	

```

> table(sapply(ii, function(x) x@expPubMed))

```

10997907	11722727	12065423	12100548	12359875	12644235
1	4	5	3	1	1
12738765	14711369	14993308	15045029	15104138	15362849
1	3	9	4	13	1
15491161	16796675	17662947	17803906	17994626	18854156
4	19	7	8	3	4
19450516	19800110				
1	5				

We can see that 41 of the interactions are from Y2H screens and 12 are from AP-MS experiments. From the second call to `table` we see that 18 different PubMed citations are involved and that the two largest sources of data are 13 interactions from PMID:15104138 and 19 interactions from PMID:16796675.

We next can verify the quality of the downloaded interactions. For instance, we can create a function that test whether Y2H dataset have all their bait and prey information and apply this function to the entire list of interactions.

```
> filterfun <- function(x) {
+   x@interactionType=="2 hybrid" && is.na(x@bait) && is.na(x@prey)
+ }
> table(sapply(ii, filterfun))

FALSE  TRUE
   91    6
```

The result indicates that there may be some issues: 6 out of 41 have missing information.

More generally quality assessment (QA) is an crucial processing step in (any) data analysis and exploration. Experimental data (especially high-throughput) are often biased due to systematic or stochastic errors [20,21]. QA allows identifying biases (and potentially their causes) and minimizing the effects by correcting or removing erroneous data. Several Bioconductor packages offers QA methods specific to a variety technology. For example, the **ppiStats** provides tools for the analysis of Y2H and AP-MS data. The package is specifically designed to summarize graph data, identify systematically biased baits, and estimate stochastic error probabilities [22,23]. In particular, its authors [22] introduced the concepts of viable baits (VBs) and viable prey (VP) to help take into account the problem of untested interactions versus tested but not observed interactions. Experimental results are typically only reported for proteins that are observed in at least one interaction, leaving uncertainty as to which proteins were actually tested. VB is thus the set of bait proteins that successfully detect at least one interaction partner, VP is the set of all proteins detected by at least one bait, and VBP is the set of proteins that are both viable baits and viable

preys. In the following paragraph we will get an overview of the usefulness this package.

Upon reading the PPI data into R, one intuitive exploratory approach is to represent it as a graph. To extract a graph from XML data, **RpsiXML** provides 2 methods: `psimi25XML2Graph` that generates one large graph composed of all the bait-prey data across all experiments while `separateXMLDataByExpt` yields a list of graphs indexed by the PubMed ID. Here we choose the latter as the data originate from many various experimental techniques (mostly low throughput) and it might be misleading to mix them together. For similar reasons, the `separateXMLDataByExpt` function does not mix interaction data based on direct, e.g. binary physical interactions, with that obtained from indirect methods, such as AP-MS. Here we ask for `indirect`, e.g. AP-MS, interactions.

```
> graphs <- separateXMLDataByExpt(bfile, INTACT.PSIMI25,
+                               type="indirect", directed= TRUE,
+                               abstract= TRUE, verbose= FALSE)
> names(graphs)
```

This subset of the PPI data originated from 6 different works. One way of getting a descriptive summary of these experiments is to use the `createSummaryTables` function of the **ppiStats** package. This function takes a list of directed graph objects and creates a summary table based on the directed connectivity, as shown in Table 2.

```
> library("ppiStats")
> directedGraph <- sapply(graphs, isDirected)
> aSummary <- createSummaryTables(graphs[directedGraph])
```

We next consider the paper 16796675 [24] in more detail. First, we can read the abstract information.

```
> abstract(graphs[["16796675"]])

An object of class 'pubMedAbst':
Title: A new FtsZ-interacting protein, YlmF,
       complements the activity of FtsA during
       progression of cell division in Bacillus
       subtilis.
PMID: 16796675
Authors: S Ishikawa, Y Kawai, K Hiramatsu, M
         Kuwano, N Ogasawara
Journal: Mol Microbiol
Date: Jun 2006
```

Then we verify the number and type of interaction reported.

```
> Ishikawa <- sapply(ii, function(x) x@expPubMed == 16796675)
> sum(Ishikawa)

[1] 19

> IshikawaInts <- ii[Ishikawa]
> table(sapply(IshikawaInts, function(x) x@interactionType))

 2 hybrid pull down
 8          11
```

They report 19 interactions, 11 detected by pull down (most likely AP-MS technology) and 8 detected via Y2H. We next extract the graph for their data from the `graphs` object. Recall that in creating this we only accessed `indirect` interactions, so only the AP-MS-like interactions are represented.

```
> IshikawaGraph <- graphs$`16796675`
> IshikawaGraph

A graphNEL graph with directed edges
Number of Nodes = 15
Number of Edges = 32
[1] "psimi25Graph"
attr(,"package")
[1] "RpsiXML"
```

The resulting graph, `IshikawaGraph`, is a directed graph composed of 15 nodes (representing proteins) and 32 edges (representing observed indirect interactions). The IntAct data reported 11 interactions, but recall that each such interaction is between one bait and one or more prey. The nodes in the graph correspond to the unique set of proteins detected as either bait or prey, while the edges indicate which baits detected which prey. Thus, there were 32 interactions detected among 15 different proteins.

In a biological network it is often useful to know the number of interactors associated with a node. This information can be used for a variety of purposes, such as monitoring flux between components of the network, characterizing protein complexes, or in assessing the quality of proteomics bait-prey technologies by modeling experimental errors on node degree [25]. This corresponds to computing the degree of the nodes in the graph. Since the graph is directed the in-degree and out-degree are reported separately.

```
> degree(IshikawaGraph)

$inDegree
031728 034894 P17865 P28264 P37469 P94542 P54420 P33166
      4      4      4      4      0      0      2      2
P45694 P80868 P21465 P39751 P06574 P24327 P06567
      2      2      2      1      2      2      1

$outDegree
031728 034894 P17865 P28264 P37469 P94542 P54420 P33166
      3      3      3      11      8      4      0      0
P45694 P80868 P21465 P39751 P06574 P24327 P06567
      0      0      0      0      0      0      0      0
```

Two proteins, P28264 and P37469 have large out-degrees, 11 and 8 respectively. However these relationships are not reciprocated (the in-degrees are 4 and 0, respectively). In fact, in PPI experiments a major issue is that not all proteins are employed as both bait and prey and we cannot observe interactions that were not tested. In the Ishikawa experiment only four proteins were observed as both bait and prey (see [22] for terminology).

```
> idViableProteins(IshikawaGraph)$VBP

[1] "031728" "034894" "P17865" "P28264"
```

The `assessSymmetry` function from the **ppiStats** package can be used to test the symmetry of your data by calculating the reciprocated degree (nr), the nonreciprocating out (no) and in (ni) degrees. Using this information and a Binomial error model you can assess the p -value for the in and out degree of each protein [25].

In protein interaction graphs, an other question of special interest is the identification of functionally related modules (*e.g.*, protein complexes). In graph analysis this might be achieved by searching for cliques. A clique is a complete subgraph, *i.e.*, there is an edge between every pair of vertexes. Here we search for the maximum clique, *i.e.*, the largest subgraph(s), using the `maxClique` function implemented in **RBGL**. Though this algorithm needs to be applied on an undirected graph. We thus extract the underlying structure of `IshikawaGraph` using the `ugraph` function from **igraph**.

```
> library("RBGL")
> xu <- ugraph(IshikawaGraph)
> cls <- maxClique(xu)$maxCliques
```



```

> cs <- sapply(cls, length)
> maxC <- cls[cs== max(cs)]
> maxC

[[1]]
[1] "P28264" "034894" "P17865" "031728" "P94542"

```

We identify a subgraph that contains the four VBPs plus one additional protein. Figure 2 illustrates the result of the maximum clique algorithm. We do note that a clique is a very extreme notion of a cohesive subgroup and that there are many others that might be more suitable when the data are measured with error, as is the case for most PPI experiments. In particular we note that **RBGL** provides implementations of *k-cliques*, *k-cores* and many other concepts described in the social network literature [26]. Also, [27] consider this problem and provide both an algorithm and an implementation via the **apComplex** package. Figure 2 Panel (a) displays all the interactions observed by [24] using the AP-MS technology. Panel (b) renders only the maximum clique. Note, that P9542 only has out edges indicating it was a bait. Perhaps the next step in an analysis would be to identify and understand the roles and function of the proteins in the clique. The protein interaction data are recorded using UNIPROT IDs but most papers will use common names, or symbols, so we need some way to navigate between these. For many important organisms the Bioconductor project offers database-oriented packages that provide maps between the different identifiers used for that organism. In addition one can rely on external resources such as BioMart [28] which can be accessed from within R via the **biomaRt** package.

Every analysis with **biomaRt** starts with selecting a BioMart database and one of the associated datasets (see the **biomaRt** vignette for details). Here, we query the "uniprot-mart" database and its only available dataset, "UNIPROT", with the 5 selected proteins. We define the list of filters and attributes we are interested in and use the **getBM** to retrieve annotation information we would like.

```

> library("biomaRt")
> uniprot <- useMart("uniprot_mart")
> uniprot <- useDataset("UNIPROT",mart=uniprot)
> filters <- listFilters(uniprot)
> attributes <- listAttributes(uniprot)
> attributes[c(1:4,23), ]
> selectedProt <- maxC[[1]]
> getBM(attributes = c(attributes[c(1:4, 23), 1]),
+   filters = "accession", values = selectedProt, mart = uniprot)

```

The name and comments columns of the results show us that the selected proteins are indeed tightly related physically or functionally. The proteins 031728

(*sepF*), P17865 (*ftsZ*) and P28264 (*ftsA*) are part of the divisome complex (or *FtsZ complex*), while P94542 (*zapA*) is a divisome activator and O34894 (*ezrA*) is a divisome inhibitor (Figure 2b). We also remark that the protein *sepF* corresponds to the gene name *YlmF* used in [24].

4.2 Case Study 2: Response of *Escherichia coli* during an oxygen shift

In this study case, the data consist of 43 microarrays for 6 mutants in two conditions aerobic and anaerobic. The mutants are knockout strains of 5 key transcriptional regulators of the oxygen response network: $\Delta arcA$, $\Delta appY$, Δfnr , $\Delta oxyR$, $\Delta soxS$ and the double knockout $\Delta arcA\Delta fnr$. These data are part a genome-scale analysis [29]. We restrict our attention to the effect of the single transcription factor knockout data in aerobic conditions. We realize that a comparison of aerobic to anaerobic conditions would be of some importance, but such an analysis would be well beyond the scope of this chapter, where our primary goal is to inform the reader of the available tools for graph analysis and to demonstrate their use in a setting that is not especially complex. Even with that simplification, our analysis is incomplete and quite cursory. However, the reader could easily extend our example in many interesting directions.

First, we consider a model that will help to identify the set of genes that are regulated by our transcription factors. The nodes in our network will be the genes in *E. coli* that were assayed. We define an edge to exist between one of the five transcription factors listed above and any other gene if the gene expression between the wild type condition and the condition with the transcription factor knocked out is significantly different. Expression of the second gene can either increase (in which case the TF was likely a repressor) or it can decrease (in which case the TF was likely an enhancer). We note that the biological effect need not to be direct; that is, there is no reason to assume that the TF which acts in *cis*, could not affect an intermediate regulator of a second gene. One could determine which TFs are acting in *cis* by using either known TF binding sites or by performing a ChIP-seq experiment.

As a second type of analysis we use the genes identified by the knock-out experiments to identify cellular modules regulated by those transcription factors. We used the biological process component of GO [15] and KEGG pathways (www.genome.jp/kegg/) to identify the candidate pathways and sets of genes for this part of the analysis.

To get started, we create a vector with the names of the KO genes.

```
> KOgene <- c("arcA", "appY", "oxyR", "soxS", "fnr")
```

Next we load the expression data that we formatted from the online supplementary information of *Covert et al.* [29] we downloaded at (<http://www.nature.com/nature/journal/v429/n6987/supinfo/nature02456.html>). *Covert et al.* provided expression estimates for each probe based on dChip [30]. This is

slightly problematic for our purposes since dChip can give negative expression values, so we renormalized the data using the VSN algorithm [31] from the **vs**n package. The normalized data are provided on the web page that accompanies this article (<http://www.bioconductor.org/pub/MiMB>). The data were saved in a *.rda file (specific binary format for external representation of R objects). Once downloaded from the and can be reloaded in R using the function `load` and the location of the file (*e.g.*, `load(path/to/bES.rda)`).

```
> load("data/bES.rda")
> bES

ExpressionSet (storageMode: lockedEnvironment)
assayData: 4345 features, 37 samples
  element names: exprs
phenoData
  sampleNames: aerobic_wild_rep1, aerobic_wild_rep2, ...
, anaerobic_soxS_rep3 (37 total)
  varLabels and varMetadata description:
    strain:
    growth:
    genotype:
    name:
featureData
  featureNames: thrL_b0001_at, thrA_b0002_at, ..., lasT_
b4403_at (4345 total)
  fvarLabels and fvarMetadata description:
    genename: NA
experimentData: use 'experimentData(object)'
Annotation:

> pData(bES)[1:5,1:3]

      strain growth genotype
aerobic_wild_rep1 wild type aerobic      wt
aerobic_wild_rep2 wild type aerobic      wt
aerobic_wild_rep3 wild type aerobic      wt
aerobic_appY_rep1  mutant aerobic  appY
aerobic_appY_rep2  mutant aerobic  appY
```

The object `bES` is an object of class `ExpressionSet`, a container that holds both gene expression data and the corresponding, per sample, metadata.

Statistical testing is essential to evaluate contrasts between experimental conditions. Since there are relatively few arrays we use the **limma** package to test for differentially expressed genes between KO and wild-type strains.

Using the `bES` data set, we first subset the data so that only the arrays

corresponding to the aerobic condition are retained (in code not shown we have also removed the double knock-out arrays).

```
> library("limma")
> aerobia <- bES[, bES$growth=="aerobic"]
```

Every **limma** analysis starts with the definition of a design matrix, which we define below. A design matrix is a matrix of explanatory variables which here allows you to formalize your experimental design (observed data) and biological question for statistical analysis. In our example, it describes which samples have been applied to each array. Complex experimental design implies complex definition of design matrix, requiring some statistical knowledge (for more details we refer the reader to the well-documented vignette of the (limma) package).

```
> aGeno <- as.factor(pData(aerobia)$genotype)
> aDesign <- model.matrix(~ 0 + aGeno)
> colnames(aDesign) <- levels(aGeno)
```

In the following code we fit the model, define the contrast to test, and apply the moderated empirical Bayes test with Benjamini Hochberg p -value adjustment. This last step is to adjust for multiple testing concerns.

```
> afit <- lmFit(aerobia, aDesign)
> a.cont.matrix <- makeContrasts("appY"=appY-wt,
+                               "arcA"=arcA-wt,
+                               "fnr"=fnr-wt,
+                               "oxyR"=oxyR-wt,
+                               "soxS"=soxS-wt, levels=aDesign)
> afit2 <- contrasts.fit(afit, a.cont.matrix)
> afit2 <- eBayes(afit2)
> aRes <- c()
> for(i in 1:ncol(a.cont.matrix)){
+   aDE <- topTable(afit2, coef= i, n=500, adjust="BH",
+                   p.value=0.001)
+   intRes <- cbind(colnames(a.cont.matrix)[i],
+                   aDE$genename , round(aDE$logFC, 2))
+   aRes <- rbind(aRes, intRes)
+ }
> colnames(aRes) <- c("KO", "targets", "logFC")
```

We select those genes (probes) with p -value ≤ 0.001 . The number of significantly differentially expressed genes for each TF KO condition can be obtained

using the code shown next.

```
> table(aRes[,1])  
  
appY arcA fnr oxyR soxS  
33    1   65 147    1
```

We note that three of the TFs have a large number of targets identified, while two (*arcA* and *soxS*) have only one target each. This may be due to the rather stringent *p*-value threshold we used or the biological functions of those genes that depend on the oxygen condition. Another interesting feature of the data is that the transcription factor KOs $\Delta appY$, Δfnr , $\Delta oxyR$ seem to have similar effects on fairly large sets of their targets suggesting that perhaps they affect similar sets of intermediate (Figure 3).

We can display the graph of these interactions using the **graph** and **Rgraphviz** packages. The type of graph that is most helpful is one that is essentially a bipartite graph. A bipartite graph is a graph where nodes can be separated into two sets and all edges go from a member of one set to a member of the other set. The mappings between GO terms and genes form a bipartite graph, as do the mappings between KEGG pathways and genes. In this case the mapping is from the five candidate TFs to their targets. But it is not really a bipartite graph since the TFs can target themselves (self-loops). However, as you will see, rendering the graph yields an essentially bipartite graph view. There are some initial efforts to build specialized tools for bipartite graphs, or the equivalent representation as *hypergraphs* in the **hypergraph** package, but much remains to be done.

In practical terms we will construct this graph using the information from **limma** which tells us both the TFs and their targets. This is what can be referred to as a *from-to* representation of a graph and we use the function **ftM2graphNEL** to construct an instance of the *graphNEL* class. In the call we specify that the log fold change should be included in the graph as edge weights.

```
> graphAERO <- ftM2graphNEL(aRes[, c("KO","targets")],  
+                           w=as.numeric(aRes[, "logFC"]),  
+                           edgemode="directed")
```

We remove self-loops and thus the *arcA* node, which only regulates itself. The fact that *arcA* only regulates itself is intriguing. However *ArcA* is known to act primarily as a negative transcriptional regulator under anaerobic conditions thus might not be expressed (or at very low level) in aerobic condition, explaining its auto-regulation and the mild effect of its deletion [32].

```
> aeroG <- removeSelfLoops(graphAERO)
> aeroG <- removeNode("arcA", aeroG)
```

We use the fold change between KO and wild type to color the edges (orange means higher expression in KO and blue means higher expression in wild type). In the code below we demonstrate how to layout this graph using the tools in R. First, the graph is laid out, then we determine which of the edges correspond to positive log FC and which to negative. We then construct a vector named `col` that contains the value "orange" for those edges that are higher in KO, and "blue" otherwise. We apply that information to the graph using the `edgeRenderInfo` function. Finally, for readability we remove the labels from the TF targets using the `nodeRenderInfo` function and we render the graph.

```
> aeroG <- layoutGraph(aeroG, layoutType="dot")
> col <- c(rep("orange", numEdges(aeroG)))
> names(col) <- edgeNames(aeroG)
> low <- unlist(lapply(edgeWeights(aeroG), function(x) x < 0))
> col[low] <- "blue"
> edgeRenderInfo(aeroG) <- list(col=col)
> KOlabel <- nodes(aeroG)%in%KOgene
> nodeRenderInfo(aeroG)$labelY[!KOlabel] <- NA
> renderGraph(aeroG)
```

Somewhat surprisingly, there is a preponderance of orange edges and indeed 158 genes have higher expression in the KO strains while only 85 have lower expression levels. Suggesting that these TFs were largely repressing gene expression, or perhaps that their deletion caused other systems to up-regulate. Had these TFs been largely activating transcription then we would have seen blue edges, showing that their deletion corresponded to a decrease in expression.

To better understand this, we turned to the literature and found evidence that these TFs have dual roles. *fnr* is the primary transcriptional regulator mediating the transition from aerobic to anaerobic growth through the regulation of hundreds of genes. Generally speaking, this protein activates genes involved in anaerobic metabolism and represses genes involved in aerobic metabolism which may partly explained the up-regulation observed in aerobic conditions [33]. In *E.coli*, *oxyR* was until now mostly described as an activator (except for the *flu* gene [34]) but it has recently been characterized as a repressor in several other bacteria (such as *Neisseria gonorrhoeae* and *Deinococcus radiodurans* [35]). *appY* induces the expression of energy metabolism genes under anaerobiosis, stationary phase, and phosphate starvation [36]. Finally, *soxS* is a transcriptional activator of oxidative stress genes, which might explain that in physiological aerobic condition its depletion has a minor effect.

Of course we can not rule out the possibility that there was somewhat more variation in some of the experiments than in others. This would make it harder to detect differential expression under some conditions and hence could reduce the target set. A thorough analysis would engage this idea and ensure that there were no obvious differences between the conditions.

We can also use KEGG and GO categories to try to better understand whether there are particular pathways or processes that are affected. This analysis consists of testing for over-representation of categories among genes in a selected gene set based upon the Hypergeometric distribution [37]. To this aim, we make use of the methods in the **GOSTats** and **Category** packages. To perform this test we need the Entrez Gene identifiers of each *E. coli* gene on the array and in each set selected from the **limma** analysis. The **org.EcK12.eg.db** annotation package allows such mapping and more, providing various annotations such as GO categories, REFSEQ or PMID. To map *E. coli* gene name symbols to their Entrez Gene Identifiers we first select the gene name from the **bES ExpressionSet** and get the unique Entrez Gene Identifiers. We apply the same procedure to the genes associated with each KO and perform the GO and KEGG **hyperGTest** function to search for over-representation with a *p*-value cut-off of 0.001. We do note that this approach, while widely used is slightly problematic. Among the many issues that arise is the fact that we are combining genes that are both up-regulated and down-regulated by the TF KO. First we consider the results for the GO category analysis. We restricted our attention to the biological process (BP) ontology. Very similar categories are identified for *fnr*, *oxyR*, and *appY*. The set of GO categories for *fnr* was a subset of those identified for *appY*, with GO:0006066 being associated with *appY* but not *fnr*. Similarly the set of GO categories for *appY* was essentially a subset of those for *oxyR*, with GO:0006066 and GO:0006725 being associated with *appY* but not *oxyR*. These GO categories are labeled *alcohol metabolic process* and *cellular aromatic compound metabolic process* respectively. We list the complete set of GO categories for *oxyR* using the code below.

```
> sigGO <- lapply(GO, sigCategories)
> sapply(sigGO, length)

arca appY oxyR soxS fnr
    0   10   12    0    9

> lapply(GO["oxyR"], function(x) summary(x)[,c(2,3,7)])

$oxyR
      Pvalue OddsRatio
1 2.158614e-17 72.053571
2 2.158614e-17 72.053571
3 2.810665e-13 23.839286
4 6.354012e-11 14.196429
5 4.449673e-09      Inf
```

```

6 4.449673e-09      Inf
7 4.449673e-09      Inf
8 3.379156e-07 49.753846
9 1.452331e-04 5.043394
10 3.069493e-04 19.283582
11 3.069493e-04 19.283582
12 9.962969e-04 3.042772

                                Term
1                                iron ion transport
2                                metal ion transport
3                                ion transport
4                                establishment of localization
5                                enterobactin biosynthetic process
6                                catechol metabolic process
7                                siderophore biosynthetic process
8                                peptide biosynthetic process
9                                aerobic respiration
10                               thiamin biosynthetic process
11 thiamin and derivative metabolic process
12                               anaerobic respiration

```

Next we consider the results for the KEGG pathways.

```

> sigKEGG <- lapply(KEGG, sigCategories)
> sapply(sigKEGG, length)

arcA appY oxyR soxS fnr
    0    1    4    0    1

```

We see that there four significant KEGG pathways for *oxyR* and one for each of *appY* and *fnr*. All 3 TFs share the one significant pathway, *Biosynthesis of siderophore group nonribosomal peptides*, which is in accordance with the GO categories identified. We can then return to the original data to get a sense of which of the TFs had genes involved in this pathway.

```

> gnodes <- nodes(graphAERO)
> geneInKEGG <- geneIdsByCategory(KEGG[[2]])$`01053`
> symbol <- unlist(mget(geneInKEGG,
+                      org.Eck12.egSYMBOL, ifnotfound=NA))
> siderophore <- gnodes[gnodes%in%symbol]
> # level of differentially expressed gene
> as.data.frame(aRes[aRes[,2]%in%siderophore,])

```


	KO	targets	logFC
1	appY	entC	2.16
2	appY	entE	2.06
3	appY	entB	2.09
4	appY	entF	2.36
5	fnr	entC	2.72
6	fnr	entE	2.6
7	fnr	entB	2.56
8	fnr	entF	2.93
9	oxyR	entC	3.1
10	oxyR	entE	2.9
11	oxyR	entB	2.91
12	oxyR	entF	3.43

We note that several genes involved in iron metabolism are up regulated between WT and KO. And indeed it seems that KOs of *appY*, *fnr*, and *oxyR* seem to have similar effects on a largely overlapping set of genes. Microorganisms growing under aerobic conditions need iron for a variety of functions including reduction of oxygen for synthesis of ATP, reduction of ribotide precursors of DNA, for formation of heme, and for other essential purposes. Accumulation of the intracellular siderophore has also been shown to be up-regulated by oxidative stress in *Aspergillus nidulans* [38], which underscores the intertwining of iron metabolism and oxidative stress.

5 Discussion

We have provided a fairly brief and concise overview of how some of the many Bioconductor packages can be used to help analyze network data. The tools are quite varied, and certainly much remains to be done, but most of the building blocks needed for a comprehensive tool suite now exist and can fairly easily be used to accomplish complex analyses.

We believe that the current capabilities are extensive, but that perhaps the most compelling aspect of these tools is the ease with which they can be extended and adapted to different analysis problems. Interested readers can obtain our data and scripts from the accompanying web-site. They can verify our results, perhaps point out errors or bugs in our code, and then go on to carry out much more detailed analyses of these. We encourage them to do so.

Acknowledgments

Funding for this research was provided by grants from La Ligue Contre Le Cancer (RAB08008NSA; project R08010NS) . We also gratefully acknowledge the Institut national de la santé et de la recherche médicale (INSERM) for supporting this project. Funding for RG was provided by NIH grant P41 HG004059.

The authors would like to thank Wolfgang Huber, Tony Chiang, Shailesh Date and Denise Scholtens for helpful comments and for providing a stimulating research environment that has led to the many tools used here and to the ideas that underlie this research.

References

1. R Development Core Team: *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria 2009, [<http://www.R-project.org>]. [ISBN 3-900051-07-0].
2. Huber W, Carey VJ, Long L, Falcon S, Gentleman R: **Graphs in molecular biology**. *BMC Bioinformatics* 2007, **8**(6):S8.
3. Castelo R, Roverato A: **Reverse engineering molecular regulatory networks from microarray data with qp-graphs**. *J Comput Biol.* 2009, **16**(2):213–227.
4. Le Meur N, Gentleman R: **Modeling synthetic lethality**. *Genome Biol.* 2008, **9**(9):R135.
5. Csardi G, Nepusz T: **The igraph software package for complex network research**. *InterJournal* 2006, **Complex Systems**:1695.
6. Shannon P, Markiel A, Ozier O, Baliga N, Wang J, Ramage D, Amin N, Schwikowski B, Ideker T: **Cytoscape: a software environment for integrated models of biomolecular interaction networks**. *Genome research* 2003, **13**(11):2498.
7. Leisch F: **Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis**. In *Compstat 2002 — Proceedings in Computational Statistics*. Edited by Härdle W, Rönz B, Physika Verlag, Heidelberg, Germany 2002:575–580, [<http://www.ci.tuwien.ac.at/~leisch/Sweave>]. [ISBN 3-7908-1517-9].
8. Venables WN, Ripley BD: *Modern Applied Statistics with S (4e)*. New York: Springer 2002.
9. Gentleman R: *R Programming for Bioinformatics*. CRC Press 2008.
10. Chambers JM: *Software for Data Analysis: Programming with R*. New York: Springer 2008.
11. Hahne F, Huber W, Gentleman R, Falcon S: *Bioconductor Case Studies*. New York: Springer 2008.
12. Boutros M, Bras L, Huber W: **Analysis of cell-based RNAi screens**. *Genome Biol.* 2006, **7**(7):R66.
13. Hahne F, Le Meur N, Brinkman R, Ellis B, Haaland P, Sarkar D, Spidlen J, Strain E, Gentleman R: **flowCore: a Bioconductor package for high throughput flow cytometry**. *BMC Bioinformatics* 2009, **10**:106.
14. Morgan M, Anders S, Lawrence M, Aboyoun P, Pagès H, Gentleman R: **ShortRead: a bioconductor package for input, quality assessment and exploration of high-throughput sequence data**. *Bioinformatics* 2009, **25**(19):2607–8.

15. The Gene Ontology Consortium: **Gene ontology: tool for the unification of biology.** *Nat. Genet.* 2000, **25**:25–9.
16. Siek JG, Lee LQ, Lumsdaine A: *The Boost Graph Library*. Boston: Addison Wesley 2002.
17. Ellson J, Gansner E, Koutsofios E, North S, Woodhull G: **Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools.** In *Graph Drawing Software*. Edited by Junger M, Mutzel P, Springer-Verlag 2004:127–148.
18. Brückner A, Polge C, Lentze N, Auerbach D, Schlattner U: **Yeast two-hybrid, a powerful tool for systems biology.** *Int J Mol Sci.* 2009, **10**(6):2763–88.
19. Wingren C, James P, Borrebaeck C: **Strategy for surveying the proteome using affinity proteomics and mass spectrometry.** *Proteomics* 2009, **9**(6):1511–7.
20. Schuchhardt J, Beule D, Malik A, Wolski E, Eickhoff H, Lehrach H, Herzelt H: **Normalization strategies for cDNA microarrays.** *Nucleic Acids Res.* 2000, **28**(10):E47.
21. Gentleman R, Huber W: **Making the most of high-throughput protein-interaction data.** *Genome Biol.* 2007, **8**(10):112.
22. Chiang T, Scholtens D, Sarkar D, Gentleman R, Huber W: **Coverage and error models of protein-protein interaction data by directed graph analysis.** *Genome Biol.* 2007, **8**(9):R186.
23. Chiang T, Scholtens D: **A general pipeline for quality and statistical assessment of protein interaction data using R and Bioconductor.** *Nat Protoc.* 2009, **4**(4):535–46.
24. Ishikawa S, Kawai Y, Hiramatsu K, Kuwano M, Ogasawara N: **A new FtsZ-interacting protein, YlmF, complements the activity of FtsA during progression of cell division in Bacillus subtilis.** *Mol Microbiol.* 2006, **60**(6):1364–80.
25. Scholtens D, Chiang T, Huber W, Gentleman R: **Estimating node degree in bait-prey graphs.** *Bioinformatics* 2008, **24**(2):218–24.
26. Wasserman S, Faust K: *Social Network Analysis, Methods and Applications*. Cambridge: Cambridge University Press 1994.
27. Scholtens D, Vidal M, Gentleman R: **Local dynamic modeling of global interactome networks.** *Bioinformatics* 2005, **21**:3548–3557.
28. Haider S, Ballester B, Smedley D, Zhang J, Rice P, Kasprzyk A: **BioMart Central Portal—unified access to biological data.** *Nucleic Acids Res.* 2009, :W23–7.

29. Covert M, Knight E, Reed J, Herrgard M, Palsson B: **Integrating high-throughput and computational data elucidates bacterial networks.** *Nature* 2004, **429**(6987):92–6.
30. Li C, Wong WH: **Model-based analysis of oligonucleotide arrays: Expression index computation and outlier detection.** *Proceedings of the National Academy of Sciences USA* 2001, **98**:31–36.
31. Huber W, von Heydebreck A, Sueltmann H, Poustka A, Vingron M: **Variance Stabilization Applied to Microarray Data Calibration and to the Quantification of Differential Expression.** *Bioinformatics* 2002, **18 Suppl. 1**:S96–S104.
32. Iuchi S, Lin E: **arcA(dye), a global regulatory gene in Escherichia coli mediating repression of enzymes in aerobic pathways.** *Proceedings of the National Academy of Sciences, USA* 1988, **85**(6):1888–92.
33. Salmon K, Hung S, Mekjian K, Baldi P, Hatfield G, Gunsalus R: **Global gene expression profiling in Escherichia coli K12. The effects of oxygen availability and FNR.** *Journal of Biological Chemistry* 2003, **278**(32):29837–55.
34. Correnti J, Munster V, Chan T, Woude M: **Dam-dependent phase variation of Ag 43 in Escherichia coli is altered in a seqA mutant.** *Molecular Microbiology* 2002, **44**(2):521–32.
35. Chen H, Xu G, Zhao Y, Tian B, Lu H, Yu X, Xu Z, Ying N, Hu S, Hua Y: **A novel OxyR sensor and regulator of hydrogen peroxide stress with one cysteine residue in Deinococcus radiodurans.** *PLoS ONE* 2008, **3**(2):e1602.
36. Brondsted L, Atlung T: **Effect of growth conditions on expression of the acid phosphatase (cyx-appA) operon and the appY gene, which encodes a transcriptional activator of Escherichia coli.** *Journal of bacteriology* 1996, **178**(6):1556.
37. Falcon S, Gentleman R: **Using GOstats to test gene lists for GO term association.** *Bioinformatics* 2007, **23**(2):257–258.
38. Eisendle M, Schrettl M, Kragl C, Muller D, Illmer P, Haas H: **The intracellular siderophore ferricrocin is involved in iron storage, oxidative-stress resistance, germination, and sexual development in Aspergillus nidulans.** *Eukaryotic Cell* 2006, **5**(10):1596.

Figure captions

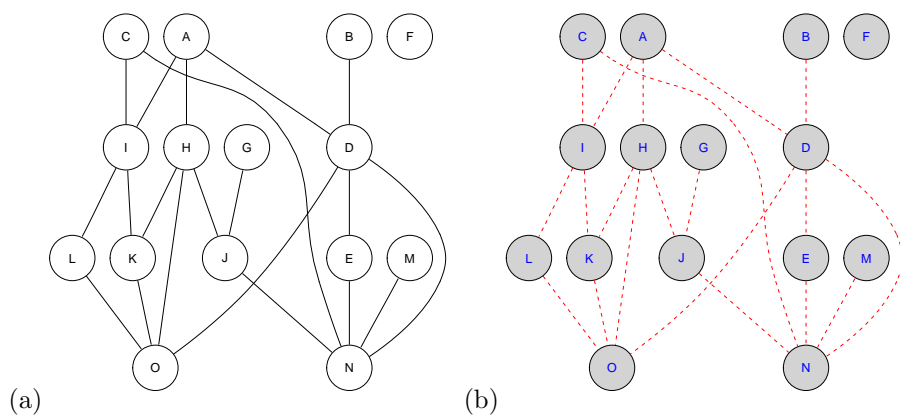


Figure 1: Flexible graph rendering with R. The graph was generated by the **graph** package. In panel (a) the default parameters of the `layoutGraph` and `renderGraph` functions were used. In panel (b) the node and edge parameters were modified using the `graph.par` function. This affected the node color, line color and type etc.

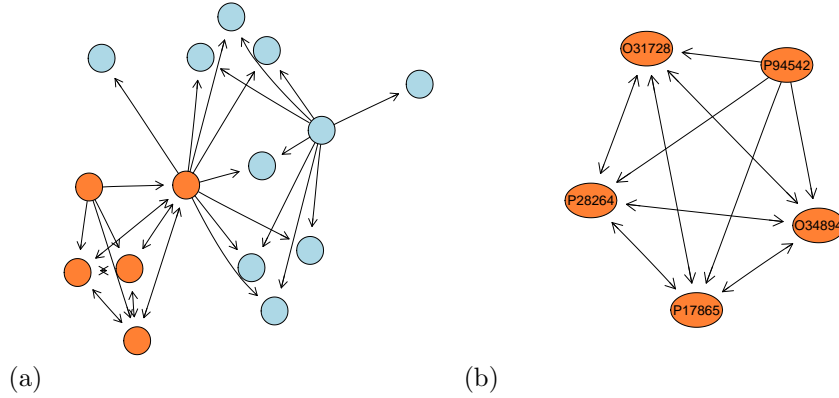


Figure 2: Graphs of interacting proteins in *B. subtilis*, during cell division, as reported by [24]. Only those interactions detected by AP-MS are rendered here. Panel a) shows all interactions and their directions (the arrows go from a bait protein to a prey protein). In orange is the maximum clique found by the **maxClique** function of the **RBGL** package. In panel (b) only the maximal clique is rendered. The proteins O31728, P17865 and P28264 are part of the divisome complex; P94542 is a divisome activator; and O34894 is a divisome inhibitor.

Tables

	VB	VP	VBP	VBP/VB	VP/VB	TI	TI/VB
12100548	1	1	0	0.00	1.00	1	1.00
15045029	4	22	3	0.75	5.50	57	14.25
15491161	3	3	2	0.67	1.00	3	1.00
16796675	6	13	4	0.67	2.17	32	5.33
17994626	2	2	1	0.50	1.00	2	1.00
19450516	1	1	0	0.00	1.00	1	1.00

Table 2: A overview of summary statistics for all AP-MS data sets. Each row displays the result reported in one paper (PMID). (VB) viable baits, (VP) viable prey, (VBP) viable bait/prey, (TI) the total number of directed interactions. The statistics within this table can differentiate between experiment type. For example, the last column details the number of interactions per VB; the experiment reported in paper 15045029 and 16796675 have remarkably high numbers of average interactions per VB, and so would produce markedly distinct features (i.e. protein complexes) from the others.

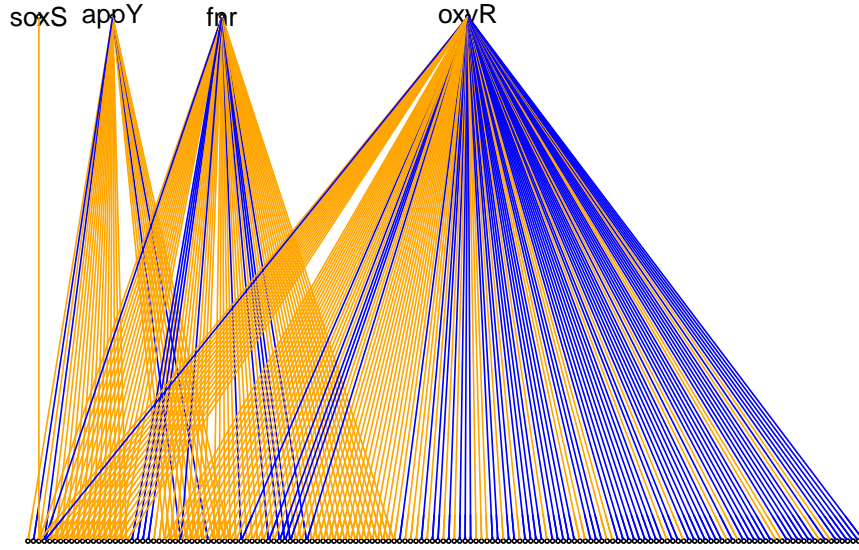


Figure 3: Unbalanced effect of the deletion of *Escherichia coli* key transcriptional regulators of the oxygen response, in aerobic condition. In this figure the TF knock-outs (top) are linked to their targets (bottom). Self-loops have been removed (auto-regulation of the TFs) as well as the node for the TF *arcA*, which only regulates itself and affects no other gene in the aerobic conditions. The edges are colored according to whether there is higher expression in WT than in KO (blue) or higher expression in KO than WT (orange). There are a surprising number of orange edges. We can also see the high level of overlap in targets between TFs.

Table 1: **Bioconductor’s graph-related packages.** Packages used in this chapter.

Package	Description	Type
graph	basic class definitions and graph handling capabilities	infrastructure
Rgraphviz	interfaces R with the AT and T Graphviz library for plotting R graph objects from the graph package	infrastructure
RBGL	interface to the graph algorithms contained in the <i>BOOST</i> library	infrastructure
RpsiXML	R interface to PSI-MI 2.5 files	infrastructure
ppiStats	tools for the analysis of protein interaction data	application
limma	Data analysis, linear models and differential expression for microarray data	application
biomaRt	interface to the BioMart software suite	annotation
GOstats	basic manipulation tools for graphs, hypothesis testing and other simple calculations on Gene Ontology.	application
Category	statistical methods for performing category analysis.	application
org.EcK12.eg.db	genome wide annotation for <i>E. coli</i> strain K12, primarily based on mapping using ENTREZ GENE identifiers from NCBI	annotation