Chapter 1

Tutorials on Agent-based modelling with NetLogo and Network Analysis with Pajek

Matthew J. Berryman[1] and Simon D. Angus[2]

[1]*Defence and Systems Institute, SPRI Building, Mawson Lakes Campus, University of South Australia, Mawson Lakes SA 5095, Australia; Email:* matthew.berryman@unisa.edu.au

[2]*Department of Economics, Monash University, Clayton Vic. 3800, Australia; Email:* simon.angus@buseco.monash.edu.au

Complex adaptive systems typically contain multiple, heterogeneous agents, with non-trivial interactions. They tend to produce emergent (larger-scale) phenomena. Agent-based modelling allows one to readily capture the behaviour of a group of heterogeneous agents (such as people, animals, *et cetera*), with diverse behaviour and important interactions, so it is a natural fit to modelling complex systems. Many complex systems (and agent-based models thereof) can be thought of as containing networks, either explicitly or implicitly. Therefore for complex systems research it is important to have a good understanding of network analysis techniques. This chapter is aimed at beginners to complex systems modelling and network analysis, using NetLogo (Section 1.1) and Pajek (Section 1.2) respectively. It is also aimed at more advanced complex systems modellers who want an introduction to these platforms.

1.1. Agent Based Modelling

1.1.1. *Introduction*

Agent-based modelling is a type of modelling in which the focus is on representing agents (such as people or animals) and their interactions.[1] In agent-based models, as in real complex systems, a set of inductively generated local rules and behaviours of agents give rise to emergent phenomena at a group or system-wide level.[2–4] Agent-based modelling allows one to effectively capture a very rich set of complex behaviours and interactions,

and is therefore highly suited to modelling complex phenomena. It has gained extensive use in the fields of economics,[5] social science,[6] ecology,[7] and biology,[8] amongst many others.

Agent-based modelling is a useful complement to more traditional model representation using a system of equations. Agent-based modelling more easily allows one to do things that are difficult to capture in traditional approaches, such as:

- Model networks of agents where the agents modify the network dynamically;
- Model agent learning and/or evolution;
- Capture a large range of different types of agents and agent behaviour; and
- Explore non-linear interactions between agents.

Agent-based modelling allows for a much richer set of behaviours than traditional variable-based modelling,[9] even if the latter is aided by the use of computers.[1,10] Agent-based modelling does not preclude the use of other styles of modelling as part of the agent-based model, or in combination with the agent-based model.

Emergent phenomena are those that arise from the low-level rules and interactions between the parts (which may be agents).[3] Sometimes the only way to observe them is to let the model run. Brock (2000) discusses the Santa Fe approach to complex systems science as stressing the identification of patterns at macro levels and trying to reproduce these using lower level rules. Agent-based models can be used to test an hypothesis in the sense that it can rule some emergent possibilities out, given a set of rules (this is discussed further in Subsection 1.1.5). Exploratory agent-based model usage, as an inductive process[a], can generate new hypotheses, but ultimately these must be tested with other methods. As with any type of scientific modelling, it is impossible to conclusively rule hypotheses as true,[11,12] rather one tests the hypothesis by trying to prove its predictions false.[12,13] To quote Blaug (1992)[14] on this logical asymmetry, 'there is no logic of proof, but there is a logic of disproof.' Falsification is a crucial element in the demarcation of science from non-science.[12]

Validation of a model's output is not sufficient, one must also verify

[a]As discussed by Epstein (2007),[11] agent-based models in and of themselves, are a form of logical deduction: given a set of rules, and initial conditions, the emergent outcomes are embedded in the rules,[4] however surprising these may be. The models remain inductive generalisations, however.

the assumptions made.[15] In the case of agent-based modelling, both the behaviours of agents and their software implementations must be verified. The following subsection describes how to implement agent-based models in software using the NetLogo package.

1.1.2. *Introduction to NetLogo*

To assist in the development of agent-based models, a number of different platforms have been developed.[16] These platforms vary in how much support they provide. Some, like Swarm and Mason, offer a set of software libraries to be used in programming a model. RePast, in its Simphony (sic) version, offers a few more tools for quick construction of agent-based models. Some other agent-based modelling platforms provide fixed sets of rules that can be used with some chosen parameters, but these are often too restricted to capture the wide range of phenomena that one might want to model. NetLogo consists of a programming language (derived from the earlier Logo language) and a set of libraries, as well as a programming environment. Like RePast and Swarm it provides a set of programming facilities, however NetLogo also provides a graphical tool for quickly constructing interfaces for running agent-based models. In the following subsections, we describe the NetLogo interface and detail some of the basics of programming in NetLogo. Note that sample code can be found at the tutorial web site http://www.complexsystems.net.au/wiki/ANUPhysicsSS2008ABM_networks. In the following subsections, all programming commands are indicated by *italic* text.

1.1.3. *NetLogo Interface*

One of the benefits of using NetLogo is its interface. An example picture of the interface is shown in Figure 1.1. By default, the interface contains just a 2D spatial view of the model environment, which is a square lattice. Unlike Mason and Repast, NetLogo does not support hexagonal grids. If this is important to your model, you should investigate these other platforms. In addition to the 2D spatial view, the developer adds other elements, such as buttons to set model parameters and graphs to monitor results. Many of these elements are detailed below.

All elements of the interface may be moved around by first selecting them by right clicking and choosing select, and then clicking and dragging. Multiple items can be selected and then moved around together. To edit

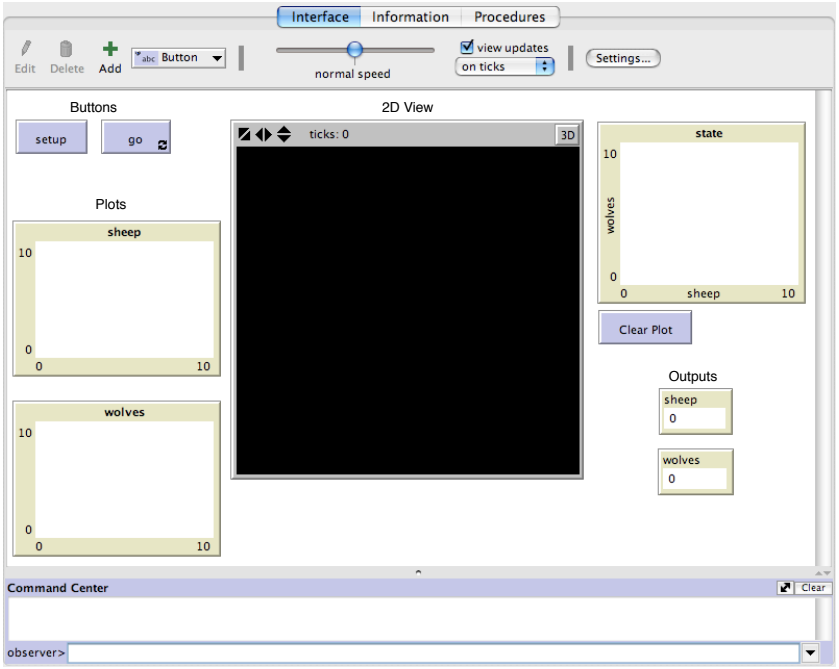

Fig. 1.1. This is the `NetLogo` interface for a sheep versus wolves predator scenario. The `NetLogo` file containing this interface, as well as the associated code, can be found at the tutorial web site.

them in order to change the details as given below, one right clicks on an element of the interface and selects 'Edit...'. They can, except for the view, be deleted by right clicking and selecting 'Delete'.

The 2D view has several different options that prove useful in modelling. The size of the grid (the number of cells) can be changed, as can the size of the cells themselves (in pixels). The edges of the model 'world' can be changed to reflect the system the model is to represent; the choices here are to treat the edges as walls, or to treat them as wrapping around onto the cells on the opposite side of the grid. The programming implications are discussed in the Topology section of the NetLogo Programming Guide.[17] The view supports inspecting patches and turtles by right clicking on the patch and inspecting it, or selecting one of the turtles and inspecting that from the right-click menu.

To run the software, buttons are typically set up in the interface. When creating a button, the piece of code to be run is specified. By default,

this is also displayed as the button name, however a different label can be specified in the button properties. The piece of code to be run can be changed by editing the button. Buttons can be set so that the code can be run repeatedly, until the button is pressed again. To specify this behaviour tick the 'forever' box. It is typical to have at least two buttons, a setup button, with a correspondingly-named procedure that clears the display and initialises the state of the model, and a go button, with the 'forever' box ticked, for running the model.

Monitors can be used to keep track of some number while the model is running. The monitor requires a reporter (procedure that returns a value) to be defined. This can either be a reporter defined in your program if one exists or it is something long, alternatively it could be a short piece of code defined in the monitor itself, for example *count turtles*. For more information on reporters, see Subsection 1.1.4. Monitors have an option to be labelled with something different than the piece of code, for example 'number of turtles' instead of the corresponding *count turtles* code.

Inputs, switches, choosers and sliders are all ways of specifying variables that can then be used by the code, either at initialisation, or while the code is running. The switches have two settings, on and off, and return a boolean value (true or false) when the global variable they define is used in code. The switches, choosers, and inputs all allow for some numerical input, with different constraints and interfaces.

Plots can be defined, along with their name (which can then be referenced in the code), their axes set up, and plot pens set up. The use of plot pens is for having multiple graphs on the same set of axes. The profile (colour, mode, and interval) for the plot pen can be set here, and then that particular pen can be drawn with by first specifying which pen is to be used in the code, followed by a list of plotting commands.

More information on the interface can be found in the NetLogo Interface Guide.[18] More information on linking these interface elements with the program can be found in the following subsection.

1.1.4. *NetLogo Programming*

Before giving details of the commands, it is necessary to define the types of agents on which commands can operate. NetLogo has a number of types of agents—turtles, patches, links, and the observer agent. The observer agent is a single agent that has a view of the whole NetLogo "world" (turtles, links and patches), and is used for running the main parts of the program

(linked to buttons on the interface) as well as providing a way of interacting command by command on the main interface. The patches represent square (in 2D) or box (in 3D) cells on the main 2D (or 3D) view of the world. The turtles are agents that can move around on the world surface, and draw. Links represent relationships between turtles.

In programming `NetLogo`, one starts with primitives—built-in `NetLogo` commands—and combines these into larger modules of code—procedures and reporters—that can be used themselves as commands, in particular as given to or used by turtles. The distinction between a procedure and a reporter is that a reporter returns some value at the end, using the command *report*, and is defined differently as follows. The block of code that forms a procedure begins with *to procedure-name* on a line by itself and on the next lines come a set of commands (primitives, or other defined procedures or reporters), and ends with the keyword *end*. It is a good idea, though not necessary in `NetLogo`, to indent code to indicate code structure. Note that the procedure name should not contain any spaces, in `NetLogo`, whitespace is used to separate out the different primitives, variables, *et cetera* that make up a line of code. The only time a space does not break up the code in this way is if it is part of a string (such as in a plot name), strings being enclosed by double quote marks. The block of code that forms a begins with *to-report reporter-name* and ends with the keyword *end*. A reporter should contain one or more lines that have the keyword *report* followed by some value, either a variable, or a call to another reporter which returns some value. Two or more *report* statements could be used depending on whether some condition is true or false—see the *if* or *ifelse* statements described in Table 1.2. Once *report* has been called, it will exit the reporter immediately, so it should come as the penultimate line in the reporter, unless it is run only under some condition.

A table of frequently-used basic primitives can be found in Table 1.1. Many primitives have shortened forms, to save on typing. Where these exist they are listed separated by commas; only one form of the primitive needs to be used in each instance. Some of the primitives act globally, other ones must be used in a particular context, for example by asking a turtle or patch to run them. This can be done using the *ask* command on an agent or agentset (set of agents); more on this later.

`NetLogo` features a number of commands for working with links to represent relationships between turtles, which are the nodes of the network. They come in two different forms, undirected and directed, to represent different types of relationships. For example, directed links could repre-

Table 1.1. A list of several frequently-used primitive commands available in the NetLogo environment. Where a shortened form of the primitive exists, this is indicated by listing the shortened form after the long form, only one (and no comma) need be used by the programmer.

Command	**Effect**
create, *crt n*	Create n turtles.
clear-all, *ca*	Clears all turtles, patches, plots and output, as well as resetting the tick counter and other global variables.
forward, *fd* and *right*, *rt*, and etc.	commands for moving the turtle. All of these commands take a single parameter giving the distance to be moved or angle (in degrees, clockwise) to be rotated by.

sent a parent-child relationship, whereas undirected links could represent a sibling relationship. NetLogo primitives for directed links have either *-from/-to* or *-in/-out* in the command name. Those for directed links, have *-with* or nothing at all. NetLogo features a number of layout primitives for laying out networks in the 2D view, such as *layout-circle* and *layout-radial*. NetLogo has commands *layout-spring* and *__layout-magspring* that use force-based layout algorithms similar those that Pajek has, as described in Subsection 1.2.4.

Breeds are a way of specifying different classes of turtles and links. These different classes can have different state variables (described later), and can have different behaviours, by *ask*ing different breeds to run different procedures and reporters. Breeds are specified at the start of the code, by using the *breed* keyword, followed by the plural and singular forms of the breed name (in that order) enclosed in square brackets. For example, to define a breed for representing wolves, one could write: *breed [wolves wolf]*. After definition, a large number of primitives are made available automatically by NetLogo, starting with the singular or plural form as appropriate. To continue the example of wolves, by using that command one would obtain primitives like *create-wolves* and *is-a-wolf?*. For details of other primitives created, refer to the breeds section of the NetLogo Programming Guide.[17] You can change the breed of a turtle (using *set breed new_breed*) mid-way through running your program, if this makes sense for your model. You can also change the breed of a link, but you cannot change between a directed and undirected breeds—NetLogo cannot do this automatically for you, and you must program such behaviour in as required in a way that makes sense for your particular model.

Variables are placeholders for storing information. There are several

different types of variables in NetLogo:

- Global variables hold values that are accessible anywhere in a NetLogo program. Global variables are defined at the start of a NetLogo program, using the *globals* command, followed by a list of whitespace-separated variables, the list being enclosed by square brackets. For example, the command *globals [dead_turtles found_money]* defines *dead_turtles* and *found_money* to be global variables. Any variables defined in the interface (such as sliders and monitors) do not need to be specified in the list of global variables as they are available by default.
- Local variables hold variables that are accessible only within a certain block of code. A block of code is something enclosed in square brackets, or within a procedure or reporter. Local variables are defined using the *let* keyword. This creates a variable, and assigns it an initial value, which must be specified. For example, *let food 3* creates a variable named *food* and assigns it the value of 3. Once defined with the *let* keyword, a variable can then be used on any subsequent line in that block of code.
- Variables owned by turtles, links, or patches. These can be defined by using the *-own* suffix on a type of agents or the plural form of a breed name, and then listing the variables. For example, *links-own [capacity latency]* defines all links to have variables named *capacity* and *latency*. A number of owned variables are predefined for turtles, patches and links. The NetLogo Dictionary[19] has a section that lists all the built-in variables owned by turtles, patches and links. Some important ones to note are *xcor* and *ycor*, the co-ordinates of the turtle (for patches these are named *pxcor* and *pycor*), and *who*, the turtle's unique identity number. There is also a reporter named *who*, callable in a turtle context, that returns a turtle's who number.

All variables, once they exist, can be set (or assigned a value) using the *set* command. For those who are used to programming in languages that have a C (programming language) or C-derived syntax, note that NetLogo uses *set* for variable assignment and the = symbol for testing for equality (whereas in a C-style syntax the = symbol is used for assignment and so == is used for testing for equality).

There are two types of data structures available: agentsets (using the same terminology as the NetLogo manual) and lists. Agentsets are just

that—sets of agents. They are useful because they then provide an easy way of giving commands to a group of agents that share some property, for example giving a set of commands to turtles that are on neighbouring patches to the current turtle. A set of commonly-used commands for constructing agentsets and for working with them can be found in the Agentsets section of the NetLogo Programming Guide.[17] Every time an agentset is used, its agents will be used in a random order. If you wish to use a specific order, then you should use a list. Lists are way of structuring agents or variables, unlike agentsets, which only store agents. A list can also store other lists, allowing for 2D or higher dimensional data structures. To change the order after a list is created, a new list must be created, and items copied over in the desired order into the new list. Primitives such as *replace-item* and *sort* simplify modification of lists, but note that they will return a new list, which must be assigned to something using *set*, or using *let* if the variable is defined and assigned in one step.

Most interesting algorithms have some control logic for repeating parts of the code, or selecting between alternative bits of code depending on some test. A summary of the NetLogo primitives for these can be found in Table 1.2. More details on these can be found in the NetLogo Programming Guide.[17]

NetLogo has a set of commands for writing output to files. To open a file for reading or writing, one uses the *file-open* command, followed by whitespace and then the filename in double quote marks, to denote that it is a string. For example, to open a file called 'output.txt', one uses *file-open "output.txt"*, where the quotes are part of the command. Whether it is opened for reading or writing is determined by whether the next command given is a command for reading from the file or a command for writing to the file. Once a file is opened for reading, it cannot be written to, and *vice versa*. To switch modes, use *file-close*, and then *file-open* again. The *file-open* command can also be used to select between multiple files that have been opened, by just using *file-open* on the file you want to access again (even if it was opened before). If a file is opened for writing and it already exists, then any write commands will write to the end of the file.

NetLogo features a number of commands for plotting. For these to work, at least one plot (distinct drawing area) must be defined as described in Subsection 1.1.3. If only one plot is in the interface, and it only has one pen (for one graph in the plot area) then commands like *plot*, for plotting points on a graph, and *plot-pen-down*, for ensuring that a point is drawn when the plotting point is set, can be used immediately. If, however, there

Table 1.2. Table of control logic commands, for changing the flow of execution of a program.

Command	**Effect**
if test [commands when true]	This runs a test, and then if true runs the command. The test must be some reporter that returns true or false, for example *xcor > 0* or *who = 3*.
ifelse test [commands when true] [commands when false]	Like the *if* command, except that this command has a section of commands (enclosed in a second pair of square brackets) to be run when the returns false.
while [reporter] [commands]	While the reporter returns true, this command runs the commands listed in the second set of square brackets. Note that one should be careful that the reporter will at some stage return false, otherwise this will loop forever (until the program is stopped manually).
foreach [list] [commands]	This applies the set of commands to every element of the list.
map [reporter] [list]	This returns a new list, formed from applying the reporter to each element of the list specified in the command.
repeat n *[commands]*	This repeats the block of commands n times.

are multiple plots and/or pens, then these should be selected using the function *set-current-plot "plot name"* to select a plot, and then if there are multiple pens in the plot the pen can be selected using *set-current-plot-pen "pen name"*. The plot name and pen name strings should match what has been defined in the interface. There are two primary commands for plotting points, *plot* and *plotxy*. The difference is that the first one uses an x-value from the last time *plot* was used (for that particular plot and pen) plus some interval. The x-value starts with a value of zero for the first time *plot* is used for that plot and pen. The interval can be changed in the pen settings. The *plotxy* command is used when one wants to specify both x and y values.

1.1.5. *The Art of Agent-Based Modelling*

As with any type of modelling, it is part art, part science. The following is a discussion of several points in modelling, mainly about agent-based modelling, that we feel are important to make. A more detailed discussion of these and other points can be found in Appendix B of Miller and Page.[1]

Part of the art of modelling is simplicity. This is important in modelling, because in modelling we are trying to gain a better understanding of

the real system through appropriate simplifications. To quote from Shalizi (1998):[20] 'Models are only good if they're easier to handle and learn about than what they model, and if they really do accurately map the relations we're interested in.' Part of picking appropriate simplifications is in picking simplifications that give better understanding, while still capturing important parts of what you are trying to model, including interactions with other parts of the system / other systems. The benefit of agent-based modelling over traditional variable-based modelling is that simplifications do not need to be made just to make the model mathematisable or solvable using a set of mathematical tools. In agent-based modelling it is important not to oversimplify, however. The power of agent-based models is based on their ability to accurately and falsifiably explain and model the complexity of real-world interactions.[21,22]

In some agent-based models, humans take the roles of some agents. These participatory agent-based models are useful for educating people about complexity and emergence in general, or on specific complex systems. They are also useful for model verification (of the rules) and validation (of the types of emergent behaviour). Feedback can be provided on the agent rules, in order to verify them. For validation, participatory actors can provide the quantitative data to serve the purpose of validation. They can, however, also provide qualitative feedback on the emergent findings.

To engage people who are experienced in the real-world system to participate in your model and provide you with feedback[23] `NetLogo` supports participatory agent-based modelling through its HubNet facility. More information can be found in the NetLogo HubNet Guide[24] and HubNet Authoring Guide.[25]

Another technique that is valuable in verifying the software is to embed within the model certain cases that may be modelled with other techniques, like equations. For example, in an agent-based SIR (susceptible-infected-recovered) model of epidemiology,[26] one could try embedding within the agent-based model a simple case that could also be modelled using mean-field theory. In the example code provided on the tutorial web site, we give sample code for a predator-prey model, with a few changes this could be compared with results from the the Lotka-Volterra equations. This helps rule out major bugs in the software; however since ABMs capture a wider range of behaviours, this form of verification is somewhat limited.

Good coding practice dictates that code is commented. Try and imagine yourself reading your code in ten years time. Good comments also help others read and understand your code. This is important as code should

be shared as part of the model verification process. Code should be shared more generally as part of good scientific practice, since results should be repeatable by others. Papers are usually too short to put down all the details required for replication.

1.2. Pajek

1.2.1. *Introduction to* `Pajek` *with Reference to Other Network Analysis Tools*

`Pajek`, pronounced 'Payek' and meaning 'spider' in Slovenian, is a social network analysis and visualisation tool. It was specifically designed to manipulate and analyse very large networks having on the order of 10^3 – 10^6 nodes. It is not the only tool of its kind, though it has some pleasing features, which has led to its wide adoption amongst academic practioners.

Other comparable software tools include `UCINet` and the `statnet` set of packages available for the open-source statistical program `R`. Both of these tools, like `Pajek`, are able to produce visualisation *and* analysis of large networks.

`UCINet` is a commerical product and its particular strength is in linear algebra measures and graph manipulations. On the other hand, it is computationally greedy and although it claims to handle over 32,000 nodes, its practical limit due to computational requirements is of the order of only 10^4 nodes, limiting the software in some very large network applications.

In contrast to `UCINet`, the open-source `statnet` meta-package in `R` provides perhaps the most flexible analysis option. Whilst the start-up costs are high for learning a scripting language such as `R`, the benefits include the embedding of network analysis within other statistical routines, and the ability to batch run the same network analysis on many different networks using a script.

`Mathematica` has a number of combinatoric and other graph theoretic packages, along with various visualisation packages. There is also a `NetLogo-Mathematica` link package available,[27] which provides a real-time link between `Mathematica` and `NetLogo`. This allows one to analyse, collect, and display data from a `NetLogo` ABM in real time, using social network and other graph theoretic measures.

Additionally, for visualisation purposes only, the multi-platform, GPL licenced software `GUESS` is certainly worth investigating. Whilst it does not feature network analysis tools, it is able to take network data input

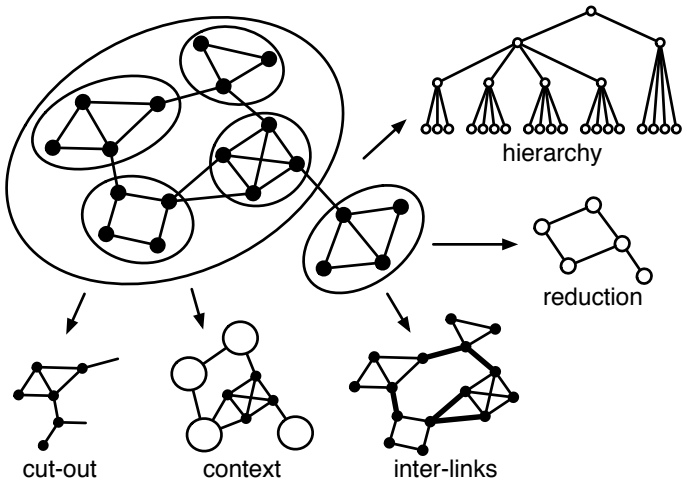

Fig. 1.2. Different ways of analysing a large graph (after the Pajek manual).[28]

from many sources (including Pajek) and output attractive visualisations, such as graph layouts over the top of an image layer (for example an airline's routing network overlayed on a map), in many popular image formats (including vector graphic formats such as EPS and SVG).

Pajek is situated in this context as a dedicated, flexible, but intuitive network visualisation and analysis software. It was designed with large scales in mind and is able to handle very large networks. Similarly, the Pajek graphical user interface (GUI) enables the management of multiple networks, components and analysis outputs at once, including those made within a Pajek session. Furthermore, it has various visualisation algorithms in-built and these outputs have been used in academic publications for several years.

To cope with a very large network dataset, it is useful to represent the graph in a variety of means (see Figure 1.2). Some of these representations require little analysis (for example cut-outs), whilst others require multiple processing steps (for example hierarchy analysis). Pajek is able to deal with many many different analytical approaches. The software is in constant development as new techniques that are derived in the academic literature make their way into the Pajek toolset.

In the following subsections, this tutorial discusses how to:

(1) Write and load `Pajek` format network files;
(2) Visualise a network in a number of ways;
(3) Generate random graph networks (random graphs) for comparison;
(4) Run common graph measures over networks.

1.2.2. *Importing Network Data*

`Pajek` can accept a range of information about a network. There are several fundamental types of data for `Pajek`:

- Networks: Vertices and edges/arcs (are indexed by their generation number);
- Partitions: Vertex information of a particular kind—they are actually class information, such that all the vertices are part of a particular class; for example 'Male' could be partition 1, and 'Female' partition 2;
- Vectors: Information on each vertex, on a vertex by vertex basis, for example 'Male' could be represented by 1 and 'Female' by 2, but likewise, each vertex can have all kinds of unique data attached to it, such as age, height or weight. Use Partitions for data that more than one vertex naturally matches, and Vectors for information that is likely to be matched by only one vertex at a time;
- Permutations: These are produced by functions applying to the whole network that create a new network based on the original network information. A permutation is a reordering of the nodes.
- Cluster: These are produced by certain functions that group together nodes according to some measure, for example their distance in the network.
- Hierarchy: A tree structure containing the vertices of the graph, and representing relationships between the nodes; for example the subtrees (parts of the tree) could represent communities, and these subtrees when joined up could represent a larger community of communities.

The `Pajek` interface manages each of these data types separately (see Figure 1.3).

`Pajek` uses several different file formats to manage the various data types. Below we give details of the file formats `Pajek` uses to read and save data in. In specifying a network, the usual components of a network are mandatory, such as vertices (nodes) and edges (links between nodes) or arcs (directed edges). Note that `Pajek` rightly treats edges and arcs differently.

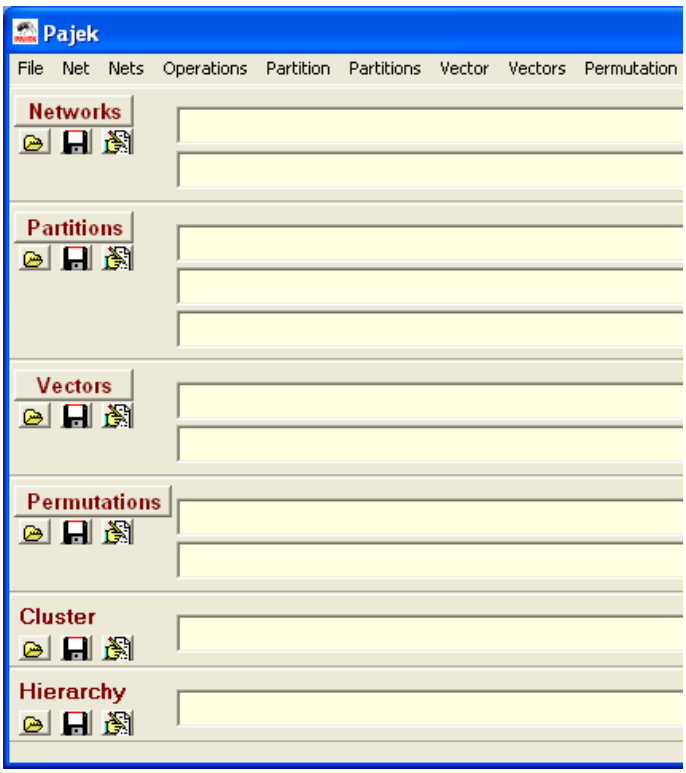

Fig. 1.3. Each of the different data types `Pajek` accepts has its own space in the interface where data can be opened (folder icon), saved (disk icon) and viewed or edited (edit button, or double click on the data file name). One can start with any number of files and generate the rest if needed through algorithms provided by `Pajek`. One could even start with no files and generate a random network as discussed in Subsection 1.2.6. Note that for image clarity reasons, not all menu options are shown here.

See Figure 1.4 to see an example network containing both edges and arcs.

A `Pajek` network file first begins with `*vertices` followed by a space and then the number of vertices in the network. If any of the nodes are labelled, this can be specified following the `*vertices` line by specifying the number of the vertex that you want to label, then a space, and then a label. If the label has a space in it, then it needs to be surrounded by double quote marks. Edges can be specified by writing `*edges` on a new line, and on the following lines writing each edge (one per line) by specifying the two nodes, separated by a space. Arcs can be specified in one of two ways:

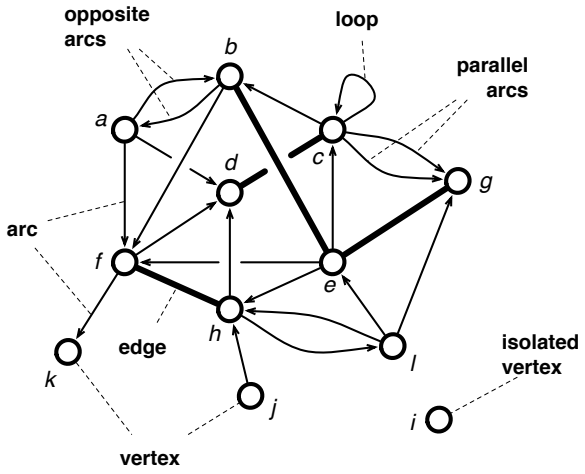

Fig. 1.4. This figure shows a set of vertices (or nodes) denoted by circles, and edges (links without arrowheads) and arcs (directed links, with arrowheads) joining them.

(1) by writing the keyword `*arcs` on a line by itself, and then on the following lines specify one arc by writing the starting node, destination node, and arc weight, all separated by whitespace; or
(2) by writing the keyword `*arcslist` on a line by itself, and then on each of the following lines specify a starting node, followed by a list of destination nodes, to be joined in separate arcs (of weight 1) from the starting node. In this case whitespace separates the nodes listed.

An example network file (`example.net`) for the graph in Figure 1.4 is given in Figure 1.5.

There are several things to note about the `Pajek` network file format

- Vertices cannot be indexed 0 (start with 1);
- Any amount of white space can be used to separate elements on a line, and whitespace consists only of space characters (not tabs);
- Note that `Pajek` won't draw arcs and edges that are from a node to itself, but they are still there in the data structure.
- `Pajek` does not support Unix text files, lines need to be ended with a carriage return and line feed (if you are working on Windows do not worry about this);
- A network file consists of just the basic information about the net-

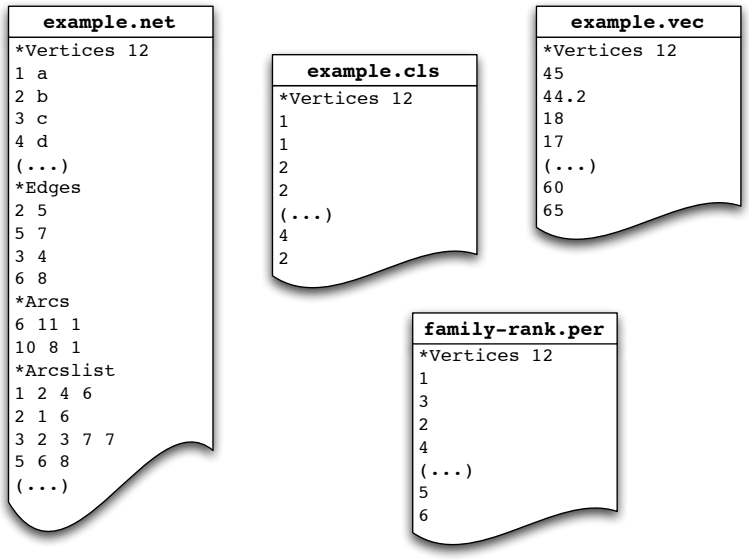

Fig. 1.5. Example files referred to in the text.

work (vertices and connections), other information such as each vertex's rank, cluster and/or other information for each vertex can be added as separate files (see Figure 1.3); and

- Each file can be read into `Pajek` and then saved as a Project: *File* → *Pajek Project File* → *Save*

In `Pajek`, one can assign vertices to clusters. The clusters can either be generated through a clustering algorithm (see Subsection 1.2.7) or loaded from a file (either one generated previously, or entered manually). To load from a file, in `Pajek` select *File* → *Cluster* → *Read*. An example clustering file (`example.cls`) for the graph given in Figure 1.4 is given in Figure 1.5. Examples of permutation and vector files for the graph in Figure 1.4 can also be found in Figure 1.5.

1.2.3. *Visualising Networks*

There are several ways to visualise a network. In fact, this is an area of active current research—consider the difference between the layout of a subway network versus an electical diagram versus a social network versus a gene expression network, *et cetera*. Each network has its own 'natural'

way of laying out. Sometimes these are connected strongly to physical interpretations (for example an international flight network overlaid on the world map), but many networks do not have a physical interpretation (for example a sexual activity network)—rather, they are just networks: objects which indicate the relationships between elements (not their location).

Some visualisation tools to try out are:

(1) With your network loaded, use *Draw* → *Draw* (or just hit Ctrl+G);
(2) You will see a simple representation, by default, the nodes are arranged on a circle—try changing this representation, or layout, by using the different options in *Layout* → *...*
(3) If the network looks strange to you, try using a different starting position (*Layout* → *Energy* → *Starting Positions* → *...*) (see Subsection 1.2.4);
(4) Now, try a common layout: *Layout* → *Energy* → *Fruchterman Reingold* → *3D*;
(5) Since your screen is in 2D, and the graph is now laid out in 3D, we need to alter the perspective to 'see' it. This can be done this as follows: *Spin* → *Spin Around (accept 360 Degrees)*.
(6) This likely was too quick for you .. in which case, use *Spin* → *Step in degrees* → *...* and set this value to something very small (like 0.05), now use *Spin* → *Spin Around* again.

1.2.4. *Force-Directed Layouts*

'Force-directed' and 'spring-directed' refer to a class of 'energy' algorithms where one can conceive of the network as a collection of balls, connected by springs of a certain length (refer to Algorithm 1.1). These are a type of graph embedding algorithm.[29,30] The idea is to imagine starting with this web of balls and springs in a random configuration, and then allowing them to 'relax'. The ultimate configuration should be one where spring tension is minimised, such that connected balls are close together in the layout space. The initial position is quite important. It can be the case that this algorithm, like a human untangling many inter-connected springs, might actually not be able to untangle them all into a 'good' layout. Hence, the option is given in `Pajek` to vary the starting positions.

Algorithm 1.1 A Force-Directed Layout Algorithm (pseudo-code)

```
Positions ← Generate-random-position-for-all-vertices (X, Y)
Energy ← Calculate-total-force-in-springs
while Energy ≥ Energy-threshold do
    Displacements ← Calculate-resultant-force-on-all-vertices
    Positions ← Positions + Displacements
    Energy ← Calculate-total-force-in-springs
end while
```

1.2.5. *Exporting the Network Layout to Another Program*

To get an image file of your current layout, use the *Export* → *2D* → menu item. Choose a file format that you wish, for example EPS/PS (mainly used for importing into LaTeX), or bitmap (for a web page or `MS Word` application).

1.2.6. *Generating a Comparison Network*

It is often useful to compare a given network to one that is like it in some ways, but different in others. Random graphs, where a random process generates a graph, with some statistics the same as the original, are useful null models.[31–34] For instance the work of Watts and Strogatz, in their influential *Nature* paper,[35] use comparisons between a given graph and its random graph equivalent (same number of nodes and average degree). We will do the same.

(1) First, we will generate a random Erdős-Rényi network: *Net* → *Random network* → *Erdos-Renyi* → *Undirected* → *General*. Specify a number of Vertices, and the Average Degree (for example 50, and 4). Obviously, for a comparison net you will want to use the same number of vertices and average degree as your network of interest.
(2) The network will appear in your 'networks' row on the home area of `Pajek`.
(3) Let us visualise it. You can do that as before.
(4) Now, since this is probably our first large network, we will do a bit more with our visualisations: Back in the home area, with the new Erdős-Rényi random network highlighted, we will create a Partition (of the vertices) by the property 'Degree' (number of incident edges): *Net* → *Partitions* → *Degree* → *Output*. New rows should have been added in the main area of the interface.

(5) Highlight the original Erdős-Rényi random network (networks) and use *Draw* → *Draw Partition* (or Ctrl+P). This should bring up a coloured visualisation.
(6) Make the vertices bigger by using *Options* → *Size* → *of Vertices* (a good size to use is 10).
(7) Now let us get a better idea of the partitions. Try *Layout* → *Circular* → *using Partition*. This will place similar vertices (by their degree/-partition) together in space.

1.2.7. *Common Network Measures*

In this subsection we will look at some common network measures. We will continue working with the large Erdős-Rényi random network we made earlier. A random network by itself, even visualised with one of the pretty force-directed placement algorithms, is still just a random network—it is hard to get much out of it by just visualising it. First we will generate some outputs from the network, using the sample network file for Figure 1.4. This file can be loaded by clicking on the folder icon under in the networks section of the `Pajek` interface or by clicking *File* → *Network* → *Read*. The process of generating outputs is going to get a bit confusing unless you pay close attention to the way that `Pajek` organises and operates on data. Recall from Subsection 1.2.2 the different data types for `Pajek`.

Once a network is loaded using the interface, one can calculate a number of network measures. A limitation of `Pajek` is that for some of these, it doesn't handle multiple lines and loops well, in that it doesn't take them into account when normalising. So it is a good idea to remove them from the network. It is also a good idea for some network algorithms (in general, and not just in `Pajek`) to represent edges by arcs going in both directions. These pre-processing steps can be found in the *Net* → *Transform* submenu in `Pajek`. The ones that need to be applied are *Edges* −> *Arcs*, *Remove* → *Multiple Lines* → *Single Line* and *Remove* → *Loops*. You can create a new network or just overwrite at each step. If you create a new network at each step, the last one is the one that will automatically be selected for the following operations.

(1) Degree distribution: What we wish to do is assign each vertex a number that describes the number of input, output or either input/output edges that are incident at the given vertex. Hence, we will be aiming to produce a vector output of (normalised) degrees. The way this is done, is actually to do a partition of the vertices into their degree numbers

(we kill two birds with one operation):

(a) Select from the menu *Net* → *Partitions* → *Degree* → *All*. Notice that this produces two new lines in Partition and Vectors;
(b) Double-click on the text window of Vectors (see Figure 1.3), to bring up a simple text file view of the data—notice that each vertex now has a number associated with it: this is the normalised degree of each vertex;
(c) Double-click on the text window of Partitions (see Figure 1.3), this file shows for each vertex, based on its normalised degree number, which class (or 'Partition') of vertex it falls into. Find the vertex with the highest partition number (in this case, this is the degree)—you should find it is vertex 6 ('f'). Go back to the 'Vector'; information for normalised degree and look at that vertex's normalised degree. The degrees are normalised (that is, divided) by a factor of $n - 1$.
(d) To make a degree distribution, you will have to export the normalised degree vector into another application (for example `MATLAB`, `Excel`, `R`, `SPSS`, or another package) and run a histogram function over the data. In fact, porting to `R` and `SPSS` is integrated into `Pajek` for this purpose: *Tools* → *R* → *Send to R* → *Current Vector*.

(2) Clustering Coefficient: Now what we want to do is calculate the clustering coefficient amongst the 1-neighbourhood of each node. Each node's 1-neighbourhood is all the other nodes that it shares an edge with. For the example graph in Figure 1.4, the 1-neighbourhood of node a is $\{b, d, f\}$. For k the total degree of the node (in this example, a has $k = 3$ neighbours), then there are at most $k(k-1)$ arcs in a directed network or $k(k-1)/2$ edges in an undirected network. The clustering coefficient is simply the number of arcs or edges between the neighbours, divided by the maximum possible for the type of network, $k(k-1)$ or $k(k-1)/2$. In the example network, there is one arc between the three neighbours (the arc from b to f, the other edges are all back to node a which isn't in its neighbourhood). So the clustering coefficient of node a is $1/(3 \times 2) = 1/6$. Note that `Pajek` doesn't compute clustering coefficients for networks with parallel arcs, these can be removed by either editing the file and selecting *Net* → *Transform* → *Remove* → *Multiple Lines* → *Single Line*. To compute the clustering coefficients in Pajek use *Net* → *Vector* → *Clustering Coefficients* → *CC1*. The *CC2* option is for computing the clustering coefficient of the

2-neighbourhood (all of a node's neighbours, and all of their neighbours excluding the node you are computing the clustering coefficient for).

(3) Shortest Paths: Now what we want to do is to find the shortest path between nodes. There are two options for this: *Net* → *Paths between 2 vertices* → *One shortest* and *Net* → *Paths between 2 vertices* → *All shortest*. The difference is that the first shows only one path if multiple exist, the latter shows both. Both options will prompt for the starting node and end node to find shortest path(s) for. Both options also ask two questions. The first is 'Forget values on lines?' If yes is selected, then any edge weights will be ignored, and a shortest path is one with the fewest edges. If no is selected, then edge weights are used, and a shortest path is one with minimum sum of edge weights. The second question asked is 'Identify vertices in source networks'. If yes is chosen, then a partition is created, partitioning the edges of the original network into belonging to the shortest path(s), indicated by a '1', or not, indicated by a '0'. The resulting network has all the nodes that are in shortest paths, with the edges that lie on shortest path(s) in the original network being the only edges in the new network.

(4) Betweeness Centrality measures the shortest paths going through each vertex.[36,37] If there are multiple (equal) shortest paths between two nodes, then instead of adding to the count by one, one adds the fraction of the shortest paths going through the vertex. The overall count can then be normalised by $(n-1)(n-2)$, the total number of paths, as is done in `Pajek`. In `Pajek` one can use *Net* → *Vector* → *Centrality* → *Betweeness* to calculate the betweenness. `Pajek` does not calculate the newer, but similar, edge betweenness measure.[38]

(5) The diameter of a network is the longest shortest path between two vertices. So one considers every pair of vertices, finds the shortest path between each of the pairs, and then the diameter is the longest of these shortest paths. It can be computed in `Pajek` through *Net* → *Paths between 2 vertices* → *Diameter*

(6) Hierarchical Decomposition: *Net* → *Hierarchical Decomposition* → *Clustering* → *Run*. One runs a hierarchical decomposition when you wish `Pajek` to automatically group vertices based solely on the pattern of connections amongst them without reference to their type. In this way, hierarchical decomposition can be a revealing method to uncover similarities in vertex context that may otherwise not be prominent to the observer. Note that this analysis produces a 2D specific hierarchical network output which is different to `Pajek`'s normal 'Draw' output. The

output is sent directly to an external, encapsulated Postscript (EPS) file that can be converted to a PDF file or other image formats using a graphics tool like Photoshop, GIMP, or Ghostscript.

There are many measures that you can use—the above are just a starter to get you going. Exploration in Pajek is rewarding!

References

1. J. H. Miller and S. E. Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life (Princeton Studies in Complexity)*. (Princeton University Press, March 2007). ISBN 0691127026.
2. M. Prokopenko, F. Boschetti, and A. J. Ryan, An information-theoretic primer on complexity, self-organisation and emergence, *Complexity*. (2008).
3. A. J. Ryan, Emergence is coupled to scope, not level, *Complexity*. **13**(2), 67–77, (2007). doi: http://dx.doi.org/10.1002/cplx.20203. URL http://dx.doi.org/10.1002/cplx.20203.
4. R. Abbott, Emergence explained: Abstractions: Getting epiphenomena to do real work, *Complexity*. **12**(1), 13–26, (2006). doi: http://dx.doi.org/10.1002/cplx.20146. URL http://dx.doi.org/10.1002/cplx.20146.
5. L. Tesfatsion, Agent-based computational economics: modeling economies as complex adaptive systems, *Information Sciences*. **149**, 263–269, (2003).
6. J. Epstein and R. Axtell, *Growing Artificial Societies. Social Science from the Bottom Up*. (MIT Press, 1996).
7. V. Grimm, E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H. H. Thulke, J. Weiner, T. Wiegand, and D. L. Deangelis, Pattern-oriented modeling of agent-based complex systems: lessons from ecology, *Science*. **310**, 987–991, (2005).
8. S. L. Spencer, R. A. Gerety, K. J. Pienta, and S. Forrest, Modeling somatic evolution in tumorigenesis., *PLoS Computational Biology*. **2**(8) (August, 2006). ISSN 1553-7358. doi: 10.1371/journal.pcbi.0020108. URL http://dx.doi.org/10.1371/journal.pcbi.0020108.
9. E. R. Smith and F. R. Conrey, Agent-based modeling: A new approach for theory building in social psychology, *Pers Soc Psychol Rev*. **11**(1), 87–104 (February, 2007). doi: 10.1177/1088868306294789. URL http://dx.doi.org/10.1177/1088868306294789.
10. E. D. Beinhocker, *Origin of Wealth: Evolution, Complexity, and the Radical Remaking of Economics*. (Harvard Business School Press, September 2007). ISBN 1422121038.
11. J. M. Epstein, *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton Studies in Complexity, (Princeton University Press, January 2007). ISBN 0691125473.
12. K. R. Popper, *The Logic of Scientific Discovery*. (Routledge, March 1972), 2nd edition. ISBN 0415278449.

13. K. R. Popper, *Conjectures and Refutations; The Growth of Scientific Knowledge*. (Routledge, August 2002). ISBN 0415285941.
14. M. Blaug, *The Methodology of Economics: Or, How Economists Explain*. Cambridge Surveys of Economic Literature, (Cambridge University Press, July 1992). ISBN 0521436788.
15. D. M. Hausman, Ed., *Testability and approximation*, In ed. D. M. Hausman, *The Philosophy of Economics: An Anthology*, chapter 8, pp. 179–181. Cambridge University Press, 3rd edition, (2008).
16. M. J. Berryman. Review of software platforms for agent based models. Technical Report DSTO-GD-0532, Defence Science and Technology Organisation, Edinburgh, Australia (April, 2008).
17. U. Wilensky. Netlogo programming guide (November, 2008). URL `http://ccl.northwestern.edu/netlogo/docs/programming.html`.
18. U. Wilensky. Netlogo interface guide (November, 2008). URL `http://ccl.northwestern.edu/netlogo/docs/interface.html`.
19. U. Wilensky. Netlogo dictionary (November, 2008). URL `http://ccl.northwestern.edu/netlogo/docs/primindex.html`.
20. C. R. Shalizi. John holland, emergence, (1998). URL `http://www.cscs.umich.edu/~crshalizi/reviews/holland-on-emergence/`.
21. K. M. Carley. Simulating society: The tension between transparency and veridicality. In *Proceedings of Agent 2002 Conference on Social Agents: Ecology, Exchange and Evolution* (October, 2002).
22. S. Odubbemi and T. Jacobson, Eds., *Governance Reform Under Real World Conditions: Citizens, Stakeholders, and Voice*. (World Bank, June 2008).
23. A. M. Ramanath and N. Gilbert, The design of participatory agent-based social simulations, *Journal of Artificial Societies and Social Simulation*. **7**(4) (October, 2004).
24. U. Wilensky. Netlogo hubnetguide (November, 2008). URL `http://ccl.northwestern.edu/netlogo/docs/hubnet.html`.
25. U. Wilensky. Netlogo hubnet authoring guide (November, 2008). URL `http://ccl.northwestern.edu/netlogo/docs/hubnet.html`.
26. M. J. Berryman. *A Complex Systems Approach to Important Biological Problems*. PhD thesis, The University of Adelaide (May, 2007).
27. U. Wilensky. Netlogo-mathematica link (November, 2008). URL `http://ccl.northwestern.edu/netlogo/docs/mathematica.html`.
28. V. Batagelj and A. Mrvar. *Pajek: Program For Analysis and Visualization of Large Networks—Reference Manual*. University of Ljubljana, Slovenia, 1.24 edition (December, 2008).
29. V. Chandru and J. Hooker, *Optimization Methods for Logical Inference*. (Wiley Interscience, 1999).
30. J. L. Gross and J. Yellen, Eds., *Handbook of Graph Theory (Discrete Mathematics and Its Applications)*. (CRC, December 2003), 1 edition. ISBN 1584880902.
31. A.-L. Barabási and R. Albert, Emergence of scaling in random networks, *Science*. **286**(5439), 509–512 (October, 1999).
32. A.-L. Barabási, *Linked: How Everything Is Connected to Everything Else*

and What It Means for Business, Science, and Everyday Life. (Plume Books, April 2003). ISBN 0452284392.
33. D. J. Watts, *Six Degrees: The Science of a Connected Age*. (W. W. Norton & Company, February 2004). ISBN 0393325423.
34. E. Ravasz and A.-L. Barabási, Hierarchical organization in complex networks, *Phys. Rev. E.* **67**(2), 026112 (Feb, 2003). doi: 10.1103/PhysRevE.67.026112.
35. D. J. Watts and S. H. Strogatz, Collective dynamics of 'small-world' networks., *Nature.* **393**(6684), 440–442 (June, 1998). ISSN 0028-0836.
36. L. C. Freeman, A set of measures of centrality based on betweenness, *Sociometry.* **40**, 35–41, (1977).
37. L. C. Freeman, Centrality in social networks: Conceptual clarification, *Social Networks.* **1**, 215–239, (1978–1979).
38. M. Girvan and M. E. Newman, Community structure in social and biological networks., *Proc Natl Acad Sci U S A.* **99**(12), 7821–7826 (June, 2002). ISSN 0027-8424.

Subject Index