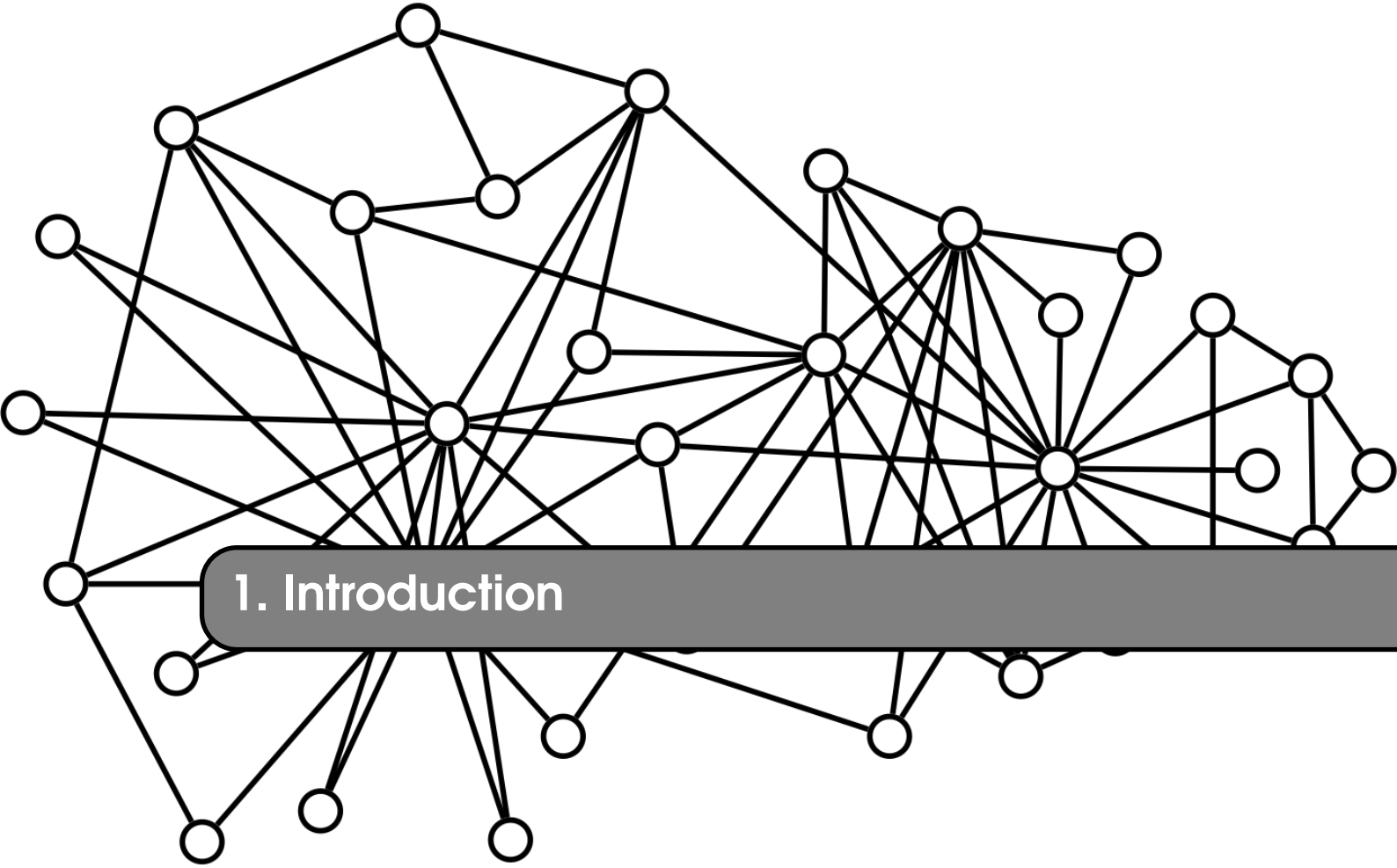
Graph Mining

Efficient Julia Programs for Understanding Our World

Pierluigi Crescenzi

L'Aquila, January 31, 2022

Contents



1. Introduction

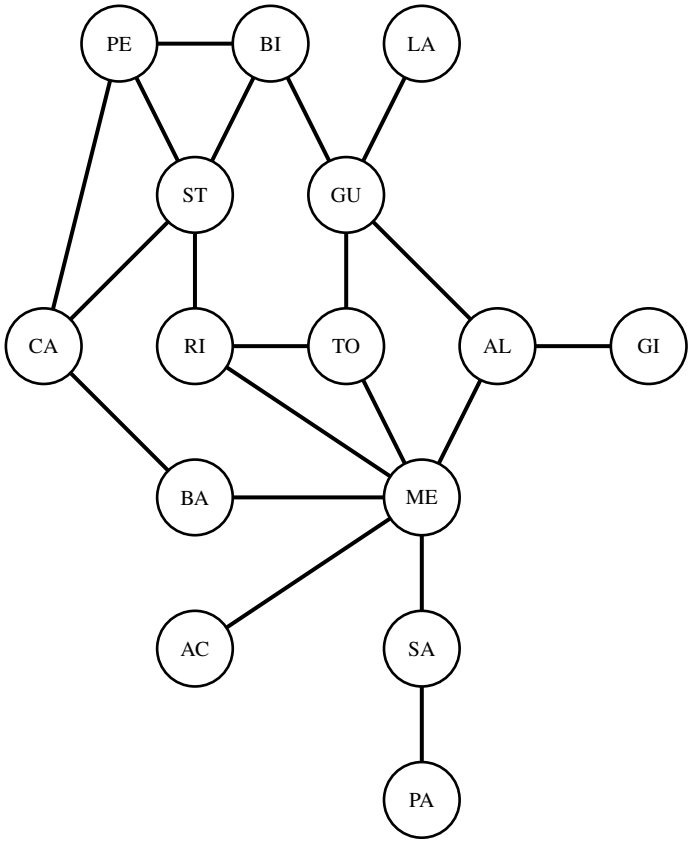
The fundamental difference between a social network explanation and a non-network explanation of a process is the inclusion of concepts and information on relationships among units in a study.

S. Wasserman, K. Faust, "Social Network Analysis: Methods and Applications", 1994

1.1 Graph mining

Graphs are everywhere. Mainly because they are a simple, elegant and powerful tool to represent and analyze binary relationships, and our life is full of binary relationships of very different types. Some typical examples of such binary relationships are *communications* (such as, for example, talking on the phone or sending emails or messages on Telegram), *collaborations* (such as, for example, co-authoring a scientific paper or co-acting in a movie), *ratings* (such as, for example, indicating a "like" for a photo on Instagram or rating a movie on Netflix), *membership* (such as, for example, being a member of a university department or of a political party), *dependencies* (such as, for example, citing a paper within another paper or including a link to a web-page on another web-page), and *transfers* (such as making a bank transfer or selling a car). Each instance of these relationships creates a connection between the two agents involved, who, more often than not, are human beings. These connections are the *arcs* of a graph, whose *nodes* are the agents themselves.

Consider, for example, a simplification of one of the most popular social networks, which concerns a group of families in the city of Florence in the fifteenth century. In this case, the nodes of the graph are the following fifteen Florentine families: Acciauioli, Albizzi, Barbadori, Bischeri, Castellani, Ginori, Guadagni, Lamberteschi, Medici, Pazzi, Peruzzi, Ridolfi, Salviati,



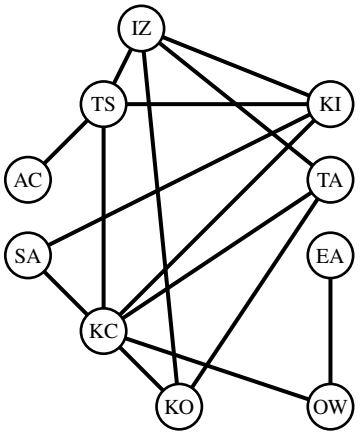
AC: Acciauoli
AL: Albizzi
BA: Barbadori
BI: Bischeri
CA: Castellani
GI: Ginori
GU: Guadagni
LA: Lamberteschi
ME: Medici
PA: Pazzi
PE: Peruzzi
RI: Ridolfi
SA: Salviati
ST: Strozzi
TO: Tornabuoni

Figure 1.1: The social network of Florentine families in the fifteenth century (see Table 1 in [16]): for each family, the graph includes a node labeled with the first two letters of the family name and, for each pair of families, the graph includes an arc between the two corresponding nodes if at least one member of one family has married at least one member of the other family.

Strozzi, and Tornabuoni. The binary relationship between these families is that at least one member of a family has married at least one member of another family. The graphical representation of this relationship is shown in Figure 1.1, where a label is associated with each node of the graph, formed by the first two letters of the name of the corresponding family. In this case, therefore, the social network includes fifteen nodes and twenty arcs: we can already observe how the Medici family plays an important role in this social network, being the one with the greatest number of connections (six), much greater than the average of the connections of each family which is equal to $\frac{2\times 20}{15} = 40/15 \approx 2.7$ (note that the total number of connections must be multiplied by two as each arc must be considered from the point of view of the two nodes that the arc itself joins).

The main advantage of representing binary relationships through mathematical objects, such as graphs, is that, on these objects, we can use the many mathematical and computer tools, that have been developed in the field of graph theory. In other words, we can analyze mathematical properties of a graph, and we can design, analyze and develop efficient algorithms that compute these properties. The final aim will be to reach conclusions that are interesting from the point of view of the specific domain, such as, for example, the existence of patterns that perhaps were not initially foreseen.

In the next paragraphs of this chapter, we will describe some examples of application of this methodology to three different fields of research: the study of archaeological sites in third century Japan, the identification of communities within movie collaboration networks, and the prediction of



AC: Asian continent
SA: Sanuki-Awa
EA: East
IZ: Izumo
KC: Kinki-core
KI: Kibi
KO: Koshi
OW: Owari
TA: Taniwa
TS: Tsukushi

Figure 1.2: The graph of archaeological sites in third century Japan (see Figure 7.5 in [70]): for each site, the graph includes a node labeled with the initials of the site, and, for each pair of sites, the graph includes an arc between the two corresponding nodes if, at the two sites, they were found artifacts of the same type.

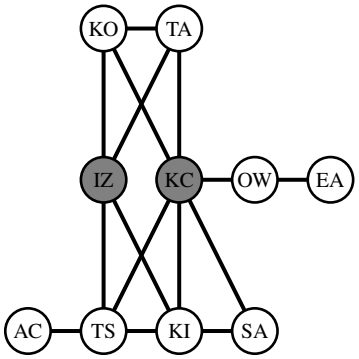
the emergence of new relationships in the future of a social network. In all three cases, we will show how simple graph theory concepts (such as neighborhood, connectivity, and shortest path) can be used to analyze the network, and derive possible conclusions about its agents and about their relationships. We will conclude the chapter by introducing more formally the basic concepts of graph theory, and by describing and analyzing a simple graph algorithm, which we will often refer to throughout the rest of the book.

1.2 Interpreting the past: the network of Japanese archaeological sites

The exchange of valuables can be seen as a communication tool that allows us to reproduce the communication systems spread across a space-time dimension. This idea can be applied in particular to reconstruct the communication systems of the past, and to identify the participants in the systems themselves who have played a more significant role.

As an example, let us consider a particular period in Japanese history, called Kofun, which begins around the middle of the third century. In this period, it began the construction of specific burials or mounds in the shape of a "keyhole", of relatively large dimensions (about three hundred meters in length) and containing objects of different types, such as mirrors, bronzed tools, and weapons. In particular, nine archaeological sites have been identified over the years, to which the Asian continent, from which various tools and valuables were imported, must be added. Each of these ten sites corresponds to a node of a graph: for each pair of sites, the graph includes an arc between the two corresponding nodes if artifacts of the same type are found at the two sites (see Figure 1.2).

The representation of this graph shown in Figure 1.2 does not help too much to understand which site played a more important role than the others. However, if we redraw the graph as shown in Figure 1.3, it becomes clear that the two sites Izumo and Kinki-core are more "central" than the others. In this book we will analyze various measures of centrality (including the number of connections we have already referred to, when talking about the social network of the Florentine families). For now we limit ourselves to introducing a measure of centrality based on the concept of *distance* between two nodes of the graph. This distance indicates the minimum number of connections that must be "crossed" to go from one node to another. For example, in the case of the graph of Figure 1.3, the distance from the node with the label `KC` to the nodes with labels `KO`, `TA`, `TS`, `KI`, `SA`, and `OW` is equal to 1, since `KC` is directly connected to these nodes. Its distance from the



AC: Asian continent
SA: Sanuki-Awa
EA: East
IZ: Izumo
KC: Kinki-core
KI: Kibi
KO: Koshi
OW: Owari
TA: Taniwa
TS: Tsukushi

Figure 1.3: The graph of archaeological sites in Japan drawn in such a way as to highlight the existence of two more "central" nodes (in gray): these two sites seem to play a role of passage point for communication between the others sites and, in particular, the KC site seems to be more important than the IZ site, as it allows the sites on its right to communicate with all other sites.

node labeled IZ is instead 2, since at least two connections must be crossed: for example, the one from KC to KO and, therefore, the one from KO to IZ. Similarly, we can conclude that the distance of KC to the nodes with labels AC and EA is also equal to 2.

The distances between all the pairs of nodes of the graph in Figure 1.3 are summarized in the following table, in which a node is associated with each row and column, and each cell in the table includes the distance between the two nodes corresponding to the row and the column of the cell.

	AC	AS	ES	IZ	KC	KI	KO	OW	TA	TS	**Closeness**
AC	0	3	4	2	2	2	3	3	3	1	23
AS	3	0	3	2	1	1	2	2	2	2	18
ES	4	3	0	4	2	3	3	1	3	3	26
IZ	2	2	4	0	2	1	1	3	1	1	17
KC	2	1	2	2	0	1	1	1	1	1	12
KI	2	1	3	1	1	0	2	2	2	1	15
KO	3	2	3	1	1	2	0	2	1	2	17
OW	3	2	1	3	1	2	2	0	2	2	18
TA	3	2	3	1	1	2	1	2	0	2	17
TS	1	2	3	1	1	1	2	2	2	0	15

Note that the values of the cells corresponding to two equal nodes are set equal to 0, since the distance in this case is clearly null. We also note that the table is "symmetric", in the sense that the distance from a node x to a node y is equal to the distance from the node y to the node x: indeed, the same path can be traversed in both directions. The last column of the previous table shows the sums of the distances of each node from all other nodes (in other words, the sum of the elements of each row). This value indicates how "close" a node is to the other nodes: the smaller this value is, the closer a node is to the other nodes and, therefore, the more central. For this reason, we call this value *closeness* of a node.

In our case, the most central node is the one labeled KC, which corresponds to the Kinki-core site: the sum of its distances from the other nodes is equal to 12. Next, we have the Kibi site (with the KI label) and the Tsukushi site (with the TS label) with a closeness measure of 15, which precede the Izumo site (with the IZ label) which has a measure of closeness equal to 17. This partly contradicts the idea we had, based on the graphical representation of the graph shown in Figure 1.3, that this last site was more central.

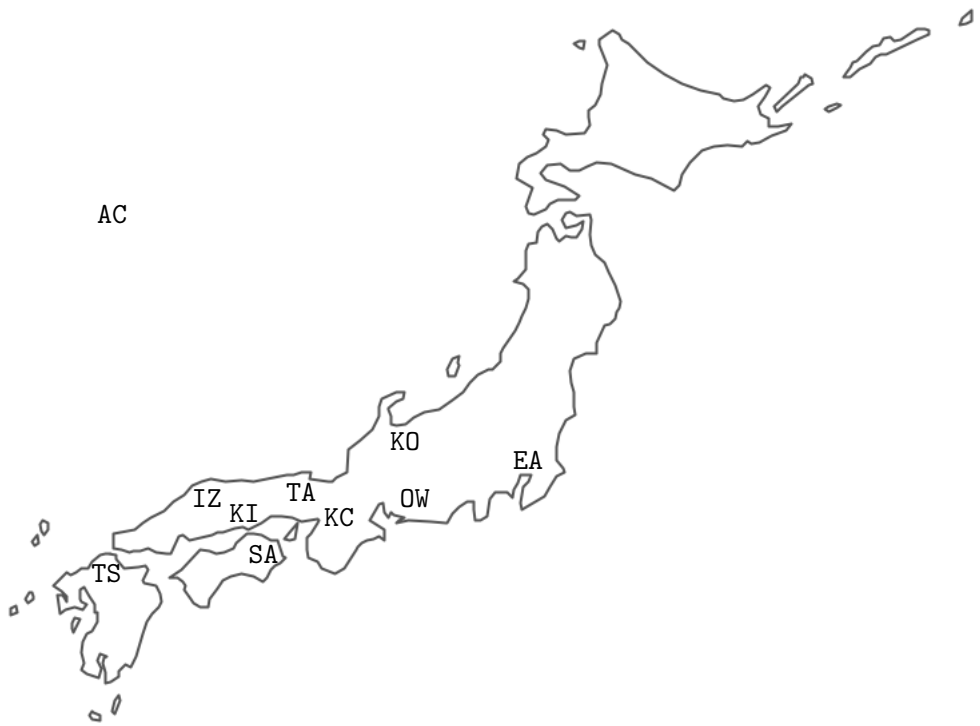


Figure 1.4: The map of Japan indicating the nine archaeological sites of the Kofun period, in addition to the Asian continent indicated with the AC label (see Figure 7.4 in [70]). The Kinki-core site (labeled KC), which is the most central in the graph with respect to the closeness measure, is also the most central geographically. More surprising is the fact that the Tsukushi site (labeled TS), which is rather geographically decentralized, is instead a central site in the social network with respect to the closeness measure.

At this point, it is interesting to analyze the graph of archaeological sites by referring to their geographical location. As shown in Figure 1.4, the Kinki-core site (labeled KC) is geographically positioned quite centrally with respect to the other sites: it is therefore not surprising that this site is also central in the graph. However, we observe that the graph was defined by referring only to the presence of similar artifacts in two different sites: the analysis of the graph therefore indicates that there could be a correlation between the latter property and the relative geographical locations of the sites themselves. This hypothesis, however, has to face a different situation as regards the Tsukushi site (with the label TS), which is rather peripheral from a geographical point of view but, as we have seen, is very central from the point of view of the closeness measure. Indeed, it may surprise us that in the graph there is a direct connection between the node labeled KC and the node labeled TS: this is due to the fact that in one of the most important mounds of Tsukushi, objects worked in the Kinki-core style have been found. The graph, therefore, suggests that the interactions between the different sites are more complex than the geographic location of the sites themselves can suggest.

1.3 Understanding the present: the collaboration network of actors

The *Internet Movie DataBase* (for short, *IMDB*) is a service available on the World Wide Web, which allows us to access information of various kinds relating to films, actors, directors, screenwriters and, in general, to all the staff of a film production [47]. Created in the early nineties, it has

developed mainly thanks to the collaboration of its users, who, once registered, can contribute to the archive with new information that is monitored before being made public. In addition to being an online social network, in which members can interact, not only by updating information but, for example, by also providing film ratings, IMDB has the advantage of making all the data collected publicly available and providing, therefore, a very rich data-set for a researcher, who wishes to carry out an analysis of social networks. As of mid-2018, IMDB included information relating to nearly nine million people and nearly five million movies.

Starting from this data-set, we can, for example, create one of the largest collaboration networks currently known. A *collaboration network* is a particular type of social network, in which the relationship between two participants in the network itself consists in having collaborated in the realization of some artifact (be it a book, a film, or a software). In the case of IMDB, the corresponding collaboration network includes, as participants, all the actors registered in the data-set: two actors are linked together if they have acted together in at least one movie. For example, Marcello Mastroianni and Sofia Loren are linked together having acted together in several films, including the beautiful *A special day* by Ettore Scola. On the contrary, Marcello Mastroianni is not connected to Christian De Sica, as the two actors have never acted together in the same film. We note that in the definition we have just provided of the IMDB collaboration network, we are not taking into account the number of collaborations that took place between two actors: this information, on the other hand, could obviously be interesting if our research were to take into account in some way the "strength" of existing relationships.

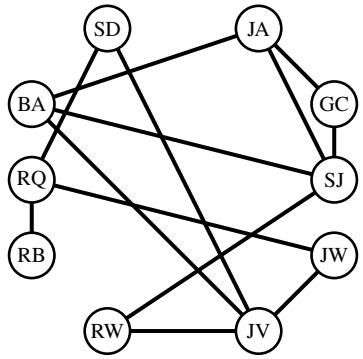
Let us now see how the collaboration network between actors can be used to deduce information about the actors themselves, which was not explicitly included in the IMDB data-set. For this purpose, let us consider a tiny part of the entire IMDB collaboration network, made up of just the following ten American actors: Ben Affleck, Jennifer Aniston, Roseanne Barr, George Clooney, Stacey Dash, Scarlett Johansson, Randy Quaid, Jon Voight, Reese Witherspoon, and James Woods. Based on the IMDB data-set,[1] we can state that:

- Ben Affleck collaborated with Jennifer Aniston and with Scarlett Johansson in *He's Just Not That Into You* (2009), and with Jon Voight in *Pearl Harbor* (2001);
- Jennifer Aniston collaborated with George Clooney in *Waiting for Woody* (1998) and with Scarlett Johansson in *He's Just Not That Into You* (2009);
- Roseanne Barr collaborated with Randy Quaid in *Home on the Range* (2004);
- George Clooney collaborated with Scarlett Johansson in *Hail, Caesar!* (2016);
- Stacey Dash collaborated with Randy Quaid in *Moving* (1988), and with Jon Voight in *Roe v. Wade* (2020);
- Scarlett Johansson collaborated with Reese Witherspoon in *Sing 2* (2021);
- Randy Quaid collaborated with James Woods in *Curse of the Starving Class* (1994);
- Jon Voight collaborated with Reese Witherspoon in *Four Christmases* (2008), and with James Woods in *An American Carol* (2008).

On the ground of this information, we can build the corresponding collaboration network between these ten actors shown in Figure 1.5, in which each of the actors is represented by a node with a label equal to the initials of his first and last name. Also in this case, the random visualization of the graph does not help to steal from the graph itself new information that can be useful for understanding the social mechanisms on which the graph was formed (in addition to those related to co-participation in a movie).

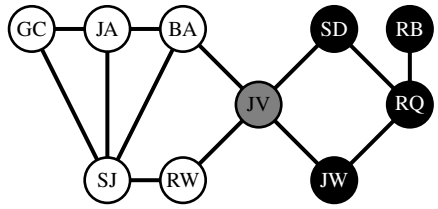
However, we can redraw the collaboration network of the ten actors as shown in Figure 1.6.

[1] Actually, for the sake of simplicity, we used the web service available at `https://oracleofbacon.org/movielinks.php` in order to find the collaborations between the ten actors (last visit on December 28, 2021). All the movie, actor, and actress data used on this web page are extracted from the English-language Wikipedia pages: these data might then differ from the IMDB data-set.



BA: Ben Affleck
JA: Jennifer Aniston
RB: Roseanne Barr
GC: George Clooney
SD: Stacey Dash
SJ: Scarlett Johansson
RQ: Randy Quaid
JV: Jon Voight
RW: Reese Witherspoon
JW: James Woods

Figure 1.5: The collaboration network of ten American actors: for each of the actors, the graph includes a node labeled with the initials of his name and surname and, for each pair of actors, the graph includes an arc between the two corresponding nodes if the two actors starred together in at least one movie.



BA: Ben Affleck
JA: Jennifer Aniston
RB: Roseanne Barr
GC: George Clooney
SD: Stacey Dash
SJ: Scarlett Johansson
RQ: Randy Quaid
JV: Jon Voight
RW: Reese Witherspoon
JW: James Woods

Figure 1.6: The collaboration network of the ten American actors drawn in such a way as to highlight the existence of two communities: the one represented by the white colored nodes (which correspond to actors and actresses who have endorsed Joe Biden's campaign for President of the United States in the 2020 U.S. presidential election) and the one represented by the black colored nodes (which correspond to actors and actresses who publicly indicated support for Donald Trump in the 2020 U.S. presidential election). The gray node also corresponds to a supporter of Donald Trump, but evidently open to collaborating with actors with politically different ideas.

In this visualization it is evident that there are two communities formed by five and four nodes, respectively, and shown in the figure with white and black color, respectively. The nodes of one community are not connected to any of the nodes of the other community: if it were not for the node in gray, the two communities would be totally detached from each other. Can we interpret the existence of these two communities from a more sociological point of view? In other words, is there any sociological reason that justifies the appearance of these two communities?

The answer to this question might be positive. Indeed, according to two Wikipedia pages [91, 92], all the actors included in the community whose nodes are colored in white have endorsed Joe Biden's campaign for President of the United States in the 2020 U.S. presidential election, while all the actors included in the community whose nodes are colored in black and the actor corresponding to the node colored in gray have publicly indicated support for Donald Trump in the 2020 U.S. presidential election. With the exception of Jon Voight (who corresponds to the gray node), this analysis seems to indicate that, in this set of actors, the democrats prefer to work together with the democrats and the republicans together with the republicans.

The search for communities within a graph is one of the problems most often faced in the field

of the analysis of large graphs. Indeed, when a graph includes thousands if not millions of nodes, no visualization will ever be able to highlight the existence of such communities. Their identification must therefore be carried out by an automatic method which, among other things, has to be also very fast. Many methods for identifying such communities have been proposed over the years and in this book we will analyze some of them in detail. Let us now see, in a more intuitive way, a simplification of one of these methods applied to the collaboration network of the ten actors, which will allow us to identify exactly the two communities of democratic and republican actors.

The idea is to identify among all the nodes the one which is needed more than the others for the (indirect) connection between all the pairs of the other nodes. Informally, a node a is needed to link two nodes b and c, if after removing a from the graph, there is no way to reach c starting from b and the other way around. For example, the node with label RQ of the collaboration network of the ten American actors is necessary for the connection between the node with the label JV and the one with the label RB: if we delete the node RQ from the graph, there is no way to reach RB from JV.

In the case of the ten-actor collaboration network shown in Figure 1.6, each white node must go through the gray node if it wants to reach a black node: therefore, the gray node is needed to connect $5 \times 4 = 20$ pairs of nodes (i.e. all pairs formed by any of the five white nodes and by any of the four black nodes). The black node labeled RQ is needed to connect any other node to the black node labeled RB: therefore, this node is needed to connect $8 \times 1 = 8$ pairs of nodes (that is, all pairs formed by any of the eight nodes with labels other than RQ and RB and the node with label RB). All other nodes in the collaboration network are not needed for any pair of nodes: for example, the node labeled JA is not needed to connect the node labeled GC to the node labeled BA, as this connection can still be obtained by passing through the node labeled SJ.

We call *articulation point* a node that is needed for at least one pair of nodes. If in Figure 1.6 we observe the two nodes with labels JV and RQ, respectively, we see how these nodes play the same role in the graphs as a joint of our human body, which has the task of holding together different bone segments. Similarly, an articulation point holds two components of the graph together: the JV node holds the white and black communities together, while the RQ node holds the RB node together with the rest of the graph. An articulation point, if eliminated from the graph, makes it impossible to connect those pairs of nodes that needed it to be connected, possibly highlighting those communities of nodes that, on the contrary, are less dependent on the removed node. Among all the articulation points, we consider the one most "necessary", that is the one that holds together the greatest number of pairs of nodes: in the case of the graph shown in Figure 1.6, it is the gray node. If we remove this node from the graph, the graph itself is broken into the two communities shown in the figure, namely the one with the white nodes and the one with the black nodes.

Despite its simplicity, this example shows how it is possible to divide a graph into two or more groups with an interesting "semantics", by analyzing only the structure of the graph itself and without making use of any additional information. In reality, things are more difficult than we have described in this section, and the methods used to identify communities are, generally, more complicated. Moreover, the validation of these methods is not always easy to carry out, as the very notion of community can often be ambiguous and difficult to quantify through a classification of the type white/black (i.e. democrat/republican), as we did in the example of the collaboration network of the ten American actors.

1.4 Imagining the future: predicting new relationships

The last example we consider in this introductory chapter refers to one of the best known social networks [98]. It is a social network associated with a university karate club, which has become a very popular example of a two-community structure in the field of complex network analysis. The social network includes 34 members of a karate club, linked together in case there was a documented interaction between two members outside the club itself: in total, the graph includes

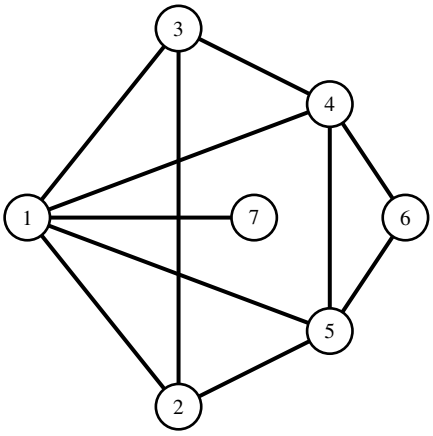


Figure 1.7: A part of the social network corresponding to a university karate club (see Figure 1 of [98]): the node labeled 1 corresponds to the teacher of the club, while the other nodes are some members of the club who interacted with the teacher outside the club itself.

78 arcs. Due to a conflict that arose between two members of the club (in particular, between the administrator and the teacher), the club itself split into two clubs: half of the original club members created a new club after finding a new teacher, while the remaining half of the members remained in the original club.

A graphical representation of this social network is shown as the background of the title of this chapter: in this graph, as we said, there are 34 nodes, which are connected by 78 arcs. In Figure 1.7 we show a small part of this graph consisting of the node corresponding to the teacher and some of his connections (this part corresponds to a "zoom" of the rightmost part of the graphic representation of the entire social network).

The question we ask about the graph shown in Figure 1.7 consists in predicting what future interactions will take place between network participants: in other words, predicting which new connections will form from those already existing. For example, node 2 and node 4 are both connected to node 1: based on the principle that "my friends' friends are my friends", we can assume that in the future the two nodes will interact directly by creating a new arc between them. Obviously, the same is true for nodes 2 and 7, 3 and 5, 3 and 7, 4 and 7, and 5 and 7 (as they are both connected to node 1), for nodes 1 and 6 and 3 and 6 (as they are both connected to node 4), and for nodes 2 and 6 (as they are both connected to node 5). But of all these potential new arcs which ones are most likely to be created in the near future? Obviously an exact answer is impossible, as no one knows the future of a network of social relations. However, we will see in this book some techniques that are particularly effective in carrying out this task. For now, we show one of the simplest of such techniques, based on the analysis of the "neighborhood" of a node.

The *neighborhood* of a node consists of all the nodes directly connected to it. For example, the neighborhood of node 1 of the graph shown in Figure 1.7 consists of nodes 2, 3, 4, 5 and 7. In the following table, we show in the second column, for each node of this graph, its neighborhood. The following columns show the elements in common between the neighborhood of the node i corresponding to the row and the neighborhood of the nodes different from i which are not in the neighborhood of node i (for simplicity, we have indicated with $\cap_j$ the elements in common with the neighborhood of node j, which is not in the neighborhood of node i).

Node	Neighborhood	$\cap_1$	$\cap_2$	$\cap_3$	$\cap_4$	$\cap_5$	$\cap_6$	$\cap_7$
1	{2,3,4,5,7}						{4,5}	
2	{1,3,5}				{1,3,5}		{5}	{1}
3	{1,2,4}					{1,2,4}	{4}	{1}
4	{1,3,5,6}		{1,3,5}					{1}
5	{1,2,4,6}			{1,2,4}				{1}
6	{4,5}	{4,5}	{5}	{4}				{}
7	{1}		{1}	{1}	{1}	{1}	{}	

The last seven columns of the previous table suggest a way of giving each of the connections that do not yet exist in the social network a "probability" that such a connection will be generated in the near future. Intuitively, the probability that an arc is created between the node i and the node j will be greater if greater is the cardinality of the set in the table in correspondence of the row labeled i and the column labeled $\cap_j$. Therefore, of all the possible candidate connections to be generated, the one between node 2 and node 4 and the one between node 3 and node 5 are more probable, since their neighborhood intersection contains three elements.

In conclusion, we can predict that in the near future node 2 will interact with node 4 or that node 3 will interact with node 5. Unfortunately, we do not have the data to verify the correctness of this prediction and, in general, verifying forecasts for the development of a social network is not an easy task, due to the lack of sufficient data to carry out this verification. However, there are situations and techniques that allow us to measure the "goodness" of a forecast: we will discuss them in this book.

1.5 Some definitions

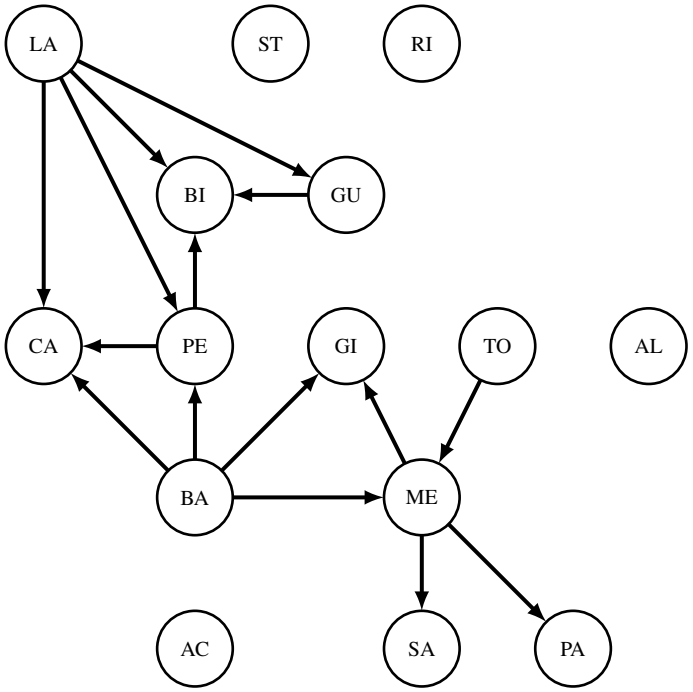
A *graph* $G = (V,E)$ is a pair formed by a set V of *nodes* and a set $E \subseteq V \times V$ of *arcs*. In general, we will denote by n the number of nodes and by m the number of edges.

1.5.1 Arc direction

If two actors participated in the same movie, a symmetrical relationship is created between them: in fact, if the actor x participated in the same movie as the actor y, then also the actor y has participated in the same movie as the actor x. In this case, we speak of a *undirected* graph (or simply graph): in formulas, if $(x,y) \in E$, then also $(y,x) \in E$. For simplicity and clarity of explanation, we will consider these two arcs as a single arc, and assume that the number of arcs m is equal to the number of such undirected arcs (that is, $m = |E|/2$). In many cases, however, the social relation represented by the graph is not symmetric. For example, if a scientific paper a_1 mentions another scientific paper a_2 in its bibliography, it is very likely that a_2 does not cite the paper a_1 in its bibliography, as a_2 has been be published later. In this case, we speak of a *directed* graph: in Figure 1.8 we show the direct social network corresponding to the business relations between the Florentine families of the fifteenth century, of which we had already seen the graph corresponding to family relations.

1.5.2 Density

We said that the social network of the university karate club has 34 nodes and 78 arcs. Can we say that this graph is "dense"? That is, are the nodes of the network connected to the maximum of their possibilities? To answer this question, we must first calculate the maximum number of edges in a social network containing n nodes. If the graph is directed, then each node can have at most $n-1$ arcs connecting it to the remaining $n-1$ nodes: therefore, the maximum number of directed edges is equal to $n(n-1)$. If the graph is undirected, then this number is simply divided by 2: therefore, in an undirected graph, the maximum number of edges is equal to $\frac{n(n-1)}{2}$. When an



AC: Acciauoli
AL: Albizzi
BA: Barbadori
BI: Bischeri
CA: Castellani
GI: Ginori
GU: Guadagni
LA: Lamberteschi
ME: Medici
PA: Pazzi
PE: Peruzzi
RI: Ridolfi
SA: Salviati
ST: Strozzi
TO: Tornabuoni

Figure 1.8: The social business network between Florentine families in the fifteenth century (see Table 1 in [16]): for each family, the graph includes a node labeled with the first two letters of the family name and, for each pair of families, the graph includes an arc from the first family to the second one if there has been an economic transaction (such as, for example, a loan).

undirected graph has the maximum number of edges, we will say that it is a *complete graph* or a *clique*: in Figure 1.9 we show a clique with five nodes and, therefore, with $\frac{5\cdot 4}{2} = 10$ arcs.

The *density* of a graph G with n nodes can then be defined as the ratio between the number m of edges in the graph and the maximum possible number of edges in a graph with n nodes, that is

$$\delta_G = \frac{m}{\frac{n(n-1)}{2}} = \frac{2m}{n(n-1)}$$

(similarly we can define the density of a directed graph). In the case of the social network of the university karate club, we have that the density is equal to

$$\delta = \frac{2 \cdot 78}{(34 \cdot 33)/2} = \frac{156}{1122} \approx 0.14,$$

where we omitted to specify the name of the graph as a subscript (which we will do whenever the referenced graph itself is clear from the context). This value is close to 0, so we can conclude that the karate club's social network is not dense, that is, it is *sparse*. As we will see in this book, this is the situation that most frequently arises when dealing with real graphs.

1.5.3 Node labels

In the social network shown in Figure 1.7 we have not associated any additional information to the nodes (apart from the identification number of each node): in this case, the graph is said to be *unlabeled*. In the other examples of this chapter, we have instead associated to each node a *label* which indicated, for example, the name of a Florentine family, the name of an archaeological site, or the name of an actor: in this case, we are talking about a *labeled* graph. The label of a node can

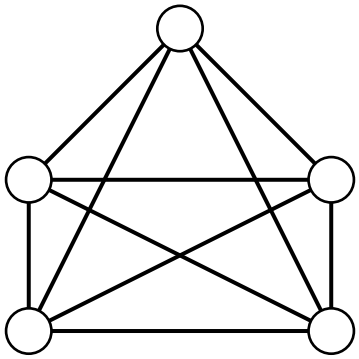


Figure 1.9: A complete graph or clique with 5 nodes and $\frac{5\cdot 4}{2} = 10$ arcs.

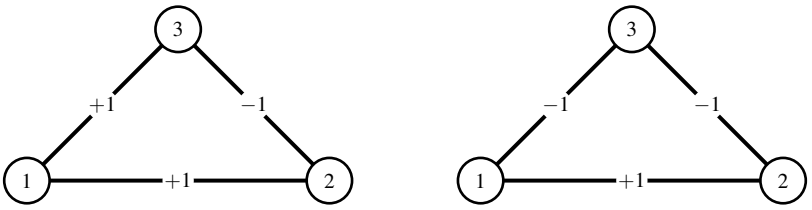


Figure 1.10: Two examples of a graphs weighted with $+1$ and -1 weights: the left graph is unstable as node 1 is "friend" of two nodes that are not friends, while the right graph is stable as nodes 1 and 2 are friends and both are not friends of node 3.

include multiple information: for example, in addition to the name of an actor, in Figure 1.6 we have represented by the color of the node another information, namely its "political orientation". Other typical information included in node labels may be age, gender, or school attended (although, for *privacy* reasons, this information is much more difficult to find and use).

1.5.4 Edge weights

In the graphs shown in the previous paragraphs, no additional information has been associated with their edges: in this case, we speak of *unweighted* graphs. However, in many cases it is natural to associate each edge with a *weight* that somehow indicates the strength of the relationship represented by the arc itself: in this case, we speak of a *weighted* graph. For example, the social network of actors shown in Figure 1.5 could be modified by associating each arc of the graph with an integer indicating the number of movies in which the two actors, linked by the arc, have performed together. A particular type of weighted graph, which we will discuss in this book, is one in which the weights of the edges can only be $+1$ and -1 (see Figure 1.10): in other words, these weights indicate whether the relationship between the two nodes is a relationship of "friendship" or "enmity", such as, for example, alliances between different political states. The two graphs shown in the figure represent two radically different situations with respect to an intuitive concept of stability of the graph. In the first graph, node 1 is a friend of both node 2 and node 3, but these last two nodes are not friends: such a situation (quite frequent in reality) creates stress at node 1. It is, hence, likely that sooner or later the situation will change. On the other hand, in the second graph node 1 and node 2 are friends and both are not friends of node 3: this situation (even more frequent than the previous one) should not create any tension. The network is likely to remain stable in this configuration.

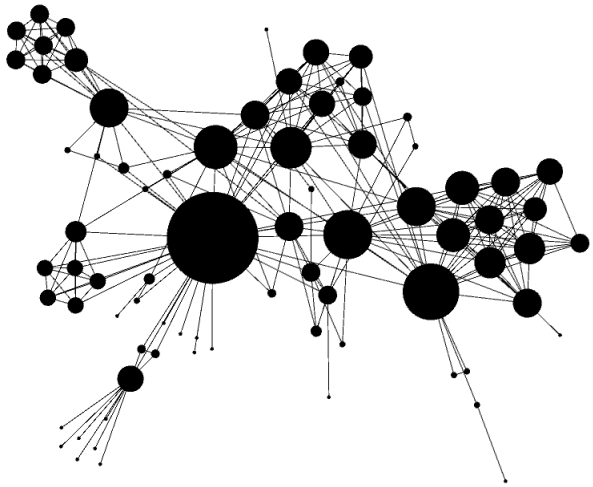


Figure 1.11: The graph of the characters of the novel *Les miserables* [52]: each node corresponds to a character and, for each pair of nodes, there is an arc if the two characters appear in the same chapter. The size of the nodes indicates the degree of the nodes themselves: as expected, the largest node (i.e. the one with the highest degree) corresponds to Jean Valjan, who is the protagonist of the novel.

1.5.5 Neighbors, neighborhood, and degree

In a graph the *neighbors* of a node x are all the nodes connected to x by an arc: for example, in the graph of Figure 1.7 the neighbors of node 1 are nodes 2, 3, 4, and 7. The *neighborhood* $N(x)$ of a node x is the set of all its neighbors: for example, in the graph of Figure 1.7 $N(1) = \{2,3,4.7\}$. The *degree* $d(x)$ of a node x is the number of arcs that exit from the node itself, that is the cardinality of $N(x)$: for example, in the graph of Figure 1.7 $d(1)$ is equal to 4. The degree of a node is often considered as an index of the importance of the node itself. For example, consider the "virtual" social network whose nodes are characters from Victor Hugo's *Les miserables*, and whose arcs indicate whether two characters have appeared in the same chapter of the novel (see Figure 1.11). In this network, the node with the highest degree is the node that corresponds to the main character of the novel, namely Jean Valjan: his rank is 32. The node with the second largest degree is the node that corresponds to the character Gavroche, a boy who lives on the streets of Paris and who plays a short but important role in the novel (for a very interesting book on the analysis of social networks based on narratives, we refer the reader to [63]). We observe that in a complete graph each node has degree equal to $n-1$, while in a *star*, that is a graph in which a node, called *center* of the star, is connected to the others $n-1$ nodes and there are no other edges, the center has degree $n-1$ and all the other nodes have degree 1.

In the case of directed graphs, we distinguish between the *in-degree* of a node x (that is, the number of edges (y,x) that "enter" in x) and the *out-degree* (that is, the number of edges (x,y) that "exit" from x). For example, in the directed graph of the Florentine families shown in Figure 1.8, the in-degree of the node corresponding to the Medici family is equal to 2, as there are two arcs entering this node: one from the node corresponding to the Barbadori family and one from the node corresponding to the Tornabuoni family. The out-degree of the same node is instead equal to 3, as there are three arcs that exit from this node: one directed to the node corresponding to the Ginori

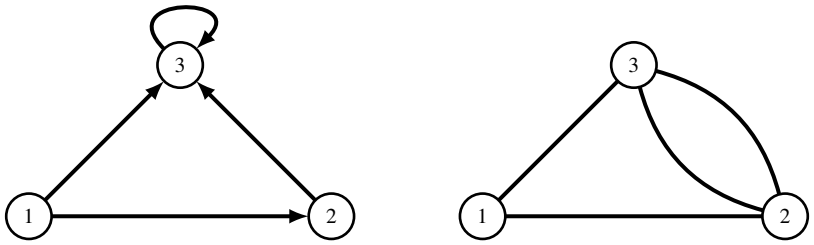


Figure 1.12: An example of a directed graph with self-loop (left) and an example of a graph with multi-arcs.

family, one to the node corresponding to the Pazzi family, and one to the node corresponding to the Salviati family. In some cases, it may be useful to make the arcs "symmetrical" by eliminating their direction, and then consider the degree of the undirected graph resulting from this operation: in the case of the directed graph of the Florentine families, we would have, therefore, after this operation of eliminating the direction of the arcs, that the degree of the node corresponding to the Medici family would be equal to 5.

1.5.6 Simple graphs

In the graph of the American actors, we have not included an arc between a node and itself (after all, an actor participates with itself in the same movie). Such kind of arc (that is, an arc of the type (x,x)) is called *self-loop* (see the left part of Figure 1.12). Furthermore, in all the examples we have seen so far, only one arc can exist between two nodes: however, if the social network wants to represent multiple social relations it may make sense to include multiple arcs between two nodes. In this case we speak of *multi-arcs* (see the right part of Figure 1.12). A graph that does not include self-loops and multi-arcs is said to be *simple*. In this book we will analyze only simple graphs, but it is good to know that in some situations the introduction of self-loops and of multi-edges is useful, if not necessary.

1.5.7 Paths, eccentricity, and diameter

We have already talked about the concept of indirect connection between two nodes of a graph in the examples relating to the Japanese archaeological sites and to the American actors. A *path* between two nodes x and y is a sequence of arcs $(x,x_1),(x_1,x_2),\ldots,(x_k,y)$: the *path length* is equal to the number of arcs in the sequence, that is k. For example, in the graph of the American actors there are two paths of length 3 between the nodes labeled BA and RQ, i.e. the path $(\texttt{BA},\texttt{JV}),(\texttt{JV},\texttt{JW}),(\texttt{JW},\texttt{RQ})$ and the path $(\texttt{BA},\texttt{JV}),(\texttt{JV},\texttt{SD}),(\texttt{SD},\texttt{RQ})$. We observe that in a graph every path between a node x and a node y of length k is also a path of the same length between the node y and the node x. This statement is not true if the graph is directed. For example, in the directed graph of the Florentine families shown in Figure 1.8 there exists a path of length 2 from the node corresponding to the Barbadori family to the node corresponding to the Pazzi family, which passes through the node corresponding to the Medici family, but there is no path that goes from the node corresponding to the Pazzi family to the node corresponding to the Barbadori family.

In many cases, we will be interested in finding a path of minimum length, called *shortest path*. The shortest path length between two nodes x and y is called the *distance* between x and y and is denoted by $d(x,y)$. Clearly, if the graph is undirected, then $d(x,y) = d(y,x)$. If there is no path between the two nodes x and y, then, by convention, we set $d(x,y) = \infty$. The following table (similar to the one we have seen in the case of the Japanese archaeological sites) shows all the distances between the nodes of the graph of the American actors shown in Figure 1.6.

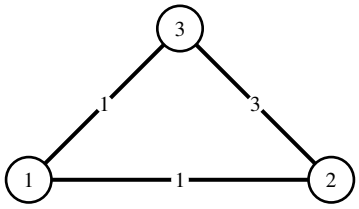


Figure 1.13: In this weighted graph, the shortest path between node 2 and node 3 has length 2 and passes through node 1: in fact, the arc connecting node 2 to node 3 has weight equal to 3 and, therefore, is a longer path.

	BA	JA	RB	GC	SD	SJ	RQ	JV	RW	JW	**Eccentricity**
BA	0	1	4	2	3	1	3	1	2	2	4
JA	1	0	5	1	3	1	4	2	2	2	5
RB	4	5	0	6	2	5	1	3	4	2	6
GC	2	1	6	0	4	1	5	3	2	4	6
SD	3	3	2	4	0	3	1	1	2	2	4
SJ	1	1	5	1	3	0	4	2	1	3	5
RQ	3	4	1	5	1	4	0	2	3	1	5
JV	1	2	3	3	1	2	2	0	1	1	3
RW	2	2	4	2	2	1	3	1	0	2	4
JW	2	2	2	4	2	3	1	1	2	0	4
								Diameter			6

The last column of the previous table indicates the *eccentricity* $e(x)$ of a node x, which is defined as its maximum distance to the other nodes it is connected to through a path: in other words,

$$e(x) = \max_{y:d(x,y)\neq\infty} \{d(x,y)\}.$$

The *diameter* D_G of a graph G is defined as the maximum eccentricity of its nodes: for example, the diameter of the graph of the American actors is equal to 6.

The above definitions refer to an unweighted graph. If the graphs is weighted, then the same definitions apply apart from the definition of the length of a path: in this case, in fact, the length of a path is defined as the sum of the weights of its arcs. We note that a shortest path in a weighted graph does not necessarily coincide with the path containing the least number of arcs, as shown in the example of Figure 1.13.

1.5.8 Connected graphs

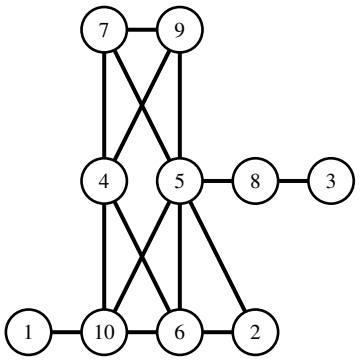
If, for each pair of nodes x and y of a graph, there exists a path from x to y, then the graph is called *connected*. For example, the graph of the Japanese archaeological sites is connected. If the graph is directed, then we distinguish between two cases. If, for each pair of nodes x and y there exists a path from x to y, then the graph is called *strongly connected*. If a directed graph is not strongly connected but, by eliminating the orientation of the arcs, we obtain a connected graph, then the directed graph is called *weakly connected*. For example, the directed graph of the Florentine families shown in Figure 1.8 is neither strongly nor weakly connected, as the nodes corresponding to the families of Acciauoli, Albizzi, Ridolfi and Strozzi are not reachable by any path. However, if we eliminate these families, the resulting graph is weakly connected (since the graph obtained by eliminating the orientation of the arcs is connected), but not strongly connected (since there is, for example, no path from the node corresponding to the Pazzi family to the node corresponding to the Medici family).

1.6 Graph representations

The representation of a graph used within a computer program is a very important factor for the effective management of the graph itself and for the design of efficient algorithms. There are two main ways of representing a graph within a computer's memory: adjacency matrices and adjacency lists.

1.6.1 Adjacency matrices

The *adjacency matrix A* of a graph is a two-dimensional table with as many rows and columns as there are nodes in the graph. The rows and columns are indexed with the node indices, so if the graph includes n nodes, then the rows and columns are indexed with the numbers from 1 to n. The element $A[i][j]$ of the table, corresponding to the row with index i and to the column with index j, is equal to 1 if there is an arc connecting node i to node j. Otherwise, that element is equal to 0. For example, referring to the social network of the Japanese archaeological sites, we assume that node indices are assigned according to the alphabetical order of node labels. The following table (on the right) is the adjacency matrix of this social network (which on the left is redrawn with the indexes of the nodes shown instead of the labels).



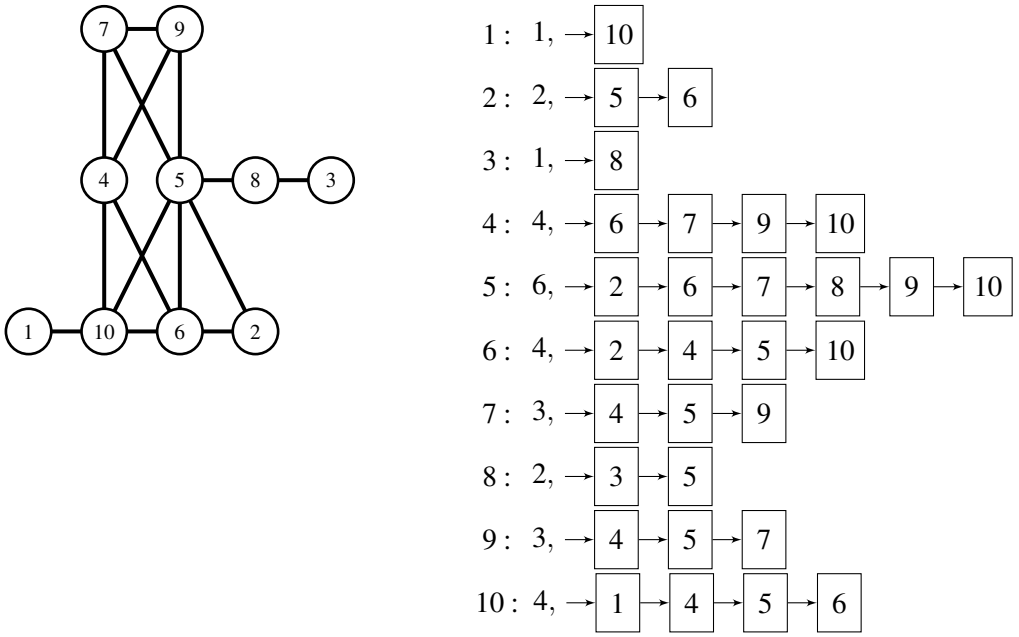
	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	1	1	0	0	0	0
3	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	1	1	0	1	1
5	0	1	0	0	0	1	1	1	1	1
6	0	1	0	1	1	0	0	0	0	1
7	0	0	0	1	1	0	0	0	1	0
8	0	0	1	0	1	0	0	0	0	0
9	0	0	0	1	1	0	1	0	0	0
10	1	0	0	1	1	1	0	0	0	0

Note that, since the graph is undirected, the previous table is symmetric, that is $A[i][j] = A[j][i]$. Clearly, this statement is not true if the graph is directed. The advantage of this representation is, first of all, the fact that it immediately allows us to know if two nodes are connected by an arc. Furthermore, the representation of graphs by means of matrices allows the use of various tools typical of linear algebra and useful for developing relatively efficient algorithms, especially in the case in which such matrices are sparse (i.e. they contain "few" 1s). On the other hand, this representation has the disadvantage of using a lot of computer memory: in principle, an adjacency matrix requires a number of elements equal to the square of the number of nodes, regardless of the number of arcs present in the graph (and, therefore, by the number of 1s present in the matrix). For example, the previous table requires 100 elements (45, if we only consider the upper right part due to the fact that the matrix is symmetric), but the table itself contains only 30 elements equal to 1 (15, if we consider only the upper right). Furthermore, a relatively simple operation such as calculating the degree of a node requires, with the adjacency matrix, a time equal to the number of nodes in the graph, regardless of how many are the neighbors of the node (and, therefore, the 1s present in the row corresponding to the node).

If the graph is weighted, then each element of the table contains the weight of the arc connecting two nodes i and j, if such an arc exists. Otherwise, the element $A[i][j]$ is set equal to the special symbol $\perp$.

1.6.2 Adjacency lists

To avoid the memory waste problem that arises especially in the case of sparse graphs (that is, graphs with relatively few arcs), another representation of graphs is often used. The *adjacency list* of a node i is the sequence of its neighbors, i.e. the list of all nodes j such that the (i, j) arc exists. To create this list within the computer memory, each node is associated with the number of its neighbors (that is, the size of its adjacency list), and with each element of the list a reference to the next element of the list is associated if it exists (otherwise, the special value $\perp$ is associated). For example, the representation by adjacency lists of the graph of the Japanese archaeological sites is shown in the right part of the following figure, in which for each node of the graph the degree of the node is shown followed by the list of its neighbors.



The adjacency list of a node occupies a memory that is proportional to the number of its neighbors, that is to its degree. Overall, remembering that, in an undirected graph, the sum of the degrees of a graph is equal to twice the number of arcs of the graph itself, the memory occupied by this representation of an undirected graph is equal to $\mathsf{c}\sum_{i=1}^{n} d(i) = 2\mathsf{c}m$, where c is a constant value. If the graph is directed, then we usually associate to a node both the list of its out-neighbors (that is, the nodes j such that the arc (i, j) exists) and the list of its in-neighbors (that is, the nodes j such that the arc (j, i) exists): this last list is called *incidence list*. The memory occupation of this representation is always proportional to the number of arcs of the graph. Finally, in the case in which the graph is weighted, it will be sufficient to include in the elements of the adjacency list (and, possibly, of the incidence list), in addition to the neighbor index, also the weight of the arc connecting the two nodes.

The main disadvantage of representing a graph by means of adjacency (and incidence) lists is that, in order to decide whether a node j is a neighbor of a node i, it is necessary to scroll, in the worst case, the entire adjacency list of i. Therefore, this operation requires a time proportional to the degree of the node i, which could also be close to the number of nodes in the graph. However, as we have already observed, real-world graphs are usually sparse, so the degree of a node is generally a very small value compared to the number of nodes in the graph itself.

Finally, the representation of a graph through adjacency lists has the advantage of being very effective for the development of algorithms for the "visit" of a graph, as we will see in the next paragraph.

1.7 A first algorithm: the breadth-first search

Once we have decided how to represent a graph, we want to make a visit of it that allows us to calculate other properties of the graph itself (besides, for example, the maximum degree or the average degree of its nodes). One of the most popular algorithms for carrying out such a visit, in the case of unweighted graphs) is the *breadth-first search* (in short, *BFS*), which starts from a specific starting node s and analyzes all the nodes that can be reached through a path. The main purpose of the BFS is to assign two values to each node x of the graph: the first value is the length of the shortest path from s to x, while the second value is the node y that precedes it in one of the shortest paths from s to x. In other words, the goal is to partition the nodes of the graph into successive levels: the nodes located in the level h with $h \geq 1$ are at distance h from s and have associated one and only one node that is at the previous level (assuming that at level 0 there is only the starting node s).

Let us consider, for example, the graph of the Japanese archaeological sites in Figure 1.3, which we show also in the left part of Figure 1.14. The BFS carried out from node 1 partitions the remaining nodes of the network into 4 levels (see the right part of the figure): the first level contains only the node 10, which is at distance 1 from node 1, the second level contains the nodes 4, 5, and 6, which are at distance 2 from node 1, the third level contains the nodes 2, 7, 8, and 9, which are at distance 3 from node 1, and the fourth level contains only node 3, which is at distance 4 from node 1. Each node (except node 1) is connected to a node at the previous level that precedes it in a path from node 1 to it. For example, node 8, which is at the third level, is connected to node 5 of the second level, as 5 precedes 8 *in the only* shortest path from node 1 to node 8 (which passes through the node 10 and the node 5), while the node 9 is connected to the node 4, as 4 precedes 9 *in one* of the shortest paths from node 1 to node 9 (the one that passes through node 10 and through node 4). We note that, in the case of node 9, it could also have been connected to the node 5 that precedes it in another shortest path from node 1 to node 9.

1.7.1 Implementation of the BFS

The implementation of the BFS makes use of a very popular data structure in computer science, namely the *queue*. This data structure manages the insertion and extraction of values in a completely similar way to how a waiting queue is managed (for example, at the post office): the first value to be inserted is also the first to be extracted (for this reason, a queue is also said to be a *FIFO* data structure, from *First In First Out*). The implementation of a queue is a classic topic of an algorithm and data structure course of the first year of a degree in computer science and it is included in numerous libraries developed in different programming languages. In this section, we show how a queue can be used to perform the BFS of a graph.

For this purpose, the BFS "marks" each node the first time it "visits" it and keeps track of the nodes that have already been visited but not yet "explored". More precisely, during the execution of the BFS, each node will be in one of three possible states: *not visited*, *visited*, and *explored*. A node that is not visited has not yet been reached by the BFS: in other words, that node has not yet been assigned to a specific level. The first time a node is reached, it becomes visited (and, therefore, is assigned a level and a predecessor) and is placed in a queue waiting to be "served" (i.e. waiting to examine its neighborhood). When a node is served, that is, it is pulled from the queue, it becomes explored and will no longer play any role within the BFS.

More precisely, the BFS performs the following operations (we assume that at the beginning all nodes are marked as not visited).

1. The starting node s is added to the queue, after being marked as visited and after assigning it 0 as a level and itself as a predecessor.
2. While the queue is not empty
   (a) The first node x of the queue is extracted from the queue and marked as explored: let l

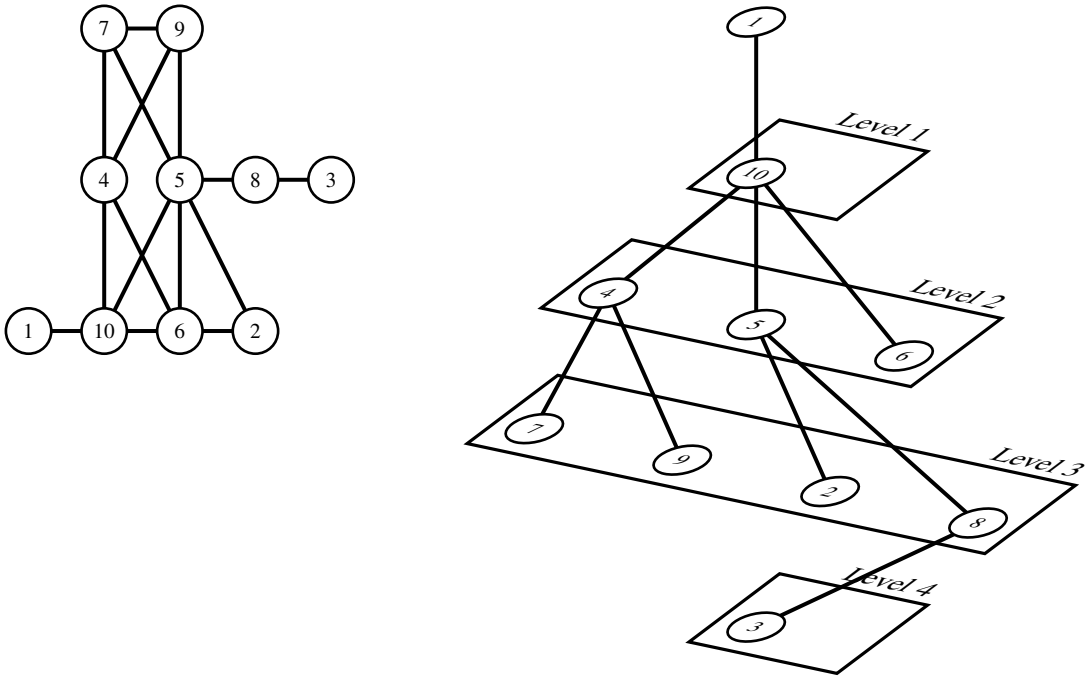
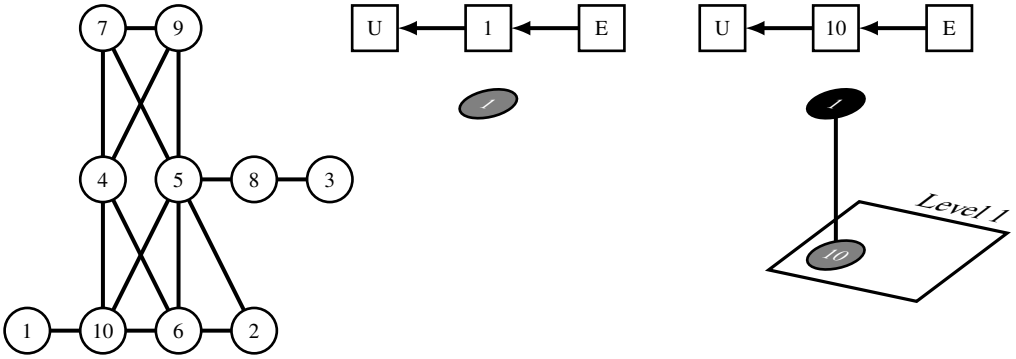


Figure 1.14: The subdivision into levels (right) carried out by the BFS on the graph of the Japanese archaeological sites (left), starting from the node 1. The nodes at level i are at distance i from node 1 and are connected to the node of the previous level that precedes them in a shortest path from 1 to them.

be the level assigned to x.

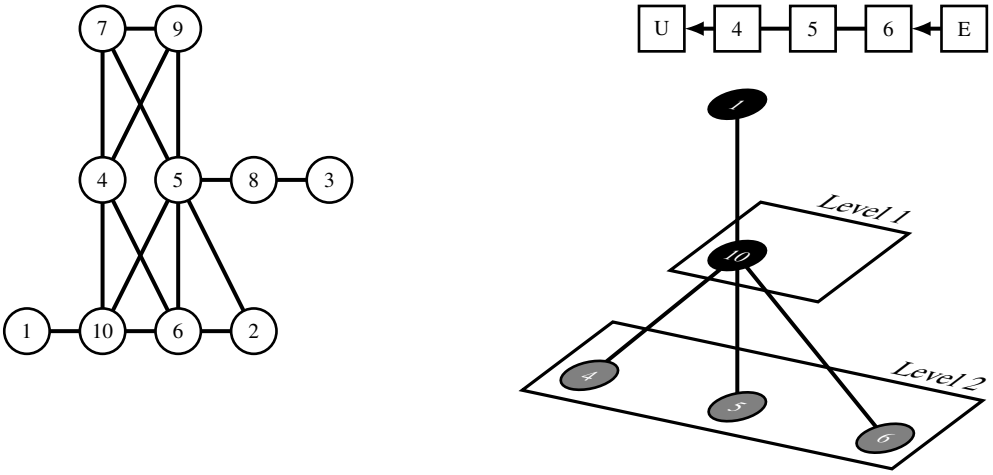
(b) For every neighbor y of x (i.e., for every node y in the adjacency list of x), if y is neither visited nor explored, then y is inserted in the queue, after being marked as visited and after assigning it $l+1$ as a level and x as a predecessor.

Let us see how the BFS operates on the graph of the Japanese archaeological sites, starting from the node 1. Initially, this node is marked as visited and placed in the queue, as shown in the central part of the following figure, where the queue is shown at the top of the figure (with an entry point E and an exit point U) and a visited node is shown in gray color.

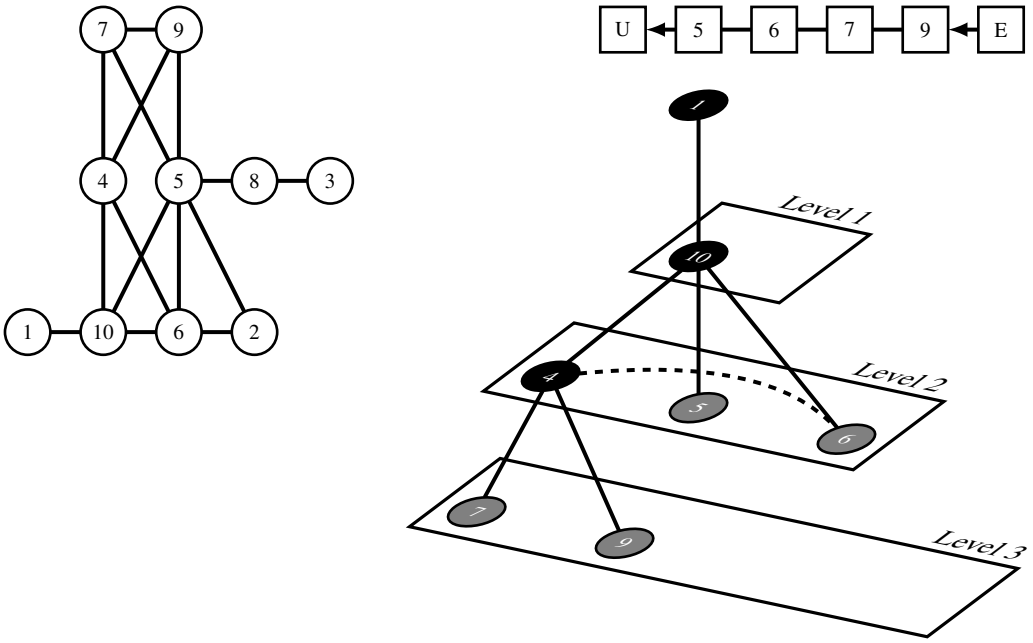


Next, node 1 is extracted from the queue and marked as explored and its neighbors are examined. In this case we have only one neighbor, that is the node 10 which is neither visited nor explored: therefore, the node 10 is inserted in the queue, added to the level 1 and connected to the node 1 which is its predecessor, as shown in the right part of the previous figure, where an explored node is shown in black.

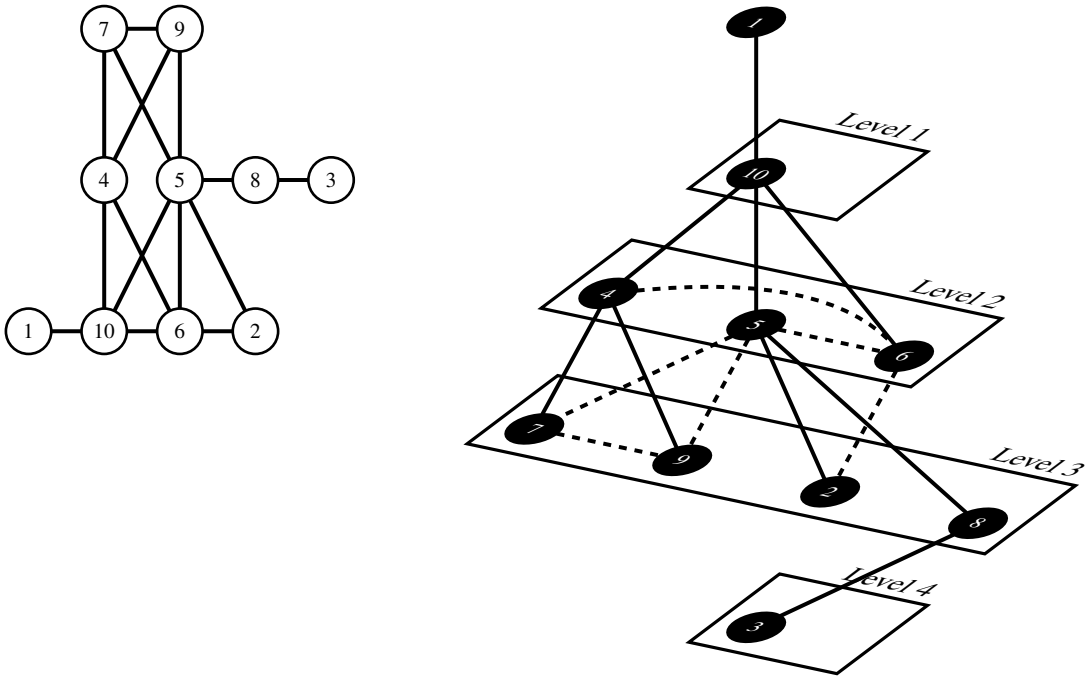
The node 10 is now pulled from the queue and marked as explored and its neighbors are examined. The node 1 is explored and, therefore, is not further considered: the other neighbors of the node 10, that is the nodes 4, 5, and 6 are neither visited nor explored. Therefore, these nodes are queued, added to level 2 and linked to the node 10 which is their predecessor.



At this point, node 4 exits the queue and is marked as explored. Its neighbors are the nodes 6, 7, 9 and 10. The node 6 is visited (and is in fact present in the queue), so it is not put back in the queue. Nodes 7 and 9 are neither visited nor explored. Therefore, these nodes are queued, added to level 3 and linked to the node 4 which is their predecessor. Finally, the node 10 is explored and, therefore, is not considered further. We note that in this case, the arc that connects the node 4 to the node 6 is not used for the connections between the various levels: it is, in reality, a "crossing" arc that we represent in the following figure as dashed.



In the subsequent iterations performed by the visit, the node 5 is extracted from the queue (which causes the nodes 2 and 8 to be inserted in the queue), the nodes 6, 7, 9, 2, 8 are extracted (the node 8 causes the node 3 to be queued), and the 3 is extracted from the queue. The final result is the expected one and is shown in the following figure, which shows not only the connecting arcs between levels but also the crossing ones.



According to the definition of the BFS, each node is explored only once, that is when it is extracted from the queue. When this happens all its neighbors are examined (and eventually added to the queue): therefore, the exploration of a node requires a number of operations proportional to the degree of the node itself. Therefore, the total number of operations of the BFS is proportional to the sum of the degrees of the nodes, which, as we have already seen, is proportional to the number of arcs of the graph. In this case, we say that the *time complexity* of the BFS is *linear* in the dimension of the graph: in this book, we will always try to design algorithms that have linear time complexity, since the size of real-world graphs is so large that a more than linear number of operations may be unacceptable.

1.7.2 Using the BFS

In this book, we will refer to the `Julia` programming language [49] and, in particular, the `Graphs` package [38]. This package includes, of course, an implementation of the BFS. In particular, it includes the `bfs_parents` function which, receiving in input a graph and a starting node, returns for each node of the graph its possible predecessor in a shortest path from the starting node.

```
g = loadgraph("graphs/japan.lg","graph")
p = bfs_parents(g, 1)
println(p)
```

By executing the previous code (in which we assume that the file `japan.lg` is the representation in `Graphs` format of the graph of the Japanese archaeological sites and is contained in the `graphs` directory), the list of predecessors is printed, for each node of the graph, that is the following list: $[1,5,8,10,10,10,4,5,4,1]$ (we observe that the predecessor of the starting node of the BFS is set by convention equal to itself). For example, the predecessor of node 3 is node 8 (which is in the third position in the list), the predecessor of node 8 is node 5 (which is in the eighth position), the predecessor of node 5 is node 10 (which is in the fifth position), and the predecessor of node 10 is node 1 (which is in the tenth position): hence, a shortest path from the node 1 to the node 3 is the one passing through nodes 10, 5, and 8.

Another function of the `Graphs` package that uses the BFS is the function that checks whether

the graph is connected: in fact, this is true if and only if all the nodes of the graph are visited by BFS (in other words, if all nodes of the graph are included in at least one level of the BFS). The following code performs this verification.

```
c = is_connected(g)
println(c)
```

Running the above code prints the value `true` indicating that the graph of the Japanese archaeological sites is connected.

Note that if the graph is not connected, the BFS contains in its levels a *connected component*, i.e. a maximal sub-graph of the graph that is connected and that contains the starting node of the BFS. To find the other connected components, it is sufficient to carry out the BFS starting from a node that has not been visited. By repeating this procedure until we have visited all the nodes, we can calculate all the connected components of the graph. The `Graphs` package includes a function that performs this procedure to compute all the connected components of a graph, as shown in the following code (where we assume that the `notconnectedjapan.lg` file is contained in the `graphs` directory and is the representation in `Graphs` format of the graph of the Japanese archaeological sites in which the arc from the node 8 to the node 3 has been removed).

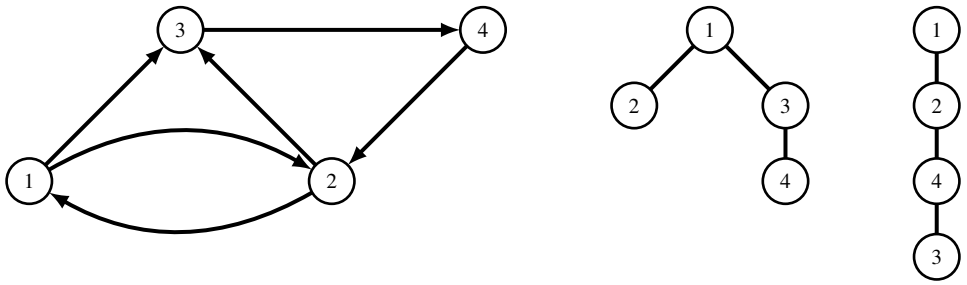
```
g = loadgraph("graphs/notconnectedjapan.lg","graph")
cc = connected_components(g)
println(cc)
```

By executing the previous code, the following list of the two connected components is printed: $[[1,2,4,5,6,7,8,9,10],[3]]$. The first component includes all nodes except the node 3, which alone forms the second connected component.

In conclusion, the BFS is a simple, efficient and powerful tool that is often useful in the analysis of graphs (in particular, undirected).

1.7.3 Directed and/or weighted graphs

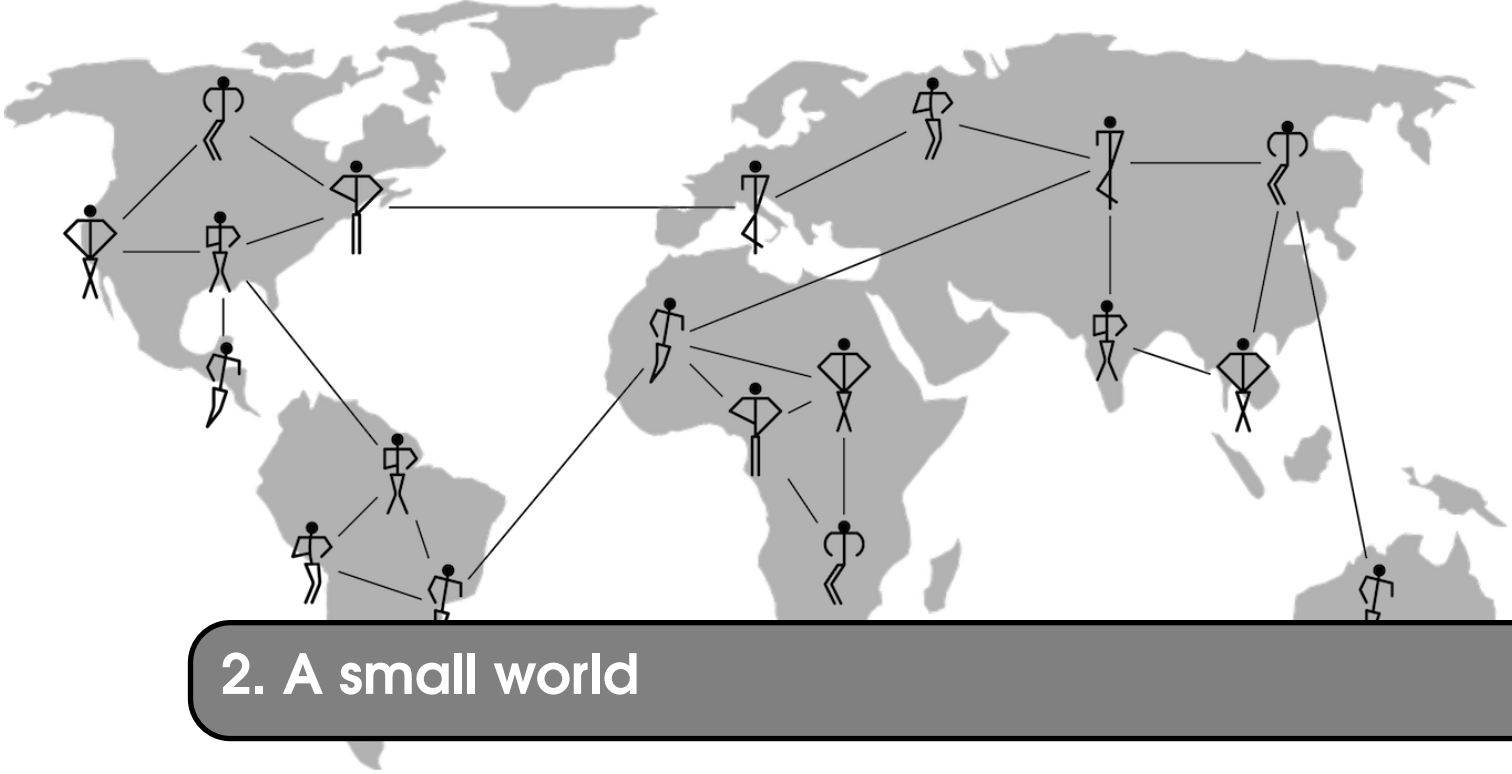
The BFS of a directed graph allows us to find all the nodes reachable by the starting node, even if it is not said that this node can in turn be reached by all such nodes. For this reason, a single BFS is not able to verify if a directed graph is strongly connected. If we wanted to do this test with BFSs, we would have to do two BFSs: one on the original graph and one on the graph with all arcs "inverted". The first BFS is called the *forward* BFS, while the second is called the *backward* BFS. If both BFSs visit all the nodes of the graph, we can conclude that the graph is strongly connected, as each node is reachable from the starting node (forward BFS) and each node can reach the starting node (backward BFS). For example, consider the directed graph on the left side of the following figure: this graph is strongly connected. The forward BFS starting from node 1 produces the level partition of the graph shown in the central part of the figure, while the backward BFS produces the partition shown in the right part of the figure.



Since all nodes of the graph are visited by both BFSs, we can conclude that the graph is strongly connected.

However, this procedure does not apply effectively (i.e., in linear time) to the problem of calculating the *strongly connected components* of a directed graph, that is, the maximal directed sub-graphs of the graph that are strongly connected. To solve this problem, it is better to make use of a different type of visit, called *depth-first search* (in short, *DFS*). The `Graphs` package includes the implementation of the DFS and also a function for calculating the strongly connected components: this will be discussed later in the book.

Finally, we have already noticed that in a weighted graph, the length of a shortest path does not correspond to the number of arcs crossed by the path. For this reason, the BFS is unable to compute shortest paths in a weighted graph. For this purpose, more sophisticated algorithms are used, the most popular being the Dijkstra algorithm (which, however, only works if the weights of the edges are not negative). The `Graphs` package includes the implementation of this algorithm and, therefore, a function for computing shortest paths in a weighted graph (actually, it includes more than one).



2. A small world

I read somewhere that everybody on this planet is separated by only six other people. Six degrees of separation between us and everyone else on this planet. The President of the United States, a gondolier in Venice, just fill in the names. I find it extremely comforting that we're so close. I also find it like Chinese water torture, that we're so close because you have to find the right six people to make the right connection... I am bound, you are bound, to everyone on this planet by a trail of six people.

J. Guare, "Six degrees of separation", 1990

2.1 The small world problem

What is the probability that I will know you, reader? And if I don't know you, what is the probability that you and I have a mutual friend? And if we don't have a mutual friend, what is the probability that a friend of mine will know a friend of yours? The answer to this kind of question is what Stanley Milgram calls the *small world problem* in a famous 1967 article [68]. Actually, the problem Milgram faces is more precisely the following: what is the average number of acquaintances we *must* go through in order to be able to connect any person in the American population of the sixties to any other person of the same population?

To estimate this number, Milgram designed and carried out the following experiment. Approximately 300 people living in Nebraska or Kansas were asked to send a letter to a person X residing in Boston, Massachusetts, according to the following rules. If the person knew X, then they could send

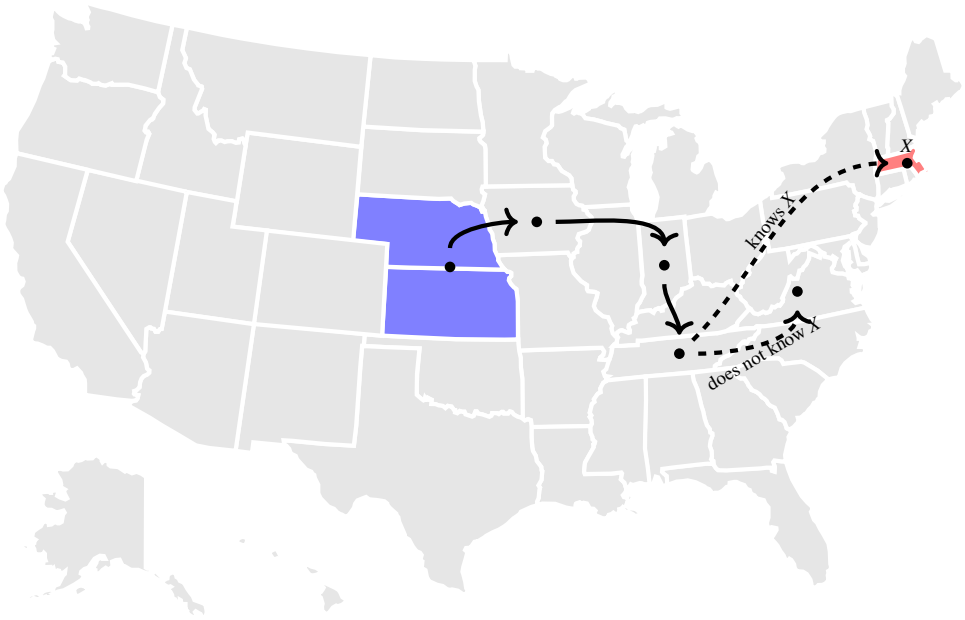


Figure 2.1: Milgram's small world experiment: starting with people who were in Nebraska or Kansas, each person who received the letter had to send it to the final recipient X (who was in Boston, Massachusetts) if they knew it, otherwise to a person they knew and believed to be closest to the recipient.

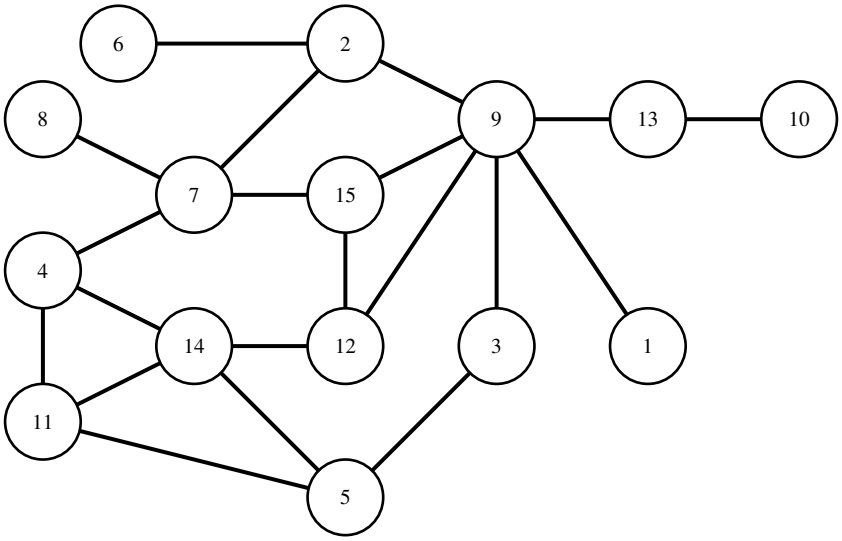
them the letter directly. Otherwise, they had to send it to another person they knew who, in his/her opinion, might be the closest to X (see Figure 2.1). Of the 300 letters, 64 reached X. Referring to these 64 letters, the average number of intermediaries that were used by the shipping chain to reach the recipient was between 5 and 6. From this experiment (carried out in collaboration with Jeffrey Travers), Milgram concluded that in the United States of America there are "six degrees of separation". This hypothesis turned out to be so fascinating that a play was written in 1990 [45] and later translated into a film, in 1993: this comedy was probably the reason why the phrase "six degrees of separation" became so popular. More recently, a game has also been produced, called *Six Degrees of Kevin Bacon game*, which consists of finding a series of movie collaborations that allow you to connect the American actor Kevin Bacon to any other actor or actress.

In this chapter, we analyze the problem of calculating the degrees of separation of a graph. However, let us immediately clarify that the goal is only to verify if a graph includes several very short shortest paths, which make the average distance between two nodes of the graph relatively small compared to the size of the graph itself. This is a purely "topological" result: the structure of the graph implies a low value of the degrees of separation. On the contrary, Milgram, in his experiment, aimed to verify even if, in the presence of very short shortest paths, people were able to *find* such paths in a distributed and independent way. This is a more "algorithmic" result, which we will not consider in this chapter, however (the interested reader may refer to the seminal paper by Kleinberg [51]).

Finally, we observe that in this chapter we will focus on undirected, connected, and unweighted graphs, but the results and algorithms that we will present apply, appropriately modified, also to other types of graphs, whether they are not connected, directed or weighted.

2.2 How to compute the degrees of separation

Generalizing the small world problem to any graph, the question is what is the average number of arcs we *must* go through in order to connect any node of the graph to any other node. Consider, for example, the graph made up of the 15 Florentine families of the fifteenth century, which we have already spoken about in the previous chapter, and which we show again in an "anonymous" way in the following figure, remembering that the connection between two families indicates that at least two members of the two families have joined in marriage.



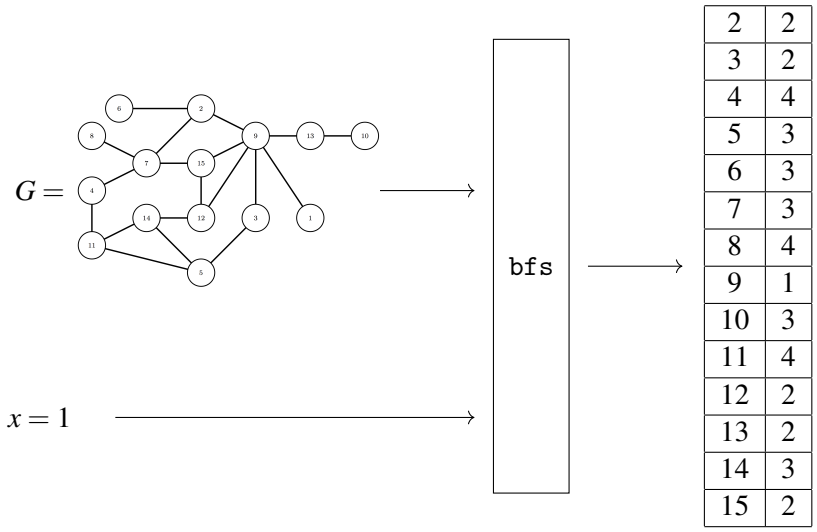
We note that to connect node 1 to node 9, we don't need any intermediary node, while to connect node 1 to node 2 we must *necessarily* go through an intermediary node (that is, node 9). Although there are other ways of connecting node 1 to node 2 (for example, via nodes 9, 15, and 7), the small world problem relates to the number of arcs that *must be* used, so we can reasonably ignore these alternative ways of connecting node 1 to node 2. In other words, we are only interested in the *shortest paths* within the graph, as we defined them in the previous chapter.

We have also seen in the previous chapter how it is possible, for each node x of a graph G, to calculate the length of all the shortest paths that connect x to the other nodes of the graph. For this purpose, we have introduced the BFS procedure, which in this chapter we can see as an algorithmic black box which, with input the graph G and a node x, outputs the distances of x from the other nodes of G. For example, if the graph is that of the Florentine families of the previous figure and the node x is the node 1, then the output of the BFS is the table shown in the right part of Figure 2.2. For example, the length of the shortest path from $x = 1$ to node 10 is equal to 3, as we have to go through nodes 9 and 13 to connect x to node 10.

Referring to the `Graphs` package we talked about at the end of the previous chapter, this package includes the `gdistances` function which, receiving a graph and a starting node, returns the list of the distances of the others nodes from the starting node (that is, the second column of the table in Figure 2.2). We can then define the following function.

```
function distances(graph::String, x::Int64)::Array{Int64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    d::Array{Int64} = gdistances(g, x)
    return d
end
```

By executing the instruction `println(distances("florence.lg", 1))` (in which we assume that the `florence.lg` file is the representation in `Graphs` format of the graph of the Florentine



2	2
3	2
4	4
5	3
6	3
7	3
8	4
9	1
10	3
11	4
12	2
13	2
14	3
15	2

Figure 2.2: With input a graph G and a node x of G, the BFS computes the table of distances of all the other nodes of G from x.

families), the list of the distances from node 1 to each other node of the graph is printed, that is, $[0,2,2,4,3,3,3,4,1,3,4,2,2,3,2]$ (observe that, clearly, the distance of the node 1 from itself is equal to 0).

We can repeat this procedure for all the nodes of the graph obtaining the *distance matrix* shown in Table 2.1, analogous to those we have already seen in the previous chapter regarding the graph of the Japanese archaeological sites and the graph of the American actors. At this point, we can add all the values contained in this table and divide this sum by the number of pairs of distinct nodes of the graph: the latter is equal to $15 \cdot 14 = 210$. The sum of the table values equals 522. Therefore, the degrees of separation between the Florentine families is equal to $\frac{522}{210} \approx 2.49$. As was expected, in such a small graph, the degrees of separation are much lower than those estimated by Milgram in his experiment, in which the graph potentially included over 200 million nodes.

Using the `Graphs` package, we can perform the operations just described in the following function, in which we make use of the `sum` function of the `Julia` programming language, which allows us to add all the elements of a list.

```
function degrees_of_separation(graph::String)::Float64
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    s::Int64 = 0
    for x::Int64 in vertices(g)
        d::Array{Int64} = gdistances(g, x)
        s = s + sum(d)
    end
    return s / (nv(g) * (nv(g) - 1))
end
```

By executing the instruction `println(degrees_of_separation("florence.lg"))`, the value `2.4857142857142858` is printed.

Can we then apply this methodology also in the case of the Milgram experiment? Even if we were able to reconstruct the graph of knowledge relations of the population of the United States of America (a far from simple task), we would have to use the BFS box as many times as there are

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	2	2	4	3	3	3	4	1	3	4	2	2	3	2
2	2	0	2	2	3	1	1	2	1	3	3	2	2	3	2
3	2	2	0	3	1	3	3	4	1	3	2	2	2	2	2
4	4	2	3	0	2	3	1	2	3	5	1	2	4	1	2
5	3	3	1	2	0	4	3	4	2	4	1	2	3	1	3
6	3	1	3	3	4	0	2	3	2	4	4	3	3	4	3
7	3	1	3	1	3	2	0	1	2	4	2	2	3	2	1
8	4	2	4	2	4	3	1	0	3	5	3	3	4	3	2
9	1	1	1	3	2	2	2	3	0	2	3	1	1	2	1
10	3	3	3	5	4	4	4	5	2	0	5	3	1	4	3
11	4	3	2	1	1	4	2	3	3	5	0	2	4	1	3
12	2	2	2	3	2	3	2	3	1	3	2	0	2	1	1
13	2	2	2	4	3	3	3	4	1	1	4	2	0	3	2
14	3	3	2	1	1	4	3	4	2	4	1	1	3	0	2
15	2	2	2	2	3	3	1	2	1	3	3	1	2	2	0

Table 2.1: The matrix of the distances of the graph of the Florentine families of the fifteenth century.

members of that population. According to Wikipedia [89], in 2020 the number of inhabitants of the United States of America was more than 300 million.

As we saw in the previous chapter, the execution of a BFS requires a number of steps proportional to the number of arcs in the graph, in our case the number of knowledge relations. Also in this case it is not easy to determine exactly this number, but based on some experiments carried out already in 2010 [65], we can say that, on average, each inhabitant has a number of acquaintances approximately equal to 300. Hence, the number of arcs (counted only once for each pair of connected nodes) is most likely very close to 45 billions.

In conclusion, if we want to apply the method just described to calculate the degrees of separation of the current American population, we will have to perform a number of steps approximately equal to

$$4.5 \cdot 10^{10} \cdot 3 \cdot 10^{8} = 1.35 \cdot 10^{19}.$$

At the moment, the fastest supercomputer is capable of performing approximately $4.5 \cdot 10^{17}$ operations per second [94]. With this supercomputer, the time required to calculate the degrees of separation would be approximately equal to

$$\frac{1.35 \cdot 10^{19}}{4.5 \cdot 10^{17}} = 30 \text{ seconds}.$$

These estimates are based on the fastest supercomputer currently available and on the analysis of the American population alone. It is therefore evident that the methodology described cannot be applied to the case of graphs of enormous size, such as those we are used to hearing about in recent years (such as Facebook or Twitter), and of more affordable computers, which typically can perform order of 10^9 operations per second. For this reason, we now show how it is possible to calculate a good approximation of the average distance of a graph through a much more efficient algorithm.

2.3 Distance distribution, sampling, and degrees of separation

The basic idea of the new algorithm, mostly inspired by [37], repeatedly proposed in the literature (see, for example, [60, 97]), and formally analyzed in [25], consists of two fundamental elements.

1. Instead of calculating the table of distances, we calculate the *distance distribution*, which intuitively specifies, for each value h, the percentage of pairs of nodes x and y of the graph that are at distance h.
2. We approximate the distance distribution by means of a *sampling* mechanism, that is, carrying out the BFS only starting from a subset of nodes of the graph.

The combination of these two elements will lead us to the design of an algorithm for estimating the degrees of separation (that is, the average distance) of a graph, which is very efficient and, at the same time, very precise.

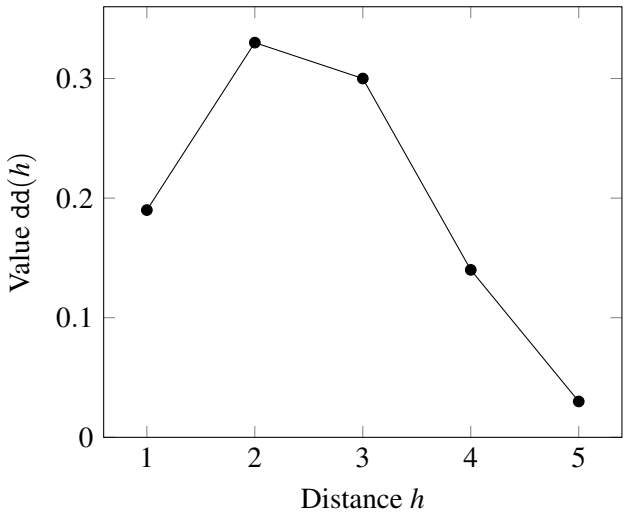
2.3.1 Distance distribution

To estimate the average distance of a graph, we can focus our attention on the *distance distribution* function $\mathtt{dd}$ of the graph itself. For each $h > 0$, the value $\mathtt{dd}(h)$ is defined as

$$\mathtt{dd}(h) = \frac{|\{(x,y) \in V \times V : d(x,y) = h\}|}{n(n-1)},$$

where V denotes the set of nodes of the graph, $n = |V|$ and $d(x,y)$ denotes the length of the shortest path from x to y. We note that $0 \leq \mathtt{dd}(h) \leq 1$, as $n(n-1)$ is the maximum number of possible pairs on a set of n elements.

For example, in the case of the graph of the Florentine families, we can deduce from the distance table examined above that there are 40 pairs of nodes at distance 1 (that is, directly connected), 70 pairs of nodes at distance 2, 64 at distance 3, 30 at distance 4, and 6 at distance 5. Since $n = 15$, then $n(n-1) = 210$. So, $\mathtt{dd}(1) = \frac{40}{210} \approx 0.19$, $\mathtt{dd}(2) = \frac{70}{210} \approx 0.33$, $\mathtt{dd}(3) = \frac{64}{210} \approx 0.3$, $\mathtt{dd}(4) = \frac{30}{210} \approx 0.14$ and $\mathtt{dd}(5) = \frac{6}{210} \approx 0.03$. The plot of this function is shown in the following figure.



We note that if we have the distance distribution function available, then the average distance of the graph (that is, the degrees of separation) can be easily calculated as follows:

$$\sum_{u,v \in V} \frac{d(u,v)}{n(n-1)} = \sum_{h=1}^{n-1} h \frac{|\{(x,y) \in V \times V : d(x,y) = h\}|}{n(n-1)} = \sum_{h=1}^{n-1} h \cdot \mathtt{dd}(h)$$

(we observe that in an unweighted graph, the maximum distance between two nodes is equal to $n-1$, as in the worst case scenario it is necessary to pass through all the remaining nodes of the graph).

For example, in the case of the graph of the Florentine families, the average distance is approximately equal to

$$1 \cdot 0.19 + 2 \cdot 0.33 + 3 \cdot 0.3 + 4 \cdot 0.14 + 5 \cdot 0.03 = 0.19 + 0.66 + 0.9 + 0.56 + 0.15 = 2.46$$

which corresponds to the degrees of separation we calculated in the previous paragraph (the difference is due to the approximations we made on the values of the distance distribution function). In conclusion, if we are able to approximate the values of the distance distribution function, we are also able to approximate the value of the degrees of separation of a graph.

2.3.2 Approximation of the distance distribution function

We now show how it is possible to calculate an approximation of the distance distribution function, by means of a technique well known in the field of data analysis and very often used to carry out opinion polls: the *sampling* technique. The basic idea of this technique is to *estimate* the distribution of distances by analyzing only the distances of all nodes from a *sample* of such nodes, that is, a relatively small subset of randomly chosen nodes.

For example, always referring to the graph of the Florentine families, suppose we choose a sample of only two nodes, in particular the nodes 2 and 8 whose distances from the other nodes of the graph are shown in the following table.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	2	0	2	2	3	1	1	2	1	3	3	2	2	3	2
8	4	2	4	2	4	3	1	0	3	5	3	3	4	3	2

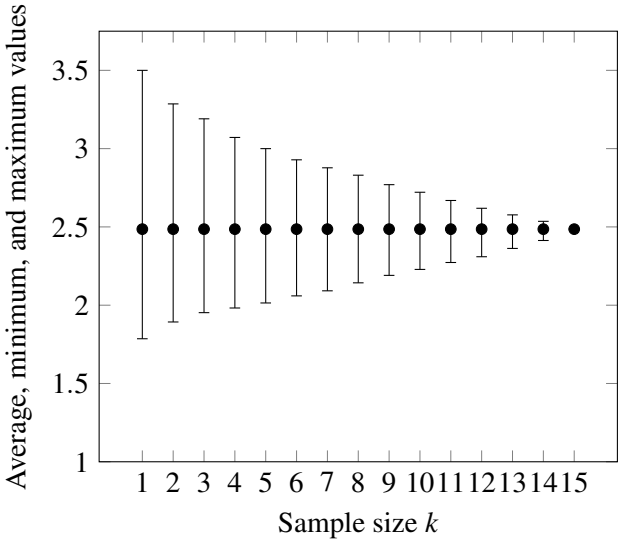
There are 4 node pairs at distance 1 (that is, directly connected), 10 at distance 2, 9 at distance 3, 4 at distance 4, and 1 at distance 5. The total number of possible pairs that we can form with a sample of 2 nodes is equal to $2(n-1) = 2 \cdot 14 = 28$. Thus, the estimate of $\mathtt{dd}(1)$ that we obtain with this sample is equal to $\frac{4}{28} \approx 0.14$, that of $\mathtt{dd}(2)$ is equal to $\frac{10}{28} \approx 0.36$, that of $\mathtt{dd}(3)$ is equal to $\frac{9}{28} \approx 0.32$, that of $\mathtt{dd}(4)$ is equal to $\frac{4}{28} \approx 0.14$, and that of $\mathtt{dd}(5)$ is equal to $\frac{1}{28} \approx 0.04$. The average distance calculated by these approximations of the values of the distance distribution function is equal to

$$1 \cdot 0.14 + 2 \cdot 0.36 + 3 \cdot 0.32 + 4 \cdot 0.14 + 5 \cdot 0.04 = 0.14 + 0.72 + 0.96 + 0.56 + 0.2 = 2.58$$

which is a value not too far from the exact value calculated above, which is 2.49.

In the next section we will show mathematically that this result is not a coincidence, but that in general, whatever the sample size, the expected estimate of the distance distribution function (and, therefore, of the degrees of separation) is equal to the exact one. The size of the sample, however, is important to have the "certainty" that, whatever the sample chosen, the error made by the estimate is very limited.

Let us consider, once again, the graph of the Florentine families and let us try to run the algorithm based on the sampling technique with the sample size that can vary from 1 (that is, a sample consisting of a single node) to 15 (that is, a sample that includes all nodes). In this case, since the number of nodes is very limited, we can examine all the possible samples whose number is equal to the number of possible subsets of a set of 15 elements, that is $2^{15} = 32768$. If we group these samples into 15 groups, according to their size, if for each of them we repeat the operation we have done in the case of the sample consisting of the nodes 2 and 8, and if, for each dimension, we calculate the average, minimum, and maximum estimate of the degrees of separation we obtain the result summarized in the following figure.



First of all we observe that, regardless of the size of the sample, the average value of the estimate obtained (represented in the figure by the black dots) is always equal to about 2.5, that is the exact value. On the contrary, as the size of the sample increases, the magnitude of the error that can be committed is reduced more and more, up to, obviously, a null error in the case in which the sample size is equal to 15, which means that it contains all nodes. For example, in the case of samples with only one node, one can oscillate between an estimate equal to about 1.79 and one equal to 3.5, in the case of samples with 5 nodes, one can oscillate between an estimate of about 2.01 to one of 3 and, in the case of samples with 10 nodes, we can oscillate between an estimate of about 2.22 to one of 2.71.

The question we ask at this point is how large the sample must be to be almost sure that the error is limited. In particular, in the next paragraph we will demonstrate that if we want to be sure that the absolute error of the estimates of the distance distribution function is not greater than a certain value t, then we can choose a sample of size $\log_2(n)/t^2$. In other words, a logarithmic sample size with respect to the number of nodes of the graph is sufficient to have a good approximation of the distance distribution function and, therefore, of the degrees of separation of a graph. We will see in the last paragraph of this chapter an application of this method which will show how effective it is in practice. But we now show its effectiveness by considering a graph much larger than that of the Florentine families, but sufficiently "small" to be able to compare the sampling technique with the exhaustive one described in the previous paragraph.

2.3.3 An example: Slashdot friendship social network

Slashdot is a news website, mainly of a technological nature, which are submitted, possibly published, and, therefore, evaluated by the users themselves and by the website editors. Each news (also called a story) is associated with a comment section, to which users can contribute. Founded in 1997, Slashdot has been hugely successful until reaching approximately five and a half million users in 2006. It has received several awards including the *Webby Award* for best news site. In 2010 it was bought by an online jobs company for twenty million dollars, then sold to a San Diego company for a price that is unknown [93].

In 2002 Slashdot introduced what was called the *Slashdot Zoo*, which allowed each user to mark another user as friend or foe. It is, therefore, one of the first examples of an online friendship/enmity network, created two years before Facebook [56]. The corresponding graph is a directed and weighted graph: in fact, if a user marks another user as friend or foe, the other user is not said to do the same, and the arcs have a value equal to $+1$ or -1 depending on the type of marking (friend or

foe, respectively). In this paragraph, however, we use an undirected and unweighted version of the graph available at [59], obtained by eliminating the direction of the arcs and their weights (in other words, there is a connection between two Slashdot users if at least one of the two users marked the other one, either as friend or as foe).

It is a graph that, in 2009, included 82168 nodes and 504230 arcs: difficult, therefore, to think of being able to visualize it in any way that can help to understand it (as, for example, we did in the first chapter of this book). We can certainly say that it is a sparse graph: in fact its density is equal to $\delta = \frac{2 \cdot 504230}{82168 \cdot 82167} = \frac{1008460}{6751498056} \approx 0.00015$ (therefore, very close to 0). Regarding the degrees of its nodes, the average degree is equal to $\frac{1008460}{82168} \approx 12.27$. However, this relatively low value does not seem to correspond to an equal distribution of degrees. We can, in fact, calculate the minimum degree and the maximum degree of the graph, using the `degree_centrality` function of the `Graphs` package, as shown in the following code.

```
function min_max_degree(graph::String)::Tuple{Int64,Int64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    degree::Array{Int64} = degree_centrality(g, normalize = false)
    return minimum(degree), maximum(degree)
end
```

By executing the instruction `println(min_max_degree("slashdot.lg"))` (where we assume that the `slashdot.lg` file is the `Graphs` representation of the Slashdot Zoo graph), the two values 1 and 2552 are printed. In other words, while the average degree is approximately 12, there are nodes that have a degree 1 and nodes that have a degree 2552 (we will better elaborate on the degree distribution of a graph in a subsequent chapter of the book).

By still using the `Graphs` package, we can check that the graph is connected. We can, therefore, define the following function which computes the exact values of the distance distribution function (in order to execute this function, we also need the `StatsBase` package). The body of this function is similar to what we saw earlier with the difference that, whenever distances from a node are calculated, it is also updated the sequence of the values of the distribution function (we observe that, in the absence of further information, the length of this sequence is equal to the number of nodes minus one).

```
function distance_distribution(graph::String)::Array{Float64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    dd::Array{Int64} = zeros(Int64, nv(g) - 1)
    for x::Int64 in vertices(g)
        dd = dd + counts(gdistances(g, x), 1:nv(g)-1)
    end
    return dd / (nv(g) * (nv(g) - 1))
end
```

Before executing the instruction `dd = distance_distribution("slashdot.lg")`, let us be prepared to wait for some time. In fact, we are running a BFS starting at 82168 nodes: each BFS requires a number of operations proportional to the number of arcs of the graph, that is, 504230. Therefore, the total number of operations will be proportional to $41431570640 \approx 41 \cdot 10^9$. If our computer could perform four billion operations per second, the execution of the previous code would take about 10 seconds. However, this is a very optimistic hypothesis, as our computer in the meantime performs many other operations associated with other running programs and, furthermore, we are not considering the constant of proportionality that comes into play in the number of operations performed by the code. All this depends on the computer we have at our disposal, on the programs we have running, and on the implementation of the BFS. We can, despite

this, get an idea of the advantage of using the sampling technique if we run our code on the same computer, under the same conditions, and with the same implementation of the BFS. For example, on my computer, the above instruction takes about 740 seconds, which is about 12 minutes.

The result of the execution is shown in the first column of the table in the left part of Figure 2.3. The first observation we can make is that, starting from the distance equal to 14, all the values of the distance distribution function are equal to 0 (this can be verified by executing the instruction `findlast(x -> x > 0, dd)`, which returns the value 13): in other words, the diameter of this graph is equal to 13. We observe that Milgram's hypothesis of small degrees of separation implies nothing about the diameter of the graph: we could have a very small average distance, but at the same time have a fairly high maximum distance. Yet in the case of the Slashdot Zoo graph, as well as in the case of the vast majority of real-world graphs, the diameter is also very small (we will talk about it in the next chapter). The second observation that we can make from the results shown in the first column of the table is that, using these results, we can calculate the average distance of the Slashdot Zoo graph, by using the following function.

```
function degrees_of_separation(dd::Array{Float64})::Float64
    last_distance::Int64 = findlast(x -> x > 0, dd)
    return dot(1:last_distance, dd[1:last_distance])
end
```

By executing the instruction `println(degrees_of_separation(dd))`, a value approximately equal to 4.07 is printed. In other words, the degrees of separation of the Slashdot Zoo graph are less than the six degrees hypothesized by Milgram: once again, however, we must remember that we are talking about a network that is several orders of magnitude smaller than the knowledge relation graph among the entire American population.

At this point we can apply the sampling technique on the Slashdot Zoo graph, to verify its efficiency and accuracy. We have already said that the results we will see in the next paragraph suggest taking a sample of logarithmic size with respect to the number of nodes in the network: being $n = 82168$, we can, for example, take a sample of $100 \cdot \log_2(82168) \approx 1600$. Since the number of operations performed by a single BFS does not depend on the starting node, we can expect that with such a sample the execution of the approximation algorithm will require $1600 * \frac{740}{82168} \approx 14$ seconds (against the 12 minutes required by the exact algorithm). To test this hypothesis, we define the following function.

```
function distance_distribution(graph::String, k::Int64)::Array{Float64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    dd::Array{Int64} = zeros(Int64, nv(g) - 1)
    for _ in 1:k
        dd = dd + counts(gdistances(g, rand(1:nv(g))), 1:nv(g)-1)
    end
    return dd / (k * (nv(g) - 1))
end
```

The execution of the instruction `add = distance_distribution("slashdot.lg", 1600)` requires (on my computer under the same conditions) approximately 14 seconds (as expected). The values calculated are shown in the second column of the table in the left part of Figure 2.3 (clearly, these values can change from one execution to another, because of the random choice of the sample): as we can see, these values are very close to the exact ones. The plot shown in the right part of the figure shows how the two distributions are practically identical and, therefore, indistinguishable. The sampling technique, therefore, seems to be an excellent tool that allows us to reconcile efficiency with precision and that allows us to calculate an estimate of the degrees of

	Exact	Approximate
1	0,000149368	0,000149756
2	0,0145728	0,0152348
3	0,214857	0,219024
4	0.506805	0,508865
5	0.21979	0,213167
6	0.0383402	0,0377504
7	0.00489839	0,00528811
8	0.000527452	0,000476872
9	5,42057e-5	4,01241e-5
10	5,59135e-6	3,40769e-6
11	3.71473e-7	2,20587e-7
12	2.07361e-8	7,60646e-9
13	5.92461e-10	0,0

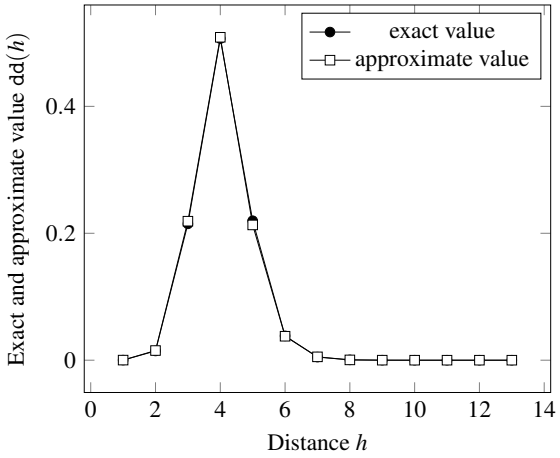


Figure 2.3: The exact and approximate distribution of distances in the Slashdot Zoo graph: the approximate values were calculated using a sample of 1600 nodes out of a total of 82168 nodes included in the graph.

separation of even very large graphs. In the case of the graph of the Slashdot Zoo, if we execute the instruction `println(degrees_of_separation(add))`, the printed estimate of the degrees of separation is approximately 4.07 and, hence, very close to the exact value.

2.4 Probabilistic analysis of the sampling algorithm

The approximation algorithm of the distance distribution function, with input a graph G with n nodes and a value k, operates as follows (see also the code `Julia` seen at the end of previous paragraph where $k = 100 \cdot \log_2(n)$).

1. It randomly chooses a sequence U of k nodes in G (*not necessarily distinct*) and, for each h with $1 \leq h \leq n-1$, initializes to 0 the value $\mathtt{dd}_U(h)$.
2. For each node x in U:
   (a) it computes the distances of x from all other nodes of G;
   (b) for each h with $1 \leq h \leq n-1$, it increments the value of $\mathtt{dd}_U(h)$ by the number of nodes that are at distance h from x.
3. For each h with $1 \leq h \leq n-1$, it "normalizes" the value $\mathtt{dd}_U(h)$, by dividing it by $k(n-1)$.

We observe that the execution time of this algorithm is proportional to km (in the case of undirected and unweighted graphs): the smaller the sample, the more efficient the algorithm. On the other hand, the greater efficiency of the algorithm can cost in terms of accuracy of the approximation. In the remainder of this section, we calculate what size the sample should be (that is, how much k should be), so that a guarantee on the quality of the approximation can be obtained.

If $U = [u_1, u_2, \ldots, u_k]$ with $u_i \in V$ is the sequence of nodes (not necessarily distinct) chosen at random by this algorithm, then the approximation of the function `dd` obtained by the algorithm itself can be expressed in the following way:

$$\mathtt{dd}_U(h) = \frac{|\{(u_i, v) : i \in \{1,2,\ldots,k\} \wedge v \in V \wedge d(u_i, v) = h\}|}{k(n-1)}.$$

In other words, the percentage of node pairs that are at distance h is approximated by the percentage of node pairs, with the first node of the pair included in the sample, that are at distance h. We note

that, for each h with $1 \leq h \leq n-1$, $\mathtt{dd}_U(h)$ is a *random variable*, that is a variable that assumes different values in dependence of a random event. In our case, the random event is the choice of the k nodes in the sample (with possible repetitions): the value that the random variable $\mathtt{dd}_U(h)$ can assume is always between 0 and 1.

We first show that the expected value of the values of the approximated function is equal to the exact one, that is, the expected value of the random variable $\mathtt{dd}_U(h)$ is equal to $\mathtt{dd}(h)$ (in other words, the approximate function is a *unbiased estimator* of the distance distribution function). The *expected value* of a random variable that depends on a set Ω of possible events is defined as the sum, for all possible events $\omega \in \Omega$, of the product of the value of the random variable corresponding to ω times the probability of ω itself. In our case, rather than directly applying this definition to the $\mathtt{dd}_U(h)$ random variable, we show how it can be expressed as the sum of k random variables relative to samples of a single node, and then use the *linearity property* of the expected value (that is, the property that the expected value of the sum of k random variables is equal to the sum of the expected values of the k random variables).

From the definition of $\mathtt{dd}_U(h)$ it follows that

$$\mathtt{dd}_{\{u_i\}}(h) = \frac{|\{(u_i,v) : v \in V \wedge d(u_i,v) = h\}|}{n-1}$$

and, hence, that

$$\begin{aligned}
\mathtt{dd}_U(h) &= \frac{|\{(u_i,v) : i \in \{1,2,\ldots,k\} \wedge v \in V \wedge d(u_i,v) = h\}|}{k(n-1)} \\
&= \frac{1}{k}\frac{\sum_{i=1}^{k} |\{(u_i,v) : v \in V \wedge d(u_i,v) = h\}|}{n-1} \\
&= \frac{1}{k}\sum_{i=1}^{k}\frac{|\{(u_i,v) : v \in V \wedge d(u_i,v) = h\}|}{n-1} = \frac{1}{k}\sum_{i=1}^{k}\mathtt{dd}_{\{u_i\}}(h) = \sum_{i=1}^{k}\frac{\mathtt{dd}_{\{u_i\}}(h)}{k}.
\end{aligned}$$

Thus, the random variable $\mathtt{dd}_U(h)$ can be expressed as the sum of the k random variables $\frac{\mathtt{dd}_{\{u_i\}}(h)}{k}$, which depend on the random event consisting of the random choice of a single node. If each node u_i is uniformly chosen in V (that is, with probability equal to $\frac{1}{n}$), then

$$\mathbb{E}[\mathtt{dd}_{\{u_i\}}(h)] = \sum_{u \in V}\frac{1}{n}\cdot\mathtt{dd}_{\{u\}}(h) = \sum_{u \in V}\frac{\mathtt{dd}_{\{u\}}(h)}{n} = \mathtt{dd}_V(h) = \mathtt{dd}(h).$$

In other words, we have just proved mathematically what we have already experimentally verified, namely that, even with a sample of only one node, the expected value of the estimate of the average distance is equal to the exact value. From the linearity property of the expected value we have that this statement is true for any sample of any size: in fact,

$$\mathbb{E}[\mathtt{dd}_U(h)] = \mathbb{E}\left[\sum_{i=1}^{k}\frac{\mathtt{dd}_{\{u_i\}}(h)}{k}\right] = \sum_{i=1}^{k}\mathbb{E}\left[\frac{\mathtt{dd}_{\{u_i\}}(h)}{k}\right] = \sum_{i=1}^{k}\frac{\mathbb{E}[\mathtt{dd}_{\{u_i\}}(h)]}{k} = \sum_{i=1}^{k}\frac{\mathtt{dd}(h)}{k} = \mathtt{dd}(h).$$

The correctness of the estimator, however, does not assure us that, in certain cases, the approximate value of the distance distribution function cannot be very far from the real one. To demonstrate that this is not the case, we will use the following result of concentration of the values of *independent* random variables, that is, random variables such that, intuitively, knowing something about the value of one of them does not bring any information about the value of another (in the following we denote by $\Pr[X \geq t]$ the probability that a random variable X has a value greater than or equal to t).

Theorem 2.4.1 — Hoeffding bound. Let $X_1, \ldots, X_k$ be k random independent variables such that $a_i \leq X_i \leq b_i$ and let $X = \sum_{i=1}^{k} X_i$. Then, for any $t > 0$,

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq 2e^{\frac{-2t^2}{\sum_{i=1}^{k}(b_i - a_i)^2}}.$$

Before proving this theorem, let us see how it is possible to apply it to our case, that is, to measure the concentration, for each h, of the expected value of the estimate of the value of the distance distribution function obtained through the sampling algorithm. Specifically, in this case we have that

$$X_i = \frac{\mathtt{dd}_{\{u_i\}}(h)}{k}, \qquad X = \sum_{i=1}^{k} \frac{\mathtt{dd}_{\{u_i\}}(h)}{k} = \mathtt{dd}_U(h) \qquad \text{and} \qquad \mathbb{E}[X] = \mathtt{dd}(h)$$

(we note that the X_i variables are independent as the random choices of the nodes to be included in the sample are made independently of each other, with possible repetitions). Furthermore, for $1 \leq i \leq k$, $0 \leq \mathtt{dd}_{\{u_i\}} \leq 1$ and, therefore, $0 \leq X_i \leq \frac{1}{k}$: in other words, $a_i = 0$ and $b_i = \frac{1}{k}$. In conclusion, from the Hoeffding bound it follows that

$$\Pr\left[|\mathtt{dd}_U(h) - \mathtt{dd}(h)| \geq t\right] \leq 2e^{\frac{-2t^2}{\sum_{i=1}^{k}\frac{1}{k^2}}} = 2e^{-2kt^2}.$$

If we choose $k = \frac{\alpha}{2} t^{-2} \ln n = \frac{\alpha}{2\log_2 e} \frac{\log_2 n}{t^2}$ where $\alpha > 0$ is a constant, we have that this probability is bounded by $2/n^{\alpha}$. This sample size, therefore, guarantees that the absolute error committed with the sampling algorithm is limited by t *with high probability*, that is, with a probability that decreases in a way inversely proportional to a power of the number of nodes of the graph. We note that the sample size increases as the t error decreases: in principle, if we want an absolute error less than 0.001 and if we assume that $\alpha = 1$, the sample should have size equal to $500000 \ln n$, which may be larger than the number of nodes in the graph. However, as we saw in the previous section, in practice a sample whose size is equal to $100 \log_2 n$ already gives very precise results.

The proof of the Theorem 2.4.1 is structured in three steps: the Markov inequality, the Chernoff method, and, finally, the actual Hoeffding bound (these and other concentration results are superbly described in [34]) .

Markov inequality

The *Markov inequality* states that, for any random non negative variable X and for any $t > 0$,

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t},$$

where $\mathbb{E}[X]$ denotes the expected value of X. Showing this inequality is relatively simple. Indeed,

$$\mathbb{E}[X] = \sum_{u<t} u \Pr[X = u] + \sum_{u \geq t} u \Pr[X = u] \geq \sum_{u \geq t} u \Pr[X = u] \geq t \sum_{u \geq t} \Pr[X = u] = t \Pr[X \geq t].$$

Chernoff method

By making use of the Markov inequality, we can describe the so called *Chernoff bounding method* which is based on the following statement.

Proposition 2.4.2 — Chernoff bound. If X is a random variable, then, for any $t > 0$,

$$\Pr[X \geq t] \leq \min_{s>0} e^{-st}\mathbb{E}[e^{sX}].$$

To prove such a statement, let $s > 0$. We have that

$$\Pr[X \geq t] = \Pr[sX \geq st] = \Pr[e^{sX} \geq e^{st}] \leq e^{-st}\mathbb{E}[e^{sX}],$$

where the last inequality follows from the Markov inequality. Since s has been set arbitrarily, the above inequality holds for any value of s and, in particular, for the value minimizing $e^{-st}\mathbb{E}[e^{sX}]$. Therefore, the Proposition 2.4.2 turns out to be proved.

Hoeffding bound

To prove Theorem 2.4.1, we first show that

$$\Pr[X - \mathbb{E}[X] \geq t] \leq e^{\frac{-2t^2}{\sum_{i=1}^{k}(b_i-a_i)^2}}, \tag{2.1}$$

where we remember that $X = \sum_{i=1}^{k} X_i$ and that $X_1, \ldots, X_k$ are k independent random variables such that $a_i \leq X_i \leq b_i$. This inequality follows from the application of the Chernoff bound and from the following statement.

Proposition 2.4.3 If Y is a random variable such that $\mathbb{E}[Y] = 0$ and $a \leq Y \leq b$, then, for any $s > 0$,

$$\mathbb{E}[e^{sY}] \leq e^{\frac{s^2(b-a)^2}{8}}.$$

If we apply the Chernoff bound to the random variable $X - \mathbb{E}[X]$, we have that

$$\Pr[X - \mathbb{E}[X] \geq t] \leq \min_{s>0} e^{-st}\mathbb{E}[e^{s(X-\mathbb{E}[X])}].$$

For any $s > 0$, we have that

$$\begin{aligned} e^{-st}\mathbb{E}[e^{s(X-\mathbb{E}[X])}] &= e^{-st}\mathbb{E}[e^{s(\sum_{i=1}^{k} X_i - \sum_{i=1}^{k}\mathbb{E}[X_i])}] = e^{-st}\mathbb{E}[e^{s(\sum_{i=1}^{k}(X_i-\mathbb{E}[X_i]))}] \\ &= e^{-st}\mathbb{E}[\prod_{i=1}^{k} e^{s(X_i-\mathbb{E}[X_i])}] = e^{-st}\prod_{i=1}^{k}\mathbb{E}[e^{s(X_i-\mathbb{E}[X_i])}] \end{aligned}$$

where the first equality derives from the definition of X and from the linearity property of the expected value, while the last equality derives from the fact that the random variables X_i are independent (that is, the expected value of their product is equal to the product of their expected values). By applying, at this point, k times Proposition 2.4.3 with $Y = X_i - \mathbb{E}[X_i]$ we obtain that

$$\Pr[X - \mathbb{E}[X] \geq t] \leq \min_{s>0} e^{-st + \frac{s^2\sum_{i=1}^{k}(b_i-a_i)^2}{8}}$$

(we observe that $\mathbb{E}[Y] = 0$ and that $a = a_i - \mathbb{E}[X_i] \leq Y \leq b_i - \mathbb{E}[X_i] = b$, so that $b - a = b_i - a_i$). The function $y = -st + \frac{s^2\sum_{i=1}^{k}(b_i-a_i)^2}{8}$ reaches its minimum value in $s = \frac{4t}{\sum_{i=1}^{k}(b_i-a_i)^2}$. By substituting this value, we obtain the inequality (2.1).

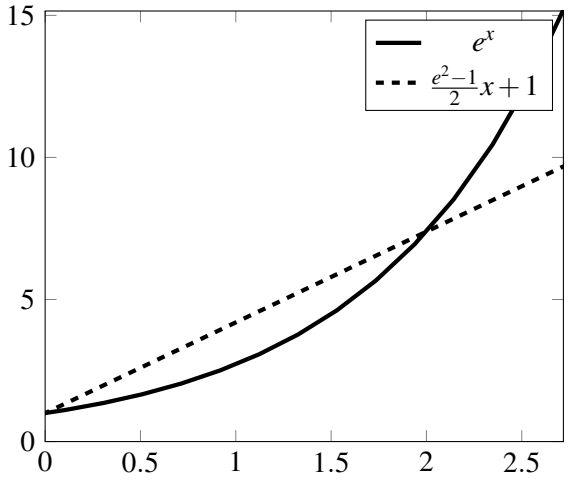


Figure 2.4: The function $y = e^x$ is convex, that is, taken any two points on the plot of the function, the segment joining them is entirely above the plot itself. For example, considering the two points $(0,0)$ and $(2,e^2)$, the line passing through these two points has the equation $y = \frac{e^2-1}{2}x+1$ and its segment joining the two points lies entirely above the function $y = e^x$.

We then have to prove Proposition 2.4.3. To this aim, let us first observe that, since the function $y = e^{sx}$ is convex (see Figure 2.4), the following inequality holds:

$$e^{sx} \leq \frac{x-a}{b-a}e^{sb} + \frac{b-x}{b-a}e^{sa}.$$

Since $\mathbb{E}[Y] = 0$, we have that

$$\mathbb{E}[e^{sY}] \leq \frac{b}{b-a}e^{sa} - \frac{a}{b-a}e^{sb}.$$

Let $p = \frac{b}{b-a}$ and $u = s(b-a)$. Then,

$$\begin{aligned}\log(\frac{b}{b-a}e^{sa} - \frac{a}{b-a}e^{sb}) &= \log(pe^{sa} + (1-p)e^{sb}) \\ &= sa + \log(p + (1-p)e^{s(b-a)}) \\ &= (p-1)u + \log(p + (1-p)e^u) = \varphi(u).\end{aligned}$$

From Taylor's theorem, it follows that

$$\varphi(u) = \varphi(0) + \varphi'(0)u + \frac{1}{2}\varphi''(\xi)u^2,$$

where $\frac{1}{2}\varphi''(\xi)u^2$ is called the Lagrange remainder (where ξ is a real number between 0 and u). Since both $\varphi(0)$ and $\varphi'(0)$ are equal to 0, and since $\varphi''(\xi) \leq \frac{1}{4}$, we have that

$$\varphi(u) \leq \frac{u^2}{8} = \frac{s^2(b-a)^2}{8}$$

and Proposition 2.4.3 turns out to be proved.

We observe that, by repeating the same proof with the random variables $-X_1, \ldots, -X_k$, we can prove the following inequality:

$$\Pr[X - \mathbb{E}[X] \leq -t] \leq e^{\frac{-2t^2}{\sum_{i=1}^{k}(b_i - a_i)^2}}.$$

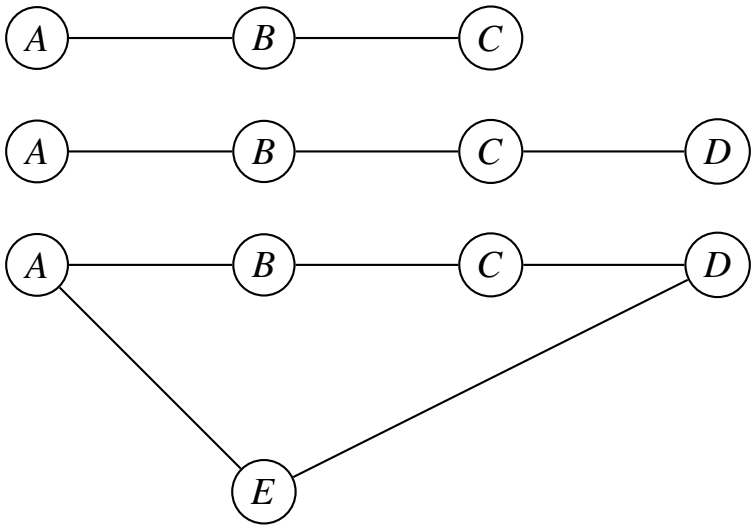
By combining the two inequalities, we get the Hoeffding bound of Theorem 2.4.1.

2.5 Degrees of separation over time

The results of the previous paragraph tell us that, by using a sample of nodes of logarithmic size with respect to the total number of nodes, we can obtain a good approximation of the distance distribution function and, therefore, of the degrees of separation of a graph.

As a further application of this result, let us ask ourselves the following question: as time passes, do the degrees of separation of a graph increase or decrease? The answer to this question does not appear so simple because, although adding new arcs can only decrease the degrees of separation, adding new nodes can cause both an increase and a decrease in the degrees of separation.

For example, let us consider the evolution of a tiny graph shown in the following figure.



At the beginning, the network, consisting of only three nodes, has an average distance of

$$\frac{1 \cdot 4 + 2 \cdot 2}{6} = \frac{8}{6} \approx 1.3.$$

Adding the node D and the arc that connects it to the node C node causes an increase in the degrees of separation. In fact, the new average distance is equal to

$$\frac{1 \cdot 6 + 2 \cdot 4 + 3 \cdot 2}{12} = \frac{20}{12} = \frac{5}{3} \approx 1.7.$$

Finally, the addition of the node E and of the two arcs that connect it to the nodes A and D causes a decrease in the degrees of separation. In fact, the new average distance is equal to

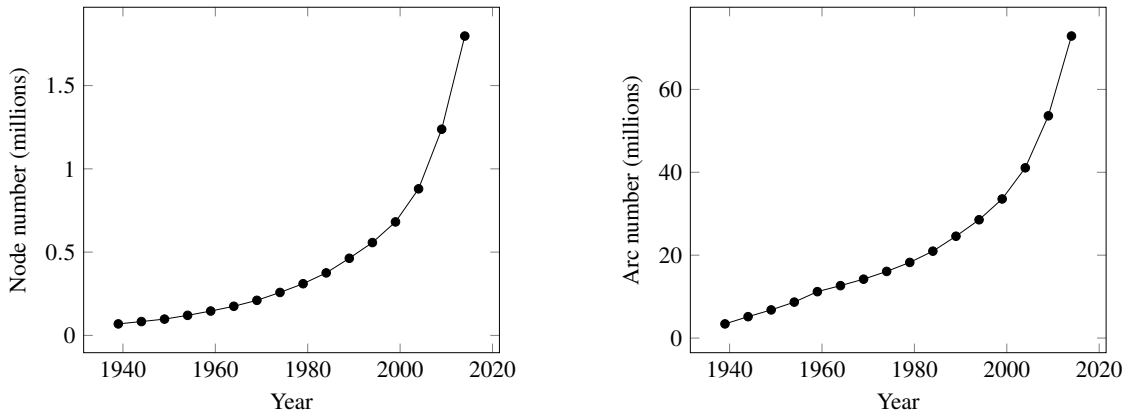
$$\frac{1 \cdot 10 + 2 \cdot 10}{20} = \frac{30}{20} = \frac{3}{2} = 1.5.$$

What, then, is the trend of the degrees of separation in a "real" graph? For example, let us consider the network of movie collaborations, of which it is possible to have a snapshot for each year since the 1940s. We note that this graph contains several hundred thousand nodes (which correspond to the actors) and several million arcs (which correspond to the actors). It is therefore not conceivable to use the exact calculation method of the average distance for this network and we must instead resort to the approximation algorithm based on sampling.

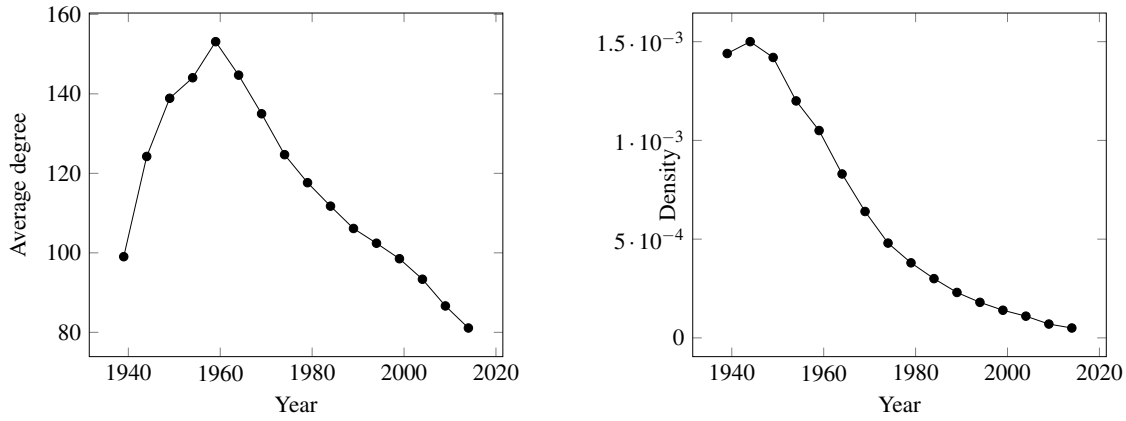
2.5.1 The IMDB graph

We have already talked in the first chapter of the Internet Movie DataBase, an online service that allows us to access various information relating to movie productions of different genres. The data available to the user concern movie productions starting from before the Second World War up to the present day. In this book, we will make use of this data-set to create one of the largest graphs to experiment with: for this purpose, however, we will ignore, for obvious reasons, the data related to broadcasts of awards (such as, for example, the Academy Awards), documentaries, TV quizzes, news and talk shows.

In the graph related to IMDB the nodes are the actors and the arcs represent the co-acting in at least the same movie: we note that the graph is not weighted, in the sense that we do not distinguish between multiple or single collaborations. Since every movie production has a release date (that is, the year of release), the graph related to IMDB is actually a network of which we can know, quite precisely, the temporal evolution. For example, in the following figure we show, on the left, the number of actors present in the network from 1939 to 2014 (indicating this number every five years), while on the right we show the number of arcs (that is, the number of collaborations without multiplicity) in the same time frame.



The IMDB graph had just over $69,000$ nodes and just over 3400000 arcs in 1939 and had nearly 1800000 nodes and nearly 73000000 arcs in 2014. It is a graph that has remained sparse over time: in the following figure, we show the trend of the average degree and of the density of the graph, in the same time span as the previous figure.



We can see how, at the beginning of the evolution of the graph, the average degree and the density increased: in other words, initially more movie were made than the new actors who entered the graph. Subsequently, both the average degree and the density of the graph decrease, which suggests that the number of movies produced each year does not keep pace with the number of new actors who decide to enter the movie business.

2.5.2 Evolution of the degrees of separation of the IMDB graph

In the year 1999, the IMDB graph included 681358 nodes and 33564142 arcs. If we want to compute exactly the distance distribution function of this graph and, therefore, the degrees of separation, we would need (always assuming a computer capable of performing four billion operations per second) a time proportional to

$$\frac{681358 \cdot 33564142}{40^9} = \frac{6813.58 \cdot 3.3564142}{4} \approx \frac{22870}{4} \approx 5717\text{seconds}$$

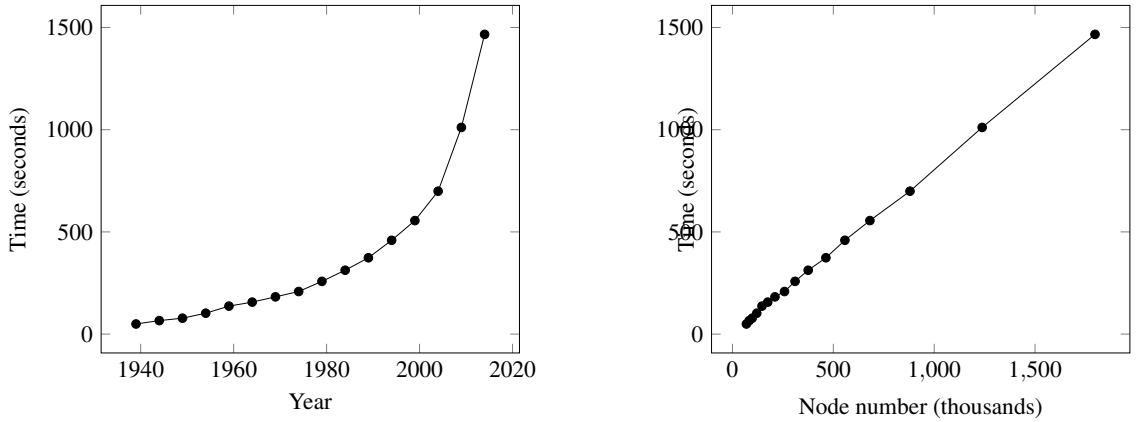
(or about two hours). And this wouldn't even be the largest network to analyze. Therefore, if we want to study the evolution of the degrees of separation in the IMDB graph and receive an answer in a reasonable time, we should make use of the algorithm based on the sampling technique, contenting ourselves with obtaining an approximate estimate.

Recall that this algorithm, to ensure that the absolute error is limited, requires a size that is proportional to the logarithm of the number of nodes in the graph. As in the case of the Slashdot Zoo graph, we will also use a proportionality constant of 100 in the case of the IMDB graph: the sample, therefore, will contain $100\log_2(n)$ randomly selected nodes.

Let us define the following function (in which we assume that, for each year examined yyyy, the file `imdbyyyy.lg` is the representation in `Graphs` format of the IMDB graph in that year).

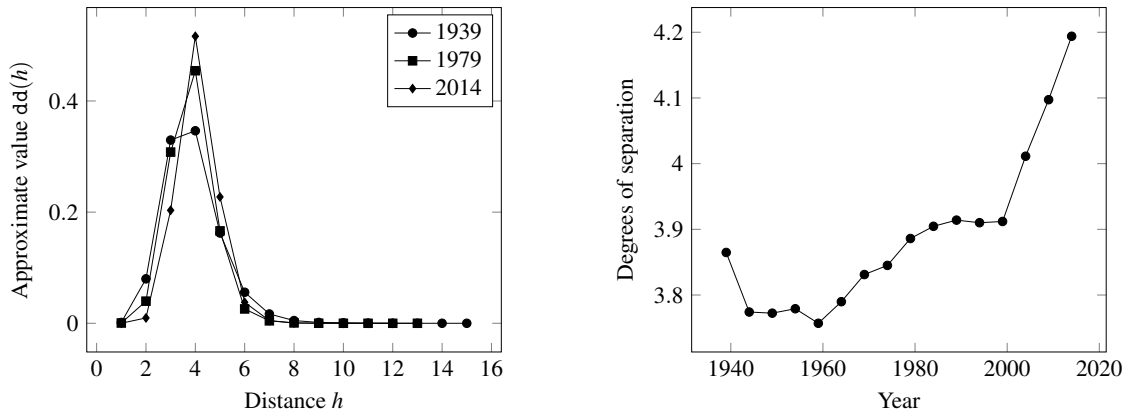
```
function imdb_degrees_separation()::Array{Float64}
    ds::Array{Float64} = zeros(16)
    for year in 1939:5:2014
        graph::String = "imdb/imdb$year.lg"
        g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
        k::Int64 = 100 * trunc(log2(nv(g)))
        dd::Array{Float64} = distance_distribution(graph, k)
        ds[1+div(year-1939,5)] = degrees_of_separation(dd)
    end
    return ds
end
```

Running the instruction `ds = imdb_degrees_separation()` takes 4648 seconds on my computer, which is about an hour and a half. In the figure below, the graph on the left shows the run times for each graph from the year 1939 to 2014 according to the year.



The trend of the plot on the left is very similar to what we have seen previously regarding the number of nodes of the various IMDB graphs. For this reason, in the right part of the previous figure, we show the execution times as a function of the number of nodes. From this plot, it is evident that the execution time is approximately linear in the number of nodes: in other words the logarithmic factor that we have to pay to have a guarantee of the accuracy of the results does not affect that much (as it was expected considering that the logarithm function grows very slowly).

As for the distance distribution function, it does not seem to vary too much over the years. In the left part of the following figure, we show the distribution functions for the years 1939, 1979 and 2014. These functions are quite similar, even if it is possible to notice a greater concentration of the values on the peak value corresponding to the distance 4. We then observe that, on the basis of these values, we can deduce that the diameter of the graph decreases with the passing of the years: from a value of 15 in 1939 to a value of 13 in 1979 and one of 12 in 2014. However, we must take into account that the values of the distance distribution function are approximate values and that, therefore, they may not give a correct estimate of the diameter: we will return to the exact calculation of the diameter of larger graphs later in this book.



Faced with a decreasing diameter, the degrees of separation seems to increase, as shown in the right part of the previous figure. The degrees of separation goes from roughly 3.87 in 1939 to roughly 4.19 in 2014. In other words, it appears that while the more “peripheral” nodes are approaching

TECHNOLOGY

Separating You and Me? 4.74 Degrees

By JOHN MARKOFF and SOMINI SENGUPTA NOV. 21, 2011

The world is even smaller than you thought.

Figure 2.5: In November 2011, a group of scholars from Facebook and the University of Milan discovered that the degrees of separation within the Facebook graph is equal to 4: in other words, as suggested by the 'beginning of this New York Times article at the time, the world is smaller than you thought.

over the years, overall the average distances between the nodes increase. In a graph such as that of IMDB we must, among other things, take into consideration that some actors have died in the course of these over seventy years and that, therefore, it will no longer be possible for other actors to connect to them (unless repertoire scenes are used). Finally, it should not be surprising that the degrees of separation is much lower than that assumed by the Milgram experiment. After all, we are talking about a graph made up of participants who do the same job and who probably frequent the same environment: in other words, the world of cinema is a world even smaller than that of normal relationships.

2.5.3 The degrees of separation of Facebook

More surprising, however, were the results obtained by some Facebook researchers in collaboration with some researchers of the University of Milan [4]. Facebook is an online friendships service born in 2004 and initially aimed only at Harvard University students. The service had a very rapid success that led it to exceed 100 million users in 2008, reaching more than 2 billion users today. In their study, researchers from Facebook and the University of Milan examined the graph of Facebook friends in the year 2011: this graph included over 700 millions of nodes and nearly 70 billions of arcs. It is therefore a gigantic graph, for which even the algorithm based on the sampling technique can have problems in completing its execution (we cannot verify this hypothesis as the graph is not made publicly available for *privacy* reasons). In fact, the analysis of the degrees of separation of this graph has been carried out by making use of more sophisticated probabilistic algorithms, which are able to save not only in the execution time but also in the computer memory used [10, 74].

The result is that the degrees of separation are about 4, instead of the five degrees assumed by Milgram's experiment (see Figure 2.5). More recently, this number has dropped further: at the beginning of 2016, the estimated average distance was approximately 3.57 [9]. Hence, it should be concluded that the creation of a gigantic network of online friendships has truly made the world smaller. Even in this case, however, it can be argued that Facebook users represent a non-representative sample of the entire world population and that friendship relationships in Facebook often do not have much to do with those in the real world.

Beyond these considerations that are difficult to verify, we recall what was said at the end of the first paragraph of this chapter. The computation of the average distance between two nodes of a

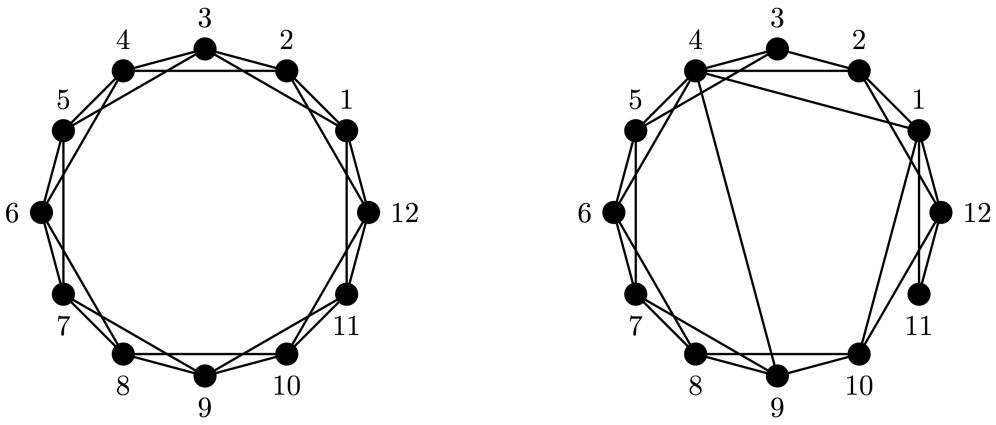


Figure 2.6: The generative model proposed by Watts and Strogatz to replicate the phenomenon of the small world: a "regular" graph (like the one shown on the left) is "perturbed" by replacing, for each node, one of its arcs with an arc that connects it to another node chosen at random (as in the graph on the right).

graph captures a purely topological aspect of the graph itself. The result obtained by the researchers of Facebook and of the University of Milan does not tell us much about what the "algorithmic" distance is between two people, that is, with how many steps a person would actually be able to reach another person, making use of the graph topology. We therefore invite the interested reader to read the good book on this subject by Duncan Watts [87], one of the two researchers who in the nineties proposed a generative model of random graphs that presented the phenomenon of the small world [88].

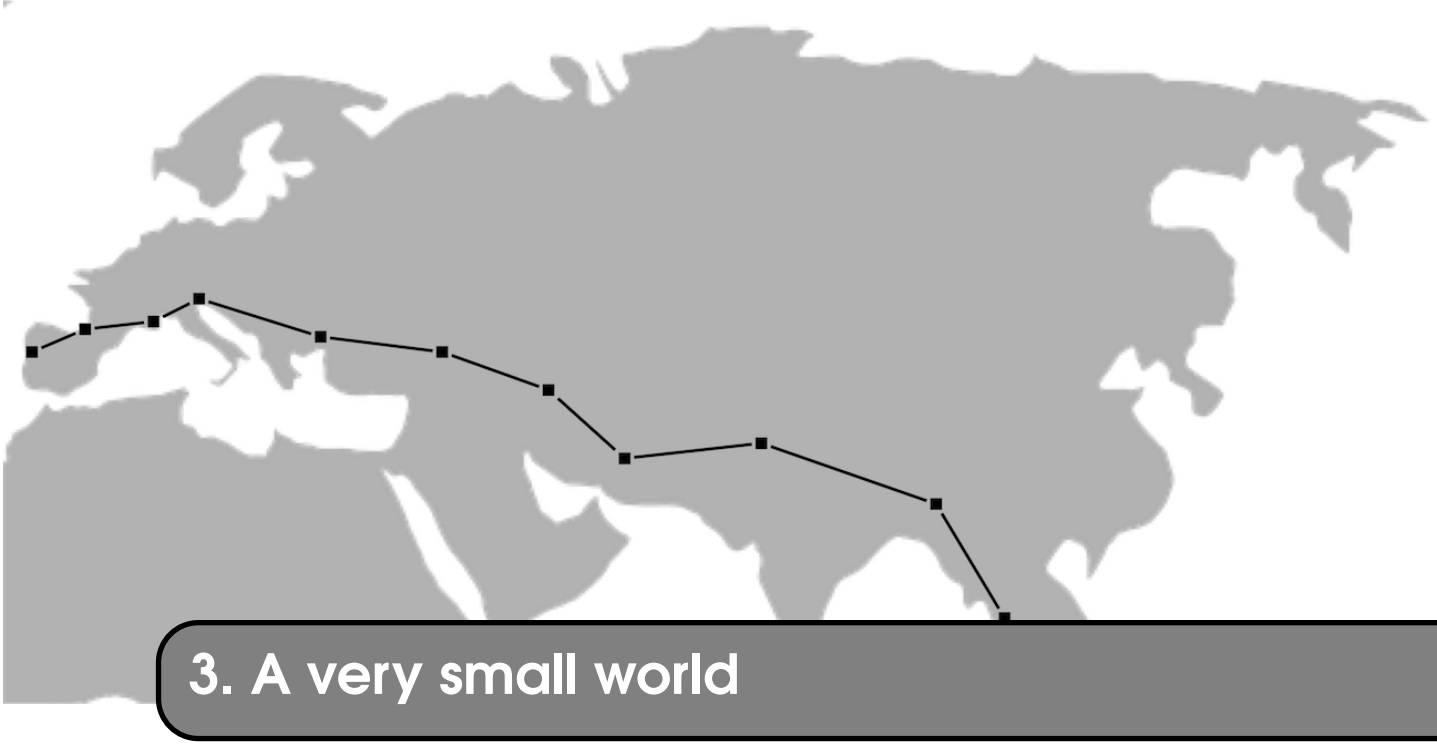
Intuitively, this model consists in randomly "perturbing" a *regular* graph (that is, in which each node had the same number of neighbors and with the same structure), replacing, for each node, an arc with an arc connecting it to another node chosen at random (the `Graphs` package includes, among other things, the `watts_strogatz` function that allows us to generate a random graph using this model).

For example, we start from a graph in which the nodes are positioned along a circumference and each node is connected to its two neighbors in the circumference and to the two nodes on the circumference that are at a distance 2 from it (see the left part of the Figure 2.6). More formally, node i is connected to nodes $i+1$, $i-1$, $i+2$,and $i-2$, where addition and subtraction operations are performed in a "circular" way, that is, modulo the number n of nodes.

The graph thus constructed is modified by carrying out the following stochastic process. For each arc (x,y), with probability β this arc is replaced with an arc (x,z) where z is chosen at random among all nodes other than x. For example, in the right part of Figure 2.6, the arc $(1,3)$ that was present in the original graph has been deleted and replaced by the arc $(1,4)$, the arc $(10,11)$ has been deleted and replaced by the arc $(10,1)$, and the arc $(9,11)$ has been deleted and replaced by the arc $(9,4)$. Note that by doing so, some nodes (like node 1) increase their degree, which was initially equal to 4, while others decrease it (like node 11).

We can see, therefore, how the more "local" arcs that were present in the starting graph (that is, the arcs between nodes at distance 1 or 2 on the circumference) have been replaced with arcs between nodes more "distant": it is precisely these "jumps" introduced randomly that intuitively justify the appearance of the phenomenon of the small world. They allow us to jump from one part of the graph to the other in one step, thus making the average distance of the graph smaller. The final structure of the graph clearly depends on the value of the β parameter. If β is equal to 0 the original graph is not affected, while if β is equal to 1, then the graph becomes a purely random

graph. Intermediate values of this parameter allow to generate graphs with different values of the degrees of separation and, at the same time, different values of another measure of graphs, called the clustering coefficient, which we will analyse later in this book.



3. A very small world

"All right," said Deep Thought. "The Answer to the Great Question..." "Yes..!" "Of Life, the Universe and Everything..." said Deep Thought. "Yes...!" "Is..." said Deep Thought, and paused. "Yes...!" "Is..." "Yes...!!!...?" "Forty-two," said Deep Thought, with infinite majesty and calm.

D.N. Adams, "The Hitchhiker's Guide to the Galaxy", 1979

3.1 Degrees of separation and diameter

We have already said, in the second chapter, that the degrees of separation of a graph are not necessarily related to the *maximum* distance between two nodes of the graph, that is, to the diameter of the graph. In fact, the degrees of separation indicate the *average* distance between any two nodes of the graph: the fact that the degrees of separation are few, therefore, does not imply that there are not two nodes very distant from each other.

For example, consider the graph shown in Figure 3.1, which is made up of a clique of nine nodes (that is, nodes from 1 to 9) to which a path of two edges (that is, the edges $(9,10)$ and $(10,11)$) is connected. Recall that in a clique each node is directly connected to each other node of the clique: therefore, the diameter of this graph is equal to 3, which corresponds to the distance of the node 11 from any node of the clique. Let us now calculate the sum of the distances between any pair of nodes in the graph. For each node of the clique (with the exception of node 9), the sum of its distances from the other nodes of the graph is equal to $8 \cdot 1 + 2 + 3 = 13$, where the

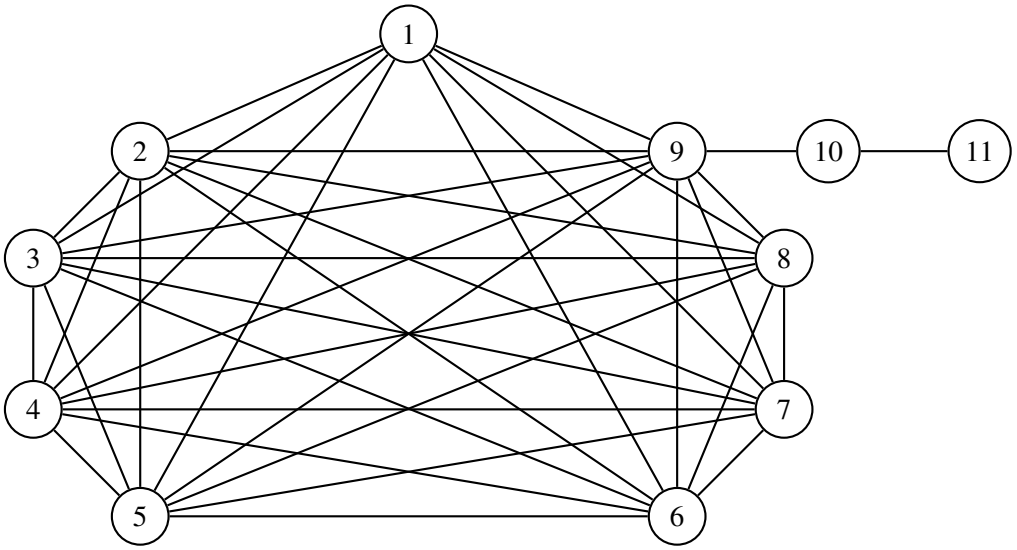


Figure 3.1: An example of a graph where the diameter is more than twice the average distance: each node from 1 to 8 contributes to the sum of the distances with $8+2+3=13$, the node 9 contributes $8+1+2=11$, the node 10 with $16+1+1=18$, and the node 11 with $24+2+1=27$. Therefore, the average distance is equal to $\frac{8\cdot 13+11+18+27}{11\cdot 10} \approx 1.46$, while the diameter is equal to 3.

first addend corresponds to the distances from the other nodes of the clique, the second one at the distance from the node 10, and the third one at the distance from the node 11. As regards node 9, the sum of its distances from the other nodes of the graph is equal to $8\cdot 1+1+2=11$ (since node 9 is directly connected to node 10). For the node 10, the sum of its distances from the other nodes of the graph is equal to $8\cdot 2+1+1=18$, while for the node 11, the sum of its distances from the others nodes of the graph is equal to $8\cdot 3+2+1=27$. Therefore, the sum of all distances is equal to $8\cdot 13+11+18+27=104+11+18+27=160$. The degrees of separation are, therefore, $\frac{160}{11\cdot 10} \approx 1.46$, which is less than half the diameter.

The above example can be generalized to show that the diameter can be arbitrarily larger than the average distance.

Proposition 3.1.1 For every integer $m > 2$, there exists a graph whose diameter is equal to m and whose degrees of separation are less than 3.

Consider a graph consisting of a clique K_n of n nodes $k_1, \ldots, k_n$ and a path of m nodes $p_1, \ldots, p_m$, where $p_1 = k_n$ (that is, the path starts from one of the n nodes of the clique). As in the case of the graph of Figure 3.1, the diameter is equal to the number of nodes in the path, that is m. Now let us calculate the sum of the distances between any pair of nodes in the graph.

- For each node k_i with $1 \leq i < n$, we have that this node is at distance 1 from every other node of K_n and at distance i from the node p_i of the path. Therefore, the sum of its distances from the other nodes of the graph is equal to

$$(n-2) + \sum_{i=1}^{m} i = (n-2) + \frac{m(m+1)}{2},$$

where we considered the node $k_n = p_1$ only in the second addend and we used the Gauss formula for the sum of the first m integers [35].

- For each node p_i with $1 \leq i \leq m$, this node is at distance i from each node k_j of the clique with $1 \leq j < n$, at distance $i-j$ from each node p_j with $1 \leq j < i$, and at a distance of $j-i$ from each node p_j with $i < j \leq m$. Therefore, the sum of its distances from the other nodes of the graph is equal to

$$i(n-1) + \sum_{j=1}^{i-1}(i-j) + \sum_{j=i+1}^{m}(j-i) \leq i \cdot n + \sum_{j=1}^{m} j = i \cdot n + \frac{m(m+1)}{2}.$$

The total sum of the distances is therefore bounded above by

$$(n-1)(n-2) + (n-1)\frac{m(m+1)}{2} + \sum_{i=1}^{m} i \cdot n + m\frac{m(m+1)}{2} \leq n^2 + nm(m+1) + m^2(m+1).$$

The total number of node pairs is equal to $(n+m)(n+m-1) \geq n^2$. If we choose $n = m^2$ and $m \geq 3$, we have that the average distance (that is, the degrees of separation) is at most

$$\frac{m^4 + m^3(m+1) + m^2(m+1)}{m^4} \leq 3$$

and the proposition turns out to be proved.

The Proposition 3.1.1 states, therefore, that there are graphs in which the diameter can be very large, even though having a very small average distance. However, the proof of the proposition makes use of a very particular graph, constituted by the composition of a clique and a path. The question we ask ourselves at this point is whether this situation also occurs in "real" graphs and, above all, in the large ones. For this reason we will now see a very effective algorithm for calculating the diameter of a graph: in the next paragraph, we will consider only unweighted graphs, while in the next one we will see how to extend the procedure to weighted graphs.

3.2 How to compute the diameter in large graphs

Many algorithms have been proposed and analyzed for computing the diameter of a graph. In the general case, the best known solution to date still consists in computing, more or less, the lengths of all shortest paths between each pair of nodes (in other words, solving what is called the *all-pairs shortest path* problem). In the case of undirected and unweighted graphs, this problem is solved in time proportional to nm (where n indicates the number of nodes and m the number of arcs) by performing a BFS starting from each node of the graph. In the case of dense graphs, the computation time can be reduced by making use of efficient algorithms for the multiplication of matrices: in this case, the execution time is proportional to n^ω, where $\omega \approx 2.373$ [2]. We already know that this computation time is not acceptable if we are dealing with graphs with several millions of nodes and several millions if not billions of arcs. Although it is possible to prove that, in theory, it is not possible to design algorithms with an execution time that is proportional to the number of arcs in the graph, now let us see how to design an algorithm that, in practice, is able to compute the diameter of a connected undirected and unweighted graph, by performing a "very small " number of BFSs (we observe that if the graph is not connected, we can apply this algorithm to each of its connected components).

3.2.1 Lower and upper bounds on the diameter

We begin by showing how it is possible to calculate a lower and an upper bound on the diameter value by performing only one BFS.

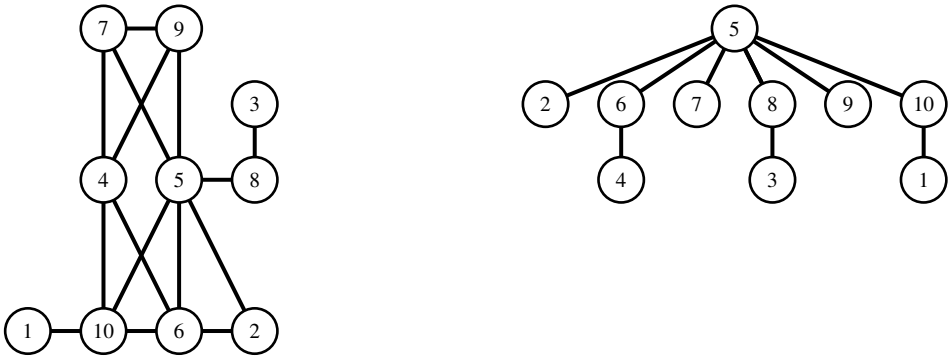


Figure 3.2: The graph of the Japanese archaeological sites and the partition into levels of the BFS starting from node 5: the diameter is at least equal to the number of levels (that is, 2) and at most equal to twice the number of levels (that is, 4).

Proposition 3.2.1 Let G be a connected, undirected, and unweighted graph and let h be the number of levels of the BFS performed starting from a node v of G. Then, the diameter of G is at least equal to h and at most $2h$.

If the number of levels of the BFS performed starting from v is equal to h, this means that there is at least one node u in the level h: as we observed in the first chapter, this implies that $d(u,v) = h$. Since the diameter is the maximum distance between two nodes, we have that the diameter is at least equal to h. To prove the upper bound, we observe that, for each pair of nodes u_1 and u_2, there exists a path starting from u_1, reaching v, and then going to u_2, and which uses the connecting arcs between the various levels of the BFS. Since the number of levels is h, then $d(u_1,v) \leq h$ and $d(v,u_2) \leq h$: therefore, $d(u_1,u_2) \leq 2h$. Since this inequality holds for any pair of nodes u_1 and u_2, we have that the diameter is at most $2h$, and the proposition is proved.

For example, let us consider the graph of the Japanese archaeological sites that we reproduce in the left part of Figure 3.2. The partition into levels of the BFS performed from node 5 is shown in the right part of the figure. In this case, there are two levels. The diameter of the graph is at least 2: for example, the distance between node 5 and node 4 is equal to 2. On the other hand, for each pair of nodes, it is possible to go from one to the other via the node 5. For example, we can go from node 4 to node 5 (passing through node 6), then go from node 5 to node 3 (passing through node 8): therefore, $d(4,3) \leq 2+2 = 4$. Hence, the diameter is at most 4.

We observe that, in the previous example, the upper limit is exactly equal to the diameter: in fact, the maximum distance occurs, for example, precisely in the case of the two nodes 4 and 3 and is equal to 4 (passing through the node 5). In general, however, we do not know what the exact value of the diameter is: the only thing we can say with certainty is that its value is included in the set $\{h, h+1, \ldots, 2h-1, 2h\}$ and that any of these values assures us that the error we make is at most a factor of 2.

An example: the graph of scientific collaborations

The *DataBase systems and Logic Programming* (in short, *DBLP*) is an online service that provides almost complete information on scientific publications in the field of computer science. It is a digital bibliography which, at the moment, covers over 1700 journals and 5600 conferences, for a total of almost six million publications and almost three million authors [28, 61]. Which journals and

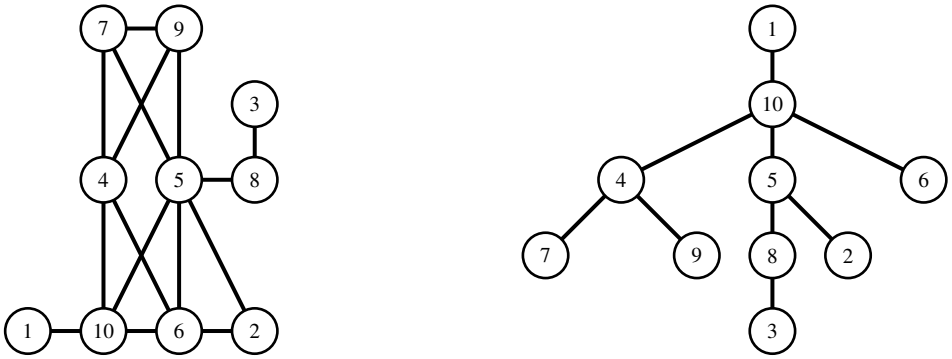


Figure 3.3: The graph of the Japanese archaeological sites and the partition into levels of the BFS starting from node 1: the diameter is at least equal to the number of levels (that is, 4).

which conferences are included in the bibliography also depends on the reports made by the authors themselves, who can ask the editors of DBLP to add a specific journal or a specific conference to the bibliography. The system provides different search tools (for example, by author), allowing you to apply different filters (for example, journal publications only). It is fair to say that, at the moment, DBLP is one of the most used tools for the analysis of the scientific production of researchers working in the field of computer science.

In this chapter, we refer to the data relating only to the conferences included in the DBLP archive in 2015. Using this data, it is possible to create a graph whose nodes are the authors of the articles and in which there is an arc between two nodes if the two corresponding authors have written at least one article together (that is, published at least one article in a computer science conference indexed in DBLP). We observe that, as in the case of the graph of movie collaborations, we are not considering repeated collaborations (we will return to this topic at the end of the chapter). Considering the largest connected component of this graph, we have a graph with 563192 nodes and 2226374 arcs: calculating the diameter of this social network could therefore take a very long time (that is, several hours on my computer). For this reason, we can apply Proposition 3.2.1 to compute an upper and lower bound on the diameter. In particular, by choosing a random node u, we can perform the BFS starting from u, and set the lower bound equal to the number of levels and the upper bound equal to twice the number of levels: this is what the following function does.

```
function diameter_lower_upper_bound(graph::String)::Tuple{Int64,Int64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    lb::Int64 = maximum(gdistances(g, rand(1:nv(g))))
    return lb, 2 * lb
end
```

By executing the instruction `diameter_lower_upper_bound("dblp/lg")` (where we assume that the `dblp.lg` file is the `Graphs` representation of the DBLP graph), we might obtain a lower bound of 15 and, therefore, an upper bound of 30: we observe that this computation required, on my computer, less than three seconds. From this result, it follows that the diameter is at least 15 and that it is at most 30. Deciding what the exact value is does not seem to be possible, if not performing all the BFS starting from each of the 563192 nodes: as we have already said, performing this calculation could take a very long time. We can, however, already conclude that the world of scientific collaborations in the field of computer science is *really* a small world. Not only is the number of degrees of separation very low (by applying the method we described in the second chapter, we can verify that this value is approximately equal to 6.14), but even the maximum distance between any two nodes of the graph is not greater than 30.

3.2.2 Improving the lower bound

The method we have described in the previous paragraph to compute a lower and upper bound on the diameter value can obviously be repeated for a certain number of nodes chosen at random, in order to obtain more precise estimates [69]. In the case of the DBLP graph, for example, by choosing a thousand nodes at random, both the lower bound and the upper bound are likely to improve (for example, to 20 and 24, respectively): however, this requires carrying out a thousand BFS without having, among other things, obtained an exact estimate of the diameter.

A technique to drastically reduce the number of BFSs consists in choosing in an "appropriate" way the nodes from which to start the BFS. Referring, for example, to Figure 3.2, once the BFS has been performed starting from node 5, we could decide to start the next BFS starting from any of the nodes found in the last level (that is, the nodes 1, 3 or 4). If we choose the node 1, the second BFS includes four levels, as shown in the right part of Figure 3.3. Since from Figure 3.2 we had derived that the diameter could not be greater than 4 and from Figure 3.3 we can deduce that the diameter is at least equal to 4 , we can now conclude that the diameter of the graph of the Japanese archaeological sites is 4: we have reached this conclusion by carrying out only two BFSs.

From this example, we can derive a very efficient algorithm that computes a lower bound on the diameter, which, in practice, is almost always very close to the true value: this algorithm, also called *2-sweep*, operates in the the following way (see also Figure 3.4) [64]. A random node x is chosen and the BFS is executed starting from x. At this point, a node y present in the last level of the BFS just performed is selected, and a BFS is executed starting from y. Finally, the maximum between the number of levels of the two BFSs is returned as the lower bound.

As shown in Figure 3.4, the intuitive idea of the algorithm is that the first BFS is used to reach the "border" of the graph. The node y is located on this border, so that the BFS starting from it generates a number of levels very close to the value of the diameter, as it must reach the "opposite border" of the graph.

The graph of the scientific collaborations and the 2-sweep algorithm

Let us define the following function.

```
function two_sweep(graph::String)::Int64
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    lb::Int64, y::Int64 = findmax(gdistances(g, rand(1:nv(g))))
    return max(lb, maximum(gdistances(g, y)))
end
```

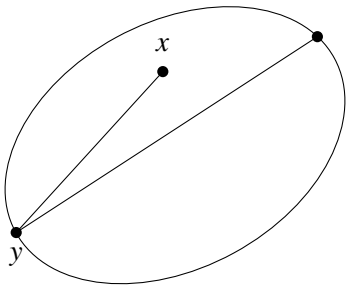


Figure 3.4: The 2-sweep algorithm: after performing a BFS starting from a randomly chosen node x, a BFS is performed starting from any of the nodes at the last level (indicated with y). The value returned by the algorithm is the maximum between the number of levels of the two BFSs.

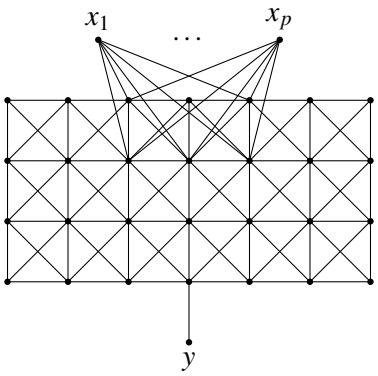


Figure 3.5: With input this grid with k rows and $1+3k/2$ columns, the 2-sweep heuristics can return k, while the diameter of the graph is instead $3k/2$.

If we execute the instruction `two_sweep("dblp.lg")`, most likely the answer will be equal to 22, which is, as we might expect, a value between 20 and 24 (that is, the lower and upper bounds calculated with a thousand BFSs). Clearly, we still do not have the certainty that this value is equal to the diameter of the graph: however, numerous experiments carried out on graphs of different types (and not only) have validated the hypothesis that the value returned by the algorithm is the correct one. If, therefore, our goal is to obtain a reasonable estimate of the diameter (giving up the accuracy of this estimate), the 2-sweep algorithm seems to be the ideal choice: with only two BFSs, this algorithm gives us such an estimate. Before proceeding, let us first observe that it is not difficult to find graphs in which the value returned by the 2-sweep algorithm is quite far from the exact value.

Proposition 3.2.2 For every even integer $k > 3$ such that $k \equiv 0 \bmod 4$, there exists a graph whose diameter is equal to $1+3k/2$ and such that the value returned by the 2-sweep algorithm is equal to k.

Let us consider the grid-like graph shown in Figure 3.5. The diameter of this graph, in which the grid has k rows and $1+3k/2$ columns, is clearly $1+3k/2$, that is, the distance from one node on the left border of the grid to one node on the right border of the grid. On the other hand, if p is sufficiently large, it is likely that the 2-sweep algorithm (in which the initial node r is chosen randomly) will execute a BFS starting from the node x_i, for some i with $1 \leq i \leq p$. The farthest node from x_i is the node y, which is at distance k, since all the other nodes are at distance at most $k-1$ (because x_i is connected to the second row of the grid and because of the "diagonal" arcs in the grid). The BFS executed starting from y will have k levels, so that the 2-sweep algorithm will return the value k.

The previous proposition states that the value returned by the 2-sweep algorithm can be $2/3$ of the exact value, which is not a very good estimate. In practice, however, the value returned by the 2-sweep algorithm is quite always very close to exact value of the diameter.

3.2.3 Computing the exact value of the diameter

As we said, the 2-sweep algorithm gives a good (lower) bound on the diameter but no guarantee that this bound is accurate. For this purpose, let us now see how it is possible to design an algorithm,

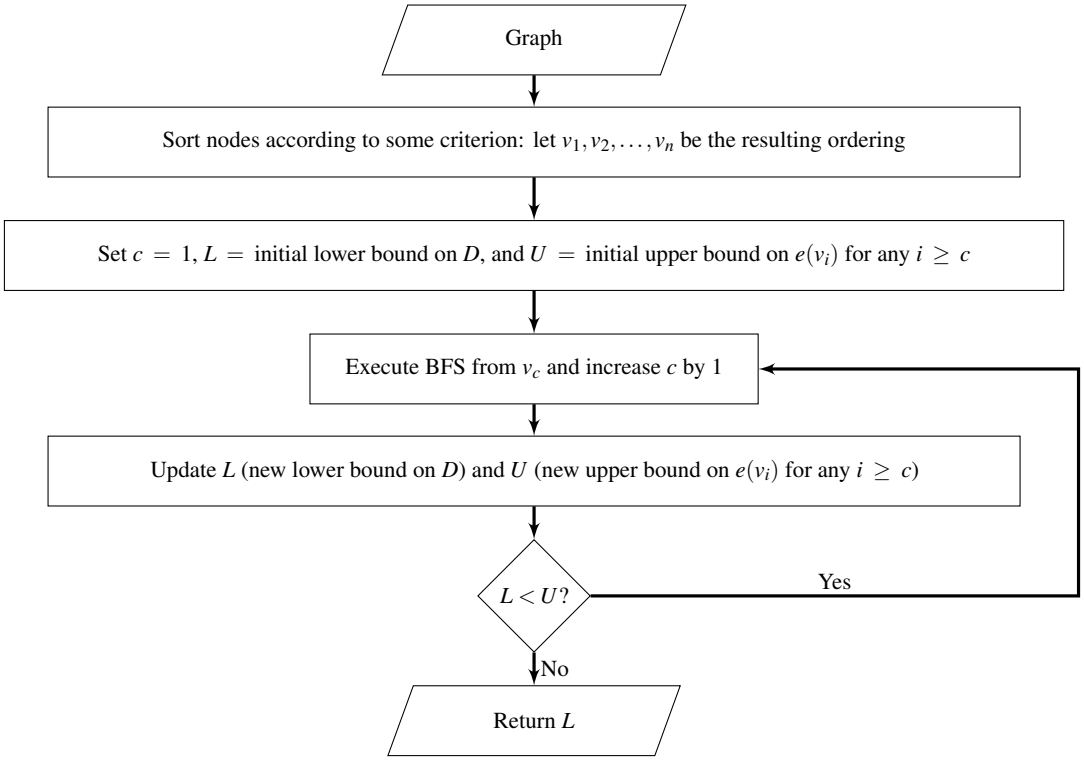


Figure 3.6: The iFUB algorithm scheme, where $e(v_i)$ denotes the eccentricity of node v_i and D denotes the diameter of the graph.

which, by performing a "small" number of BFSs, is able to "refine" more and more the estimate of the diameter value until it reaches the exact value [27]. Even if, in the worst case, we have no guarantee on the efficiency of this algorithm, we will see that, in the case, for example, of the DBLP graph, it can calculate the diameter with a very limited number of BFSs.

The basic idea of this algorithm (see Figure 3.6) is to first sort the nodes according to some criterion. Then, starting from the first node in the resulting ordering, it performs a BFS for *each* of them: in doing so, the algorithm updates the lower bound on the diameter value and, at the same time, updates an upper bound on the *eccentricity* of the nodes from which it has not yet carried out the BFS (remember that the eccentricity of a node is the number of levels of the BFS performed from it). If the upper bound becomes not greater than the lower bound, the algorithm ends the execution and returns the lower bound as the diameter value. Otherwise, the algorithm continues with the next nodes in the ordering. Clearly, in order to implement this algorithm we need to specify how to order the nodes, and how to initialize and update the lower and the upper bounds. The ordering is obtained by executing a BFS from a randomly chosen node u, and by sorting (in non-decreasing way) the nodes according to the level of the BFS they belong to. The lower bound is simply set equal to the maximum eccentricity of the nodes from which a BFS has been executed, while, as we will see, the upper bound is connected to the level of such nodes.

To describe this algorithm more precisely and demonstrate its correctness, we introduce some notations (see also the definitions given in the first chapter). Given a node u, we denote by $e(u)$ the eccentricity of node u, that is the number of levels of the BFS performed starting from u, and we denote by $L_i(u)$, for $1 \leq i \leq e(u)$ the set of nodes that are at the level i of this BFS. Hence, for every node $v \in L_i(u)$, we have $d(u,v) = i$. Moreover, for each i with $1 \leq i \leq e(u)$, we denote by $I_i(u)$ the maximum eccentricity of a node in $L_i(u)$, that is $I_i(u) = \max_{v \in L_i(u)} e(v)$ (we note that, from

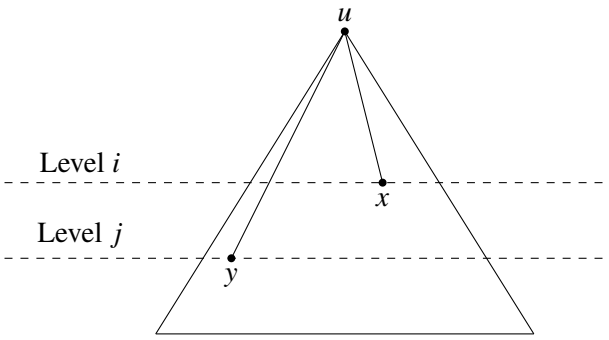


Figure 3.7: Two nodes at two different levels of the BFS are at distance at most the sum of the levels they belong to.

what has been said in the previous paragraphs, $I_i(u)$ is a lower bound on the value of the diameter). Finally, we denote by D the diameter of the graph, that is, D is equal to the maximum eccentricity of the nodes of the graph.

Proposition 3.2.3 For each node u, for each i with $1 \leq i < e(u)$, for each k with $1 \leq k < i$, and for each node $x \in L_{i-k}(u)$ such that $e(x) > 2(i-1)$, there exists a node $y \in L_j(u)$ with $i \leq j \leq e(u)$ such that $e(y) \geq e(x)$.

First of all we observe that, for each i and j with $1 \leq i, j \leq e(u)$, for each node $x \in L_i(u)$ and for each node $y \in L_j(u)$, we have $d(x,y) \leq i + j \leq 2\max\{i, j\}$ (see Figure 3.7). Since $e(x) > 2(i-1)$, then there must exist a node y_x whose distance from x is equal to $e(x)$ and, therefore, greater than $2(i-1)$. If y_x were in $L_j(u)$ with $j < i$, then from the previous observation it would follow that

$$d(x,y_x) \leq 2\max\{i-k, j\} \leq 2\max\{i-k, i-1\} = 2(i-1)$$

and we would have a contradiction. Therefore, y_x must be in $L_j(u)$ with $j \geq i$.

From Proposition 3.2.3 it follows that if L is the maximum eccentricity of all nodes at or below level i, then the eccentricities of all nodes above level i cannot be larger than the maximum between L and $2(i-1)$. This is what the next proposition states.

Proposition 3.2.4 For each i with $1 \leq i \leq e(u)$, let $L = \max_{i \leq j \leq e(u)} I_j(u)$. Then, for each j with $1 \leq j < i$ and for each node $x \in L_j(u)$, $e(x) \leq \max(L, 2(i-1))$.

For every x in $L_j(u)$ with $1 \leq j < i$, the following two cases can occur.

- $e(x) \leq 2(i-1)$. In this case, $e(x)$ is clearly less than or equal to $\max\{L, 2(i-1)\}$.
- $e(x) > 2(i-1)$. In this case, from Proposition 3.2.3 it follows that there exists a node y in $L_k(u)$ with $i < k \leq e(u)$ whose eccentricity $e(y)$ is greater than or equal to $e(x)$. Hence, $L \geq e(y) \geq e(x)$ and $e(x) \leq \max\{L, 2(i-1)\}$.

At this point, it should be clear enough how to go about computing the diameter value. After executing the BFS starting from the node u, the BFSs are executed for the nodes of the various levels, starting from the last level (that is, from the nodes in $L_{e(u)}(u)$) moving upwards. For each level i, $I_i(u)$ is computed (that is, the maximum eccentricity of the nodes included in that level) and the value of L is updated (if necessary): if $L \geq 2(i-1)$, then we can terminate the process, as all the eccentricities of the nodes of the higher levels cannot be larger than L. In other words, the algorithm, which from now on we will call *iFUB* (from *iterative Fringe Upper Bound*), is the following.

- Pick a node u (at random).
- Set $i = e(u)$, $L = I_i(u)$, and $U = 2(i-1)$.
- If $L \geq U$, then return the value L. Otherwise, decrease i by 1, set $L = \max\{L, I_i(u)\}$ and $U = 2(i-1)$, and repeat this step.

We observe that in the worst case, this algorithm could "go up" many levels of the BFS performed starting from u and, therefore, require a computation time proportional to nm, where n denotes the number of nodes and m the number of arcs. In particular this is true when the nodes all have the same eccentricity, and the structure of their BFSs is very similar.

Proposition 3.2.5 For every odd integer $n > 2$, there exists a graph with n nodes on which the iFUB algorithm performs $\frac{n-1}{2}$ BFSs.

Let us consider a loop of n nodes, as shown in the left part of Figure 3.8. The diameter of this graph is equal to $\frac{n-1}{2}$: in fact, for any node u the BFS divides the remaining nodes into $\frac{n-1}{2}$ levels, each containing two nodes, as shown in the right part of the figure. At the end of this BFS, the value of i in the IFUB algorithm is set equal to $\frac{n-1}{2}$ (that is, the eccentricity of u). Furthermore, the two nodes of this level also have an eccentricity equal to $\frac{n-1}{2}$, so the value of lb is also set equal to $\frac{n-1}{2}$. Unfortunately, the algorithm does not yet "know" that this value is the exact value of the diameter and must, therefore, continue going up the levels of the BFS, as long as lb is less than or equal to $2(i-1)$. Since at each level i, the value $I_i(u)$ (that is, the maximum eccentricity of the two nodes at level i) is always equal to $\frac{n-1}{2}$, the value of lb does not change and the algorithm ends when the level i has been visited and the following inequality holds $\frac{n-1}{2} > 2(i-1)$, that is $i < \frac{n+1}{4}$. Thus, the number of levels visited is at least $\frac{n+1}{4} - 1 = \frac{n-3}{4}$: for each of these levels, two BFSs are executed to which must be added the BFS performed starting from u. Thus, the number of BFSs performed by the iFUB algorithm is at least equal to

$$2\frac{n-3}{4} + 1 = \frac{n-3}{2} + 1 = \frac{n-1}{2}.$$

The proposition is therefore proved.

The previous proposition tells us that, in the worst case, the iFUB algorithm requires a computation time proportional to the product of the number of nodes by the number of arcs and, therefore, it is not very efficient. In practice, however, graphs do not have that "regularity" that makes the iFUB algorithm inefficient: for this reason, this algorithm almost always manages to calculate the diameter of a social network by performing a very small number of BFSs, especially if the initial node u is not chosen at random but it is a node of maximum degree. This node can be selected by making use of the following function.

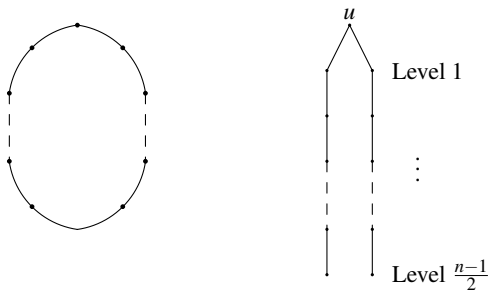


Figure 3.8: An example of a graph on which the iFUB algorithm performs a number of BFSs proportional to the number of nodes of the graph: in particular, whatever the starting node u is, the algorithm performs $\frac{n-1}{2}$ BFSs.

```
function max_degree_node(graph::String)
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    _, u = findmax(degree_centrality(g, normalize = false))
    return u
end
```

The diameter of the graph of the scientific collaborations

We have seen that the DBLP graph includes 563192 nodes and 2226374 arcs, and that computing its diameter by performing a BFS from each node can be too expensive in terms of execution time. Let us see how, using the iFUB algorithm, it is possible to compute the diameter of the DBLP graph by performing only 4 BFSs, with an execution time approximately equal to six seconds (on my computer). As suggested at the end of the previous paragraph, we choose a node of maximum degree as the initial node u of the algorithm. At this point, we perform the BFS starting from u, we set the index i of the level at the maximum distance from u (that is, at its eccentricity), and compute the maximum eccentricity of the nodes located at this level, storing this value in L, and setting the value of U equal to $2(i-1)$. The algorithm then continues, as long as L is less than U, moving to the previous level $i-1$, computing the maximum eccentricity of the nodes located in this level, storing this value in L if it is greater, and setting the value of U equal to $2(i-1)$. All of this is done in the following function, where for each BFS that is performed, a counter increases (so that we ultimately know how many BFSs the algorithm has performed).

```
function ifub(graph::String, u::Int64)::Tuple{Int64,Int64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    d::Array{Int64}, nbfs::Int64 = gdistances(g, u), 1
    node_index::Array{Int64} = sortperm(d, alg = RadixSort, rev = true)
    c::Int64, i::Int64, L::Int64, U::Int64 = 1, d[node_index[1]], 0, nv(g)
    while (L < U)
        L, U = max(L, maximum(gdistances(g, node_index[c]))), nv(g)
        nbfs, c = nbfs + 1, c + 1
        if (d[node_index[c]] == i - 1)
            U, i = 2 * (i - 1), i - 1
        end
    end
    return L, nbfs
end
```

Note that, in order to be consistent with the scheme described in Figure 3.6, the function first orders the node according to their distance from the node u, and then executes a BFS starting from each node in the resulting ordering $v_1, \ldots, v_n$ (where n is the number of nodes). At the end of the execution the BFS starting from the node v_c, the upper bound is set equal to $2(i-1)$ if v_c is the last node at level i, otherwise it is set equal to n (in order to be sure that all nodes at level i are examined).

If we execute the above code, it will be communicated that the diameter of the DBLP graph is equal to 22: to calculate this value, the algorithm only performed 4 BFSs! We observe that if the initial node u were chosen at random, the number of BFSs can also increase significantly: for example, by choosing node 324311, the number of visits performed rises up to 114 and the execution time is approximately equal to 15 seconds (obviously, the diameter is always equal to 22).

In conclusion, the DBLP graph, despite having over half a million nodes and "only" just over two million arcs, is a graph in which the maximum distance between two nodes is equal to 22 and, therefore, very small. The phenomenon of the very small diameter is common to almost all real-world graphs and, like the phenomenon of the distribution of degrees and the phenomenon of the small world, is a characteristic of real-world graphs that is often taken into account when evaluating mathematical models capable of generating synthetic graphs, as we will see later in this book.

3.2.4 Extension to directed graphs

The findings described in the preceding paragraphs extend quite easily to strongly connected directed graphs [26]. For this purpose, for each node u of the graph G we can define its *forward eccentricity* $e_F(u)$, defined as the number of levels of the BFS performed on G starting from u, and its *backward eccentricity* $e_B(u)$, defined as the number of levels of the BFS performed on the *transposed* graph G^t starting from u, where the transposed graph is obtained by inverting the directions of all arcs.

Lower and upper bounds on diameter

Similarly to what we have said for undirected graphs, the maximum value between $e_F(u)$ and $e_B(u)$ is a lower bound on the value of the diameter. Furthermore, the sum of $e_F(u)$ and $e_B(u)$ is an upper bound on the value of the diameter: in fact, the BFS in the transposed graph provides all shortest paths reaching u, while the BFS in the original graph gives all shortest paths starting at u. Therefore, for any pair of nodes x and y, their distance $d(x,y)$ cannot be greater than $d(x,u)+d(u,y)$, that is, the length of the path obtained by going first from x to u and then from u to y.

Improving the lower bound

The analogue of the 2-sweep algorithm in the case of directed graphs, once a node u has been chosen at random, performs a BFS starting from u in the graph G, and then performs one BFS starting from a node of the last level in the transposed graph G^t. Next, it performs a BFS starting from u in the transposed graph G^t, and then performs a BFS starting from a node of the last level in the original graph G. The algorithm returns as the lower bound the maximum number of levels of the four BFSs that have been performed (see Figure 3.9). This directed version of the 2-sweep algorithm is also very effective (especially if the node u is chosen either as the one with maximum out-degree or as the one with maximum in-degree).

Computing the exact value of the diameter

We define an analog of the iFUB algorithm for directed strongly connected graphs. For $1 \leq i \leq e_F(u)$, let $L_i^F(u)$ be the set of nodes at level i of the BFS performed starting from node u in the graph, and, for $1 \leq i \leq e_B(u)$, let $L_i^B(u)$ be the set of nodes at level i of the BFS performed

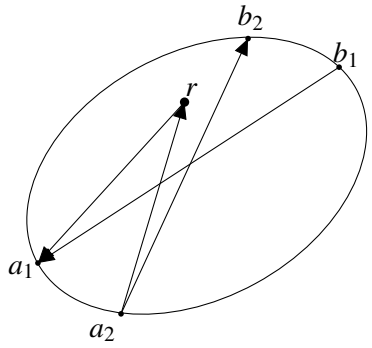


Figure 3.9: The 2-sweep algorithm for directed graphs. After performing a BFS starting from a randomly chosen node r, a "backward" BFS is performed starting from any of the nodes at the last level (indicated with a_1). After performing a backward BFS starting from r, a BFS is performed starting from any of the nodes at the last level (indicated with a_2). The value returned by the algorithm is $\max\{d(b_1,a_1),d(a_2,b_2)\}$.

starting from u in the transposed graph. Moreover, for $1 \leq i \leq e_F(u)$, let $I_i^F(u)$ be the maximum "backward" eccentricity of a node in $L_i^F(u)$, that is $I_i^F(u) = \max_{v \in L_i^F(u)} e_B(v)$ (by convention, we assume that if $i > e_F(u)$, then $I_i^A(u) = 0$). Finally, for $1 \leq i \leq e_B(u)$, let $I_i^B(u)$ be the maximum forward eccentricity of a node in $L_i^B(u)$, that is $I_i^B(u) = \max_{v \in L_i^B(u)} e_F(v)$ (by convention, we assume that if $i > e_B(u)$ then $I_i^B(u) = 0$). Propositions 3.2.3 and 3.2.4 can be suitably adapted to the case of directed graphs, to obtain the following algorithm, similar to the iFUB one, which we call *iDiFUB*.

- Choose a random node u, and set $i = \max(e_F(u), e_B(u))$, $L = \max(I_i^F(u), I_i^B(u))$, and $U = 2(i-1)$.
- If $L \geq U$, then return the value L. Otherwise, decrease i by 1, set L equal to $\max\{lb, I_i^F(u), I_i^B(u)\}$ and $U = 2(i-1)$, and repeat this step.

Let us see how the iDiFUB algorithm works on the graph shown in the left part of Figure 3.10, and whose distance matrix is the following.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	e_F
v_1	0	2	1	3	1	3	2	3	2	4	1	2	4
v_2	1	0	2	1	2	3	2	3	3	4	2	3	4
v_3	2	1	0	2	3	2	1	2	4	3	3	4	4
v_4	1	3	2	0	2	2	1	2	3	3	2	3	3
v_5	3	2	1	2	0	3	2	2	1	3	4	5	5
v_6	2	4	3	1	3	0	2	3	4	4	3	4	4
v_7	3	4	3	2	2	1	0	1	3	2	4	5	5
v_8	4	3	2	3	1	4	3	0	2	1	5	6	6
v_9	2	4	3	1	2	3	2	1	0	2	3	4	4
v_{10}	5	4	3	4	2	5	4	1	3	0	6	7	7
v_{11}	2	4	3	5	3	5	4	5	4	6	0	1	6
v_{12}	1	3	2	4	2	4	3	4	3	5	2	0	5
e_B	5	4	3	5	3	5	4	5	4	6	6	7	

In the center part of the figure, we show the result of the BFS executed starting from v_1 in the graph, while in the right part we show the result of the BFS executed starting from v_1 in the transposed graph. The execution of the iDiFUB algorithm goes through the following steps.

- At the beginning $i = \max\{e_F(v_1), e_B(v_1)\} = \max\{4,5\} = 5$, $I_5^F(v_1) = 0$ (since $5 > \mathrm{ecc}_F(v_1)$), $I_5^B(v_1) = e_F(v_{10}) = 7$, $L = \max\{0,7\} = 7$, and $U = 2(i-1) = 8$.

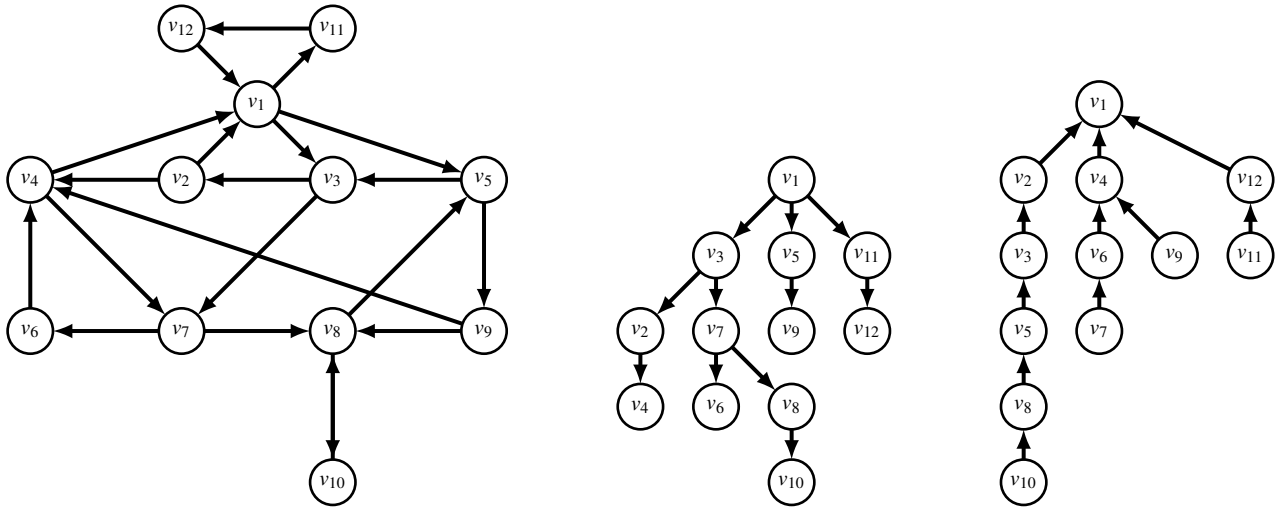


Figure 3.10: A directed graph (left), the BFS executed starting from v_1 (center), and the BFS executed starting from v_1 in the transposed graph.

- Since $L = 7 < 8 = U$, the algorithm decrease i by 1, that is, set $i = 4$. Since $I_4^F(v_1) = e_B(v_{10}) = 6$ and $I_4^B(v_1) = e_F(v_8) = 6$, the algorithm set $L = \max\{7,6,6\} = 7$ and $U = 2(i-1) = 6$.
- Since $L = 7 > 6 = U$, the algorithm returns the value 7, which is the correct diameter value.

As we can see, the algorithm has computed the exact value of the diameter by executing five BFSs, instead of performing the twelve BFSs, that would have been done by the exhaustive algorithm. In general, the iDiFUB also appears to be extremely efficient when applied to real-world graphs. As in the case of the iFUB algorithm, the choice of the initial node u can be performed in a more "clever" way (for example, by choosing the node with the maximum number of outgoing arcs or the one with the maximum number of incoming arcs).

3.3 Diameter in weighted graphs: Dijkstra's algorithm

So far we have only considered unweighted graphs. In order to extend the methods described in the previous paragraphs to weighted graphs, we must introduce one of the best known algorithms for the calculation of shortest paths in graphs, namely the *Dijkstra's algorithm* [32]. This algorithm operates in a similar way to the BFS, but is different from the latter for two main characteristics: the use of a priority queue (instead of a queue, as in the case of the BFS) and the update of the distance from the starting node when a better path is identified (which cannot happen in the case of a BFS).

A *priority queue* is another very common data structure in computer science, typically taught in a first course of algorithms and data structures of any degree course in computer science. Unlike a simple queue, such as the one we used for the BFS, the priority queue allows us to store elements characterized by an identifier and a numeric value (normally called *priority*). In particular, the priority queue allows us to perform the following operations.

- *Insertion* of an element x with priority p.
- *Extraction* of the element x with minimum priority.
- *Update* of the priority of an element x by a new value p (usually, this new value is smaller than the previous one).

There are several implementations of a priority queue that allow us to perform each of the three previous operations in a time proportional to the logarithm of the number of elements in the priority queue. Unlike, therefore, the queue for which the insertion and the extraction of an element both require a constant time (independently, that is, from the number of elements in the queue), the

priority queue requires a "small" price to be payed to have the possibility to associate a numerical value to the elements and to quickly identify the element with the smallest value.

At this point we are able to define the Dijkstra's algorithm, which, given as input a weighted graph and a starting node s, returns the distances of all the other nodes from s (remember that, in the case of weighted graphs, the distance between two nodes is the sum of the weights of the arcs of the shortest path and, therefore, does not necessarily coincide with the number of arcs of the path itself).

The algorithm performs the following operations (very similar to those of the BFS).

1. The starting node s is inserted in the priority queue with a priority equal to 0, while all the other nodes are inserted in the priority queue with priority ∞ (or, anyway, an arbitrarily large priority). Furthermore, the predecessor $p[s]$ of the starting node is set equal to itself.
2. As long as the priority queue is not empty, it does the following.
   (a) The node u with lowest priority is extracted from the priority queue. Let $d[u]$ be the priority of u: then, the definitive distance of u from s is set equal to $d[u]$.
   (b) For every neighbor v of u (that is, for every node v in the adjacency list of u) that has not already been taken from the priority queue, if $d[v] > d[u] + w(u,v)$, where $d[v]$ indicates the priority of v and $w(u,v)$ indicates the weight of the arc (u,v), then the priority of v is updated and set equal to $d[u] + w(u,v)$ and the predecessor $p[v]$ is set equal to u.

Before proving the correctness of this algorithm and to better understand its behavior, let us consider the weighted graph shown in Figure 3.11. This graph is a sub-graph of the graph of the characters of the novel *Les miserables*, in which the weights are inversely proportional to the number of times two characters appear in the same chapter [52]. We can, in fact, imagine that the more two characters appear in the same chapter, the more "close" they are to each other in the novel. In particular, the weights were assigned according to the following rule: if the characters x and y appear together in c chapters, then the weight of the arc (x,y) is set equal to $\frac{100}{c}$. For example, the character Myriel (that is, the node 1) appears with the character Valjean (that is, the node 2) in five chapters: for this reason, the weight of the arc $(1,2)$ is $\frac{100}{5} = 20$. We observe that, in this case, the graph is not directed, but, in general, Dijkstra's algorithm applies to both directed and undirected graphs.

Suppose we want to compute the shortest paths starting from node 1. Initially, therefore, the priority queue contains nine elements, all with priority equal to ∞, except the element 1 which has priority 0 (see the first row of the Table 3.1). The element 1 is then extracted from the priority queue and its distance becomes permanently equal to its priority, which is 0. Since 1 has only one neighbor, namely node 2, and since $\infty > 0 + 20$, the priority of node 2 is updated and set equal to 20 (see the second row of the table). At this point, the node 2 is extracted from the priority queue and its distance becomes permanently equal to its priority, which is 20. Node 2 has 5 neighbors, of which one is node 1 already taken from the priority queue. For the other neighbors, that is, the nodes 3, 4, 5, and 6 nodes, the priority is updated as $\infty > 20 + 11$, $\infty > 20 + 9$, $\infty > 20 + 9$, and $\infty > 20 + 6$, respectively. Also, the ancestor of these four nodes becomes the node 2 (see the third row of the table). Now, the node 6 is taken from the priority queue and its distance becomes permanently equal to its priority, which is 26. Node 6 has 5 neighbors, of which one is node 2 already taken from the priority queue. For the other neighbors, that is, the nodes 3, 4, 5, and 8, only the latter's priority is updated as $31 < 26 + 20$, $29 < 26 + 20$, $29 < 26 + 100$ and $\infty > 26 + 100$, respectively. Also, the predecessor of the node 8 becomes the node 6 (see the fourth row of the table). At this point, the node 4 is extracted from the priority queue and its distance becomes permanently equal to its priority, which is 29. The node 4 has 7 neighbors, of which two are the nodes 2 and 6 already taken from the priority queue. For the other neighbors, that is, nodes 3, 5, 7, 8, and 9, only the priority of the nodes 7, 8, and 9 is updated as $31 < 29 + 100$, $29 < 29 + 100$,

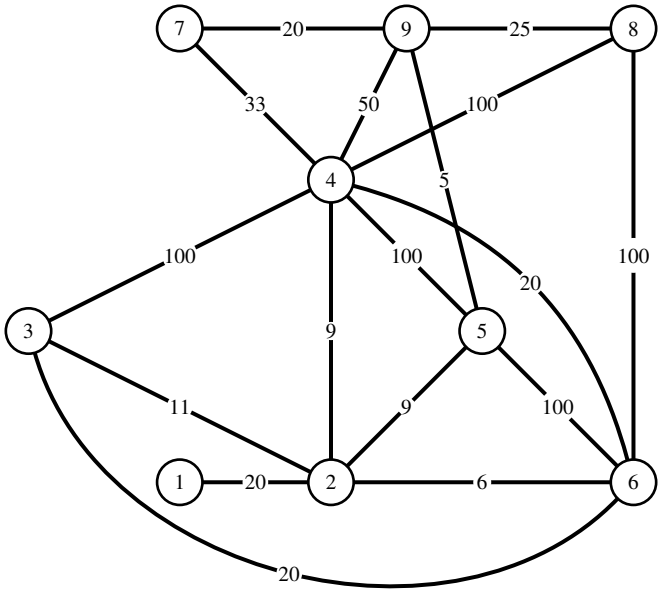


Figure 3.11: The weighted graph of some of the characters in the novel *Les miserables*: the weights of the arcs are inversely proportional to the number of times the two characters appear in the same chapter (that is, the smaller the weight of an arc, the more the two characters connected by the arc appear in the same chapter).

$\infty > 29+33$, $\infty > 29+100$, and $\infty > 29+50$, respectively. Also, the predecessor of the nodes 7, 8 and 9 becomes the node 4 node (see the fifth row of the table). Now, the node 5 is taken from the priority queue and its distance becomes permanently equal to its priority, which is 29. The node 5 has 4 neighbor, of which three are the nodes 2, 4, and 6, already taken from the priority queue. For the other neighbor, the node 9, its priority is updated as $79 > 29+5$. Also, the predecessor of the node 9 becomes the node 5 (see the sixth row of the table). The node 3 is now taken from the queue and its distance becomes permanently equal to its priority, which is 31: in this case, all the neighbors have already been extracted from the queue. At this point, the node 9 is extracted from the priority queue and its distance becomes permanently equal to its priority, which is 34. The node 9 has 4 neighbors, of which two are the nodes 4 and 5, already taken from the priority queue. For the other two neighbors, the nodes 7 and 9 nodes, their priority is updated as $62 < 34+20$ and $126 > 34+25$, respectively. Also, the predecessor of the nodes 7 and 8 becomes the node 9 (see the seventh row of the table). Finally, the two nodes 7 and 8 are extracted from the queue and their distance becomes definitively equal to their priority, that is, 54 and 59, respectively. The algorithm ends at this point.

To prove that Dijkstra's algorithm correctly calculates the distances from the starting node s to all the other nodes, it is sufficient to verify that the priority $d[v]$ of a node v at the moment of its extraction from the priority queue is equal to $d(s,v)$, that is, at a distance of node v from node s. For this purpose, for each integer $i > 0$, we indicate with F_i the *frontier* of the Dijkstra's algorithm at step i, that is the set of the first i nodes extracted from the priority queue.

Proposition 3.3.1 For each i with $1 \leq i \leq n$ and for each node $u \in F_i$, the priority of u at the

	1		2		3		4		5		6		7		8		9	
	d	p	d	p	d	p	d	p	d	p	d	p	d	p	d	p	d	p
u	**0**	1	∞	1	∞	1	∞	1	∞	1	∞	1	∞	1	∞	1	∞	1
1		1	**20**	1	∞	1	∞	1	∞	1	∞	1	∞	1	∞	1	∞	1
2		1		1	31	2	29	2	29	2	**26**	2	∞	1	∞	1	∞	1
6		1		1	31	2	**29**	2	29	2		2	∞	1	126	6	∞	1
4		1		1	31	2		2	**29**	2		2	62	4	126	6	79	4
5		1		1	**31**	2		2		2		2	62	4	126	6	34	5
3		1		1		2		2		2		2	62	4	126	6	**34**	5
9		1		1		2		2		2		2	**54**	9	59	9		5
7		1		1		2		2		2		2		9	**59**	9		5
8		1		1		2		2		2		2		9		9		5

Table 3.1: The evolution of the priorities of the nodes of the weighted graph of the characters of the novel *Les miserables* during the execution of the Dijkstra's algorithm: for each node, not only the priority d is indicated but also the predecessor p along the shortest path corresponding to this priority. The values in bold are the distances of the nodes from the starting node, that is,the node 1.

time of its extraction is equal to $d(s,v)$.

The proposition is clearly true after the first extraction, that is, for $i = 1$. In this case, in fact, the frontier F_1 is the only node s and its priority, at the moment of its extraction, is equal to 0, which is equal to the distance of s from itself. Reasoning by induction, suppose that the proposition is true for every k with $1 \leq k < i \leq n$: that is, the priority of each node $v \in F_k$ at the time of its extraction is equal to $d(s,v)$. Let u the i-th node extracted from the queue: we must prove that the priority $d[u]$ at the time of its extraction is equal to $d(s,u)$. Suppose, on the contrary, that this is not true, that is, that $d(s,u) < d[u]$. Since $u \notin F_{i-1}$, then there must exist a shortest path π from s to u which, at some point, passes through a node not in F_{i-1}: let (x,y) be the first arc of this path with $x \in F_{i-1}$ and $y \notin F_{i-1}$ and let π_x be the sub-path of π from s to x. Thus, $l(\pi_x) + w(x,y) \leq l(\pi) < d[u]$, where $l(\cdot)$ denotes the length of a path (that is, the sum of the weights of its arcs). Since $d(s,x) \leq l(\pi_x)$, then we have that $d(s,x) + w(x,y) \leq l(\pi) < d[u]$. Since $x \in F_{i-1}$ and $y \notin F_{i-1}$, then when taking x from the priority queue, the priority of y must have been "examined" and, therefore, $d[y] \leq d[x] + w(x,y) = d(s,x) + w(x,y)$, where the equality follows from the inductive hypothesis. Therefore, at the time of extracting u from the priority queue, we have $d[y] \leq l(\pi) < d[u]$. On the other hand, if u is the extracted node, then $d[u] \leq d[y]$ (since the node with the least priority is extracted). In conclusion, we have that $d[u] \leq d[y] \leq l(\pi) < d[u]$, that is a contradiction. We have thus proved the inductive step and, therefore, the proposition.

The correctness of Dijkstra's algorithm follows from Proposition 3.3.1 by considering $i = n$, that is, after all nodes have been extracted from the priority queue. As regards the execution time, as in the case of the BFS, each arc is examined only once: when it is examined, it can modify the priority of an element present in the queue and, therefore, request a time proportional to the logarithm of the number of nodes of the graph. If we also consider the time needed to initialize the priority queue and extract all nodes, we can conclude that the execution time of Dijkstra's algorithm is proportional to $(n+m)\log_2(n)$. In other words, this algorithm requires an execution time slightly higher than

that of the BFS, but by a factor that we can consider "reasonable". By using this algorithm all the techniques described in the previous paragraphs can be adapted to the case of weighted (directed) graphs (provided that they are (strongly) connected). In general, the corresponding iFUB and iDiFUB algorithms still are very effective (with the most remarkable exception of road networks). We suggest the interested reader to choose one large weighted real-world graph and apply the weighted version of these algorithms, by making use of the `dijkstra_shortest_paths` function included in the `Graphs` package.

3.4 Can we do better?

Since the very beginning of theoretical computer science and until recent years, the duality between NP-hard problems and polynomial-time solvable problems has been considered the threshold distinguishing "easy" from "hard" problems [43]. However, polynomial-time algorithms might not be as efficient as one expects: for instance, in real-world graphs with millions or billions of nodes, also quadratic-time algorithms for computing the diameter might turn out to be too slow in practice, and a *truly sub-quadratic* algorithm would be a significant improvement, where an algorithm is said to be truly sub-quadratic if its time-complexity is $O(n^{2-\varepsilon})$ for some $\varepsilon > 0$, where n is the input size. Following the main ideas behind the theory of NP-completeness, researchers have recently started to prove that the existence of such an algorithm would imply faster solutions for other well-known and widely studied problems [1]. As an example, a great amount of work started from the analysis of the Strong Exponential Time Hypothesis (in short, *SETH*), which has been used as a tool to prove the hardness of polynomial-time solvable problems [48]. This hypothesis says that there is no algorithm for solving the well-known k-SAT problem in time $O((2-\varepsilon)^n)$, where $\varepsilon > 0$ does not depend on k. SETH has become a starting point for proving the "hardness" of many other problems, especially in the field of graph mining. Note that all these results do not deal with the notion of completeness, but they simply prove that "a problem is harder than another", using a kind of reduction between problems, relying on the fact that the easiest problem has been studied for years and no efficient algorithm has been found.

In general, a (combinatorial) problem A is a function which associates to any input $x \in I_A$, where I_A is the the set of *instances* of the problem, a set of outputs $A(x)$ (each string in $A(x)$ is called a *solution* of x). If, for any $x \in I_A$, $A(x) \in \{\{0\},\{1\}\}$, then the problem A is said to be a decision problem. Given two problems A and B, we say that A is reducible to B is there exist two functions f and g such that, for any $x \in I_A$, $x' = f(x) \in I_B$, and, for any $y' \in B(x')$, $y = g(x,y') \in A(x)$ (see Figure 3.4). In the theory of NP-completeness, the two functions f and g are usually required to be computable in polynomial time. However, since our goal is to analyze the complexity of problems which are solvable in polynomial time, in the following we will require that f and g are computable in *linear* time. Indeed, if this is the case and if A is reducible to B, then a sub-quadratic algorithm for B would imply a sub-quadratic algorithm for A.

3.4.1 From SETH to quadratic SAT

Let us consider an "artificial" variation k-SAT* of k-SAT which is quadratic-time solvable, but not solvable in time $O(n^{2-\varepsilon})$ unless SETH is false [13, 95]. In this variation, an instance is formed by two sets of variables $X = \{x_i\}$ and $Y = \{y_j\}$ of the same size, a set C of clauses over these variables, such that each clause has at most size k, and the two power sets $\mathscr{P}(X)$ and $\mathscr{P}(Y)$ (which are used to change the input size). The solution to an instance of k-SAT* is 1 if there is an evaluation of all variables that satisfies all clauses, 0 otherwise. This problem differs from the classic one only by the input size. In this way, a quadratic-time algorithm exists for k-SAT*, which simply tries all possible assignments of the variables. However, if m is the number of variables and $n = 2^{\frac{m}{2}}$ is the input size, since both X and Y have size $\frac{m}{2}$, an algorithm running in time $O(n^{2-\varepsilon})$ with ε

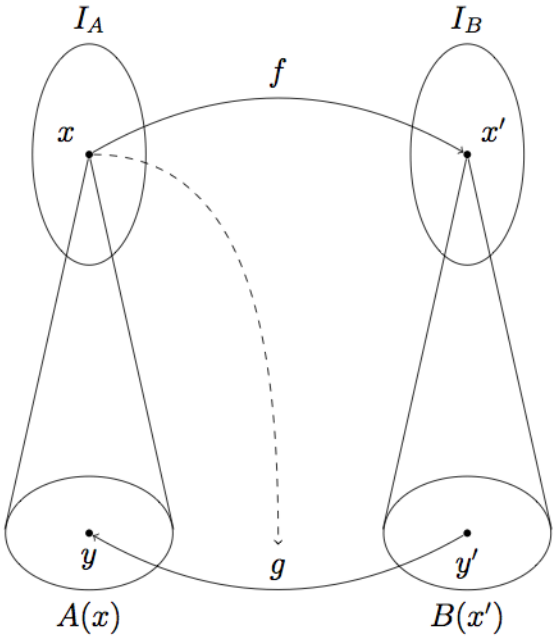


Figure 3.12: A problem A is reducible to a problem B if we can map any instance of A to an instance of B, and if we can recover a solution for A starting from a solution from B.

not depending on k would imply an algorithm solving k-SAT in time $O(2^{\frac{m}{2}(2-\varepsilon)}) = O\left(\left(2^{\frac{2-\varepsilon}{2}}\right)^m\right)$, where m is the number of variables. This latter result contradicts SETH.

3.4.2 From quadratic SAT to disjoint sets

The Two Disjoint Set (in short, *TDS*) problem is defined as follows. Given a set X and a collection $\mathscr{C}$ of subsets of X, the solution is 1 if there are two disjoint sets $C, C' \in \mathscr{C}$, 0 otherwise. Clearly, this problem can be solved in quadratic time. Let us now show that k-SAT* is reducible (in linear time) to TDS, thus implying that this latter problem is not solvable in sub-quadratic time, unless SETH is false [13].

Let X_1 be the set of the possible evaluations of $\{x_i\}$, X_2 the set of possible evaluations of $\{y_j\}$, C the set of clauses of an instance I of k-SAT*. We define $f(I) = (X, \mathscr{C})$, where $X = C \cup \{t_1, t_2\}$, and $\mathscr{C}$ is the collection made by sets of clauses not satisfied by an assignment in X_1 or X_2 (and t_1, t_2 are used to distinguish between assignments in X_1 and X_2). More formally, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$, where

$$\mathscr{C}_1 := \{\{t_1\} \cup \{c \in C : x \text{ does not satisfy } c\} : x \in X_1\},$$

and

$$\mathscr{C}_2 := \{\{t_2\} \cup \{c \in C : x \text{ does not satisfy } c\} : x \in X_2\}.$$

This way, the size of $f(I)$ is linear in the size of I. Moreover, it is possible to compute f by analyzing all sets of evaluation of variables one by one, and for each of them check which clauses are verified. For each evaluation in X_1 or X_2, the checking time is proportional to the number of clauses, which is at most $O(\log^k(n))$, where n is the input size of k-SAT*. It remains to prove that the output of the problem is preserved: we will prove that there is a bijection g between the pairs of disjoint sets and the satisfying assignments of the formula of k-SAT*. In particular, two sets that are both in $\mathscr{C}_1$ or both in $\mathscr{C}_2$ cannot be disjoint because of t_1 and t_2. As a consequence, two disjoint sets correspond to an evaluation of all variables: the evaluation satisfies ϕ if and only if for each clause there is a variable contained in the evaluation if and only if there is no clause contained in both sets.

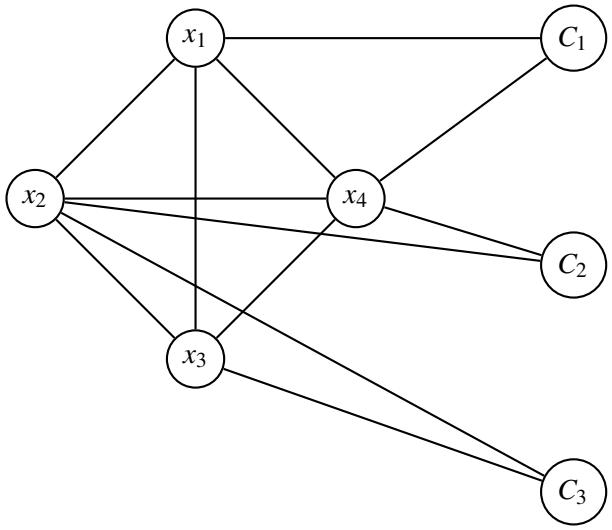


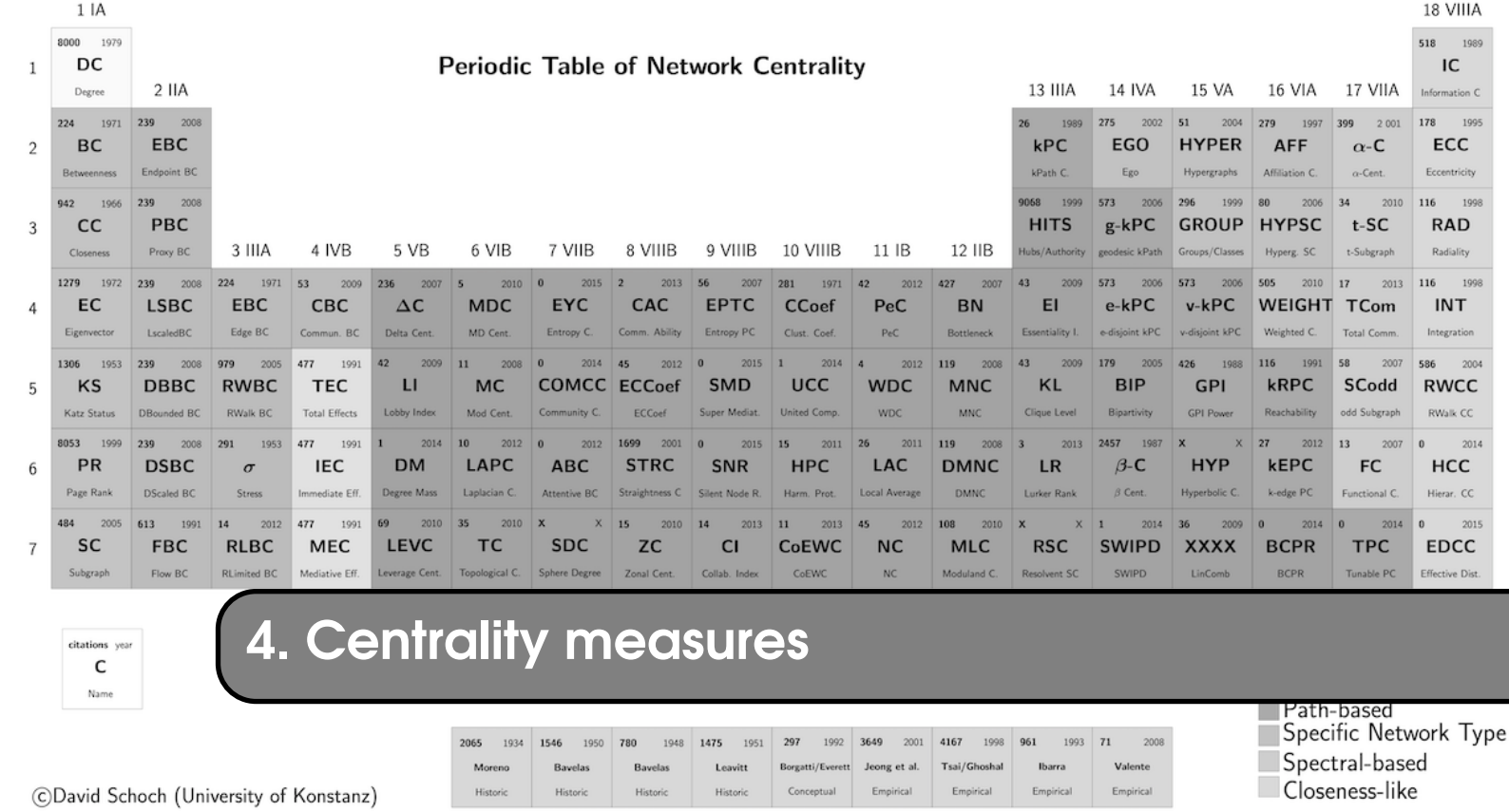
Figure 3.13: The reduction from disjoint sets to diameter computation. In this case, $C_1 = \{x_1, x_2\}$, $C_2 = \{x_2, x_4\}$, and $C_3 = \{x_3, x_4\}$. The diameter is 3 since C_1 and C_3 are disjoint.

3.4.3 From disjoint sets to diameter computation

Finally, we show that TDS is reducible (in linear time) to the problem of computing the diameter of a graph. Hence, this latter problem is not solvable in sub-quadratic time unless SETH is false [13].

Given an input $I = (X, \mathscr{C})$ of TDS, construct a graph $G = (X \cup \mathscr{C}, E)$, where each pair in X is connected, and for each set $C \in \mathscr{C}$ we add an edge from C to its elements (see Figure 3.4.3). Since each vertex is at distance 1 from X, the diameter is 2 or 3: it is 3 if and only if there exist two different vertices $C, C' \in \mathscr{C}$ with no common neighbor. It is clear that this happens if and only if C and C' are disjoint.

In conclusion, we have shown that it is likely that there is no algorithm computing the diameter of a graph in time sub-quadratic with respect to the number of nodes (note that, if the graph is sparse, this implies that it is unlikely that we can design an algorithm with complexity lower than $O(nm)$, where n is the number of nodes and m is the number of arcs). Using algorithms like iFUB or iDiFUB seems then the only available choice, when dealing with real-world graphs.



4. Centrality measures

Thus, the idea of centrality is alive and well and is being mobilized in an ever widening range of applications. Everyone agrees, it seems, that centrality is an important structural attribute of social networks. All concede that it is related to a high degree to other important group properties and processes. But there consensus ends.

L.C. Freeman, 1978

4.1 Centrality: a fuzzy notion

"Many kinds of theories have been developed to explain human behavior" [5]. So begins a quite celebrated article, appeared at the end of the forties, in which, among other things, the author initiated a line of research concerning the identification of the most important agents within a network of interactions. By referring to the study of pathways introduced ten years earlier and generally termed "hodology", he proposed to classify the nodes of a (undirected and unweighted) graph according to their eccentricities. In particular, a "central" node is a node with minimum eccentricity, while a peripheral node is a node at maximum distance from a central node. Starting from this paper, many definitions, modifications, and experiments were proposed in the next thirty years, and only at the end of the seventies a systematic organization of measures of centrality was proposed, in order to clarify and resolve some of the conceptual problems of centrality [42]. This classification made use of graph theory notions and techniques.

Capturing the notion of importance of an agent in a interaction network has always been a difficult task, mostly because of the "fuzziness" of this notion. However, when we refer to graphs,

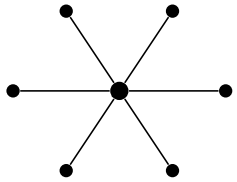


Figure 4.1: An example of a "star" graph where the bigger node at the center of the star seems to be very important: indeed, this node is the most central one with respect to several graph centrality measures, such as degree, eccentricity, closeness, betweenness.

some centrality measure seem to be able to identify the importance of a node within a graph. To explain this statement, let us consider the "star" graph shown in Figure 4.1. The bigger node at the center of the star seems to be very important: indeed, if we remove it, the graph is disconnected, and all the other nodes turn out to be isolated. This intuition is confirmed if we refer to the following four centrality measures (in the following, the indicated references are just ones of the first using the corresponding centrality measure and n will denote the number of nodes of the graph).

Degree centrality [81] In this case, the centrality $\delta(x)$ of a node x is equal to its degree $d(x)$ divided by the number of nodes minus 1, that is,

$$\delta(x) = \frac{d(x)}{n-1}.$$

The degree centrality of the node at the center of the star is $6/6 = 1$, while all other nodes have degree $1/6 \approx 0.17$.

Eccentricity centrality [5] In this case, the centrality $\varepsilon(x)$ of a node x is equal to the inverse of its eccentricity $e(x)$, that is,

$$\varepsilon(x) = \frac{1}{e(x)}.$$

The eccentricity centrality of the node at the center of the star is $1/1 = 1$, while all other nodes have degree $1/2 = 0.5$.

Closeness centrality [76] In this case, the centrality $\kappa(x)$ of a node x is equal to the inverse of its average distance to the other nodes, that is, if $\varphi(x) = \sum_{w \in V} d(x,w)$ denote the *farness* of the node x, then

$$\kappa(x) = \frac{1}{\frac{\varphi(x)}{n-1}} = \frac{n-1}{\varphi(x)}.$$

The closeness centrality of the node at the center of the star is $6/(6 \cdot 1) = 1$, while all other nodes have degree $6/(1+5 \cdot 2) = 6/11 \approx 0.55$.

Betweenness centrality [3, 41] In this case, the centrality $\beta(x)$ of a node x is equal to the fraction of shortest paths in the graph passing through x. Formally, let $\sigma_{u,w}$ denote the number of shortest paths from u to w, and let $\sigma_{u,w}(x)$ the number of shortest paths from u to w passing through x, with $u \neq w$ and $u \neq x$. Then,

$$\beta(x) = \frac{\sum_{u,w} \frac{\sigma_{u,w}(x)}{\sigma_{u,w}}}{(n-1)(n-2)}.$$

The betweenness centrality of the node at the center of the star is $(6 \cdot 5) \cdot 1/(6 \cdot 5) = 1$, while all other nodes have betweenness centrality equal to $6/30 = 0.2$.

Hence, with respect to all the four centrality measures defined above, the node at the center of the star is the one with the highest value. Note that the four centrality measures we have introduced are just a few examples of the "zoo" of centrality measures that have been introduced in the literature [77]. The periodic table of network centrality, which is shown below the title of this chapter, should give an idea of how many graph-based centrality measures have been defined in the last fifty years [78]. Dealing with all of them is clearly out of the scope of this book: indeed, we will refer only to the four measures previously defined.

Concerning the eccentricity, closeness, and betweenness centrality, we have to further specify how to deal with the case in which the graph is directed and not strongly connected (in the case in which the graph is undirected and not connected or directed and not weakly connected, the centrality measures can be computed in each connected component). In the case of the eccentricity centrality, we can just assume that if a node cannot reach any other node, then its centrality is zero. In the case of the betweenness centrality, if there is no path from a node u and a node w, we can assume that the ratio $\frac{\sigma_{u,w}(x)}{\sigma_{u,w}}$ is equal to zero, for any other node x. For what concerns the closeness centrality, two different approaches can be followed. The first one consists in generalizing the previous definition, by denoting as $\rho(x)$ the number of nodes reachable from x (including x itself). By using this notation, we can define

$$\kappa(v) = \frac{\rho(v)-1}{\varphi(v)}\frac{\rho(v)-1}{n-1} = \frac{(\rho(v)-1)^2}{(n-1)\varphi(v)}$$

(in other words, we restrict our attention to the set of nodes reachable from x, and we then normalize with respect to the total number of nodes). The other approach consists in "deleting" from the definition of the closeness centrality the contribution of the nodes which are not reachable from x. To this aim, we can define the "harmonic" version of the closeness centrality (simply called **harmonic centrality**), which is defined as follows:

$$\eta(x) = \sum_{w \in V-\{x\}} \frac{1}{d(x,w)}$$

(note that if w is not reachable from x, then $d(x,w) = \infty$, and hence $\frac{1}{d(x,w)} = 0$).

4.1.1 Are all the centrality measures the same?

A first question we can pose ourselves is whether the above defined centrality measures are related one to the other [79]. Indeed, this has been the subject of many papers, and the results are not always consistent one to the other (mostly because different data-sets are used). Just to give an example of this kind of results, in order to measure the relation between the different centrality measures, we can use the *Pearson correlation coefficient*, which is a standard measure of linear correlation between two sets of data [73]. Formally, given two sets of p values $X = x_1, \ldots, x_p$ and $Y = y_1, \ldots, y_p$, the Pearson correlation coefficient $r_{X,Y}$ is defined as

$$r_{X,Y} = \frac{\sum_{i=1}^{p}(x_i - \overline{X})\sum_{i=1}^{p}(y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{p}(x_i - \overline{X})^2}\sqrt{\sum_{i=1}^{p}(y_i - \overline{Y})^2}},$$

where $\overline{X}$ and $\overline{Y}$ denote the average value of the set X and Y, respectively (that is, $\overline{X} = \frac{\sum_{i=1}^{p} x_i}{p}$ and $\overline{Y} = \frac{\sum_{i=1}^{p} y_i}{p}$).

Let us consider the graph of the Florentine families introduced in the first chapter (see Figure 1.1). By using the `Graphs` package, the following function computes the degree, eccentricity, closeness, and betweenness centrality of all nodes in the graph (we will see in this chapter the

algorithms used by the package), and then computes, by using the `Statistics` package, the Pearson correlation coefficient, for each pair of centrality measures (note that, for each pair, we need to compute the correlation coefficient only once, since the coefficient is symmetric).

```
function correlations(graph::String)
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    d::Array{Float64} = degree_centrality(g)
    e::Array{Float64} = 1 ./ eccentricity(g)
    c::Array{Float64} = closeness_centrality(g)
    b::Array{Float64} = betweenness_centrality(g)
    return [[cor(d,e),cor(d,c),cor(d,b)],[cor(e,c),cor(e,b)],[cor(c,b)]]
end
```

Note that, in order to compute the eccentricity centrality, we have inverted all values contained in the result of the `eccentricity` function.

The results of the execution of the instruction `correlations("florence.lg")` is shown in the following table.

	δ	ε	κ	β
δ	1	0.49439	0.82451	0.84415
ε	0.49439	1	0.8299	0.57968
κ	0.82451	0.8299	1	0.80662
β	0.84415	0.57968	0.80662	1

We can conclude, by looking at the table, that all centrality measures are positively correlated one to the other. However, the degree centrality is more strongly correlated with the closeness and the betweenness centrality, while the eccentricity centrality is more strongly correlated with the closeness centrality. Finally, the closeness and the betweenness centrality seem to be quite strongly correlated.

4.1.2 Axioms for centrality

As we already said, many different definitions of centrality measures have proposed in the last fifty years. Besides analyzing their correlation, it could also be interesting to look for basic properties, also called *axioms*, that any graph-based centrality measure should satisfy. For example, consider the following *density axiom* [12].

Let K_q be clique with q nodes $x_1,\dots,x_q$, and let C_q be a directed cycle with q nodes $y_1,\dots,y_q$ (see Figure 4.2). Moreover, suppose that one node x in the clique is connected (via an undirected arc) to one node y of the cycle (note that this graph is strongly connected). Then, the centrality of x has to be greater than the centrality of y. Intuitively, this axiom says that a node participating to a very strong community (in terms of density) has to be more important than a node participating to a much weaker community (still in terms of density).

This axiom is clearly satisfied by the degree centrality, if we assume that $q > 3$. Indeed, $d(x) = q-1 > 2 = d(y)$ (if we consider the out-degree of y). Let us now compute the eccentricity of x and y. The node x has $q-1$ nodes at distance 1, and, for each i with $1 \leq i \leq q$, has one node at distance i. Hence, its eccentricity is q. On the other hand, y has $k-1$ nodes at distance 2 and one at distance 1, and, for each i with $1 \leq i \leq q-1$, has node at distance i. Hence, its eccentricity is $q-1$. Hence, the eccentricity centrality of x, that is, $1/q$, is smaller than the eccentricity centrality of y, that is, $1/(q-1)$: this implies that the eccentricity centrality does not satisfy the density axiom. Moreover, we have that

$$\varphi(x) = (q-1) + \sum_{i=1}^{q} i = (q-1) + \frac{q(q+1)}{2},$$

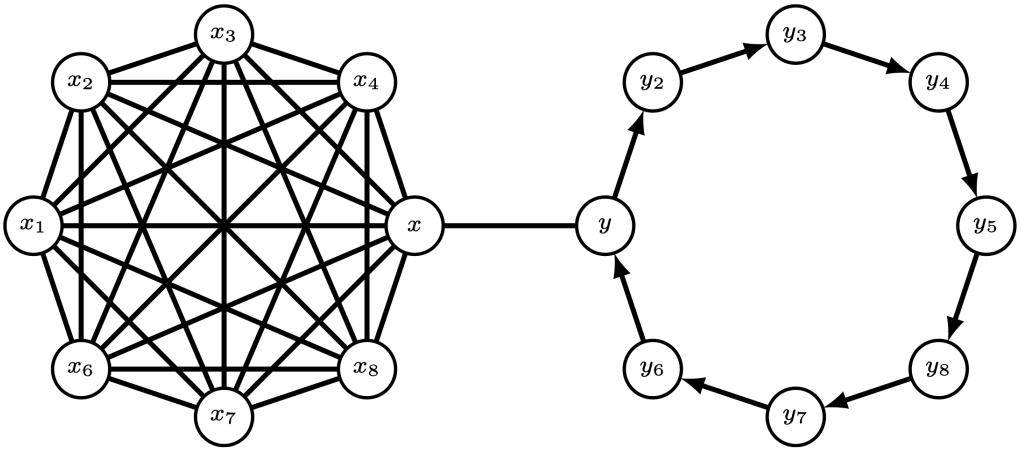


Figure 4.2: The density axiom for centrality measures. A centrality measure satisfies this axiom if the centrality of x is greater than the centrality of y.

and that

$$\varphi(y) = 2(q-1) + 1 + \sum_{i=1}^{q-1} i = 2(q-1) + 1 + \frac{q(q-1)}{2}.$$

Since

$$(q-1) + \frac{q(q+1)}{2} = (q-1) + q + \frac{q(q-1)}{2} = 2(q-1) + 1 + \frac{q(q-1)}{2},$$

we have that $\varphi(x) = \varphi(y)$. That is, the density axiom is not satisfied by the closeness centrality. Finally, for what concerns the betweenness centrality, we have that x is in all shortest path from a node of the clique to a node of the cycle, and vice versa. Hence, its betweenness is equal to $\frac{2(q-1)q}{(2q-1)(2q-2)}$. On the other hand, y is also on the same shortest paths, but it is also on any shortest path from a node u of the cycle to another node of the cycle which is between y and u. Then its betweenness is equal to $\frac{2(q-1)q+(q-1)(q-2)}{(2q-1)(2q-2)}$, which is greater than the betweenness centrality of x. Hence, the density axiom is not satisfied by the betweenness centrality. Finally, let us consider the harmonic centrality. In the case of x, because of what we have said about its distance from the other nodes, we have that $\eta(x) = (q-1) + \sum_{i=1}^{p} \frac{1}{i}$. On the other hand, $\eta(y) = \frac{(q-1)}{2} + 1 + \sum_{i=1}^{p-1} \frac{1}{i}$. Since, for $q > 3$, $(q-1) > \frac{(q-1)}{2} + 1$, we have that the harmonic centrality satisfies the density axiom.

Other axioms have been proposed in the literature. Nevertheless, it is not easy to decide which are the "correct" axioms that any centrality measure should satisfy. In conclusion, choosing the right measure of importance is a quite complicated task, which in most cases depends on the application domain. However, closeness and betweenness centrality are certainly two of the oldest and of the most widely used: almost all books dealing with graph mining discuss them, and almost all existing graph mining libraries implement algorithms to compute them. For this reason, in this chapter we will examine algorithms for computing these two centrality measures, by restricting ourselves to unweighted graphs.

4.2 Computing the betweenness centrality

A naive algorithm for computing the betweenness centrality can proceed as follows.

1. For every node v, set $\beta(v) = 0$.
2. For every node s

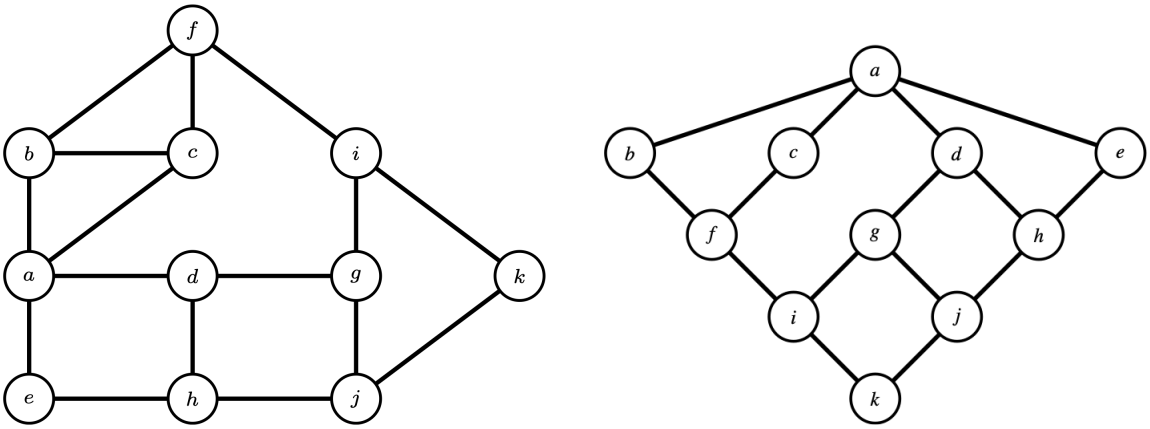


Figure 4.3: The level partition (right) obtained by executing the augmented BFS on the graph on the left. Differently from the BFS, the augmented BFS keeps trace of all the predecessors in any shortest path from the root.

(a) Perform a BFS starting from s, in order to find all shortest paths from s to all other nodes t: let $\Sigma_{s,t}$ be the set of these paths.
(b) For every node v and for each pair s,t, count the number of times v appears in a path in $\Sigma_{s,t}$, divide this value by $|\Sigma_{s,t}|$, and add it to $\beta(v)$.

3. For every node v, return $\frac{\beta(v)}{(n-1)(n-2)}$.

The computation of $\Sigma_{s,t}$, for each node $t \neq s$, can be realized by making use of a sort of *augmented BFS* starting from s. During this augmented BFS, each node y maintains a list $\Sigma_{s,y}$ of all the shortest path from the starting node s to y. Each time a node y is inserted into the queue, its list $\Sigma_{s,y}$ is set equal to the list $\Sigma_{s,x}$, where x is the node that inserted y in the queue, and, for each path in $\Sigma_{s,y}$, the node y is added at the end of the path. Successively, each time a node x has a neighbor y visited but not explored, the paths in $\Sigma_{s,x}$ are added to $\Sigma_{s,y}$ after adding to each path the node y. In this way, at the end of the augmented BFS each node t has computed and stored the correct list $\Sigma_{s,t}$ of all the shortest paths from s to t.

Let us see an example of execution of the augmented BFS on the graph shown in the left part of Figure 4.3. The augmented partition in levels computed by the augmented BFS starting from node a is show in the right part of the figure. In the figure, for each node t at level i, t is connected to all its predecessors in any shortest path from the root. Indeed, this connection indicates that t computes the list $\Sigma_{a,t}$ by joining the lists of its predecessors.

At the beginning, $\Sigma_{a,a} = \{(a)\}$. When nodes b, c, d, and e are inserted in the queue by node a, their lists become $\Sigma_{a,b} = \{(a,b)\}$, $\Sigma_{a,c} = \{(a,c)\}$, $\Sigma_{a,d} = \{(a,d)\}$, and $\Sigma_{a,e} = \{(a,e)\}$. When node f is inserted in the queue by node b, its list become $\Sigma_{a,f} = \{(a,b,f)\}$. Successively, node c discovers that its neighbor f has not been explored yet, and the list of f becomes $\Sigma_{a,f} = \{(a,b,f),(a,c,f)\}$. Analogously, $\Sigma_{a,g} = \{(a,d,g)\}$ (since g is the only predecessor of g), and $\Sigma_{a,h} = \{(a,d,h),(a,e,h)\}$. When node i is inserted in the queue by node f, its list become $\Sigma_{a,i} = \{(a,b,f,i),(a,c,f,i)\}$. Successively, node g discovers that its neighbor i has not been explored yet, and the list of i becomes $\Sigma_{a,i} = \{(a,b,f,i),(a,c,f,i),(a,d,g)\}$. Analogously, $\Sigma_{a,j} = \{(a,d,g),(a,d,h),(a,e,h)\}$. When node k is inserted in the queue by node i, its list become $\Sigma_{a,k} = \{(a,b,f,i,k),(a,c,f,i,k),(a,d,g,k)\}$. Finally, node j discovers that k has not been explored yet, and the list of k becomes $\Sigma_{a,k} = \{(a,b,f,i,k),(a,c,f,i,k),(a,d,g,k),(a,d,g,k),(a,d,h,k),(a,e,h,k)\}$.

Once we have computed all the lists of shortest paths,for each node u and for any other node v, we can compute how many times u appears in $\Sigma_{a,v}$. For example, the node d can verify that it appears in three shortest paths from a to k. Since the number of such shortest paths (that is,

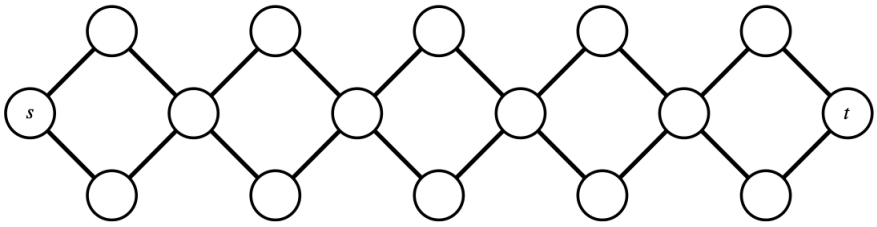


Figure 4.4: An example of a graph in which the number of shortest paths from the node s to the node t is exponential in the number of nodes. Indeed, at any crossroad a shortest path can decide to go either up or down. The number of shortest paths from s to t is 32.

the cardinality of $\Sigma_{a,k}$ is equal to 6, we can add to $\beta(d)$ the value $3/6 \approx 0.5$. We can do this computation for any two nodes u and v, and we can repeat the entire process for each starting node of the augmented BFS. At the end, we have computed the "unnormalized" betweenness of each node, and it suffices to divide it by $(n-1)(n-2)$.

The above algorithm is quite expensive in terms of space consumed. Indeed, it has to store, for any two nodes s and t all the shortest paths from s to t. In the worst case, this set can be quite large. Consider, for example, the graph in Figure 4.4. At any crossroad of this graph, the shortest path can decide to go either up or down. Hence, the number of shortest paths from s to t is 32. In general, if n is the number of nodes of the graph, this kind of graphs have $2^{(n+2)/3}$ shortest paths, that is, a number of shortest paths which is exponential in the number of nodes.

4.2.1 The Brandes algorithm

The *Brandes algorithm* is based on the naive algorithm but it avoids storing all the shortest paths, by adding to the augmented BFS, a "bottom-up" visit of the partition in levels generated by the augmented BFS [15]. This second phase of the algorithm, which uses the output of the augmented BFS, is usually called the *accumulation phase*.

The algorithm proceeds as follows.

1. For every node v, set $\beta(v) = 0$.
2. For every node s
   (a) For every node v, set $\delta_s(v) = 0$.
   (b) Perform a BFS starting from s, in order to *compute the number* of shortest paths from s to all other nodes t, that is, $\sigma_{s,t}$.
   (c) Going back to s, increment $\delta_s(v)$ *as appropriate* when node v is reached by using $\sigma_{s,\bullet}$-values.
   (d) Add $\delta_s(v)$ to $\beta(v)$.
3. For every node v, return $\frac{\beta(v)}{(n-1)(n-2)}$.

For what concerns the computation of the number of shortest paths, we can proceed as before. This time, however, instead of maintaining a list of shortest paths, we maintain only the number of shortest paths, which is initialized the first time a node is inserted in the queue, and updated each time a node has a neighbor which has been visited but not yet explored. In particular, each time a node y is inserted into the queue, its $\sigma_{s,y}$ is set equal to $\sigma_{s,x}$, where x is the node that inserted y in the queue. Successively, each time a node x has a neighbor y visited but not explored, the value $\sigma_{s,x}$ is added to $\sigma_{s,y}$. In this way, at the end of the augmented BFS each node t has computed and stored the correct number $\sigma_{s,t}$ of shortest paths from s to t.

By still referring to Figure 4.3, we have that, at the beginning, $\sigma_{a,a} = \{1\}$. When nodes b, c, d, and e are inserted in the queue by node a, we $\sigma_{a,b} = 1$, $\sigma_{a,c} = 1$, $\sigma_{a,d} = 1$, and $\sigma_{a,e} = 1$. When node f is inserted in the queue by node b, we set $\sigma_{a,f} = 1$. Successively, node c discovers that its neighbor f has not been explored yet, and the value of f becomes $\sigma_{a,f} = 2$. Analogously, $\sigma_{a,g} = 1$

(since g is the only predecessor of g), and $\sigma_{a,h} = 2$. When node i is inserted in the queue by node f, we set $\sigma_{a,i} = 2$. Successively, node g discovers that its neighbor i has not been explored yet, and the value of i becomes $\sigma_{a,i} = 3$. Analogously, $\sigma_{a,j} = 3$. When node k is inserted in the queue by node i, we set $\sigma_{a,k} = 3$. Finally, node j discovers that k has not been explored yet, and the value of k becomes $\sigma_{a,k} = 6$.

In the accumulation phase, we compute the contributions to the betweenness centrality of each node due to the source node s. Let us define the *pair-wise dependency*

$$\delta_{s,t}(v) = \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}.$$

Note that $\beta(v) = \frac{\sum_{s,t} \delta_{s,t}(v)}{(n-1)(n-2)}$. Moreover, let us define the *one-sided dependency*

$$\delta_s(v) = \sum_t \delta_{s,t}(v).$$

Hence, $\beta(v) = \frac{\sum_s \delta_s(v)}{(n-1)(n-2)}$. The basic idea of the accumulation phase is that $\delta_s(v)$ can be computed recursively by using $\delta_s(w)$ for all w immediately following v on some shortest path from s. Let $\mathrm{PH}_s(v)$ be the set of these nodes w.

In order to prove the main lemma showing how to compute $\delta_s(v)$, let us first state the following simple results (whose "graphic" proof is given in Figure 4.5).

Fact 1. If $\sigma_{s,t}(v) \neq 0$ then $\sigma_{s,t}(v) = \sigma_{s,v}\sigma_{v,t}$.

Fact 2. If the arc (u,v) belongs to a shortest path from s to t, then $\sigma_{s,t}(u,v) = \sigma_{s,u}\sigma_{v,t}$, where $\sigma_{s,t}(u,v)$ denotes the number of shortest paths from s to t passing through edge (u,v).

Fact 3. If $\sigma_{s,t}(v) \neq 0$ then $\sigma_{v,t} = \sum_{w \in \mathrm{PH}_s(v)} \sigma_{v,t}(v,w)$.

Proposition 4.2.1 For every $s \neq v$

$$\delta_s(v) = 1 + \sigma_{s,v} \sum_{u \in \mathrm{PH}_s(v)} \frac{\delta_s(u)}{\sigma_{s,u}}.$$

First, note that if $t = v$, then the contribution of t to $\delta_s(v)$ is equal to $\delta_{s,t}(t) = \frac{\sigma_{s,t}(t)}{\sigma_{s,t}} = 1$. Let T_u

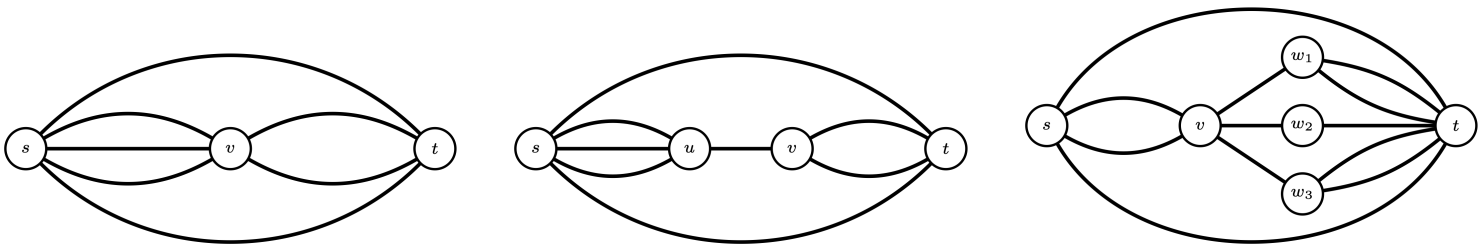


Figure 4.5: The "graphic" proof of Fact 1, 2, and 3 in the design of the Brandes algorithm.

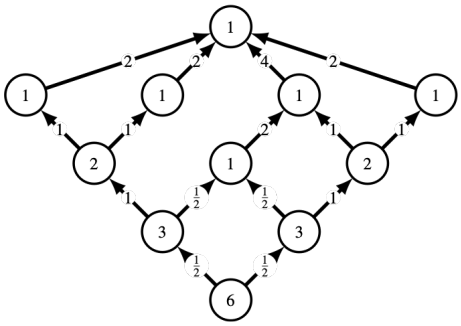


Figure 4.6: The accumulation phase of the Brandes algorithm. The node at the last level has a budget equal to 1, while all other nodes have a budget which is equal to 1 plus the contribution it receives by its successors. Each node distributed its budget to its predecessors proportionally to how many shortest paths to u pass thorough the predecessor (the number shortest paths is shown within the nodes).

be set of $t \neq v$ such that $\sigma_{s,t}(v,u) \neq 0$ and let $T = \bigcup_{u \in \mathrm{PH}_s(v)} T_u$. Then

$$\begin{aligned}
\delta_s(v) &= 1 + \sum_{t \in T} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} = 1 + \sum_{t \in T} \frac{\sigma_{s,v}\sigma_{v,t}}{\sigma_{s,t}} = 1 + \sigma_{s,v} \sum_{t \in T} \sum_{u \in \mathrm{PH}_s(v)} \frac{\sigma_{v,t}(v,u)}{\sigma_{s,t}} \\
&= 1 + \sigma_{s,v} \sum_{u \in \mathrm{PH}_s(v)} \sum_{t \in T_u} \frac{\sigma_{v,t}(v,u)}{\sigma_{s,t}} = 1 + \sigma_{s,v} \sum_{u \in \mathrm{PH}_s(v)} \sum_{t \in T_u} \frac{\sigma_{u,t}}{\sigma_{s,t}} \\
&= 1 + \sigma_{s,v} \sum_{u \in \mathrm{PH}_s(v)} \frac{1}{\sigma_{s,u}} \sum_{t \in T_u} \frac{\sigma_{s,u}\sigma_{u,t}}{\sigma_{s,t}} = 1 + \sigma_{s,v} \sum_{u \in \mathrm{PH}_s(v)} \frac{\delta_s(u)}{\sigma_{s,u}}.
\end{aligned}$$

The proposition has thus been proved.

The above proposition thus suggests how to visit the partition in levels produced by the augmented BFS in order to compute the value $\delta_s(v)$, for each node v in the graph. This value is equal to 1 for the nodes at the last level. Subsequently, each node u contributes to the value of one of its predecessor v in a way proportional to the number of shortest paths from s to v among all the shortest paths from s to u. In a certain sense, we can see this as a distribution of the value $\delta_s(u)$ to its predecessors proportional to how many shortest paths to u pass thorough the predecessor.

By still referring to Figure 4.3, let us see how the accumulation phase executes. In Figure 4.6, in each node we show the number of shortest paths from the source node computed during the augmented BFS. At the beginning, the node at the last level has to distribute its value, which is one, to its predecessors. Since the two predecessor contribute equally to the number of shortest paths, the node distributes $1/2$ to both the predecessors. The first node at the level before the last one receives $1/2$ and adds 1 to this value. So, it has to distribute $3/2$ to its predecessors: since the predecessor on the left contributes with 2 shortest paths, while the other predecessor contributes with 1 shortest path, the first predecessor receives $\frac{3}{2}\frac{2}{3} = 1$, while the second one receives $\frac{3}{2}\frac{1}{3} = 1/2$. We can continue like this until reaching the source node, who receives a value which has to be to the number of nodes minus 1.

Note that the time and the space complexity of the Brandes algorithm for computing the value $\delta_s(v)$, for each node v, is proportional to m, where m denotes the number of arcs in the graph. Hence, in order to compute the betweenness of all nodes, the Brandes algorithm requires time proportional to nm and space linear in the number of arcs. Once again, this time complexity is not affordable whenever the graph is very large (you can verify this statement, by executing the

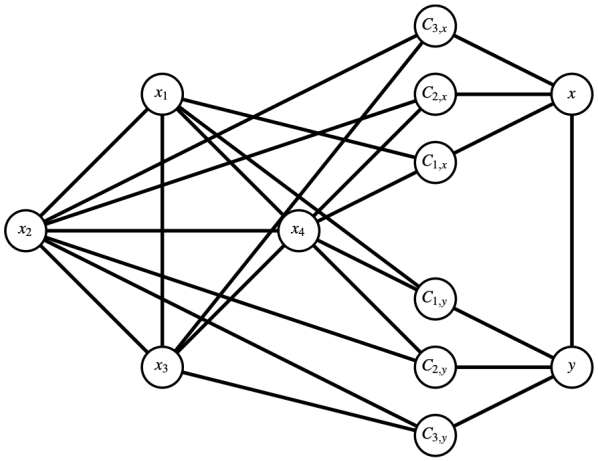


Figure 4.7: The reduction from disjoint sets to betweenness computation. In this case, $C_1 = \{x_1, x_2\}$, $C_2 = \{x_2, x_4\}$, and $C_3 = \{x_3, x_4\}$. The betweenness of node x is bigger than $6/110$ since C_1 and C_3 are disjoint.

function `betweenness_centrality` of the `Graphs` package with input any of the most recent IMDB graphs). For this reason, several approximation algorithms have been proposed, based on the sampling technique (either choosing random nodes or random shortest paths). However, the design and analysis of these algorithms are out of the scope of this book, since they require quite sophisticated probabilistic analysis techniques.

4.2.2 Hardness result

Even in the case of the computation of the betweenness centrality, we can show that it is unlikely that we can find an algorithm better than the Brandes algorithm. Indeed, we can reduce (in linear time) the Two Disjoint Set (in short, TDS) problem, introduced in the previous chapter, to the problem of computing the betweenness centrality of a specific node b [13]. This implies that even computing the betweenness of a single node cannot be done in sub-quadratic time (unless SETH is false).

Given an instance $I = (X, \mathscr{C})$ of TDS, construct a graph $G = (V, E)$ where (see Figure 4.7):

- $V = \{y\} \cup \{x\} \cup \mathscr{C}_x \cup X \cup \mathscr{C}_y$, where $\mathscr{C}_x, \mathscr{C}_y$ are two identical copies of $\mathscr{C}$;
- all pairs of nodes in X are connected;
- node x is connected to y and to each node in $\mathscr{C}_x$;
- node y is connected to x and to each node in $\mathscr{C}_y$;
- connections between $\mathscr{C}_x$ and X and connections between $\mathscr{C}_y$ and X are made according to the membership relation.

The input node b of our problem is x. First note that no shortest path can be longer than 3. This proves that any shortest path in the sum passing through x must be of one of the following forms:

- a path from y to $\mathscr{C}_x$;
- a path from $\mathscr{C}_x$ to $\mathscr{C}_y$;
- a path from y to X.

We note that the third case never occurs, because there always exists a path from y to any node in X of length 2. The first case occurs for each node in $\mathscr{C}_x$, and no other shortest path exists from y to nodes in $\mathscr{C}_x$: these nodes contribute to the sum by $|\mathscr{C}|$. Finally, the second case occurs if and only if there is a pair of nodes in $\mathscr{C}_y$ and $\mathscr{C}_x$ having no path of length 2, that is, two disjoint sets in $\mathscr{C}$.

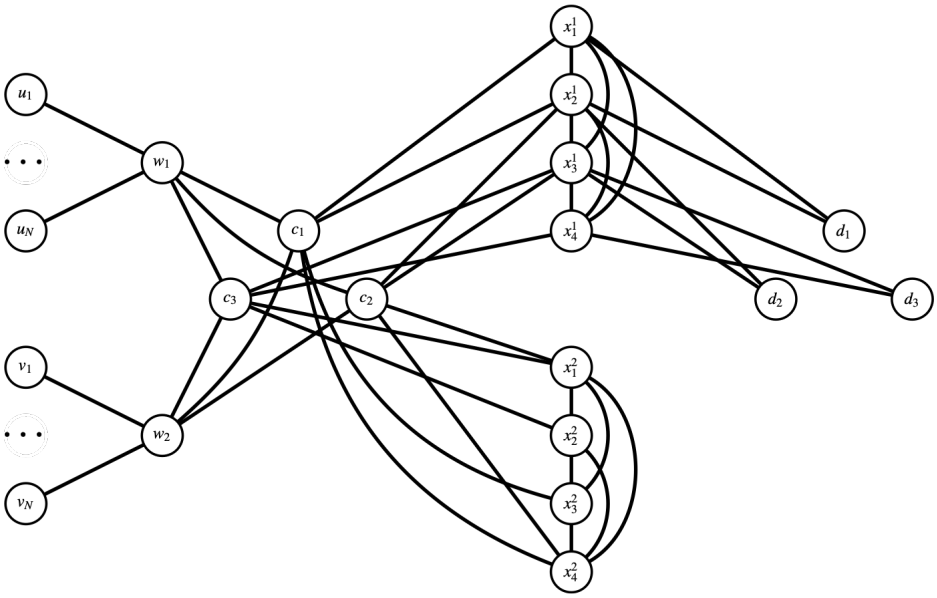


Figure 4.8: The reduction from hitting set to closeness computation. In this case, $C_1 = \{x_1, x_2\}$, $C_2 = \{x_2, x_3\}$, and $C_3 = \{x_3, x_4\}$. The farness of node c_2 is equal to $4N + 12 + 12$ since C_2 is a hitting set.

Hence, the betweenness of x is equal to

$$\beta(x) = \frac{|\mathscr{C}| + \frac{1}{2}|\{(C_i, C_j) : C_i \cap C_j = \emptyset\}|}{(|X| + |\mathscr{C}| - 1)(|X| + |\mathscr{C}| - 2)}.$$

That is, $\beta(x)$ is bigger than $\frac{|\mathscr{C}|}{(|X|+|\mathscr{C}|-1)(|X|+|\mathscr{C}|-2)}$ if and only if there are two disjoint sets in $\mathscr{C}$.

4.3 Computing the closeness centrality

Computing the closeness value for each node v can be easily done by executing a BFS starting from v. However, if we want to compute the closeness value of all nodes of the graph, then the time complexity would be proportional to nm: as usual, this time complexity is not affordable whenever we deal with very large graphs.

4.3.1 Hardness result

Also in the case of the closeness centrality, we can prove that it is unlikely that a sub-quadratic algorithm exists for computing this measure. In particular, we can now show that computing the minimum farness among all the nodes of the graph cannot be done in sub-quadratic time, unless the following problem is solvable in sub-quadratic time. The k-Hitting Set (in short, k-HS) is defined as follows: given a set X and a collection $\mathscr{C}$ of subsets of X such that $|X| \leq \log^k(|\mathscr{C}|)$, the solution is 1 if there exists one sets $C \in \mathscr{C}$ which intersects all the other sets, that is, such that, for $C' \in \mathscr{C}$, $C \cap C' \neq \emptyset$. Clearly, this problem can be solved in quadratic time. It is also conjectured that it cannot be solved in time proportional to $n^{2-\varepsilon}$, for any $\varepsilon > 0$.

Let us show that k-HS is reducible in (almost) linear time to the problem of finding the node of a graph with the smallest farness (and, hence, with the largest closeness centrality). Given an instance $I = (X, \mathscr{C})$ of k-HS, construct a graph $G = (V, E)$ where (see Figure 4.8):

- $V = U_N \cup V_N \cup \{w_1, w_2\} \cup \{x\} \cup C \cup X^1 \cup X^2 \cup D$, where U_N and V_N are two independent sets of N nodes, D, C are two independent sets of $|\mathscr{C}|$, and X^1 and X^2 are two cliques of $|X|$ nodes;

- all nodes in U_N are connected to w_1;
- all nodes in V_N are connected to w_2;
- all nodes in C are connected to w_1 and w_2;
- connections between C and X^1 are made according to the membership relation;
- connections between C and X^2 are made according to the non-membership relation;
- connections between X^1 and D are made according to the membership relation.

Let us compute the farness of a node c in C (note that the maximum closeness centrality corresponds to the minimum farness, since the graph is connected). All nodes in U_N and V_N are at distance 2 from c, w_1 and w_2 are at distance 1, all other nodes in C are at distance 2, a node $x \in X^1$ (respectively, X^2) is at distance 1 if $x \in c$ (respectively, $x \notin c$), otherwise it is at distance 2, and a node $d \in D$ is at distance 2 if c and d intersects, otherwise it is at distance 3. Hence,

$$\varphi(c) = 4N + 2 + 2(|\mathscr{C} - 1) + 3|X| + 2|\mathscr{C}| + |\{d \in \mathscr{C} : c \cap d = \emptyset\}|,$$

and we have that the farness of c is equal to $4N + 4|\mathscr{C}| + 3|X|$ if and only if c is a hitting set. It is easy to verify that all other nodes (for N sufficiently large) have a bigger farness. We can conclude that the top node with respect to the closeness centrality has farness equal to $4N + 4|\mathscr{C}| + 3|X|$ if and only if there exits a hitting set in $\mathscr{C}$.

4.3.2 Approximation through sampling

Due the above hardness result, approximation probabilistic algorithms have been proposed. Indeed, a simple algorithm for computing the closeness centrality in undirected unweighted graphs is based on random sampling [37]. This algorithm performs k BFSs from k random nodes $v_1, \ldots, v_k$ and, for any node u, return

$$\hat{\kappa}(u) = \frac{k}{n} \frac{n-1}{\sum_{i=1}^{k} d(v_i, u)}.$$

It is not difficult to prove (similarly to what we have done in the case of the distance distribution function) that, for any $t > 0$, if $k = \Theta\left(\frac{\log n}{t^2}\right)$, with high probability $\left|\frac{1}{\hat{\kappa}(u)} - \frac{1}{\kappa(u)}\right| < tD$, where D denotes the diameter of the graph. Indeed, since the k nodes in the sample are chosen uniformly at random with repetitions, we have that

$$\begin{aligned}
\mathbb{E}[1/\hat{\kappa}(u)] &= \frac{n}{n-1}\frac{1}{k}\mathbb{E}\left[\sum_{i=1}^{k} d(v_i, u)\right] = \frac{n}{n-1}\frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[d(v_i, u)] = \frac{n}{n-1}\frac{1}{k}\sum_{i=1}^{k}\sum_{v \in V}\frac{1}{n} d(v, u) \\
&= \frac{n}{n-1}\frac{1}{k}\frac{1}{n}\sum_{i=1}^{k}\sum_{v \in V} d(v, u) = \frac{n}{n-1}\frac{1}{k}\frac{1}{n}\sum_{i=1}^{k}\varphi(u) = \frac{n}{n-1}\frac{1}{k}\frac{1}{n} k\varphi(u) = 1/\kappa(u),
\end{aligned}$$

that is, $1/\hat{\kappa}(u)$ is an unbiased estimator of the inverse of the closeness centrality of the node u. Moreover, note that $0 \leq \frac{n}{n-1}\frac{d(v_i,u)}{k} \leq \frac{nD}{k(n-1)}$. We can then apply the Hoeffding bound (that is, Theorem 2.4.1) with $X_i = \frac{n}{n-1}\frac{d(v_i,u)}{k}$, $a_i = 0$, and $b_i = \frac{nD}{k(n-1)}$ in order to obtain the result.

The following function implements an algorithm for computing an approximation of the closeness centrality of all the nodes of a graph, based on the sampling technique and justified by the above result. In the case of the graph of the Florentine families (which has 15 nodes), we can choose $k = 3 \approx \log 15$.

```
function apx_closeness(graph::String, k::Int64)::Array{Float64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    farness::Array{Float64} = zeros(Float64, nv(g))
    for _ in 1:k
        farness = farness + gdistances(g, rand(1:nv(g)))
```

```
    end
    return (k * (nv(g) - 1)) ./ (nv(g) .* farness)
end
```

Even if the theoretical results states that we can bound the absolute error by randomly choosing a logarithmic number of nodes, one problem with this algorithm is that it does not guarantee that it preserves the order of the nodes according to their closeness centrality. For instance, let us consider the graph of the Florentine families, and let us choose $k = 3$. Since the graph is very small, we can execute the closeness approximation algorithm for any sample with three nodes, and we can check which node is the top one (breaking ties arbitrarily) for any sample. The following sequence shows the number of times each node has been the top one: 43, 409, 217, 229, 181, 43, 445, 43, 1015, 43, 79, 199, 121, 163, 145. We can see that the node 9 is the one which is the top one the most frequently: this is quite reasonable since this node is, indeed, the top one with respect to the closeness centrality. However, there are other nodes which appear quite frequently as the top one, such as the nodes 2 and 7: in this case, the algorithm would not be able to identify the correct top node.

An example: brain networks

Mapping the human brain is one of the great scientific challenges of this century. The *Human Connectome Project* charts the neural pathways that underlie brain function and behavior in more than one thousand healthy young adults [86]. Starting from the data of this project, a repository providing brain networks of one hundred unrelated young adults, formed from their resting state functional magnetic resonance imaging, has been made publicly available [82, 83]. Each graph has 180 nodes (corresponding to cortical regions), and each node is an identified brain region with its area description properly known. In this section we are going to use one of these graphs, which contains 4284 arcs.

First of all, by using the function `closeness_centrality` of the `Graphs` package, we can compute the closeness centrality of all nodes, and realize that the top node is node 48. If we choose an error factor $t = 0.3$, we can set $k = \frac{\log 180}{0.09} \leq 12 \log 180 \approx 84$. That is, we are going to take a sample which is a little bit less than half the number of nodes. We execute the previous code with this value of k and with input the graph `connectome.lg` one thousand times, and we check how many times the node 48 appears among the top five in the ranking of the nodes based on the approximate closeness centrality values. More or less, we have that this happens only 600 times, that is, in 40% of the experiments, the top node does not even appear among the first five nodes. We will go back to this problem at the end of the chapter.

4.3.3 Finding the top-k nodes

We then focus on the problem of computing only the k most important nodes with respect to the closeness centrality (as we said, computing an approximation of the closeness values does not guarantee that we can determine the top-k nodes). Indeed, determining such nodes is maybe even most interesting than computing the centrality values of all nodes. In the following, we will focus on the case $k = 1$, that is, the problem of finding the node with the largest closeness. The algorithm we are going to describe [6], however, can be easily adapted in order to deal with the case $k > 1$.

Once again, the simplest algorithm for computing the top node with largest closeness performs a BFS from each node v, computes its closeness $\kappa(v)$, and, finally, returns the node with biggest $\kappa(v)$ values. In order to reduce the time complexity of this algorithm (at least, in practice), a modification of this algorithm sets $\kappa(v)$ equal to the result of a *pruned* BFS, which receives in input the starting node v and a value κ_{top}, which is the biggest closeness centrality value found until now ($\kappa_{\text{top}} = 0$ if we have not processed at least one node). If this pruned BFS returns the value 0, it means that v is not the most central node, otherwise the returned value $\kappa(v)$ is the actual closeness

centrality of v, which is also the new biggest closeness centrality value (that is, v is the current most central node). In order to speed-up the pruned BFS, we want κ_{top} to be as big as possible, and consequently we need to process central nodes as soon as possible. To this purpose, we can decide to process nodes in decreasing order of degree.

This algorithms requires time proportional to $n \log n$ to sort the nodes according to their degree, and, in the worst case, time proportional to m for each partial BFS executed. The total running time, however, is proportional to $n \log n + T$, where T is the time needed to perform the pruned BFS n times (this latter time can be substantially less than nm).

The pruned BFS

The pruning of a BFS started from node v makes use of an upper bound $\tilde{\kappa}_d(v)$ on the closeness of v, which has to be updated whenever, for any $d \geq 0$, the exploration of the d-th level of the BFS tree is finished (see Figure 4.3.3). More precisely, this upper bound is obtained by proving a lower bound on the farness of v (recall that, in the case of connected graphs, $\kappa(v) = (n-1)/\varphi(v)$, and, hence, a lower bound on $\varphi(v)$ implies an upper bound on $\kappa(v)$).

If $L_d(v)$ denotes the nodes at level d of the BFS started from v, if $l_d(v) = |L_d(v)|$, and if $n_d(v)$ then

$$\varphi(v) \geq \varphi_d(v) + (d+1)l_{d+1}(v) + (d+2)(n - n_{d+1}(v)),$$

where

$$\varphi_d(v) = \sum_{i=1}^{d} i \cdot l_i(v) \qquad \text{and} \qquad n_d(v) = \sum_{i=0}^{d} l_i(v).$$

Indeed, $\varphi_d(v)$ denotes the farness at level d, that is, the sum of the distances from v to all the nodes in the first d levels. Moreover, $(d+1)l_{d+1}(v)$ is equal to the contribution to the farness of v of the nodes at level $d+1$. Finally, all the remaining nodes (that is, the nodes below the level $d+1$) contribute to the farness of v by a distance which is at least $d+2$.

Since $n_{d+1}(v) = l_{d+1}(v) + n_d(v)$, we have that

$$\varphi(v) \geq \varphi_d(v) - l_{d+1}(v) + (d+2)(n - n_d(v)).$$

At the end of the exploration of the d-th level of the BFS tree, we don't know yet the value of $l_{d+1}(v)$. However, we can certainly say that this value is not greater than the sum of the degrees of all nodes at level d minus 1 (since one arc has already been used), that is,

$$l_{d+1}(v) \leq \sum_{u \in \Gamma_d(v)} (d(v) - 1).$$

Let us denote by $n_d(v)$ the above sume. Hence,

$$\varphi(v) \geq \varphi_d(v) - \tilde{l}_{d+1}(v) + (d+2)(n - n_d(v)).$$

The lower bound on the farness of v is then defined as $\tilde{\varphi}_d(v) = \varphi_d(v) - \tilde{l}_{d+1}(v) + (d+2)(n - n_d(v))$. This lower bound on the farness implies the following upper bound $\tilde{\kappa}_d(v)$ on the closeness:

$$\tilde{\kappa}_d(v) = \frac{(n-1)}{\tilde{\varphi}_d(v)} \geq \kappa(v).$$

The pruned BFS has, hence, two inputs: the starting node v and the value κ_{top} which is the biggest closeness centrality value found until now. At the end of the exploration of the level d of the BFS, the upper bound $\tilde{\kappa}_d(v)$ is computed and compared to κ_{top}. If $\kappa_{\text{top}} > \tilde{\kappa}_d(v) \geq \kappa(v)$, the BFS

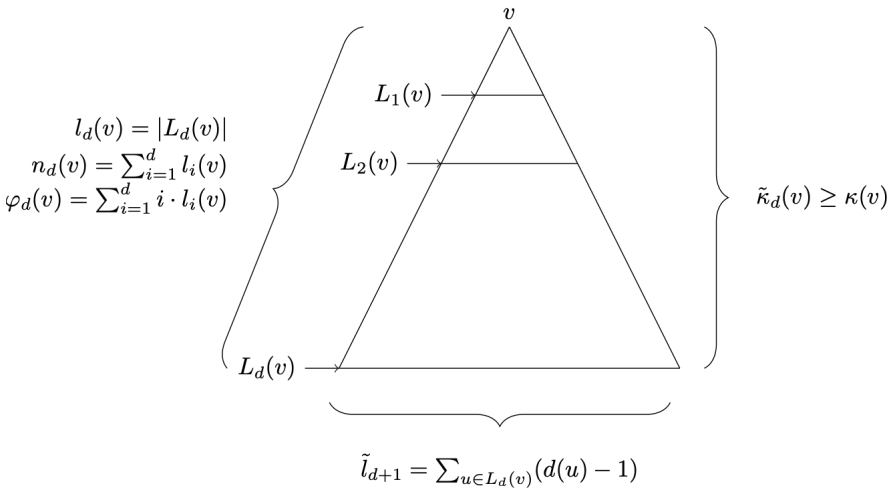


Figure 4.9: After the exploration of the first d levels of the BFS started from node v, an upper bound on the closeness value of v can be computed.

is interrupted, since for sure the node v is not among the top node. Otherwise, the BFS continues with the exploration of the next level. Note that all the information necessary to compute the upper bound $\tilde{\kappa}_d(v)$ is available at the end of the exploration of the level d of the BFS. Note also that, if the graph is undirected, during a linear time pre-processing phase we can compute the connected components of the graph, and, for each component, apply the algorithm to all the nodes in the component. The same is true if the graph is directed and strongly connected. It remains the case in which the graph is directed but not strongly connected. We will see how to to deal with this case in the next chapter.

4.3.4 Looking for the most important actor

The previously described algorithm has been implemented in C and used to analyze the IMDB graphs [6], where nodes are actors, and two actors are connected if they played together in a movie (some genres were excluded such as awards-shows, documentaries, game-shows, news, realities and talk-shows). In particular, for every snapshot of the IMDB graph, taken every 5 years from 1939 to 2014, the most central actor with respect to the closeness centrality has been computed. The obtained results are shown in Figure 4.3.4. The total time needed to perform the computation (with 30 parallel threads) was less than 40 minutes! In 2014, the most central actor is Michael Madsen, whose career spans 25 years and more than 170 movies. Among his most famous appearances, he played as Jimmy Lennox in *Thelma & Louise* (Ridley Scott, 1991), as Glen Greenwood in *Free Willy* (Simon Wincer, 1993), as Bob in *Sin City* (Frank Miller and Robert Rodriguez, 2005),and as Deadly Viper Budd in *Kill Bill* (Quentin Tarantino, 2003-2004). Note that in 2004 and 2009, the top "actor" is not really an actor: he is the German dictator Adolf Hitler. This a consequence of his appearances in several archive footages, that were re-used in several movies (he counts 775 credits, even if most of them are in documentaries or TV shows, which were eliminated). Among the movies where Adolf Hitler is credited, we find *Zelig* (Woody Allen, 1983), and *The Imitation Game* (Morten Tyldum, 2014). Among the other most central actors, we find many people who played a lot of movies, and most of them are quite important actors. However, this ranking does not discriminate between important roles and marginal roles: for instance, the actress Bess Flowers is not widely known, because she rarely played significant roles, but she appeared in over 700 movies in her 41 years career, and for this reason she was the most central for 30 years, between 1949 and 1979. Finally, it is worth noting that we never find Kevin Bacon in the top 10, even if he became famous for the "Six Degrees of Kevin Bacon" game (see `http://oracleofbacon.org`). In this

Figure 4.10: The most central actors in the IMDB graph with respect to the closeness centrality measure, computed every five years from 1939 to 2014.

game the player receives an actor x and has to find a path of length at most 6 from x to Kevin Bacon in the IMDB graph. Kevin Bacon was chosen as the goal because he played in several movies, and he was thought to be one of the most central actors: actually, he is quite far from the top. Indeed, his closeness centrality is 0.336, while the most central actor has centrality 0.354, the 10th actor has centrality 0.350, and the 100th actor has centrality0.341.

4.4 An example: protein-protein interaction graphs

Protein-protein interactions are essential to almost every process in a cell. Protein-protein interaction graphs are mathematical representations of the physical contacts between proteins in the cell, that have been repeatedly used in the literature, both in the bioinformatics and in the computer science research area. We conclude this chapter by applying the top node algorithm to a quite small protein-protein interaction graph (called HC-BioGRID), which has been made available on the BioGRIDweb site [14] (*BioGRID* is a biomedical interaction repository, in which all data are freely provided and available for download in many standardized formats [84]). This graph contains 4039 nodes and 10321 arcs: hence, is quite small. Nevertheless, we will show how efficient the top node algorithm described above is with respect to the approach based on computing a BFS starting from each node of the graph.

Before doing this, however, let us go back to the approximation algorithm based on the sampling technique, and let us verify that the top node obtained by this algorithm is not always the correct one. Indeed, let us execute the code used for analyzing the graph of the Florentine families with $k = 143$, which is bigger than $\frac{\log 4039}{0.09}$ (that is, the error factor t should be at most 0.3). It turned out that the top node with respect to the closeness centrality, that is, the node 201, is the top one with respect to the approximate values approximately in 60% of the experiments (confirming what we observed in the case of the graph of the Florentine families).

First, let us implement the algorithm in Julia. To this aim we first define a function which computes the pruned BFS procedure, by taking as input the graph, the starting node, and a centrality closeness value (which will correspond to the best value computed at a given iteration of the algorithm).

```
function prunedBFS(g::SimpleGraph{Int64}, source::Int64, topk::Float64)
    visited::Array{Bool} = falses(nv(g))
    cur_level::Array{Int64}, next_level::Array{Int64} = Vector(), Vector()
    visited[source] = true
    push!(cur_level, source)
    nd::Int64, farness::Int64, d::Int64 = 1, 0, 0
    while !isempty(cur_level)
        d, l_next_d::Int64 = d + 1, 0
        for v::Int64 in cur_level
            for i::Int64 in outneighbors(g, v)
                if !visited[i]
                    push!(next_level, i)
                    nd, farness = nd + 1, farness + d
                    l_next_d = l_next_d + degree(g, i) - 1
                    visited[i] = true
                end
            end
        end
        if (topk >= (nv(g) - 1) / farness)
            return 0
        end
        empty!(cur_level)
        cur_level, next_level = next_level, cur_level
    end
    return (nv(g) - 1) / farness
end
```

By using the above function `prunedBFS`, we can now implement the algorithm for finding the top node with respect to the closeness centrality (recall that, in order to speed up the execution, we scan the nodes ordered in non increasing order with respect to their degree).

```
function top_closeness(graph::String)::Tuple{Int64,Float64}
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    top_node::Int64, top_kappa::Float64 = 0, 0.0
    d::Array{Int64} = sortperm(degree(g), rev = true)
    for v in 1:nv(g)
        kappa_v::Float64 = prunedBFS(g, d[v], top_kappa)
        if (kappa_v > 0)
            top_node, top_kappa = d[v], kappa_v
        end
    end
    return top_node, top_kappa
end
```

By executing the function `top_closeness` with input the graph stored in the file `hcbiogrid.lg`, we get that the top node is, clearly, the node with index 201: the computation on my computer requires approximately 0.75 seconds. By using, instead, the function `closeness_centrality` included in the `Graphs` package, the computation requires approximately 5.65 seconds: hence, the algorithm based on pruned BFSs is more than seven times faster than the "textbook" approach.

5. Giant components and bow-tie graphs

In a sense the web is much like a complicated organism, in which the local structure at a microscopic scale looks very regular like a biological cell, but the global structure exhibits interesting morphological structure (body and limbs) that are not obviously evident in the local structure.

Andrei Broder et al., "Graph structure in the Web", 2000

5.1 Giant components in undirected graphs

One of the most common tasks that must be performed when analyzing a graph is to look for groups of nodes that have specific properties that make the group particularly interesting. For example, one might think of grouping the nodes into two groups, one containing the nodes with "high" degree and one containing the nodes with "low" degree. The first group would correspond to a sort of "core" of the graph, while the second group would correspond to a sort of "periphery". In this chapter, instead, we mostly analyze the possibility of grouping nodes into distinct sets based on their connectivity properties. In particular, in this paragraph we will refer to the connected components of an undirected graph, which we recall are defined as maximal (strongly) connected subgraphs.

A *subgraph* of a graph $G = (V,E)$ is the graph obtained from a subset $S \subseteq V$ of nodes and including only the arcs connecting nodes in S. For example, the graph shown in the right part of Figure 5.1 is the subgraph obtained starting from the graph shown in the left part of the figure and considering only the nodes 1, 2, and 4. We note that only the arcs of the original graph connecting any two of these three nodes are included in the subgraph.

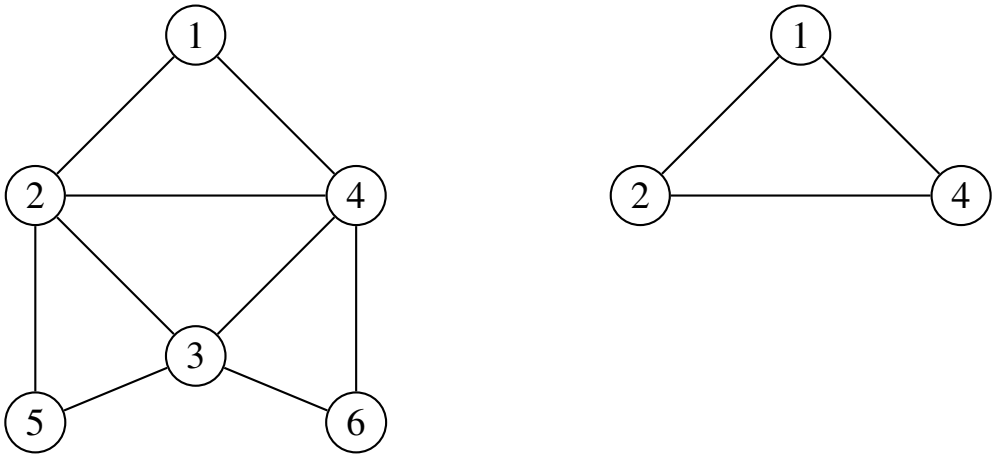
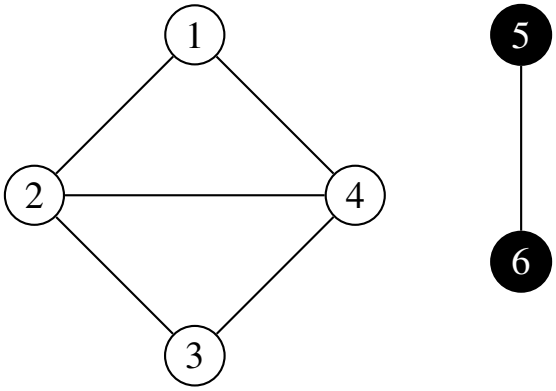


Figure 5.1: The graph on the right is a subgraph of the one on the left obtained by considering only the nodes 1, 2, and 4 and only the edges that connect these nodes.

As mentioned at the end of the first chapter, a *connected component* of an undirected G graph is a connected subgraph such that no other nodes of G can be added to the subgraph while maintaining the connectivity property (clearly if a graph is connected then it contains only one connected component). For example, in the graph of the following figure we have two connected components, that of nodes in white and that of nodes in black: we observe that the subgraph formed by only nodes 1, 2, and 4 is not a connected component as it is not *maximal*, since node 3 can be added to the subgraph and still get a connected subgraph.



We also saw, in the first chapter, how the BFS allows us to calculate the connected components of an undirected graph in time proportional to the number of arcs of the graph. For example, in the case of the graph in the previous figure, we could perform a first BFS starting from node 1: this BFS would include in the first level the nodes 2 and 4 and in the second level the node 3, and it would execute a number of operations proportional to the number of arcs connecting the visited nodes (that is, 5). Since at the end of the BFS there are still nodes marked as unvisited, we perform another BFS starting, for example, from the node 5, which would include in the first level the node 6 and would perform a number of operations proportional to the number of arcs connecting the visited nodes (that is, 1). At the end of this second BFS, all the nodes have been visited. We have thus found the two connected components of the graph: the first component includes the nodes visited by the first BFS, while the second includes those visited by the second BFS.

5.1.1 An example: the graph of the friendships on YouTube

YouTube is an online video sharing service founded in early 2005 and purchased by Google in late 2006 [62]. It allows users to post, view, share and judge videos: it has been and still is hugely successful, at the point that it is estimated that over 500 hours of videos are posted on YouTube every minute and a billion hours are viewed by users every day [21]. For a while, YouTube also allowed users to create friendships, much like what happens on Facebook today. The social network we are analyzing here refers to this YouTube feature, which, apparently, has now been disabled. This is a YouTube "snapshot" from 2007: a graph with 3223585 nodes and 9375370 arcs, where the nodes are YouTube users and the arcs indicate friendship relationships [54, 55]. Hence, it is a sparse graph.

If we apply the method just described to compute the connected components of this graph, we first realize that the graph is not connected. Specifically, running the following code we discover that the graph contains 2172 connected components (we note that, on my computer, it takes just over a second to run).

```
g = loadgraph("youtube.lg","graph")
cc = connected_components(g)
println(length(cc))
```

That the graph is not connected is perhaps not surprising: after all, we can imagine that YouTube users form unrelated "communities". However, more surprising is the fact that of these 2172 connected components, one contains almost all the nodes of the graph: in particular, we can verify this by executing the following code (after executing the previous code) .

```
println(maximum(length.(cc)))
```

The result gives the size of the largest connected component: in this case, that size is 3216071. In other words, the 99.8% of the nodes are included in only one connected component. This phenomenon is very common in social networks and is called the *giant connected component* phenomenon. We note that if a giant connected component exists, it is highly unlikely that another one exists: in fact, if there were two giant connected components, the probability that there are no two connected nodes, one each in the two different components, would be very low. Thus, the giant connected component, if any, is typically unique. In our case, in particular, all the other connected components are very small: as shown in Figure 5.2, the largest connected component (excluding the giant one) contains less than 50 nodes and the majority of connected components contain only two nodes (that is, they consist of a single arc). In conclusion, we can say that the YouTube graph is a very connected network, with few exceptions that form small "communities" disconnected from the rest of the network itself: as already mentioned, this feature is practically present in all real-world graphs, of whatever kind they are.

To conclude the analysis of the graph of YouTube friends, we can apply the approximation algorithm of the distance distribution function seen in the second chapter (which assumed that the input graph was undirected, connected and unweighted) to calculate the degrees of separation of the giant connected component of this graph. To do this, by making use of the `Graphs` package, we can get the subgraph induced by the giant connected component and save it in a new file, as shown in the following code.

```
sg = induced_subgraph(g, cc[1])
savegraph("youtubelcc.lg",sg[1],compress=false)
```

By executing at this point the code seen in the second chapter with input the file `youtubelcc.lg` and $k = 10\log(n)$, we obtain that the average distance in the connected giant component of the

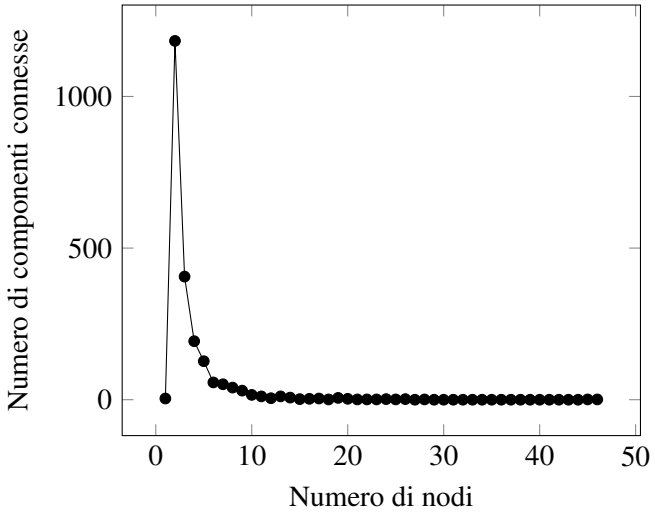


Figure 5.2: The distribution of the connected components of the YouTube graph, excluding the giant one: the maximum size is 46 and the maximum frequency is 1183 corresponding to connected components with two nodes (i.e., an arc).

YouTube graph is approximately equal to 5, respecting once again the hypothesis of the small world. Moreover, by using the iFUB algorithm, described in the third chapter, we can verify that the diameter of this graph is 31 (by executing only 22 BFSs if we choose as the starting node the node 1845582, which is one node of highest degree): that is this graph is a very small world.

5.2 Triangles in graphs and clustering coefficient

Before seeing how the existence of giant connected components can be experimentally verified also in the case of directed graphs, let us consider a measure of the degree to which the nodes in an undirected graph tend to cluster together, that is, the *clustering coefficient* of a graph. This coefficient, in particular, measures the degree to which a node's neighbors are themselves neighbors: more precisely, the *clustering coefficient* of a node u is the fraction of its pairs of neighbors that are themselves connected by an arc [85]. That is,

$$cc(u) = \frac{|\{(v,w) \text{ is an arc} : v \in N(u) \wedge w \in N(u)\}|}{\frac{d(u)(d(u)-1)}{2}},$$

where $d(u)$ denotes the degree of u and $N(u)$ its neighborhood (hence, $d(u) = |N(u)|$). For example, in Figure 5.1, the clustering coefficient of the node 3 is equal to $3/6 = 1/2$. It has been observed that high clustering coefficients signal groups of agents such that most pairs with a mutual friend are themselves friends. Such communities are expected to have interesting properties: for example, in such a community there is a strong disincentive to doing anything wrong to someone you are connected to, since lots of your other friends are likely to hear about it. On the other hand, a low clustering coefficient can signal the existence of a node which is well connected to different communities that are not otherwise connected to each other, and, hence, can be in a potentially advantageous situation (for example, to transfer information from one community to another).

Counting triangles in graphs is then a fundamental problem, which has received much attention because of their importance in graph mining [46, 57]. A trivial algorithm for counting the number of triangles in a graph consists in analyzing all possible triples of nodes u, v, and w, and checking whether a triple forms a triangle (that is, whether u is connected to v and w, and v is connected

to w). The number of triples examined by this algorithm is clearly cubic in the number of nodes, and, hence, this algorithm cannot be used whenever the number of nodes is very large. Note, however, that this complexity is unavoidable in the worst case, that is, when the graph contains a number of triangles which is cubic in the number of nodes: this happens when the graph is a clique. Nevertheless, we now describe an algorithm which analyze $O(m^{3/2})$ triples, where m is the number of arcs (note that this bound does not contradict the case of a clique, in which the number of edges is quadratic with respect to the number of nodes).

We begin by first defining a simpler algorithm, which generates paths of length 2 among the neighbors of a node u. The algorithm works by pivoting around u and then checking if there exist an arc that completes any of the resulting paths of length 2 to form a triangle. More precisely, the algorithm works as follows. For each node u and for each pair of distinct nodes $v,w \in N(u)$, if u, v, and w form a triangle, the algorithm increments a running count of triangles that include node u. It is easy to verify that the number of triples examined by this algorithm is proportional to $\sum_u (d(u))^2$: hence, if all nodes have degree linear with respect to the number of nodes, the number of triples examined by this algorithm will still be cubic with respect to the number of nodes. Even if only one node has degree linear with the number of nodes, this algorithm would examine a number of triples which is quadratic with respect to the number of nodes (and, hence, would be not practical in the case of very large graphs).

To see how to reduce the running time of the above algorithm, note that each triangle is counted six times, twice by pivoting around each node. Moreover, those pivots around high degree nodes generate far more paths of length 2 and are thus much more expensive than the pivots around the low degree nodes. To improve upon the baseline algorithm, the new algorithm is based on the idea of making the lowest degree node in each triangle be "responsible" for making sure the triangle gets counted. To this aim, we modify the above algorithm as follows (we assume that if two nodes have the same degree, than their index decides which one has the greater degree). For each node u and for each pair of distinct nodes $v,w \in N(u)$ such that $d(v) > d(u)$ and $d(w) > d(v)$, if u, v, and w form a triangle, the algorithm increment a running count of triangles that include node u. For example, in the case of the graph in the left part of Figure 5.1, node 1 will count the triangle $\{1,2,4\}$, node 2 will count the triangle $\{2,3,4\}$, node 5 will count the triangle $\{2,3,5\}$, and node 6 will count the triangle $\{3,4,6\}$.

Proposition 5.2.1 — (46, 75). The worst-case number of triples examined by the algorithm above is $O(m^{3/2})$, where m is the number of arcs.

Call a vertex u *big* if its degree $d(u)$ is at least $\sqrt{m}$, and *small* otherwise. Observe that there are at most $2\sqrt{m}$ big nodes: otherwise, the sum of the degrees of the graph would be more than $2\sqrt{m}\sqrt{m} = 2m$, which cannot be true. Thus, all but a relatively small number of nodes have relatively small degrees. If S denotes the set of small nodes, the total work done by the small nodes is $O\left(\sum_{u \in S} d(u)^2\right)$. Since $d(u) < \sqrt{m}$ for every $u \in S$ and since $\sum_{u \in S} d(u) \leq 2m$, we have that $\sum_{u \in S} d(u)^2$ is maximized when we have $2\sqrt{m}$ nodes with degree $\sqrt{m}$ each (this follows from the convexity of the function $f(x) = x^2$). With the worst-case choice of the degrees of the small nodes, we have that the total work done by the small nodes is $O(m^{3/2})$. It remains to bound the work done by the big nodes. All work done by a big node u corresponds to neighbor pairs v and w with larger degree (and hence also big). Thus, the total work done by all big nodes is at most a constant times the number of triples u, v, and w of big nodes. Recalling that there are at most $2\sqrt{m}$ big nodes, there are at most $8m^{3/2} = O(m^{3/2})$ such triples. By combining the bounds for the work done by small and by big nodes, the total work done by the algorithm is $O(m^{3/2})$, as claimed.

The following function implements the above algorithm, and its execution time can be compared with the execution time of the function `global_clustering_coefficient` included in the `Graphs` package.

```
function number_triangles(graph::String)
    g::SimpleGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    d::Array{Int64} = degree_centrality(g, normalize = false)
    c::Int64 = 0
    for u in 1:nv(g)
        for v in neighbors(g, u)
            if (d[v] > d[u] || (d[v] == d[u] && v > u))
                for w in neighbors(g, u)
                    if (d[w] > d[v] || (d[w] == d[v] && w > v))
                        if has_edge(g, v, w)
                            c = c + 1
                        end
                    end
                end
            end
        end
    end
    return c
end
```

For example, executing the instruction `number_triangles("youtube.lg")` requires (on my computer) less than 20 seconds, while the execution of the function computing the global clustering coefficient (included in the `Graphs` package) with input the YouTube graph requires several minutes.

5.3 Strongly connected components in directed graphs

In the case of directed graphs, the notions of connectivity must be adapted to the fact that the arcs now have a specific direction. In other words, the arc (x,y) implies that it is possible to go directly from node x to node y but does not necessarily imply that it is possible to go directly from node y to node x: for this to happen the arc (y,x) must also exist (obviously, it is also possible that the connection from y to x is not direct but passes through a path formed by two or more directed arcs).

Recall that a directed graph is called *strongly connected* if, for each pair of nodes x and y, there exists a path from x to y and one from y to x. A *strongly connected component* of a directed G graph is a strongly connected subgraph such that no other node of G can be added to the subgraph while maintaining the strong connectivity property (clearly if a graph is strongly connected then it contains only one strongly connected component). For example, the directed graph shown in Figure 5.3 is not strongly connected, as no node on the right side of the graph is able to reach any node on the left side of the graph. This graph contains the following five strongly connected components: $\{1,2,3\}$, $\{4,5\}$, $\{6,7,8,9\}$, $\{10,11\}$, and $\{12\}$. For example, the lower right component cannot be extended with any other node, as there are no edges that leave nodes 10 and 11 and go to nodes not already included in the component.

The computation of strongly connected components in a directed graph is more complicated than that of connected components in an undirected graph, if we want the execution time of this calculation to be proportional to the number of arcs of the graph. Indeed, the strongly connected components of a directed graph G can be easily computed using the following algorithm, which,

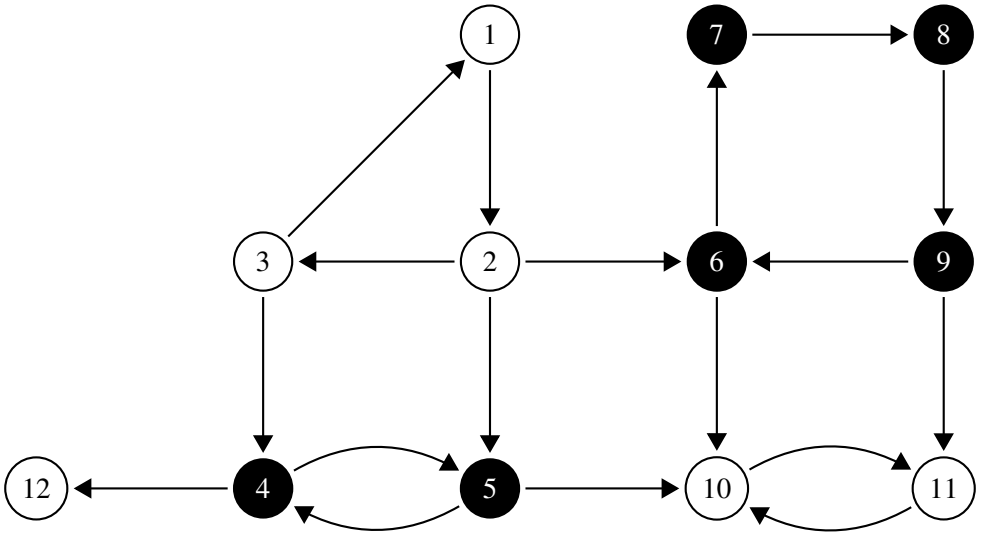


Figure 5.3: A not strongly connected directed graph: for example, from node 4 it is not possible to reach node 1 and from any node on the right side of the graph it is not possible to reach any of the nodes on the left side. The graph includes the following five strongly connected components: $\{1,2,3\}$, $\{4,5\}$, $\{6,7,8,9\}$, $\{10,11\}$, and $\{12\}$.

again, uses the BFS.

First of all, we construct the transposed graph G^T which is obtained by inverting the directions of all arcs. We start from any node x and perform a BFS starting from that node in G: let $F(x)$ be the set of visited nodes, that is, the set of all and only the nodes that can be reached from from x. At this point, we perform a BFS starting from node x in G^T: let $B(x)$ be the set of visited nodes, that is, the set of all and only the nodes that can reach x in G (remember that in the transposed graph the arcs have been inverted). The strongly connected component $C(x)$ that includes the node x is then obtained as the intersection of the two sets $F(x)$ and $B(x)$ (that is, $C(x) = F(x) \cap B(x)$): in fact, any node in $C(x)$ can reach node x (being included in the set $B(x)$) and from x it can reach any node in $F(x)$ (and, therefore, any node in $C(x) \subseteq F(x)$). Once we have computed the strongly connected component including x, we can repeat this process starting from any of the nodes not included in that component and so go on until each node is included in exactly one strongly connected component connected.

For example, in the case of the graph in Figure 5.3, the transposed graph is shown in Figure 5.4. We perform the BFS starting from node 1 in the original graph: in this case, all the nodes of the graph are visited (that is, $F(x) = V$). We then perform the BFS starting from the node 1 in the transposed graph: the nodes visited (in addition to the node 1) are the nodes 3 (at the first level) and 2 (at the second level). Thus, $B(x) = \{1,2,3\}$ and $C(x) = V \cap B(x) = \{1,2,3\}$: the strongly connected component, which includes the node 1, also includes the nodes 2 and 3 (see the upper right part of Figure 5.3). Similarly, we can calculate the other strongly connected components.

The problem with the procedure just described is that each BFS can require a number of operations proportional to the number of arcs of the directed graph that have not already been included in a strongly connected component. It is possible to show that this entails a total number of operations proportional to nm: this execution time, as we already noted in the previous chapters, is not acceptable, when we have millions of nodes and millions (if not billions) of arcs. For this reason, we will see in the rest of the chapter a much more efficient algorithm that makes use of a visit of the graph different from the BFS. But first let us analyze a directed graph small enough not

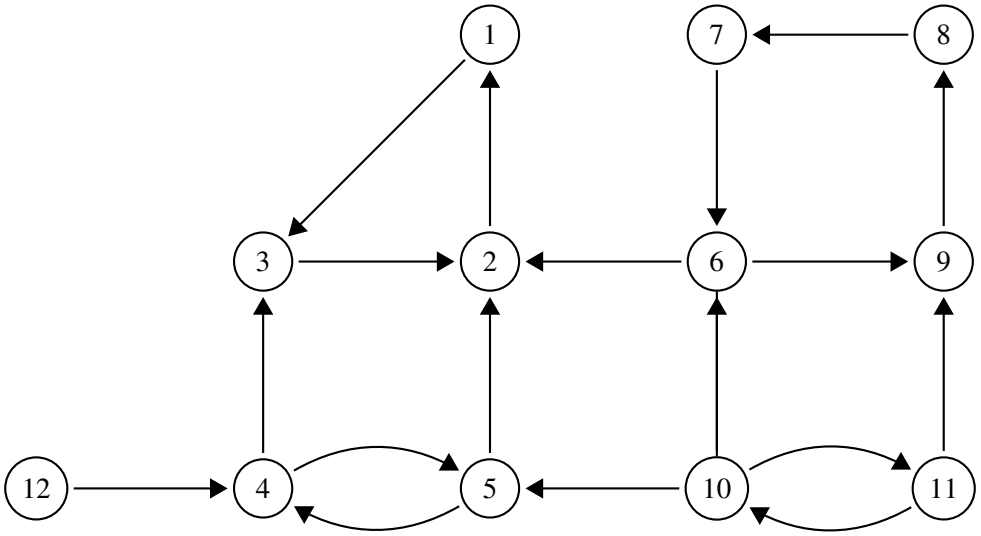


Figure 5.4: The transposed graph of the directed graph of Figure 5.3: this graph includes the same nodes and all the inverted arcs. For example, the graph includes the arc $(3,2)$ since the starting graph included the arc $(2,3)$.

to need a faster algorithm than the one just described for the computation of strongly connected components.

5.3.1 An example: the graph of the Italian parliament

OpenPolis is a foundation that provides various information and data collection and processing services in the field of Italian politics [72]. Among its online services there is *OpenParlamento* which monitors the work of the parliament on a daily basis, processing the official data available (for example, from the website of the Chamber, the Senate or the Ministry of the Interior) [71]. Several times OpenParlamento has been taken as a reference point to observe the activity of the chambers as a whole, but also of individual parliamentarians (whether they are deputies or senators). Among the many information that OpenParlamento makes available for their consultation are those relating to the presentation and, subsequently, to the process of the bills. In this paragraph, we focus on the presentation of a bill and, in particular, on parliamentarians who present a specific bill or who support it. A bill, in fact, has associated the parliamentarians who present it and sign it, assuming direct responsibility and who, for this reason, are called *first signers*, and the parliamentarians who support it by adding their signature and which, for this reason, are called *co-signers*. In some ways, being co-signer of a bill presented by another parliamentarian can be seen as an act of support or, if you like, of "political friendship". This relationship is clearly not symmetrical, as a parliamentarian A may be co-signer of a bill presented by a parliamentarian B, but B may never have co-signed a bill presented by A (in fact, it may even happen that A has never presented a bill). By making use of the data made available by OpenParlamento, we can then create a directed graph in which the nodes are the deputies of parliament in the XVI legislature (i.e., that from 2008 to 2013) and, for each pair of nodes x and y, the arc (x,y) exists if the deputy x has co-signed a bill presented by the deputy y.

This graph contains 620 nodes and 30020 arcs: its density is therefore equal to $\frac{30020}{620 \cdot 619} \approx 0.08$ and, therefore, albeit to a lesser extent, is a sparse graph (we note that, since the graph is directed, the number of edges must not be doubled). The graph is not strongly connected: this should not surprise us too much, considering that, normally, in the chamber there are government and opposition deputies and which, presumably, the former do not support the bills presented by the

latter and vice versa. However, if we calculate the strongly connected components, we find that, even in this case, there is a giant strongly connected component: in particular, the largest strongly connected component contains 506 nodes, which is about 82% of all nodes in the network. In other words, for the vast majority of the pairs of deputies x and y, the following statement holds: x supported a bill presented by a member of the parliament who supported a bill presented by a member of the parliament who supported ... a bill presented by y. To know how long this chain of bill support is on average, we can consider the average distance within the giant strongly connected component. Executing the code to calculate this value, it turns out that the average distance is approximately equal to 2.22 (even if the diameter is equal to 6).

The existence of the strongly connected giant component can perhaps be explained by considering that there are so-called "bipartisan" bills, that is supported by both government and opposition parliamentarians. But it could also be a consequence of the not rare changes of parliamentary group: in the XVI legislature there were 120 group changes which could therefore justify the rise of the strongly connected giant component.

Finally, we note that all the other strongly connected components are formed by a single node. Notably, of the remaining 114 parliamentarians, only 2 have both presented at least one bill and supported at least one bill. Everyone else has only supported at least one bill. Furthermore, the bill supported by these deputies was presented by a node included in the giant strongly connected component: in other words, these deputies "enter" the giant component and have no other type of connection. This structure begins to give a glimpse of what at the beginning of the 2000s was called the "bow-tie structure" of the web and which we will discuss in the last paragraph of this chapter.

5.4 The Kosaraju-Sharir algorithm

In this section, we describe and analyze an algorithm for the computation of the strongly connected components of a directed graph whose execution time is proportional to the number of arcs included in the graph [80]. The main problem we have to face consists in the fact that, when we perform a graph visit, we can visit nodes that are in distinct strongly connected components: for example, we have seen how the BFS of the graph in Figure 5.3 starting from node 1 reaches all the nodes of the graph. This problem, however, can be solved by determining a specific order in which to make the visits to the graph: in particular, we will see how it is possible to determine an order that guarantees that every visit made starting from a node x reaches all and only the nodes included in the strongly connected component to which x belongs.

To understand why the order in which we visit the nodes of the graph is important, let us consider the directed graph shown in Figure 5.5. In this case, if we perform the BFS starting from node 1, all the three nodes of the graph are reached, with node 2 at the first level and node 3 at the second level. However, the three nodes are not part of the same strongly connected component: in fact, the graph has two strongly connected components, one which includes only the node 1 and the other which includes the nodes 2 and 3. If, however, we perform the BFS starting from node 2 (or from node 3), then only the nodes 2 and 3 are reached, which, as we have just said, are part of the same strongly connected component. By eliminating these two nodes and performing another BFS, we determine the second strongly connected component.

To determine the right order in which to carry out the visits of the directed graph, we will make use of a different type of visit of a graph, which differs from the BFS for the way in which it decides to continue the discovery of the graph itself.

5.4.1 Depth-first search

The *depth-first search* (in short, *DFS*) of a graph is similar to the BFS. Unlike the latter, when a node x is reached for the first time and is therefore marked as visited, the DFS executes again in a

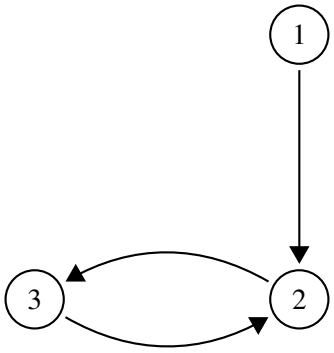


Figure 5.5: If we perform the BFS starting from node 1, all the three nodes of the graph are reached, even if they are not part of the same strongly connected component. If, on the other hand, we perform the BFS starting from node 2 or from node 3, then only the nodes 2 and 3 are reached, which are part of the same strongly connected component.

recursive way starting from x. When the DFS from this node is finished, then the node is marked as explored. As in the case of the BFS, also the DFS performs a number of operations proportional to the number of arcs of the graph. And like the BFS, the DFS partitions the graph nodes into levels (which however no longer correspond to the distances from the starting node): this partition into levels is also called *DFS tree*.

For example, let us consider the directed graph of Figure 5.3 which we show again in the left part of Figure 5.6 and suppose to perform the DFS starting from node 1 the exploration of which then begins (see the DFS tree shown in the right part of the figure). This node has only one outgoing arc that connects it to node 2: therefore, node 2 is marked as visited and the DFS starts again from it. The node 2 node has three neighbors, the nodes 3, 5, and 6: the DFS visits the node 3 node and starts over from it. The node 3 has two neighbors, namely the nodes 1 and 4: the node 1 has already been visited, so the DFS starts over at the node 4. The node 4 node has two neighbors, the nodes 5 and 12: the DFS starts over at the node 5. The latter has two neighbors, the nodes 4 and 10: the node 4 has already been visited, so the DFS starts over at the node 10. From this node it is only possible to go to node 11. At this point node 11 has only one neighbor, node 10, which has already been visited: therefore, the DFS starting from node 11 is finished, node 11 is marked as explored and the control passes back to the visit started from node 10. The latter has no other neighbors to examine and, therefore, its DFS is finished, the node is marked as explored and the visit continues with the one started from node 5 and, for the same reason, with the one started from node 4. The latter still has the node 12 as an unvisited neighbor, so the DFS starts again from it. The node 12 has no neighbors, so its DFS ends, the node is marked as explored and the DFS continues with the one started by the node 4, then with the one started by the node 3, and, finally, with the one started by the node 2, which still has the node 6 as an unvisited neighbor: the DFS starts again from the node 6, and then reaches the nodes 7, 8, and 9. At the end of the visit of the node 6, that of the node 2 also ends, and, finally, that of the node 1.

In the DFS tree of Figure 5.6 we have explicitly shown some arcs that led from the node currently examined to one already visited: these arcs have the particularity of connecting a node x of the DFS tree to a node that precedes it along the path from the starting node to x. For this reason, they are called *return arcs*: a return arc identifies a *cycle*, i.e. a path that goes from a node x to a node y and then ends with the arc (y,x). The DFS is, therefore, an efficient tool to determine if a graph has at least one cycle: when a return arc is found, we can conclude that a cycle exists. In our example, the first cycle that is determined is the one that passes through the nodes 1, 2, and 3: this

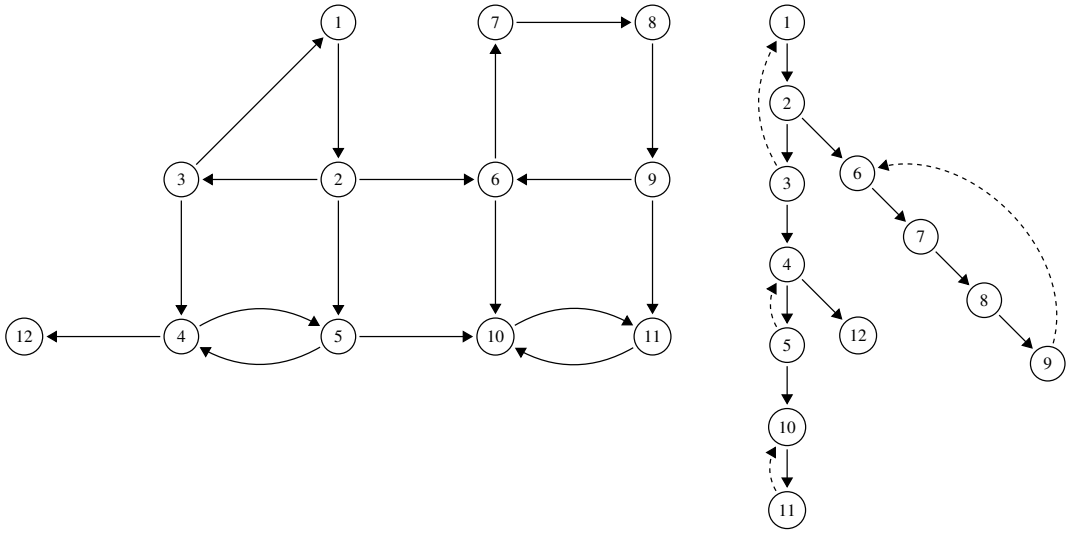


Figure 5.6: The directed graph of Figure 5.3 (left) and the corresponding DFS tree obtained starting from node 1 (right): the dashed arcs are return arcs that identify some cycles within the graph.

cycle is identified during the DFS starting from the node 3, when we notice that one of the three neighbors has already been visited and precedes it. But how can a node x know if a node y already visited precedes it on the way from the starting node to x? To answer this question, just observe that if y precedes x in the path within the DFS tree, then y is marked visited but not explored, as y is certainly at least "waiting" for the DFS started from x to end.

5.4.2 Directed acyclic graphs and topological ordering

Directed graphs that do not contain cycles are called *acyclic* or *DAG* (from *Directed Acyclic Graph*). An important property of such graphs is that their nodes can be ordered so that there is no edge in the graph that connects a node x to a node y that precedes it in the order. Such an ordering is called *topological ordering* and can be calculated using the DFS suitably modified as follows.

While carrying out the DFS, we keep a global "clock". The value of this clock increases by 1 each time the DFS is started from a node x: its value will be the *starting* value $s(x)$ associated with x. The value of the clock is also increased by 1 when the DFS started from a node x ends: its value will be the *ending* value $f(x)$ associated with x. Let us assume that at the beginning of the whole DFS, the value of the clock equals 0. In the right part of Figure 5.7, for example, the DFS tree is shown of the DAG of the left part of the figure: next to each node x of the tree is shown its starting value $s(x)$ and its ending value $f(x)$.

The topological ordering of the nodes of a DAG can be obtained by listing the nodes in descending order with respect to their ending time. For example, in the case of the graph in Figure 5.7, the topological ordering would be as follows: 1, 4, 2, 5, and 3. Each arc of the graph connects a node to another node that follows it in this order: for example, the arc $(1,4)$ connects the node 1 (whose ending time is 10) to the node 4 (whose ending time is 9). We note that if the DFS has not reached all the nodes of the graph, then it is sufficient to perform another DFS starting from an unreached node and continue in this way until all the nodes have been sorted.

Proposition 5.4.1 The decreasing ordering of the nodes of a DAG according to their ending times of the DFS is a topological ordering.

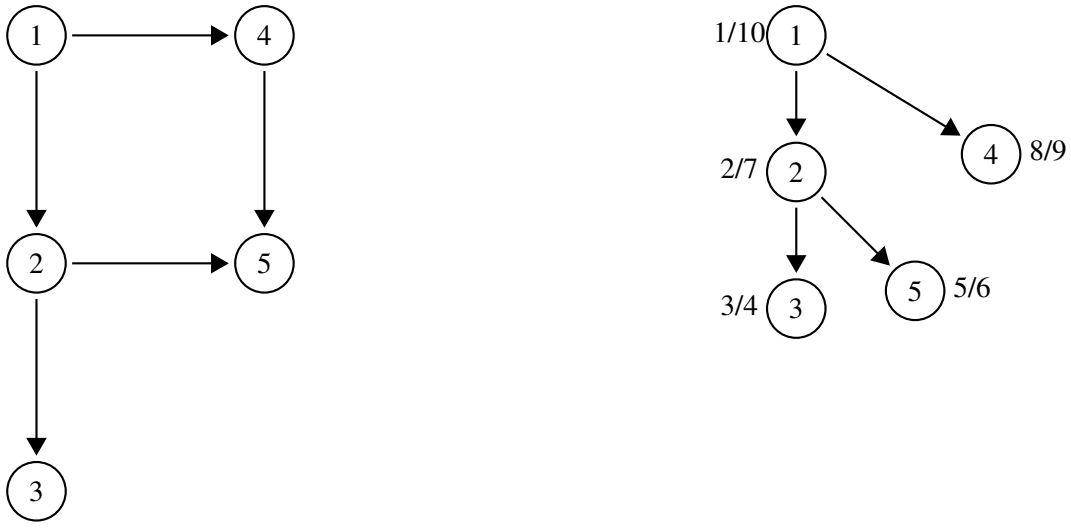


Figure 5.7: A DAG (left) and the corresponding DFS tree starting from node 1 (right): the numerical values shown next to each node x are, respectively, the value $s(x)$ at the beginning of the DFS starting from x and the value $f(x)$ at the end of the DFS itself. The corresponding topological ordering is the following one: 1, 4, 2, 5 e 3.

We must prove that, for each pair of nodes x and y such that the arc (x,y) exists, $f(y) < f(x)$ holds. From the definition of DFS, it follows that, when the arc (x,y) is examined within the DFS performed from x, the node y can either be unvisited or explored. Otherwise, in fact, the arc (x,y) would be a return arc and the graph would include a cycle, contrary to the hypothesis that the graph is a DAG. If y is explored, then the DFS starting from y is finished, while the one starting from x is still running: therefore, $f(y) < f(x)$. If, on the other hand, y is not visited, then y becomes a successor of x in the DFS tree: that is, within the DFS started from from x, the DFS starts from y. Therefore, the DFS started from x cannot end before the one started from y: also in this case, therefore, $f(y) < f(x)$. The proposition therefore turns out to be proved.

Computing the strongly connected components

To understand and analyze the algorithm for calculating the strongly connected components of a directed graph, we first introduce another very important notion when dealing with directed graphs. Given a directed graph G, the *component graph* G_{scc} is a directed graph whose nodes represent the strongly connected components of G: for each pair of nodes c_1 and c_2 in G_{scc}, the arc (c_1,c_2) exists if and only if there is at least one arc in G that connects a node of the strongly connected component corresponding to c_1 to a node of the strongly connected component corresponding to c_2. For example, the component graph of the directed graph on the left side of Figure 5.8 is shown on the right side of the figure. Indeed, the graph on the left is the same as in Figure 5.3 and we had already observed that this graph has five strongly connected components. Therefore, the component graph has five nodes c_1, c_2, c_3, c_4, and c_5 which correspond, respectively, to the five strongly connected components $\{1,2,3\}$, $\{4,5\}$, $\{12\}$, $\{6,7,8,9\}$, and $\{10,11\}$. As regards the arcs, we have that, for example, the component graph includes the arc (c_1,c_4) since the original graph has an arc (in particular, the arc $(2,6)$) that connects a node of the component corresponding to c_1 (that is, the node 2) to a node of the component corresponding to c_4 (that is, the node 6).

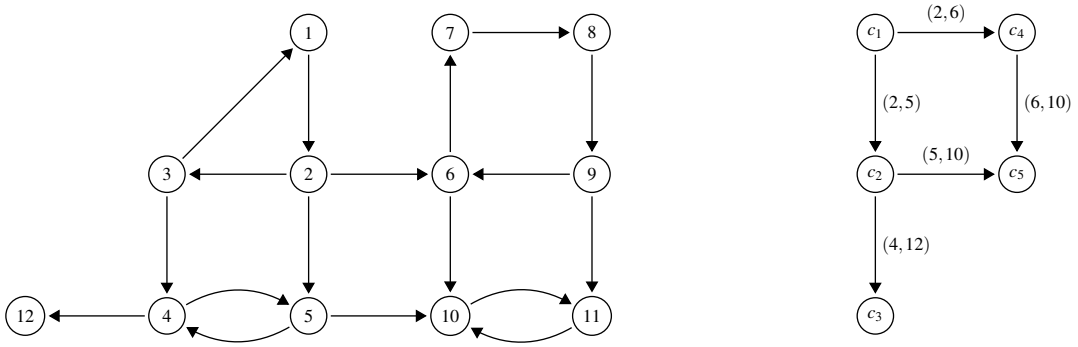


Figure 5.8: A directed graph (on the left) and the corresponding component graph (on the right): for each arc of the latter, one of the possible arcs of the graph on the left that justifies its existence is indicated on the arc itself.

Proposition 5.4.2 The component graph G_{scc} of a directed graph G is a DAG.

Suppose that there is a $c_1, \ldots, c_{k-1}, c_k = c_1$ cycle inside G_{scc}. This implies that, for every i with $1 \leq i < k$, there exists a node a_i in the strongly connected component C_i corresponding to c_i and there exists a node b_{i+1} in the strongly connected component C_{i+1} corresponding to c_{i+1} such that G includes the arc (a_i, b_{i+1}). From the definition of strongly connected component it follows that, for each node s of the strongly connected component C_i, from s we can reach the node a_i and that from a_i we can reach s. Therefore, for each node s of C_1 and for each node d of C_j with $j > 1$, there exists in G a path joining s to a_1, a_1 to b_2, b_2 to a_3, and a_3 to b_4, and so on until you get to b_j from which you can reach d. Similarly, we can show that there exists a path from every node of C_j to every node of C_1 with $i < j$ (since $c_k = c_1$). In other words, C_1 would not be maximal, contradicting the definition of a strongly connected component. The proposition therefore turns out to be proved.

Since the component graph is a DAG, its nodes can be ordered topologically. In other words, the strongly connected components of a directed graph G can be ordered so that if a component C_1 follows another component C_2, then there is no edge in G that connects a node in C_1 to one in C_2. Therefore, if we carry out a BFS starting from the last strongly connected component in this order, we are sure that all and only the nodes of this strongly connected component will be reached. We can, therefore, eliminate these nodes from the graph, and start over from the immediately preceding component, repeating this procedure until we have visited all the nodes of the graph. For example, we have seen in Figure 5.7 that a topological ordering of the component graph of Figure 5.8 is the following: c_1, c_4, c_2 , c_5, and c_3. Therefore, we can perform a BFS of the graph starting from a node of $c_3 = \{12\}$: in this case we visit only the node 12. Once this node has been eliminated, we perform the BFS starting from a node in $c_5 = \{10, 11\}$: in this case, we visit the nodes 10 and 11. And so we can continue until we have visited all the nodes.

The problem with this approach is that we are assuming to be able to construct the component graph, that is, to know which are the strongly connected components. And this is exactly what we have to do! However, the previous description suggests a way by which we can determine, at each iteration of our algorithm, a node x such that the execution of the BFS of the directed graph starting from x reaches all and only the nodes of the strongly connected component that includes x.

The basic idea is to use the ending times associated with each node of the directed graph during the execution of one or more DFS of the graph itself until all the nodes result visited. These times, in fact, will allow us to identify the strongly connected components according to their topological order in the component graph.

Given a directed graph G and given a strongly connected component C of G, we denote by $s(C)$ the minimum starting time between all nodes included in C and by $f(C)$ the maximum ending time. We observe that $s(C) = s(x_C)$ and $f(C) = f(x_C)$, where x_C is the first node of the strongly connected component to be reached by the DFS.

Proposition 5.4.3 If C_1 and C_2 are two strongly connected components of a directed graph G and if (u,v) is an arc of G with $u \in C_1$ and $v \in C_2$, then $f(C_1) > f(C_2)$.

In order to prove the proposition, we distinguish the following two cases.

1. $s(C_1) < s(C_2)$. Let x be the first node in C_1 reached by the DFS (hence, $f(C_1) = f(x)$) and let (y,z) be the first arc through which the DFS "exits" C_1 to enter C_2. Such an arc must exist since, by hypothesis, there is the arc (u,v) and this arc must be explored sooner or later (indeed, the arc (y,z) could just be the arc (u,v)). Since x is the first visited node of C_1, then the DFS started by x must end after the DFS started from any other node of C_1: therefore, $f(x) > f(y)$. Since the component graph is a DAG and there is an arc from component C_1 to component C_2, there cannot be a way to return from component C_2 to component C_1 (otherwise a cycle would be generated). Therefore, the visit of the C_2 component must end when the visit started from y ends, that is, $f(y) = f(C_2)$. Hence, $f(C_1) = f(x) > f(y) = f(C_2)$.
2. $s(C_1) > s(C_2)$. Let x be the first node in C_2 reached by the DFS. We know that the DFS started from x will reach all nodes of C_2. Also, since the component graph is a DAG and there is an arc from component C_1 to component C_2, there cannot be a way to get to component C_1 before the DFS started from x ends (otherwise a cycle would be generated). Hence, $f(C_2) = f(x) < s(C_1) < f(C_1)$.

The proposition therefore turns out to be proved.

As a result of the above discussion, we know that if we start a BFS from the node x with the largest ending time, then we will visit all nodes of the strongly connected component C_x which includes x, which is also the "first" of the strongly connected components in the topological ordering of the component graph. But how can we prevent this visit from leaving C_x? Since this component is the first in order, we are sure that there are no edges that "enter" it from other components. So to avoid "going out" of C_x we can perform the BFS starting from x in the transposed graph. We observe that the strongly connected components of the transposed graph are exactly the same as those of the starting graph: however, having inverted the arcs, it is no longer possible now to leave C_x.

In conclusion, the Kosaraju-Sharir algorithm operates in the following way.

1. It performs one or more DFSs until it visits all the nodes of the directed graph: in doing so, it calculates their ending times.
2. It computes the transposed graph G^t.
3. For each node x in G^t, in descending order with respect to the ending time, it performs a BFS starting from x in G^t, labels all the visited nodes as part of a new strongly connected component, and (logically) deletes them from G^t.

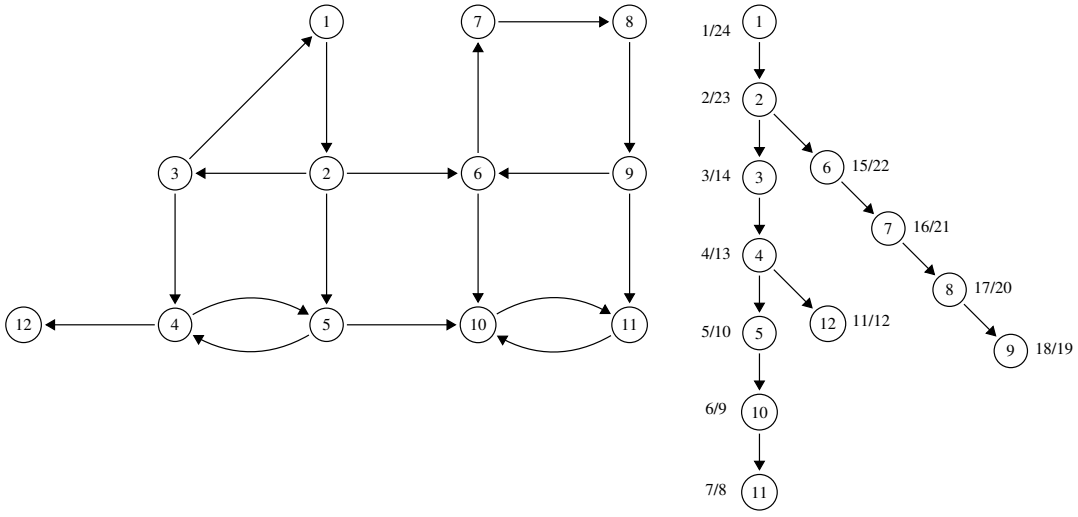


Figure 5.9: A directed graph (left) and the corresponding DFS tree (right) with the starting and ending times indicated next to each node: the ordering of the nodes to be used in the third step of the Kosaraju-Sharir algorithm is, therefore, 1, 2, 6, 7, 8, 9, 3, 4, 12, 5, 10, and 11.

The correctness of the algorithm follows from the previous discussion. As far as its execution time is concerned, this is proportional to the number of arcs of the directed graph, since also the construction of the transposed graph can be performed with a number of operations proportional to the number of arcs.

Finally, let us see the application of this algorithm to the directed graph of Figure 5.3, which we present again in the left part of Figure 5.9. By performing the DFS starting from node 1, the starting and ending times for each node are shown next to the nodes themselves in the DFS tree, shown in the right part of the figure. According to the ending times, we have that, in the third step of the algorithm, the nodes will be examined in the following order: 1, 2, 6, 7, 8, 9, 3, 4, 12, 5, 10, and 11. The second step of the algorithm consists simply in inverting the arcs of the directed graph, obtaining the transposed graph, which we have already shown in Figure 5.4 and which we present again in the left part of Figure 5.10. The third and final step is to examine the nodes one after the other following the previously established order. For each node, if it has not already been removed, a BFS is performed: all the nodes reached are included in a new strongly connected component and removed from the transposed graph. The five BFSs that are performed by this step in our example are shown in the right part of Figure 5.10: as we can see, the five strongly connected components that we had already shown in Figure 5.3 are correctly identified.

5.4.3 An example: the Wikipedia graph

Having now available a very efficient algorithm for the computation of strongly connected components, let's try to use it to analyze the structure of one of the most famous real-world graph, namely the Wikipedia graph. *Wikipedia* is a free open-collaborative online hyper-encyclopedia [40]. It is one of the most popular websites hosted by the Wikimedia Foundation, an American non-profit organization funded primarily through donations. It was launched on January 15, 2001, and currently comprises more than 55 million articles in 309 languages, attracting 1.7 billion unique visitors per month. The Wikipedia directed graph contains one node for each article and arc from a node x to a nod y if the page of x contains a hyperlink to the page of y.

Here, we are going to use a subgraph of the Wikipedia graph, which includes 1870709 nodes, and 39953145 arcs (for a large collection of Wikipedia graphs, see [55]). By using the function

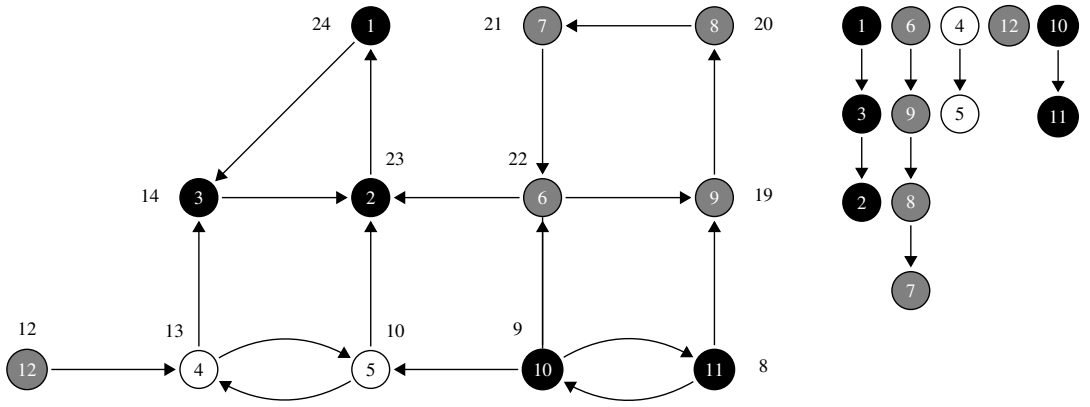


Figure 5.10: The third step of the Kosaraju-Sharir algorithm: using the previously calculated ordering of the nodes, a BFS is performed for each strongly connected component (the leveled partition of the BFSs is shown on the right) and the reached nodes reached are remove from the transposed graph.

`strongly_connected_components` of the `Graphs` package, we can verify that the Wikipedia graph has 237288 strongly connected components, but one of them is a giant component, including 1629321 nodes.

5.5 The bow-tie structure of WWW and other real-world graphs

The *World Wide Web* is probably the largest graph in the world (and, perhaps, the parent of all online graphs). In this graph, the nodes are the web pages and the arcs indicate the hypertext links between one page and another (the so-called *link*), which allow a user to move from one web page to another Web page). There are many snapshots of the World Wide Web, generally captured through programs, called *crawler*, which navigate from one web page to another and reconstruct a good part of the whole graph. One of the richest collections of such snapshots is the one available on the website of the *Web Algorithmics Laboratory* of the University of Milan [11].

In the case of this graph, it has been observed a very interesting phenomenon, called the *bowtie phenomenon* [17]. A detailed analysis of a fraction of the WWW graph (obtained by using the index of pages and links from one of the largest commercial search engines at the time) showed the existence of a giant strongly connected component *SCC* (as expected) and of two other giant sets: *IN*, the set of nodes that can reach SCC but cannot be reached from it, and *OUT*, the set of nodes that can be reached from SCC but cannot reach it. There are also nodes that belong to none of IN, OUT, or SCC (that is, they can neither reach SCC nor be reached from it). These nodes are classified as *tendrils* of the bow-tie (nodes reachable from IN that cannot reach SCC, and nodes that can reach OUT but cannot be reached from SCC), and *disconnected* (nodes that would not have a path to SCC even if we completely ignored the directions of the arcs). A special kind of tendril is the one connecting IN to OUT: we will call *tunnels* these tendrils. These findings are summarized in the very popular picture shown in Figure 5.11, which provides a high-level view of the web structure, based on its connectivity properties and how its strongly connected components fit together.

This study has been replicated on other larger snapshots of the web. Similar analyses have also been carried out for specific subset of the web, such as the Wikipedia graph. Although the snapshot of the web used in the original study comes from an earlier time in the history of the web, the mapping paradigm proposed in that study continues to be a useful way of thinking about giant

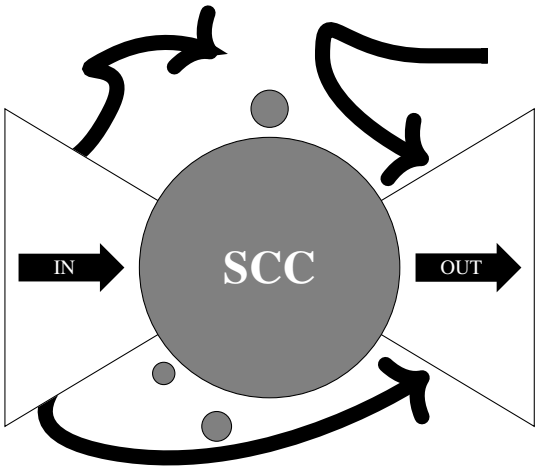


Figure 5.11: The bow-tie structure of the WWW graph.

directed graphs in the context of the web and more generally.

5.5.1 The bow-tie structure of the Wikipedia graph

Let us now compute the bow-tie structure of the Wikipedia graph that we have analyzed above. Recall that this graph contains 1870709 nodes and has a giant strongly connected component with 1629321 nodes. We now show how we can classify each node of the graph depending on whether it belongs to SCC, to OUT, to IN, to a tendril, to a tunnel, or it is disconnected. To this aim, we will make use of an array `group` that, for any node u, will contain the value 1, 2, 3, 4, 5, and 6, depending on which case the node belongs to (at the beginning this array contains all zeros). After having computed the giant strongly connected component, we will then set the value of `group` equal to 1 for each node in SCC. This is what is done in the first seven lines of the following `bowtie` function.

```
function bowtie(graph::String)::Array{Int64}
    g::SimpleDiGraph{Int64} = loadgraph("graphs/" * graph, "graph")
    group::Array{Int64} = zeros(Int64, nv(g))
    scc::Array{Array{Int64}} = strongly_connected_components(g)
    lscc_index::Int64 = argmax(length.(scc))
    lscc::Array{Int64} = scc[lscc_index]
    println("|SCC|=", length(lscc))
    for u in 1:length(lscc)
        group[lscc[u]] = 1
    end
    group_out(g, lscc[1], group)
    group_in(g, lscc[1], group)
    group_tendril_in(g, group)
    group_tendril_out(g, group)
    return group
end
```

The next step consists in identifying the nodes in OUT, that is, the set of nodes that can be reached from SCC but cannot reach it. To this aim, we can choose any node u in SCC and executes a BFS starting from it: whenever we visit a node v whose `group` value is equal to zero, we know

that v belongs to OUT. Indeed, it is reached by a node in SCC, but it cannot reach a node in SCC, since otherwise it would belong to SCC and its `group` value would be equal to 1. The computation of OUT is implemented in the following code.

```
function group_out(g::SimpleDiGraph{Int64}, u::Int64,
                group::Array{Int64})::Nothing
    visited::Array{Bool} = falses(nv(g))
    queue::Array{Int64} = Vector()
    visited[u] = true
    push!(queue, u)
    while !isempty(queue)
        u = pop!(queue)
        for v in outneighbors(g, u)
            if !visited[v]
                push!(queue, v)
                visited[v] = true
                if (group[v] == 0)
                    group[v] = 3
                end
            end
        end
    end
end
```

Similarly, we can identify the nodes in IN, that is, the set of nodes that can reach SCC but cannot be reached from it. To this aim, we first compute the transposed graph, we then choose any node u in SCC and executes a BFS starting from it on the transposed graph: whenever we visit a node v whose `group` value is equal to zero, we know that v belongs to IN. Indeed, it can reach a node in SCC (recall that we are now working on the transposed graph), but it cannot be reached by a node in SCC, since otherwise it would belong to SCC and its `group` value would be equal to 1. The computation of IN is implemented in the following code.

```
function group_in(g::SimpleDiGraph{Int64}, u::Int64,
                group::Array{Int64})::Nothing
    rg::SimpleDiGraph{Int64} = reverse(g)
    visited::Array{Bool} = falses(nv(rg))
    queue::Array{Int64} = Vector()
    visited[u] = true
    push!(queue, u)
    while !isempty(queue)
        u = pop!(queue)
        for v in outneighbors(rg, u)
            if !visited[v]
                push!(queue, v)
                visited[v] = true
                if (group[v] == 0)
                    group[v] = 2
                end
            end
        end
```

```
    end
end
```

The next step consists in identifying the tendrils going out from IN, that is, the set of nodes that can be reached from IN but cannot reach SCC. To this aim we can execute a BFS starting from *all* the nodes in IN, that is, by inserting in the queue, at the beginning of the BFS, all the nodes whose `group` value is equal to 2, and by marking them as already visited (we can also mark as visited all the nodes in SCC, since we are sure that the nodes reachable from SCC have already been classified). Whenever we visit a node v whose `group` value is equal to zero, we know that v belongs to a tendril going out from IN. Indeed, it can be reached by a node in IN, but it cannot reach a node in SCC, since otherwise it would belong to IN and its `group` value would be equal to 2. The computation of the tendrils going out from IN is implemented in the following code.

```
function group_tendril_in(g::SimpleDiGraph{Int64},
                    group::Array{Int64})::Nothing
    visited::Array{Bool} = falses(nv(g))
    queue::Array{Int64} = Vector()
    for u in 1:nv(g)
        if (group[u] == 2)
            visited[u] = true
            push!(queue, u)
        end
        if (group[u] == 1)
            visited[u] = true
        end
    end
    while !isempty(queue)
        u = pop!(queue)
        for v in outneighbors(g, u)
            if !visited[v]
                push!(queue, v)
                visited[v] = true
                if (group[v] == 0)
                    group[v] = 4
                end
            end
        end
    end
end
```

Finally, we can identify the tendrils entering OUT, that is, the set of nodes that can be reach OUT but cannot be reached from SCC, and, at the same time, the tunnels, that is, the nodes which are both in a tendril going out from IN and in a tendril entering OUT. To this aim we can execute, on the transposed graph, a BFS starting from *all* the nodes in OUT, that is, by inserting in the queue, at the beginning of the BFS, all the nodes whose `group` value is equal to 3, and by marking them as already visited (we can also mark as visited all the nodes in SCC). Whenever we visit a node v whose `group` value is equal to zero, we know that v belongs to a tendril entering OUT. Indeed, it can reached a node in OUT (recall that we are working on the transposed graph), but it cannot be reached from a node in SCC, since otherwise it would belong to OUT and its `group` value would be equal to 3. Moreover, if the `group` value of v is equal to 4, then we can conclude that v belongs to a tunnel, since it can be reached from a node in IN and it can reach a node in OUT. The computation

of the tendrils entering OUT and of the tunnels is implemented in the following code.

```
function group_tendril_out(g::SimpleDiGraph{Int64},
                    group::Array{Int64})::Nothing
    rg::SimpleDiGraph{Int64} = reverse(g)
    visited::Array{Bool} = falses(nv(rg))
    queue::Array{Int64} = Vector()
    for u in 1:nv(rg)
        if (group[u] == 3)
            visited[u] = true
            push!(queue, u)
        end
        if (group[u] == 1)
            visited[u] = true
        end
    end
    while !isempty(queue)
        u = pop!(queue)
        for v in outneighbors(rg, u)
            if !visited[v]
                push!(queue, v)
                visited[v] = true
                if (group[v] == 0)
                    group[v] = 5
                end
                if (group[v] == 4)
                    group[v] = 6
                end
            end
        end
    end
end
```

By executing the function `bowtie` on the graph `wikipedia.lg`, we have that SCC contains 1629321 nodes (as we already knew), IN contains 225017 nodes, and OUT contains 15305. Moreover, 653 nodes belong to a tendril going out from IN, 217 nodes belong to a tendril entering OUT, and 4 nodes belong to a tunnel. Finally, there are 192 nodes which are not classified. These nodes can belong to small strongly connected components disconnected from the rest of the graph (see Figure 5.11) or they can belong to tendrils entering tendrils going out from IN or going out from tendrils entering OUT. In any case, they are very few, and we can conclude that the Wikipedia graph has a very clear bow-tie structure.

5.6 Generalization of the top closeness node computation to directed graphs

We conclude this chapter by generalizing the algorithm for computing the top node with respect to the closeness centrality, in order to deal with directed graphs, which are weakly connected but not strongly connected [6]. Indeed, we already observed that all the information necessary to compute the upper bound $\tilde{\kappa}_d(v)$ is available at the end of the exploration of the level d of the BFS, since we could always determine the number $r(v)$ of nodes reachable from any node v. Indeed, if the graph is undirected and connected, then $r(v) = n$, otherwise this number can be easily computed during a

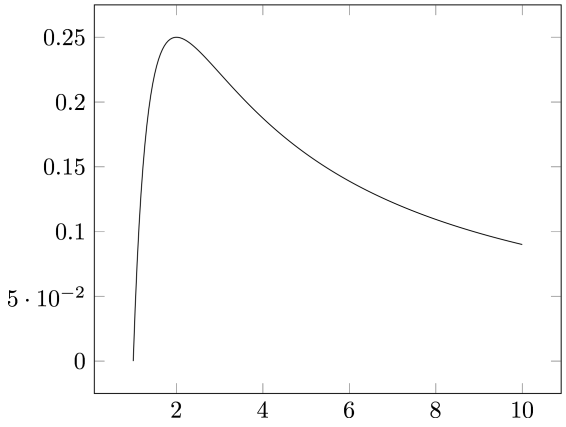


Figure 5.12: The plot of function $g(x) = \frac{ax-b}{x^2}$ with $a = b = 1$. There is a local maximum but no local minimum: hence, in each closed interval the minimum of g is reached in the extremes of the interval.

linear time pre-processing phase in which the connected components of the graph are identified, and, for each node v, $r(v)$ is set equal to the size of the component v belongs to. If the graph is directed and strongly connected, then $r(v) = n$. It remains to deal with the case in which the graph is directed but not strongly connected.

Let us assume, for now, that we know a lower (respectively, upper) bound $\alpha(v)$ (respectively, $\omega(v)$) on $r(v)$: without loss of generality we can assume that $\alpha(v) > 1$. We now show that, instead of examining all possible values of $r(v)$ between $\alpha(v)$ and $\omega(v)$, it is sufficient to examine only the two extremes of this interval. More specifically, we prove the following lower bound $\lambda_d(v)$ on $\frac{1}{\kappa(v)}$:

$$\frac{1}{\kappa(v)} \geq \lambda_d(v) = (n-1)\min\left(\frac{\tilde{\varphi}_d(v)\alpha(v)}{(\alpha(v)-1)^2}, \frac{\tilde{\varphi}_d(v)\omega(v)}{(\omega(v)-1)^2}\right).$$

If we denote $a = d+2$ and $b = \tilde{l}_{d+1}(v) + a(n_d(v)-1) - \varphi_d(v)$, we have that

$$\begin{aligned}\varphi(v) &\geq \varphi_d(v) - \tilde{l}_{d+1}(v) + a(r(v) - n_d(v)) \\ &= a(r(v)-1) + \varphi_d(v) - \tilde{l}_{d+1}(v) - a(n_d(v)-1) \\ &= a(r(v)-1) - b.\end{aligned}$$

Note that $a > 0$ because $d > 0$, and $b > 0$ because

$$\varphi_d(v) = \sum_{w \in N_d(v)} d(v,w) \leq d(n_d(v)-1) < a(n_d(v)-1)$$

where $N_d(v) = \bigcup_{i=1}^{d} \Gamma_i(v)$ and the first inequality holds because, if $w = v$, then $d(v,w) = 0$, and if $w \in N_d(v)$, then $d(v,w) \leq d$. Hence,

$$\frac{1}{\kappa(v)} \geq (n-1)\frac{a(r(v)-1)-b}{(r(v)-1)^2}.$$

Let us consider the function $g(x) = \frac{ax-b}{x^2}$. The derivative $g'(x) = \frac{-ax+2b}{x^3}$ is positive for $0 < x < \frac{2b}{a}$ and negative for $x > \frac{2b}{a}$: this means that $\frac{2b}{a}$ is a local maximum, and there are no local minima for $x > 0$. Consequently, in each closed interval $[x_1, x_2]$ where x_1 and x_2 are positive, the minimum of $g(x)$ is reached in x_1 or x_2. Since $0 < \alpha(v) - 1 \leq r(v) - 1 \leq \omega(v) - 1$,

$$g(r(v)-1) \geq \min(g(\alpha(v)-1), g(\omega(v)-1))$$

and the conclusion follows. It now remains to compute $\alpha(v)$ and $\beta(v)$ (in the case of a directed graph which is not strongly connected). This can be done during the pre-processing phase of the algorithm as follows. Let G_{scc} be the component graph of G and, for any SCC D, let $w(D)$ denote the number of nodes in D. Clearly, if v and w are in the same SCC, then $r(v) = r(w) = \sum_{D \in r(C)} w(D)$, where $r(C)$ denotes the set of SCCs that are reachable from C in G_{scc}. This means that we simply need to compute a lower (respectively, upper) bound $\alpha(C)$ (respectively, $\omega(C)$) on $\sum_{D \in r(C)} w(D)$, for every SCC C. To this aim, we first compute a topological sort $\{C_1, \ldots, C_l\}$ of V_{scc} (that is, if $(C_i, C_j) \in E_{\text{scc}}$, then $i < j$). Successively, we use a dynamic programming approach, and, by starting from C_l, we process the SCCs in reverse topological order, and we set

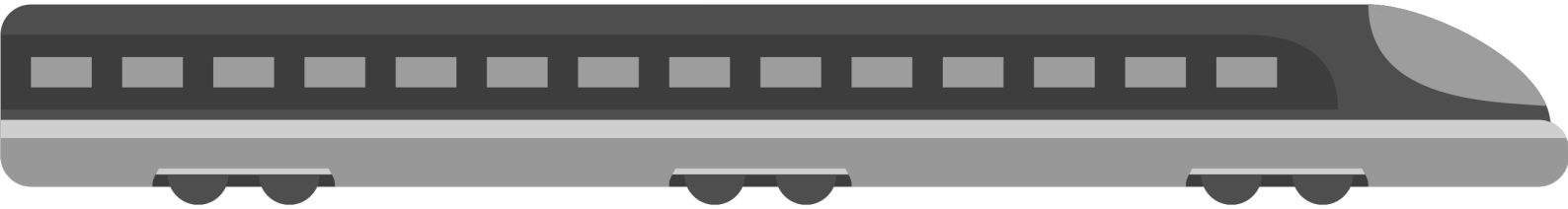
$$\alpha(C) = w(C) + \max_{(C,D) \in E_{\text{scc}}} \alpha(D)$$

and

$$\omega(C) = w(C) + \sum_{(C,D) \in E_{\text{scc}}} \omega(D).$$

Note that processing the SCCs in reverse topological ordering ensures that the values $\alpha(D)$ and $\omega(D)$ on the right hand side of these equalities are available when we process the SCC C. Clearly, the complexity of computing $\alpha(C)$ and $\omega(C)$, for each SCC C, is linear in the size of $\mathscr{G}$, which in turn is smaller than G.

Observe that the bounds obtained through this simple approach can be improved by using some "tricks". First of all, when the biggest SCC $\tilde{C}$ is processed, we do not use the dynamic programming approach and we can exactly compute $\sum_{D \in r(\tilde{C})} w(D)$ by simply performing a BFS starting from any node in $\tilde{C}$. This way, not only $\alpha(\tilde{C})$ and $\omega(\tilde{C})$ are exact, but also $\alpha(C)$ and $\omega(C)$ are improved for each SCC C from which it is possible to reach $\tilde{C}$. Finally, in order to compute the upper bounds for the SCCs that are able to reach $\tilde{C}$, we can run the dynamic programming algorithm on the graph obtained from G_{scc} by removing all components reachable from $\tilde{C}$, and we can then add $\sum_{D \in r(\tilde{C})} w(D)$.



6. Graphs over time

6.1 Temporal graphs

Temporal graphs are graphs in which nodes and edges can appear or disappear over time, due not only to failures or malfunctioning of the entities participating to the system represented by the temporal graph, but mostly to the "normal" behavior of the system itself [20]. A typical temporal graph is a person-to-person communication network within a company. In such a graph, for example, nodes can appear or disappear (depending on the recruitment policy of the company), and edges appear whenever an employee of the company sends an e-mail message to another employee of the company. In this chapter, we will focus on temporal graphs in which the set of nodes does not change over time (at least over a specified interval of time). Moreover, we consider only the case in which arcs are available at discrete time instants, so that the dynamics of the network is specified only by the appearance times of the arcs.

Temporal graphs can model a great variety of systems in nature, society and technology. Several different types of temporal graphs have, indeed, been analyzed: person-to-person communication networks (such as e-mails or phone calls), one-to-many information spreading networks (such as Twitter interactions), contact networks (such as cell phone proximity detections), biological networks (such as protein interactions), distributed computing networks (such as autonomous system communications), infrastructure networks (such as public transport timetables), and many others. It is worth observing, however, that different names have been used for denoting temporal graphs (even though the basic notion was almost the same), such as, for example, evolving graphs, time-dependent networks, link streams, and dynamic graphs.

Note that if we just remove all time information from the temporal graph (and collapse, if necessary, multiple edges between any two vertices into a single edge), we clearly loose all the

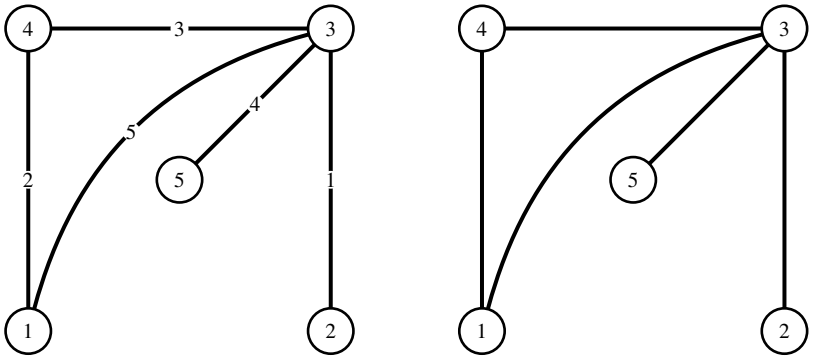


Figure 6.1: An example of a temporal graph with 5 nodes and 5 temporal edges (left), and the corresponding "non-temporal" graph in which edge temporal labels have been removed (right).

temporal information of the graph. This loss can be critical to the understanding of the reachability relationships between the nodes of the graph, and it can even lead us to deduce wrong consequences. For example, in the left part of Figure 6.1 a temporal graph with five nodes and five temporal arcs is represented, while in the right part of the figure the "non-temporal" version of the graph (in which all arc temporal labels have been removed) is shown. It is easy to verify that the two simple paths from node 1 to node 2 in the non-temporal graph do not exist in the temporal graph. Indeed, the edge from node 3 to node 2 appears at time 1, hence it cannot be used within a path starting from node 1, since all this node's edges appear after that time. Moreover, the path of length 2 from node 1 to node 5 in the non-temporal graph does not exist in the temporal graph since it is only possible to reach node 3 from node 1 in one step at time 5, while the edge from node 3 to node 5 appears at time 4. In summary, removing temporal information may let us conclude that two nodes (i.e. 1 and 2) are reachable, while they are not, or that the length of the shortest path between two nodes (i.e. 1 and 5) is smaller than it really is.

In epidemiological modeling it is common to analyze the impact of the topology of the contact graph on the dynamics of infections, in order to design new vaccination strategies and to forecast and mitigate the impact of global epidemics in the future. Not taking into account the temporal information of the contact data can lead us toward wrong estimates of the disease diffusion, since, as a consequence of a factitious transitivity, the number of paths relevant for the disease spreading can be strongly overestimated. The availability of data with high temporal resolution offers a promising way to improve epidemiological models, by consider only infection paths, with links that appear in the correct temporal order. Taking into account these time-respecting paths overcomes the limitation inherent to the static approach for realistic investigations of disease transmission.

6.2 Basic definitions

As we already said, different definitions of temporal graphs have been given in the literature. The simplest one is, maybe, the definition of *evolving graphs* [8], which are just sequences of graphs (one graph for each time instant). For example, in Figure 6.2 we show an evolving graph, which consists of three graphs at three different time instant. The first graph contains the two arcs (u_1,u_2) and (u_3,u_4), the second one contains the two arcs (u_1,u_4) and (u_2,u_4), and the third one contains the two arcs (u_1,u_2) and (u_2,u_4). It should be clear that, even if in the union of these three graphs, there exists a path from u_1 to u_3, passing through u_4, this path is not temporally feasible since the arc from u_1 to u_4 appears only after the appearance of the from u_4 to u_3.

Another very general definition of temporal graphs is the notion of time-dependent network [33]. A *time-dependent network* (in short, *TDN*) is a graph $G = (V,E)$ in which the delay or travel time of each edge changes over time. Typically, the dependence on time of the delay is specified by

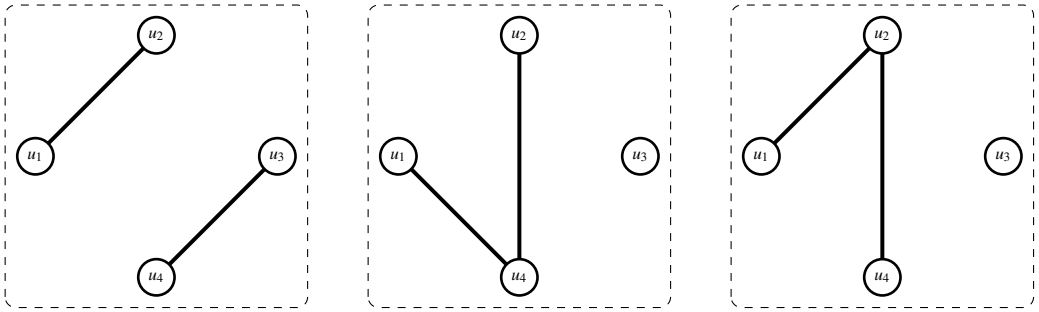


Figure 6.2: An example of an evolving graph with 4 nodes, which is a sequence of three different graphs at three different time instants.

associating to each edge $e = (u,v) \in E \subseteq V \times V$ a function $\alpha_e(t)$ which indicates, for each time t, the *arrival time* in v when the edge is traversed starting from u at time t (see, for example, the time-dependent network shown in Figure 6.3). Note that, once the arrival time function is specified, the *delay* (or travel time) of an edge e at time t can be easily computed as $\delta_e(t) = \alpha_e(t) - t$ (since we cannot yet travel back in time, this implies that $\alpha_e(t)$ has to be no smaller than t). Equivalently, arrival time functions can be easily obtained from delay functions. This general definition has been refined in several different ways in the last 30 years, by assuming different properties of the edge arrival time functions. In particular, we can identify the following hierarchy of TDN models, where each model encompasses the next one.

Piecewise linear TDN For each edge e, the function α_e is piecewise linear, like, for example, the functions $\alpha_{e_2}(t)$ and $\alpha_{e_3}(t)$ in Figure 6.3 [39].

Constant-delay TDN These are piecewise linear TDNs in which, for each edge, the slope of all linear segments of the corresponding arrival time function is equal to 1, like, for example, the function $\alpha_{e_3}(t)$ in Figure 6.3 [29].

Point-availability TDN These are constant-delay TDNs in which the domain of the arrival time functions is a finite subset $\mathbb{T}$ of the set of real numbers [96]. A commonly used representation of a point-availability TDN simply consists in listing all the quadruples (u,v,τ,δ), such that the arrival time function of the edge (u,v) at time τ is equal to $\tau + \delta$ (see the two commonly used visualizations of such representation shown in Figure 6.4).

Uniform TDN These are point-availability TDNs in which the delay is the same value δ for all edges and all time instants. For example, *temporal graphs* are uniform TDNs in which $\delta = 1$ and $\mathbb{T} \subseteq \mathbb{N}$ [23, 24, 67], while finite *link streams* are uniform TDNs in which $\delta = 0$ [58].

In this chapter, we will focus on uniform TDN, and, in particular, on temporal graphs. We conclude this short introduction to the different models introduced in the literature by mentioning the so-called *dynamic graphs*, that is, evolving graphs in which the sequence of graphs is determined by a specific probabilistic process. For example, an *edge-Markovian graph* [22] is a dynamic graph such that the graph G_{t+1} at time $t+1$ is generated, starting from the graph G_t at time t, according to the following stochastic process. For any arc (u,v) in G_t, the very same arc belongs to G_{t+1} with probability $1 - p_d$, where p_d with $0 < p_d < 1$ is the *arc death probability*. Moreover, for any pair of nodes u and v such that the arc (u,v) is not in G_t, the very same arc belongs to G_{t+1} with probability p_b, where p_b with $0 < p_b < 1$ is the *arc birth probability*.

6.2.1 Temporal graphs, static expansion, walks, and journeys

Formally, a *temporal graph* is a pair $\mathbb{G} = (V, \mathbb{E})$, where V is the set of nodes, and $\mathbb{E}$ is the set of *temporal arcs*. Each temporal arc is a quadruple $e = (u,v,t)$, where $u,v \in V$ and $t \in \mathbb{N}$ is the *appearing time* of e (in the following we will denote by $t(e)$ the appearing time of an arc e).

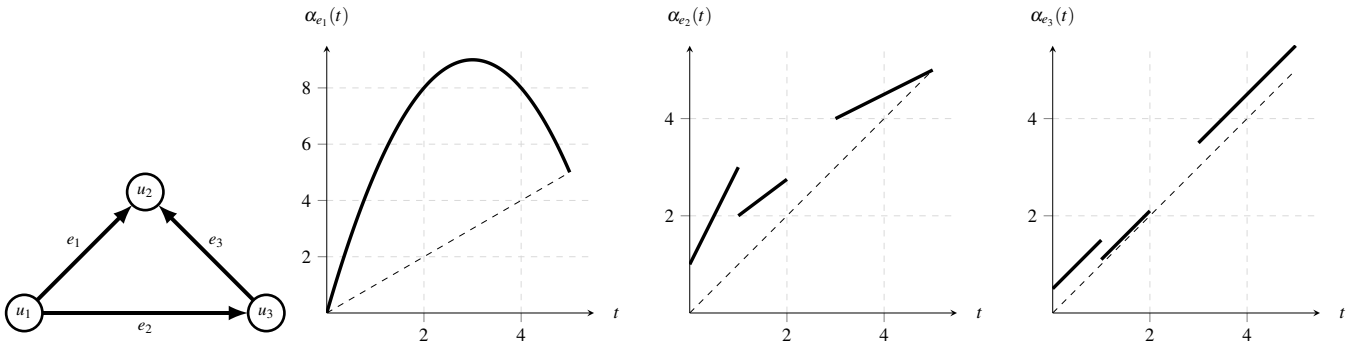


Figure 6.3: An example of time-dependent network. In this example, $\alpha_{e_1}(t) = -t^2 + 6t$. The function $\alpha_{e_2}(t)$ is piecewise linear (with segments $2t+1$ in $[0,1]$, $\frac{3}{4}t + \frac{5}{4}$ in $(1,2]$, and $\frac{1}{2}t + \frac{5}{2}$ in $[3,5]$), while the function $\alpha_{e_3}(t)$ is piecewise linear and constant-delay (with delay $\frac{1}{2}$ in $[0,1]$, $\frac{1}{10}$ in $(1,2]$, and $\frac{1}{2}$ in $[3,5]$).

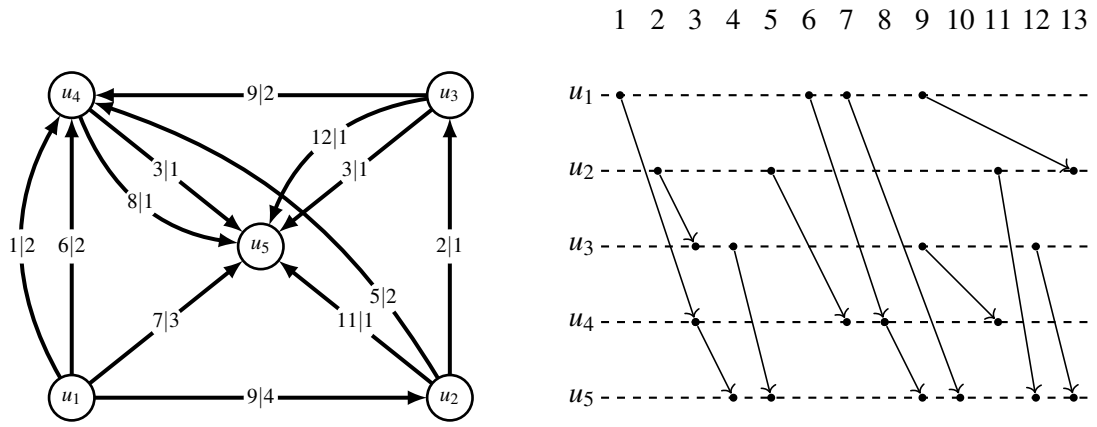


Figure 6.4: Two visualizations of an example of point-availability TDN. In the visualization on the left, each edge is labeled with its availability time and its delay. In the second visualization, the delay of each edge can be computed as the difference between its starting time and its arrival time.

Moreover, we define $t_\alpha = \min\{t(e) : e \in \mathbb{E}\}$ and $t_\omega = \max\{t(e) : e \in \mathbb{E}\}$. The *lifetime* of a temporal graph $\mathbb{G}$ is defined as $\mathbb{T} = [t_\alpha, t_\omega]$. Finally, for each $t \in \mathbb{T}$, the *snapshot* G_t of $\mathbb{G}$ at time t is defined as the graph $G_t = (V, E_t)$, where $E_t = \{(u,v) : (u,v,t) \in \mathbb{E}\}$. For example, let us consider the temporal graph shown in the left part of Figure 6.5. This temporal graph contains eight temporal arcs. Moreover, in this case $t_\alpha = 1$ and $t_\omega = 5$: hence, the lifetime is $\mathbb{T} = [1,5]$. Finally, the five snapshots of the temporal graph are shown in the right part of the figure.

Any temporal graph can be transformed into a graph by making a copy of each node at each time instant in the lifetime of the temporal graph, and by replacing each temporal arc with an arc connecting the corresponding two copies of the two nodes. Formally, given a temporal graph $\mathbb{G} = (V = \{v_1, \ldots, v_n\}, \mathbb{E})$, the *static expansion* of $\mathbb{G}$ is the graph $G = (W, E)$, where $W = \{v_{i,t} : v_i \in V \wedge t_\alpha \leq t \leq t_\omega + 1\}$, and $F = \{(v_{i,t}, u_{j,t+1}) : t_\alpha \leq t \leq t_\omega \wedge (i = j \vee (v_i, v_j, t) \in \mathbb{E})\}$. For example, the static expansion of the temporal graph, shown in the left part of Figure 6.5, is shown in Figure 6.6. Note that, since we have a copy of a node for each time instant in the lifetime of the temporal graph, the number of nodes in the static expansion is equal to $|V| \times \mathbb{T}$ (even if we can avoid including the copies corresponding to time instants t for which the corresponding snapshot has no arc, that is, for which $E_t = \emptyset$). The number of arcs in the static expansion is then $O(|V| \times \mathbb{T})$, which can be much larger than $|\mathbb{E}|$. More space efficient static representations of a temporal graph can be defined,

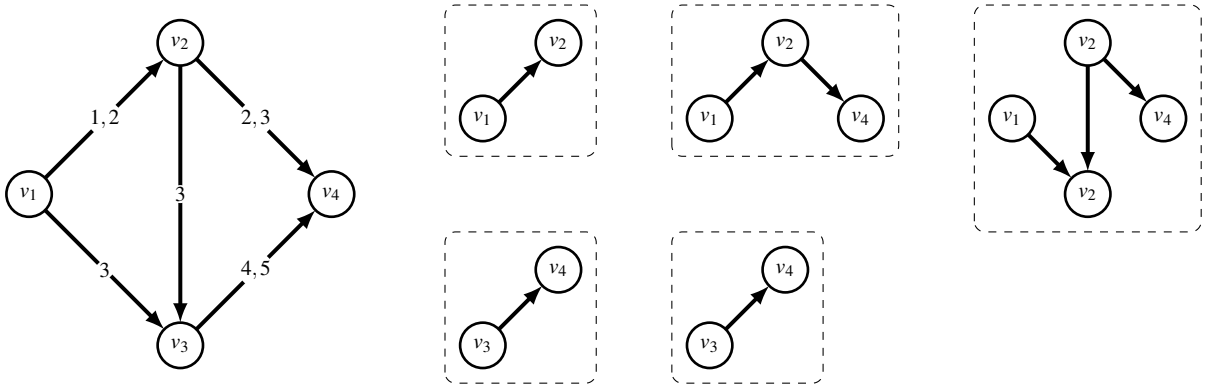


Figure 6.5: A temporal graph (left) and the corresponding five snapshots at time 1, 2, 3, 4, and 5 (right).

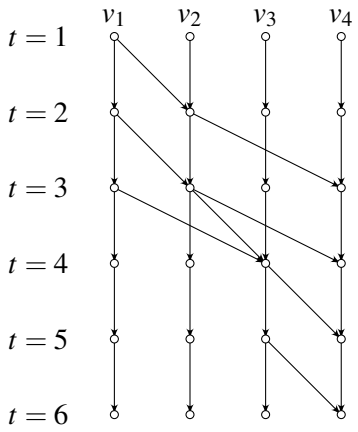


Figure 6.6: The static expansion of the temporal graph shown in the left part of Figure 6.5.

but they are out of the scope of this chapter, which is more focused on directly dealing with the temporal dimension of a graph.

Given a temporal graph $\mathbb{G} = (V, \mathbb{E})$ and two nodes $u, v \in V$, a *walk* $\mathbb{W}$ from u to v is a sequence of temporal arcs $e_1 = (u_1, v_1, t_1), \ldots, e_k = (u_k, v_k, t_k)$ in $\mathbb{E}$ such that $u = u_1$, $v = v_k$, and, for each i with $1 < i \leq k$, $u_i = v_{i-1}$ and $\tau_i > \tau_{i-1}$. The *starting time* of $\mathbb{W}$ is defined as t_1, while its *arrival time* is defined as t_k. Moreover, the *duration* of $\mathbb{W}$ is defined as $t_k - t_1 + 1$. Note that, according to this definition, it is possible to wait at a node as much as we want until some temporal arc appears and allows us to leave the node. A walk is called a *journey* if the nodes in the walk are all distinct.

6.3 Computing foremost journeys and node distances

Given a temporal graph $\mathbb{G} = (V, \mathbb{E})$, two nodes $u, v \in V$, and a time instant $t \in \mathbb{T}$, a journey $\mathbb{J}$ from u to v is a *t-foremost* journey if its starting time is at least t and its arrival time is the minimum arrival time among all the journeys from u to v starting no earlier than t. The (foremost) *t-distance* of node v from node u (denoted as $d_t(u, v)$) is then defined as the minimum duration of any t-foremost journey from u to v (if there is no t-foremost journey from u to v, we assume that $d_t(u, v) = \infty$). Let us consider, for example, the temporal graph shown in the left part of Figure 6.5. For any two nodes v_i and v_j, the t-distance from u to v is shown in the following tables, for $t = 1, 2, 3$.

$t=1$	v_1	v_2	v_3	v_4
v_1		1	3	2
v_2	∞		3	2
v_3	∞	∞		4
v_4	∞	∞	∞	

$t=2$	v_1	v_2	v_3	v_4
v_1		1	2	2
v_2	∞		2	1
v_3	∞	∞		3
v_4	∞	∞	∞	

$t=3$	v_1	v_2	v_3	v_4
v_1		∞	1	2
v_2	∞		1	1
v_3	∞	∞		2
v_4	∞	∞	∞	

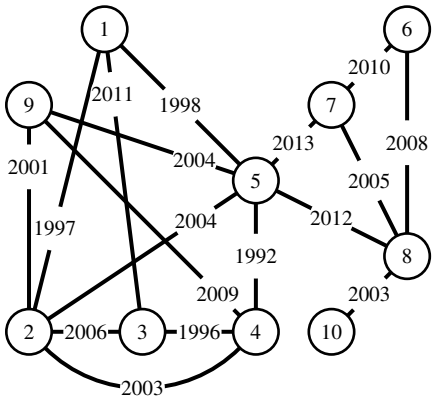
For example, the 1-distance from v_1 to v_2 is equal to 1, since there is a one-arc journey starting from v_1 at time 1, whose duration is, hence, $d_1(v_1,v_2) = 1-1+1 = 1$. For a similar reason, the 2-distance from v_1 to v_2 is equal to 1. On the contrary, the 3-distance from v_1 to v_2 is equal to ∞, since there is no journey from v_1 to v_2 starting no earlier than t (we can arrive at node v_3 and v_4, but then there is no temporal arc allowing us to go to node v_2). Analogously, the 1-distance from v_1 to v_3 is equal to 3, since a possible 1-foremost journey from v_1 to v_3 is the journey passing through v_2, which arrives at time 3 (indeed, there is also a one-arc journey which arrives at the same time): hence, $d_1(v_1,v_3) = 3-1+1 = 3$. Analogously, we have that $d_2(v_1,v_3) = 3-2+1 = 2$. Finally, $d_3(v_1,v_3) = 3-3+1 = 3$ because of the one-arc journey from v_1 to v_3. In a similar way, we can compute all the other distances shown in the previous tables.

Computing the t-distance from one source node s to all other nodes of a temporal graph can be done by appropriately modifying the BFS algorithm and by assuming that the temporal graph is available as a list of its temporal arcs, sorted in non-decreasing way with respect to their appearing time. For instance, let us consider the IMDB graph introduced in the first chapter, where to each arc of the graph we associate the year of the collaboration between the two actors (as we have specified in the first chapter). The list of the temporal arcs of this temporal graph, sorted in non-decreasing way with respect to their appearing time, is the following one (see also Figure 6.7): $(4,5,1992)$, $(3,4,1996)$, $(1,2,1997)$, $(1,5,1998)$, $(2,9,2001)$, $(2,4,2003)$, $(8,10,2003)$, $(5,9,2004)$, $(2,5,2004)$, $(7,8,2005)$, $(2,3,2006)$, $(6,8,2008)$, $(4,9,2009)$, $(6,7,2010)$, $(1,3,2011)$, $(5,8,2012)$, and $(5,7,2013)$.

By reading the list of temporal arcs one arc after the other, the t-distance from the source node s to all other nodes of a temporal graph can be computed by the following algorithm (where $\mathbf{t}$ denotes an array storing, for each node x, the arrival time of a t-foremost journey from s to x) [96].

1. We set $\mathbf{t}[s] = t-1$ and $\mathbf{t}[v] = \infty$ for all $v \neq s$.
2. For each edge $e = (u,v,t_e)$, if $t \leq t_e \wedge \mathbf{t}[u] < t_e < \mathbf{t}[v]$, then we set $\mathbf{t}[v] = t_e$. If the graph is not directed and if $t \leq t_e \wedge \mathbf{t}[v] < t_e < \mathbf{t}[u]$, then we set $\mathbf{t}[u] = t_e$.
3. We return $\mathbf{t} - (t-1)$

Clearly, the time complexity of this algorithm is proportional to $|\mathbb{E}|$, since each edge is read only once. The space complexity is, instead, linear in the number of nodes, if the temporal graph is



1: Ben Affleck
2: Matt Damon
3: Leonardo Di Caprio
4: Meryl Streep
5: Bruce Willis
6: Kristin Chenoweth
7: Dwayne Johnson
8: Vince Vaughn
9: George Clooney
10: Adam Sandler

Figure 6.7: The temporal version of a very small IMDB graph.

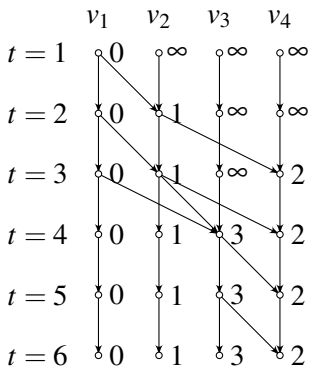


Figure 6.8: The execution of the BFS-like algorithm, for computing the 1-distances from node v_1 to all other nodes in the temporal graph shown in the left part of Figure 6.5.

stored in secondary memory.

Let us apply the above algorithm to the temporal IMDB graph shown in Figure 6.7, in order to compute the 2000-distance of Ben Affleck from all the other actors. Since the first four arcs appears before 2000, they are ignored. The next 10 arcs are also ignored, since their appearing time is not greater than the values in $\mathbf{t}$ corresponding to the nodes connected by the arc (these values are all equal to ∞). When we read the arc $(1,3,2011)$, since $2011 > \mathbf{t}[1] = 1999$ and since $2011 < \mathbf{t}[3] = \infty$, we set $\mathbf{t}[3] = 2011$. The last two arcs are ignored, since their appearing time is not greater than the values in $\mathbf{t}$ corresponding to the nodes connected by the arc (these values are all equal to ∞). In conclusion, we have that the 2000-distance from Ben Affleck to Leonardo Di Caprio is equal to $2011 - 2000 + 1 = 12$, while the 2000-distance to all the other nodes is ∞.

We said that the above algorithm is similar to the BFS algorithm. In order to further justify this statement, let us see how this algorithm executes on the static expansion of the temporal graph shown in the left part of Figure 6.5, whose ordered list of its temporal arcs is the following one: $(v_1, v_2, 1)$, $(v_1, v_2, 2)$, $(v_2, v_4, 2)$, $(v_1, v_3, 3x)$, $(v_2, v_3, 3)$, $(v_2, v_4, 3)$, $(v_3, v_4, 4)$, and $(v_3, v_4, 5)$. By executing the algorithm on this list of temporal arcs with $s = v_1$ and $t = 1$, the values assigned to the nodes, which corresponds to the arrival time of a 1-foremost journey from node v_1, are shown at the right of each node in Figure 6.8. As it can be seen from the figure, the nodes whose corresponding value is less than ∞ are, indeed, the nodes reached by the BFS started from node $v_{1,1}$ (that is, the copy of node v_1 at time 1). Hence, the only modification we have to do at the BFS in order to compute the arrival times is that, when a node u is inserted in the queue by a node $v_{t,i}$, then its arrival time is set equal to t, if $u \neq v_{t+1,i}$, and it is set equal to the arrival time of $v_{t,i}$ otherwise.

The following function implements the algorithm for computing the t-distance from a specified source node to all other nodes (we assume that the ordered list of the temporal arcs is stored in a specified text file).

```
function t_distance(fn::String, isdir::Bool, s::Int64,
                n::Int64, t::Int64)::Array{Int64}
    distance::Array{Int64} = fill(typemax(Int64), n)
    distance[s] = t - 1
    for line in eachline(fn)
        uvt::Array{String} = split(line, " ")
        u::Int64 = parse(Int64, uvt[1])
        v::Int64 = parse(Int64, uvt[2])
        te::Int64 = parse(Int64, uvt[3])
```

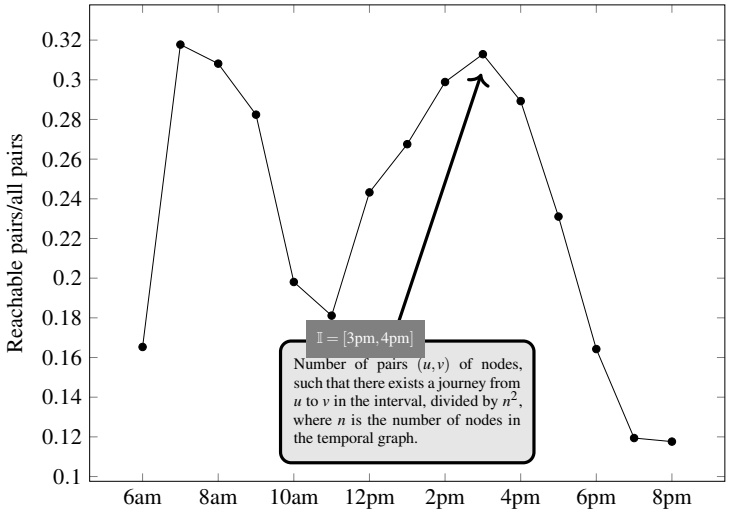


Figure 6.9: The reachability diagram of the public transport network of the city of Kuopio, in which the interval between 6am and 9pm has been split into 15 intervals of one hour each.

```
        if (te >= t)
            if (te > distance[u] && te < distance[v])
                distance[v] = te
            end
            if (!isdir)
                if (te > distance[v] && te < distance[u])
                    distance[u] = te
                end
            end
        end
    end
    return distance .- t .+ 1
end
```

6.3.1 An example: the public transportation graph of Kuopio

Kuopio is a Finnish city located in the south of Finland, with a population of 120246, which makes it the eighth-most populous city in Finland [90]. In this section we make use of a Kuopio's public transport network, which contains 549 stops (that is, nodes of a temporal graph) and 32122 temporal arcs (the data were collected on December 12, 2016) [23, 53]. The radius around the city's central point that should cover all the continuous and dense parts of the city and its public transport network is approximately ten kilometers.

Observe that, in the case of transport networks, going from one stop to another, i.e. following an arc, usually takes some time. Therefore, the temporal arc list usually contains the starting time s_e and the arrival time a_e of each temporal arc e between two nodes u and v (think about the typical timetable of a bus line). In order to accommodate such data to our definition of temporal graph, we can associate to the temporal arc e an appearing time equal to a_e. In this way, we are increasing the reachability efficacy of the network, since we allow us to use the temporal arc e even if we arrive at node u after the time s_e. But this improvement should happen systematically for each transport network and it should not have significant consequence, if our goal is the comparative analysis of the efficacy of different networks.

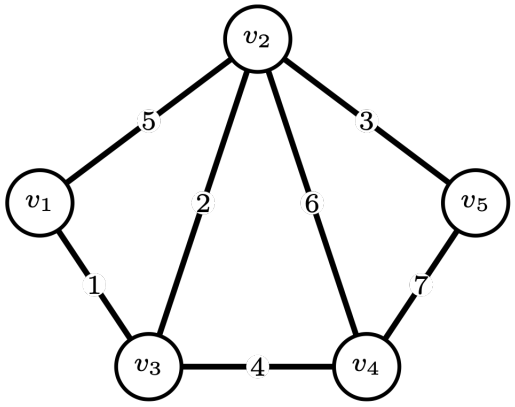


Figure 6.10: An example of temporal graph for which the Menger's theorem is false.

Our goal is, indeed, to analyze the *reachability efficacy* of the Kuopio's public transport network. To this aim, we compute the size of the temporal neighborhood corresponding to different time intervals of the day in which the data were collected. Given a time interval $\mathbb{I} = [t_\alpha, t_\omega]$, the *temporal neighborhood* corresponding to $\mathbb{I}$ is the number of pairs of nodes u and v such that $d_{t_\alpha}(u,v) < t_\omega - t_\alpha + 1$ (that is, there is a journey from u to v which starts no earlier than t_α and arrives in u no later than t_ω). We then divide the interval between 6am and 9pm into 15 intervals of length 60 minutes. For each interval $\mathbb{I} = [t_\alpha, t_\omega]$, we compute the size of the temporal neighborhood corresponding to $\mathbb{I}$ (normalized with respect to the number of all possible pairs of nodes), by using the algorithm for computing the t-distance from a specified node, suitably modified in order to ignore all arcs whose appearing time is later than t_ω. We can represent the results of these computations by the *reachability diagram* shown in Figure 6.9. We can observe how the fraction of pairs of nodes which are reachable one from the other is relatively small, never exceeding one third of the total number of pairs. It is also quite evident that the reachability efficacy of the network improves in correspondence with two time intervals (from 7am to 8am and from 3pm to 4pm), which probably correspond to peak times of the transport network. Finally, the reachability efficacy of the network drastically decreases after 5pm.

6.4 Graphs versus temporal graphs

In this section, we will show how the temporal dimension of temporal graphs makes some well-known results in graph theory not being true anymore.

6.4.1 Menger's theorem and temporal graphs

One very celebrated result in graph theory is the so-called *Menger's theorem*, which states that, for any graph $G = (V,E)$ and for any two nodes s and t in V such that $(s,t) \notin E$, the minimum number of vertices, distinct from s and t, whose removal disconnects s and t is equal to the maximum number of pairwise node-disjoint paths from s to t. This theorem is a generalization of the maybe more famous max-flow min-cut theorem, which is typically taught in an advanced course on algorithms and complexity.

It is not difficult to show that Menger's theorem does not hold in the case of temporal graphs [50]. Let us consider, for example, the temporal graph shown in Figure 6.10, and let $s = v_1$ and $t = v_5$. Note that every journey from v_1 to v_5 includes two nodes among v_2,v_3, and v_4 (that are called *inner* nodes). Hence, there exist no two node-disjoint journeys from v_1 to v_5. On the other hand, any two nodes in $\{v_2, v_3, v_4\}$ belong to a journey from v_1 to v_5. This implies that, in order to disconnect

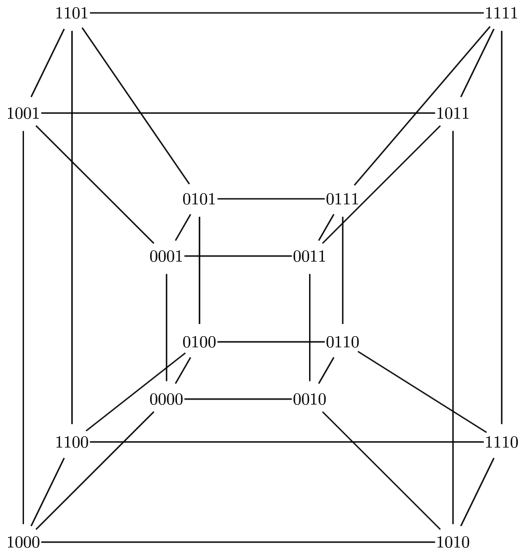


Figure 6.11: An example of a temporal hypercube with $k = 4$ (each temporal arc has appearing time equal to the bit index in which the labels of the two nodes differ): no temporal arc can be removed without disconnecting at least two nodes.

these two nodes, we have to remove at least two nodes. Note that this example can be generalized to an infinite family of graphs, by considering $2k-1$ inner nodes for any $k > 0$, and by showing that every journey from v_1 to v_{2k+1} visits at least k of these inner nodes (thus ensuring that there are no two node-disjoint journeys from v_1 to v_{2k+1}), and that there is a journey through any set of k inner nodes (thus ensuring that every "separator" of v_1 and v_{2k+1} must have size at least k). Two different ways have been proposed to overcome this negative result. On the one hand, it has been proved that the violation of the Menger's theorem does not hold if we replace node-disjointness by arc-disjointness and node removals by arc removals [7]. On the other hand, a natural analogue of Menger's theorem that is valid for all temporal networks has been proved [66].

6.4.2 Spanners in temporal graphs

A connected subgraph of a graph G is said to be a *spanner* of G if it contains all the nodes of G. If G is connected, then it is easy to show that there exists a spanner of G which contains $n-1$ arcs, where n is the number of nodes. Indeed, such a spanner is simply the tree obtained by executing a BFS starting from any node of G. Let us now show that this result is not true in the case of temporal graphs [50].

First of all, notice that a *temporal clique* in which any temporal arc appears at time 1 is clearly *temporally connected*, since, for any pair of nodes u and v, there exists always a journey from u to v simply formed by the temporal arc $(u,v,1)$. However, removing even one temporal arc from the temporal clique makes the resulting temporal graph disconnected. Indeed, assume that we remove the arc $(u,v,1)$ for some two nodes u and v. Then, there is no journey from u to v since all arcs have the same appearing time (that is, 1) and, hence, any journey is formed by one arc only. In other words, any spanner of the temporal clique, in which any temporal arc appears at time 1, has to include $O(n^2)$ temporal arcs.

Another interesting example is the *temporal hypercube* with $n = 2^k$ nodes, for any $k > 0$ (see Figure 6.11). In such a temporal graph, each node is labeled by a distinct binary string of length k. Given two nodes u and v, with labels $b_1 \cdots b_k$ and $c_1 \cdots c_k$, respectively, there exists the temporal arc (u,v,h) if and only $b_h \neq c_h$ and $b_i = c_i$ for any $i \neq h$ (that is, the two labels differ in one bit only

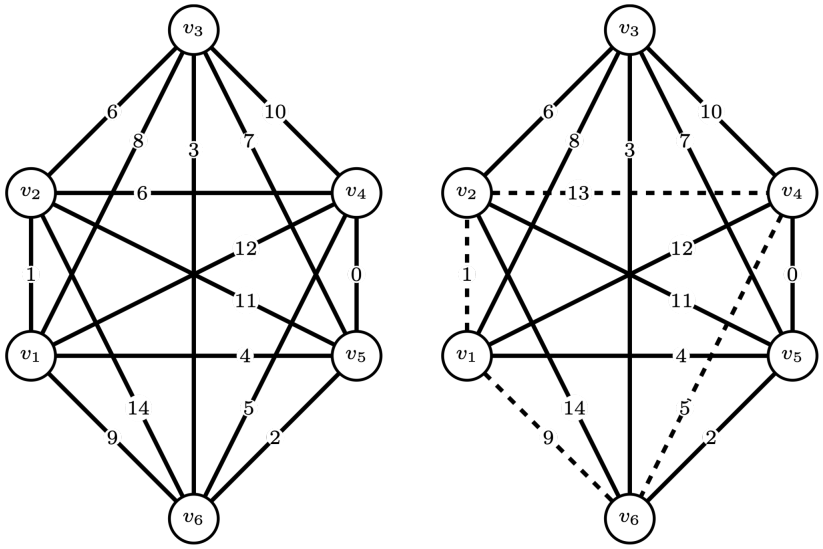


Figure 6.12: An example of a temporal clique for which a spanner exists with 4 edges less.

and the appearing time of the arc is equal to the index of this bit). Clearly, this temporal hypercube is connected. Indeed, for any two nodes u and v, there exist a journey from u to v which consists of the temporal arcs whose appearing times are the indices (in increasing order) of the bits in which the labels of u and v differ. For example, by referring to the temporal hypercube of Figure 6.11, the journey from the node with label 0101 to the node with label 1011 is formed by the following temporal arcs: $(0101,1101,1)$, $(1101,1001,2)$, and $(1001,1011,2)$. Differently from the temporal clique, the temporal hypercube is almost sparse, since the number of temporal arcs is equal to $\frac{nk}{2} = \frac{n\log n}{2}$. Even in this case, however, removing one temporal arc results in a temporal graph which is not connected. Indeed, if we remove, for example, the temporal arc (u,v,h), where the label of u is $b_1 \cdots b_k$, the label of v is $c_1 \cdots c_k$, $b_h \neq c_h$, and $b_i = c_i$ for any $i \neq h$, then there is no journey from u to v. Indeed, any journey from u to v cannot start with a temporal arc with appearing time $l \neq h$ (thus entering a node whose label has a bit in position l different from b_l), since otherwise it could never again traverse a temporal arc whose appearing time is l (thus restoring the correct value of the bit in position l). Hence, the temporal hypercube does not admit a "sparser" spanner.

Going back to temporal cliques, one natural question is whether a linear size spanner exists, if the appearing times of the temporal arcs are all different. We have seen before that all the appearing times are equal, then not even a temporal arc can be removed. On the contrary, if the appearing times are distinct (as in the case of the temporal clique shown in the left part of Figure 6.12), it might be possible to remove temporal arcs without loosing the connectivity property. For example, in the right part of the figure, we have removed the temporal arcs $(v_1,v_2,1)$, $(v_1,v_6,9)$, $(v_2,v_4,13)$, and $(v_4,v_6,5)$. In the resulting temporal graph, there still exists a journey from v_1 to v_2 consisting of the temporal arcs $(v_1,v_5,4)$ and $(v_5,v_2,11)$, and a journey from v_2 to v_1 consisting of the temporal arcs $(v_2,v_3,6)$ and $(v_3,v_1,8)$. It is easy to verify that the same is true for the other temporal arcs that have been removed. The question is then whether a temporal clique with distinct appearing times of its temporal arcs always admits a spanner with a number of temporal arcs linear in the number n of nodes. This question is still open: the best result obtained so far is that a spanner with $O(n\log n)$ temporal arcs always exists (the proof of this result is based on a clever usage of delegation of emissions and receptions to neighbors and on a recursive sparsification technique) [19].

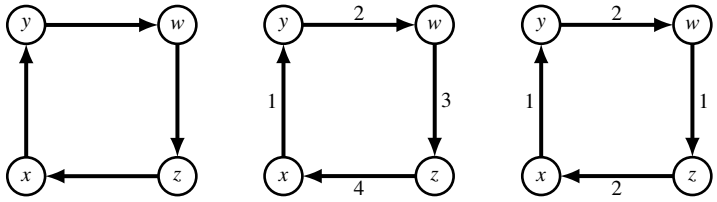


Figure 6.13: A directed graph G (left), and two edge temporalisations of D (the edge labels denote the appearing time of the edge). The temporal reachability of the first temporal graph is 13, while the temporal reachability of the second temporal graph is 13.

6.5 Making graphs temporal

A typical question that has been considered in the literature in the field of *network optimization* consists in finding the graph modification which maximizes or minimizes a specific optimization criterion (often connected to the notion of reachability and distance between nodes). For example, one well studied problem has been the following one: given a graph and given a budget k, find a set of k edges, which, when added to the graph, minimize the diameter of the new graph [31]. Other *graph operations* that have been studied are node or edge deletions or edge contractions, while other *optimization criteria* are reachability, that is, the number of pairs of nodes (u,v) such that u can reach v, information diffusion, that is, the number of nodes reachable from a specific set of source nodes, or centrality measures, such as the closeness and the betweenness centrality of a node.

In this last section of this chapter, we propose a network optimization problem which is related to the notion of reachability in temporal graphs. In such optimization problem, the graph operation will allow us to transform a directed graph into a temporal graph, while the optimization criterion will be the temporal reachability of a temporal graph, that is, the number of pairs of nodes u and v such that there exists a journey from u to v. The graph operation is the *edge temporalisation*. Given a directed graph G, this operation simply assigns to each arc (u,v) of G an appearing time t, making it a temporal edge (u,v,t). For example, the directed graph G could represent the connections between airports of an air company. An edge temporalisation of G would then correspond to an assignment of the leaving time of each flight, and the goal would be to connect as many pairs of cities as possible (clearly, the travel time of each flight should also be taken into account, and other optimization criteria would be interesting for this application, such as minimizing the average duration of all indirect connections). As an example, let us consider the directed multigraph G shown in the left part of Figure 6.13. In the middle and right parts of the figure, we show two edge temporalisations of G (the edge labels denote the appearing times assigned to the edges).

Note that, in the temporal graph in the middle part of the figure, node x can reach, besides itself, nodes y, w, and z, node y can reach, besides itself, nodes w, z, and x, node w can reach, besides itself, nodes z and x, and node z can reach, besides itself, nodes x. The temporal reachability, in this case, is equal to $4+4+3+2=13$. In the temporal graph in the right part of the figure, instead, node x can reach, besides itself, nodes y and w, node y can reach, besides itself, node w, node w can reach, besides itself, nodes z and x, and node z can reach, besides itself, nodes x. The temporal reachability, in this case, is equal to $3+2+3+2=10$. Hence, the first edge temporalisation is better than the second one.

The network optimization problem we study in this section is then the following one. Given a directed graph G, we look for an edge temporalisation which maximizes the temporal reachability of the resulting temporal graph. For example, in the case of the directed graph shown in the left part of Figure 6.13, it is easy to verify that the edge temporalisation shown in the middle part of the figure is an optimal one.

In the following, we will refer to a particular kind of edge temporalisations. Given a directed graph G with edge set E, a *schedule* S is an ordering of its edges. Note that a schedule S immediately induces an edge temporalisation τ_S of G defined as follows. If $S = e_1, \ldots, e_{|E|}$, then $\tau_S(e_i) = i$, for $i \in [|E|]$, where, for any positive integer x, $[x]$ denotes the set of integers numbers between 1 and x. On the other hand, any edge temporalization induces a schedule, by simply ordering the edges according to the appearing times assigned to them and by breaking ties arbitrarily. Given a schedule S, a node v is said to be S-reachable from a node u if there exists a journey from u to v in the temporal graph induced by G and τ_S. The *S-reachability* of G is defined as the temporal reachability of τ_S.

6.5.1 Hardness of the edge temporalization problem

We now show that there is no polynomial-time algorithm solving the edge temporalization problem, unless the satisfiability problem can be solved in polynomial time (which is believed to be false [43]). Note that in [44], it is proved that deciding whether the maximum temporal reachability is equal to the square of the number of nodes is also unlikely to be solvable in polynomial time, in the setting of undirected graphs. Moreover, similar problems have been recently considered in [30, 36], with similar results. The following result has been proved by the author in collaboration with Filippo Brunelli and Laurent Viennot while exploring a variant of the edge temporalization problem, mostly inspired by the optimization of bus/metro/tramway schedules in a public transport network [18].

Let us consider a Boolean formula Φ in conjunctive normal form, with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$. Without loss of generality we will assume that each variable appears positive in at least one clause and negative in at least one clause (otherwise the value to be assigned to the variable is determined). We first define the directed graph G, with vertex set V and edge set E, as the union of the following "gadgets" (see Figure 6.14).

Variable gadgets For each variable x_i of Φ, V contains the nodes t_i^1, t_i^2, f_i^1,and f_i^2 and E contains the edges (t_i^1, f_i^2), (f_i^2, f_i^1), (f_i^1, t_i^2), and (t_i^2, t_i^1).

Clause gadgets For each clause c_j, V contains the nodes c_j^1 and c_j^2. If the literal x_i appears in c_j, E contains the edges (c_j^1, t_i^1) and (t_i^2, c_j^2), while if the literal $\neg x_i$ appears in c_j, E contains the edges (c_j^1, f_i^1) and (f_i^2, c_j^2). Moreover, for each two clauses c_h and c_j with $h \neq j$, E contains the edge (c_j^1, c_h^2) (see the dashed edges in the figure). Finally, for each clause c_j, V also contains the nodes d_j^i and e_j^i, for $i \in [K]$ (the value of K will be specified later in the proof), and E contains the edges (d_j^i, c_j^1) and (c_j^2, e_j^i), for $i \in [K]$.

Block gadget V contains the nodes u_1, u_2, u_3, and u_4, and the nodes b_i, for $i \in [M]$ (the value of M will be specified later in the proof). E contains the edges $\{(b_i, u_1) : i \in [M]\}$, (u_1, u_2), $\{(u_2, d_j^l) : j \in [m], l \in [K]\}$, $\{(e_j^l, u_3) : j \in [m], l \in [K]\}$, (u_3, u_4), and $\{(u_4, b_i) : i \in [M]\}$.

Note that G is strongly connected. Indeed, let us consider the cycles

$$C_{i,j,l,p} = \langle u_1, u_2, d_j^l, c_j^1, t_i^1, f_i^2, f_i^1, t_i^2, c_j^2, e_j^l, u_3, u_4, b_p, u_1 \rangle,$$

$$C_{i,j,l,p} = \langle u_1, u_2, d_j^l, c_j^1, f_i^1, t_i^2, t_i^1, f_i^2, c_j^2, e_j^l, u_3, u_4, b_p, u_1 \rangle,$$

where $j \in [m]$, i is such that $\neg x_i$ is a literal of the clause c_j, $l \in [K]$, and $p \in [M]$. The union of these cycles contains each node in V, and each of these cycles contains node u_1. This proves the strong connectivity of G.

In the following, B denotes the set $\{b_i : i \in [M]\}$ and H denotes the set of nodes which do not belong to the block gadget, that is, $H = V \setminus (\{u_1, u_2, u_3, u_4\} \cup B)$ (note that $|V| = M + |H| + 4$ and that $|H| = 2(K+1)m + 4n$).

Activation of pairs of nodes in a variable gadget Consider a variable x_i and the associated variable gadget and a schedule S of the 4 edges associated to the variable gadget. We say that

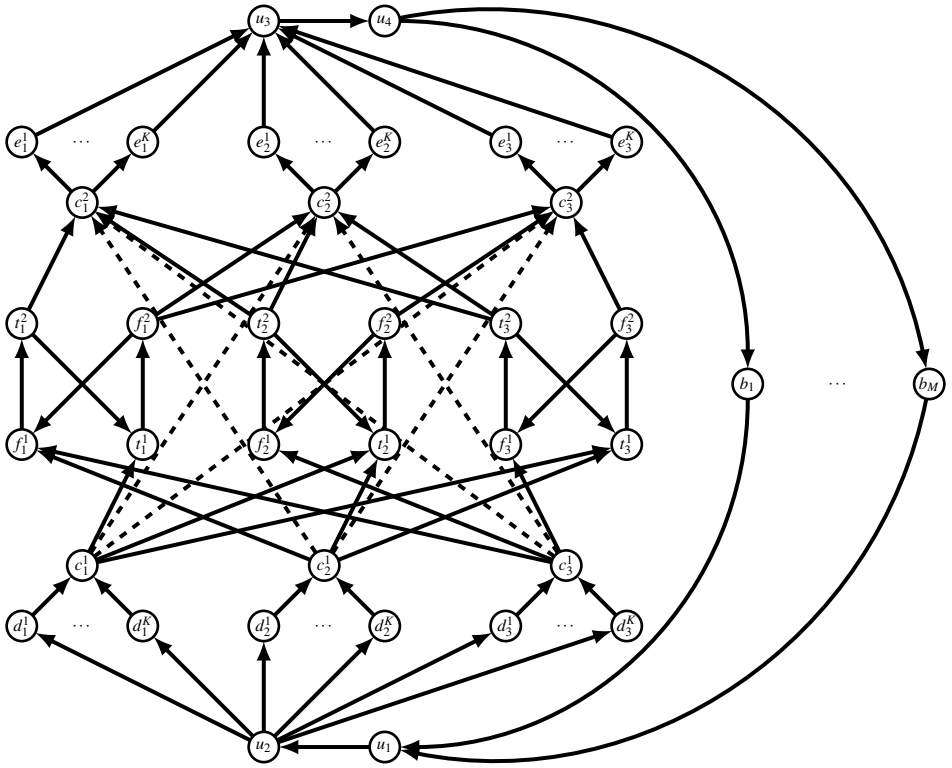


Figure 6.14: An example of the reduction of the satisfiability problem to the edge temporalization problem. The Boolean formula is $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$.

S activates the pair (t_i^1, t_i^2) (respectively, (f_i^1, f_i^2)) if t_i^2 (respectively, f_i^2) is S-reachable from t_i^1 (respectively, f_i^1) within the gadget, that is (t_i^1, f_i^2), (f_i^2, f_i^1), and (f_i^1, t_i^2) (respectively, (f_i^1, t_i^2), (t_i^2, t_i^1), (t_i^1, f_i^2)) are scheduled in that order. Note that no schedule can activate both (t_i^1, t_i^2) and (f_i^1, f_i^2) as the edge (t_i^1, f_i^2) is scheduled either before or after the edge (f_i^1, t_i^2).

Constructing a schedule from a satisfying assignment. Suppose that there is an assignment α that satisfies Φ, and let us consider the following schedule S. First we schedule the edges in $\{(b_i, u_1) : i \in [M]\}$ (in any arbitrary order), then the edge (u_1, u_2), and then the edges in $\{(u_2, d_j^i) : j \in [m], i \in [K]\}$ (in any arbitrary order). Then we schedule the edges $\{(d_j^i, c_j^1) : j \in [m], i \in [K]\}$ (in any arbitrary order), and, then, the edges going out from the nodes c_j^1, for $j \in [m]$ (in any arbitrary order). Then, for each $i \in [n]$, if $\alpha(x_i) = \text{TRUE}$, we schedule the edges (t_i^1, f_i^2), (f_i^2, f_i^1), (f_i^1, t_i^2), and (t_i^2, t_i^1) in this order (thus activating (t_i^1, t_i^2)). Otherwise (that is, $\alpha(x_i) = \text{FALSE}$), we schedule the edges (f_i^1, t_i^2), (t_i^2, t_i^1), (t_i^1, f_i^2), and (f_i^2, f_i^1) in this order (thus activating (f_i^1, f_i^2)). Then we schedule all the edges entering the nodes c_j^2, for $j \in [m]$ (in any arbitrary order), and then all the edges going out from the nodes c_j^2, for $j \in [m]$ (in any arbitrary order). Finally, we schedule the edges in $\{(e_j^l, u_3) : j \in [m], l \in [K]\}$ (in any arbitrary order), then the edge (u_3, u_4), and all the edges in $\{(u_4, b_i) : i \in [M]\}$ (in any arbitrary order).

First observe that, for any clause c_j with $j \in [m]$, there exists a literal that satisfies c_j according to the assignment α. Let x_i (respectively, $\neg x_i$) be a literal satisfying c_j. Since (t_i^1, t_i^2) (respectively, (f_i^1, f_i^2)) is activated, there exists a temporal path from c_j^1 to c_j^2 that goes through variable gadget corresponding to x_i. This means that c_j^2 is S-reachable from c_j^1 and that, for $l, l' \in [K]$, $e_j^{l'}$ is S-reachable from d_j^l. We now prove a lower bound $\mathbb{L}$ on the S-reachability by showing a lower bound

on the number of nodes temporally reachable from each possible source.

- For any $v \in V$ and for $i \in [M]$, v is S-reachable from b_i. This adds $M(M+|H|+4)$ to $\mathbb{L}$.
- For $i \in [4]$ and for $j \in [M]$, b_j is S-reachable from u_i. Moreover, all nodes in $\{u_1,u_2,u_3,u_4\} \cup H$ are S-reachable from u_1, all nodes in $\{u_2,u_3,u_4\} \cup H$ are S-reachable from u_2, u_3,u_4 are S-reachable from u_3, and u_4 is S-reachable from u_4. This adds $4M+2|H|+10$ to $\mathbb{L}$.
- For $j,h \in [m]$, $i,l \in [K]$, and $p \in [M]$, c_h^2, e_h^l are S-reachable from d_j^i (because of the above observation) and b_p is S-reachable from d_j^i. This adds $Km(M+Km+m)$ to $\mathbb{L}$.
- For $j,h \in [m]$, $l \in [K]$, and $i \in [M]$, c_h^2, e_h^l, b_i are S-reachable from c_j^1. This adds $m(M+Km+m)$ to $\mathbb{L}$.
- For $i \in [n]$, there exists $j \in [m]$ such that c_j is satisfied by $\alpha(x_i)$. Hence, for $p \in [2]$, $l \in [K]$, and $h \in [M]$, e_j^l, b_h are S-reachable from t_i^p and e_j^l, b_h are S-reachable from f_i^p. This adds $4n(M+K)$ to $\mathbb{L}$.
- For $j \in [m]$, $l \in [K]$, and $h \in [M]$, e_j^l, b_h are S-reachable from c_j^2. This adds $m(M+K)$ to $\mathbb{L}$.
- For $j \in [m]$, $l \in [K]$, and $h \in [M]$, b_h is S-reachable from e_j^l. This adds MKm to $\mathbb{L}$.

Thus, the S-reachability is at least

$$\begin{aligned}\mathbb{L} \;=\; & M(M+|H|+4)+(4M+2|H|+10)+Km(M+Km+m)\\ & +m(M+Km+m)+4n(M+K)+m(M+K)+MKm.\end{aligned}$$

Bounding reachability when Φ is not satisfiable. Let us set M equal to any value greater than $(|H|+5)^2$. We now prove that, if there exists no truth-assignment satisfying the formula Φ, then no schedule S can have S-reachability greater than or equal to $\mathbb{L}$. First notice that if S assigns to the edge (u_3,u_4) a starting time smaller than the starting time assigned to (u_1,u_2), then the S-reachability is less than $\mathbb{L}$. This is because, in this case, for $i,j \in [M]$ with $i \neq j$, b_j is not S-reachable from b_i. Hence, the S-reachability is bounded by $\mathbb{U}_1 = M(|H|+4+1)+(|H|+4)(M+|H|+4)$: this would happen if, for each node $v \notin B$, any node in V is S-reachable from v. Since $\mathbb{L} > M^2$, $\mathbb{U}_1 = M(|H|+4+1)+(|H|+4)(M+|H|+4) = 2M(|H|+4)+(|H|+4)^2+M < M(|H|+5)^2$, and $M > (|H|+5)^2$, it holds that $\mathbb{L} > \mathbb{U}_1$. We can then focus on schedules that assign to the edge (u_1,u_2) a starting time smaller than the starting time assigned to the edge (u_3,u_4). Let S be such a schedule and let G be the temporal graph induced by G and S. We now prove an upper bound $\mathbb{U}_2$ on the S-reachability by giving an upper bound on the nodes reachable from each possible source. Observe that, for any two nodes u and v, v is S-reachable from u only if in G there exists a walk from u to v that does not include the edge (u_3,u_4) before the edge (u_1,u_2).

- For $i \in [M]$, the number of nodes S-reachable from b_i is at most $|V|$. This adds $M(M+|H|+4)$ to $\mathbb{U}_2$.
- For $i \in [2]$, the number of nodes S-reachable from u_i is at most $|V|$, while the number of nodes S-reachable from u_3 is at most $M+3$ and the number of nodes S-reachable from u_4 is at most $|V| = M+|H|+4$.
- For $j \in [m]$ and $i \in [K]$, in the best case the set of nodes S-reachable from d_j^i contains d_j^i, c_j^1, the 12 nodes corresponding to the three variables appearing in c_j, and the nodes in $\{c_h^2 : h \in [m]\} \cup \{e_h^l : h \in [m], l \in [K]\} \cup \{u_1,u_3,u_4\} \cup B$/ Hence, the number of nodes S-reachable from d_j^i is at most $M+Km+m+17$. However, we can show that there exists an index j^* such that, for $l,l' \in [K]$, $e_{j^*}^{l'}$ is not S-reachable from $d_{j^*}^l$, implying that the d-nodes add at most $Km(M+Km+m+17)-K^2$ to $\mathbb{U}_2$. For defining j^*, we consider the following truth-assignment α: for any variable x_i with $i \in [n]$, $\alpha(x_i) = \text{TRUE}$ if (t_i^1, f_i^2) is scheduled before (f_i^1, t_i^2), otherwise $\alpha(x_i) = \text{FALSE}$. Note that if $\alpha(x_i) = \text{TRUE}$ (respectively, $\alpha(x_i) = \text{FALSE}$) we know that S does not activate (f_i^1, f_i^2) (respectively, (t_i^1, t_i^2)). Since the formula Φ is not satisfiable there exists $j^* \in [m]$ such that c_{j^*} is not satisfied by α. Let x_i (respectively, $\neg x_i$) be a literal in c_{j^*}. Since c_{j^*} is not satisfied by α, we that $\alpha(x_i) = \text{FALSE}$

(respectively, $\alpha(x_i) = \text{TRUE}$) and that (t_i^1, t_i^2) (respectively, (f_i^1, f_i^2)) is not activated. It is thus impossible to reach $c_{j^*}^2$ from $c_{j^*}^1$ through the variable gadget of x_i. On the other hand, in all the other walks in G that connect $c_{j^*}^1$ to $c_{j^*}^2$ the edge (u_3, u_4) appears before the edge (u_1, u_2). Hence, $c_{j^*}^2$ is not S-reachable from $c_{j^*}^1$ and $e_{j^*}^{l'}$ is not S-reachable from $d_{j^*}^l$ for $l, l' \in [K]$.

- For $j \in [m]$, in the best case the set of nodes S-reachable from c_j^1 contains c_j^1, the 12 nodes corresponding to the three variables appearing in clause c_j, and the nodes in $\{c_h^2 : h \in [m]\} \cup \{e_h^l : h \in [m], l \in [K]\} \cup \{u_1, u_3, u_4\} \cup B$. This adds $m(M + Km + m + 16)$ to $\mathbb{U}_2$.
- For $i \in [n]$ and for $j \in [2]$, in the best case the set of nodes S-reachable from t_i^j and from f_i^j contain the corresponding four variable nodes and the nodes in $\{c_h^2 : h \in [m]\} \cup \{e_h^l : h \in [m], l \in [K]\} \cup \{u_1, u_3, u_4\} \cup B$. This adds $4n(M + Km + m + 7)$ to $\mathbb{U}_2$.
- For $j \in [m]$, in the best case the set of nodes S-reachable from c_j^2 contains c_j^2 and the nodes in $\{e_j^l : l \in [K]\} \cup \{u_1, u_3, u_4\} \cup B$. This adds $m(M + K + 4)$ to $\mathbb{U}_2$.
- For $j \in [m]$ and $i \in [K]$, in the best case the set of nodes S-reachable from e_j^i contains e_j^i and the nodes in $\{u_1, u_3, u_4\} \cup B$. This adds with $Km(M + 4)$ to $\mathbb{U}_2$.

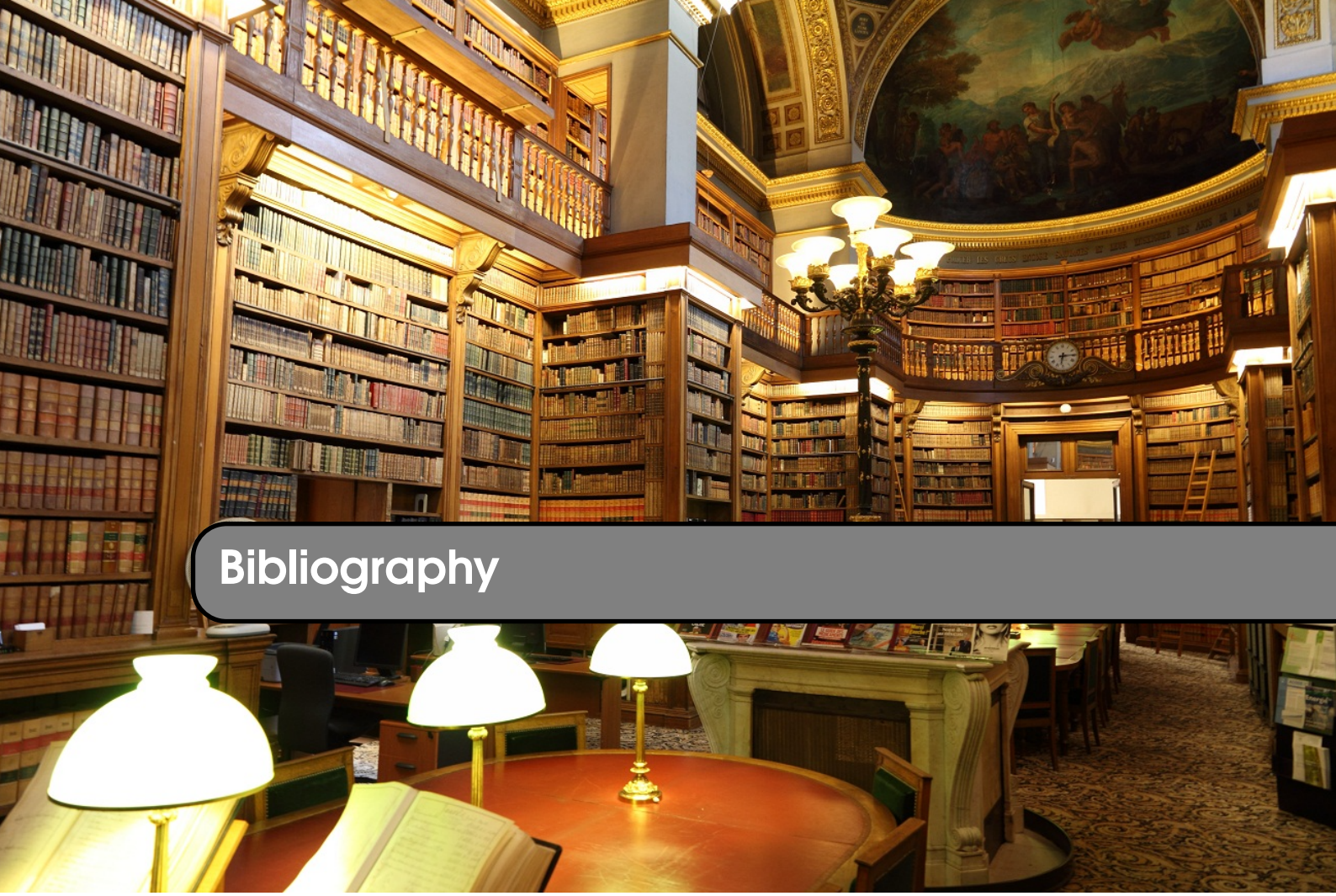
In summary,

$$\begin{aligned}\mathbb{U}_2 \;=\;& M(M + |H| + 4) + (4M + 3|H| + 15) + (Km(M + Km + m + 17) - K^2) \\ & + m(M + Km + m + 16) + 4n(M + Km + m + 7) + m(M + K + 4) \\ & + Km(M + 4).\end{aligned}$$

We have that $\mathbb{L} - \mathbb{U}_2 = -|H| - 5K^2 - 21Km - 4n(K(m-1) + m + 7) - 20m = K^2 - 23Km - 4n(K(m-1) + m + 8) - 22m - 5 > K^2 - Knm(23 + 4(1 + 1 + 8) + 22 + 5) = K^2 - 90Knm$ using $K, n, m \geq 1$. Let us set K equal to any value greater than or equal to $91nm$. We then have $K^2 > 90Knm$ and, thus, $\mathbb{L} > \mathbb{U}_2$. That is, the S-reachability has to be smaller than $\mathbb{L}$.

Conclusion. We have thus proved that the formula Φ is satisfiable if and only if there exists a schedule S such that the S-reachability of G is at least $\mathbb{L}$. This completes the proof of the hardness of the edge temporalization problem.

Note that, even if this problem is likely not to be solvable in polynomial time, it might be be that there exists a polynomial time approximation algorithm, that is, an algorithm able to compute in polynomial time an edge temporalization which produces a temporal reachability which is a constant fraction of the maximum obtainable reachability. This is indeed the main open question concerning the edge temporalization problem (and many of its variants).



Bibliography

Journal papers

[5] Alex Bavelas. “A Mathematical Model for Group Structures”. In: *Human Organization* 7 (1948), pages 16–30 (cited on pages 67, 68).

[6] Elisabetta Bergamini et al. “Computing top-k Closeness Centrality Faster in Unweighted Graphs”. In: *ACM Trans. Knowl. Discov. Data* 13.5 (2019), 53:1–53:40 (cited on pages 79, 81, 104).

[7] Kenneth A. Berman. “Vulnerability of scheduled networks and a generalization of Menger’s Theorem”. In: *Networks* 28 (1996), pages 125–134 (cited on page 116).

[12] Paolo Boldi and Sebastiano Vigna. “Axioms for Centrality”. In: *Internet Math.* 10.3-4 (2014), pages 222–262 (cited on page 70).

[13] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. “Into the Square: On the Complexity of Some Quadratic-time Solvable Problems”. In: *Electron. Notes Theor. Comput. Sci.* 322 (2016), pages 51–67 (cited on pages 64–66, 76).

[15] Ulrik Brandes. “A faster algorithm for betweenness centrality”. In: *The Journal of Mathematical Sociology* 25 (2001), pages 163–177 (cited on page 73).

[16] Ronald L. Breiger and Philippa E. Pattison. “Cumulated social roles: The duality of persons and their algebras”. In: *Social Networks* 8.3 (1986), pages 215–256 (cited on pages 2, 11).

[17] Andrei Z. Broder et al. “Graph structure in the Web”. In: *Comput. Networks* 33 (2000), pages 309–320 (cited on page 100).

[19] Arnaud Casteigts, Joseph G. Peters, and Jason Schoeters. “Temporal cliques admit sparse spanners”. In: *J. Comput. Syst. Sci.* 121 (2021), pages 1–17 (cited on page 117).

[20] Arnaud Casteigts et al. “Time-varying graphs and dynamic networks”. In: *IJPEDS* 27.5 (2012), pages 387–408 (cited on page 107).

[22] Andrea E. F. Clementi et al. “Flooding Time of Edge-Markovian Evolving Graphs”. In: *SIAM J. Discret. Math.* 24.4 (2010), pages 1694–1712 (cited on page 109).

[23] Pierluigi Crescenzi, Clémence Magnien, and Andrea Marino. “Approximating the Temporal Neighbourhood Function of Large Temporal Graphs”. In: *Algorithms* 12.10 (2019), page 211 (cited on pages 109, 114).

[24] Pierluigi Crescenzi, Clémence Magnien, and Andrea Marino. “Finding Top-k Nodes for Temporal Closeness in Large Temporal Graphs”. In: *Algorithms* 13.9 (2020), page 211 (cited on page 109).

[27] Pilu Crescenzi et al. “On computing the diameter of real-world undirected graphs”. In: *Theor. Comput. Sci.* 514 (2013), pages 84–95 (cited on page 54).

[29] Frank Dehne, Masoud T. Omran, and Jörg-Rüdiger Sack. “Shortest Paths in Time-Dependent FIFO Networks”. In: *Algorithmica* 62.1-2 (2012), pages 416–435 (cited on page 109).

[32] Edsger W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1 (1959), pages 269–271 (cited on page 60).

[33] Stuart E. Dreyfus. “An Appraisal of Some Shortest-Path Algorithms”. In: *Operations Research* 17.3 (1969), pages 395–412 (cited on page 108).

[36] J. Enright, K. Meeks, and F. Skerman. “Assigning times to minimise reachability in temporal graphs”. In: *Journal of Computer and System Sciences* 115 (2021), pages 169–186 (cited on page 119).

[37] David Eppstein and Joseph Wang. “Fast Approximation of Centrality”. In: *J. Graph Algorithms Appl.* 8 (2004), pages 39–45 (cited on pages 30, 78).

[39] Luca Foschini, John Hershberger, and Subhash Suri. “On the Complexity of Time-Dependent Shortest Paths”. In: *Algorithmica* 68.4 (2014), pages 1075–1097 (cited on page 109).

[41] Linton C. Freeman. “A Set of Measures of Centrality Based on Betweenness”. In: *Sociometry* 40.1 (1977), pages 35–41 (cited on page 68).

[42] Linton C. Freeman. “Centrality in social networks conceptual clarification”. In: *Social Networks* 1 (1978), pages 215–239 (cited on page 67).

[44] F. Göbel, J. Orestes Cerdeira, and H.J. Veldman. “Label-connected graphs and the gossip problem”. In: *Discrete Mathematics* 87.1 (1991), pages 29–40 (cited on page 119).

[46] Mohammad Al Hasan and Vachik S. Dave. “Triangle counting in large networks: a review”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8 (2018) (cited on pages 88, 89).

[48] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. “Which Problems Have Strongly Exponential Complexity?” In: *J. Comput. Syst. Sci.* 63.4 (2001), pages 512–530 (cited on page 64).

[50] D. Kempe, J. Kleinberg, and A. Kumar. “Connectivity and Inference Problems for Temporal Networks”. In: *Journal of Computer and System Sciences* 64.4 (2002), pages 820–842 (cited on pages 115, 116).

[51] Jon M. Kleinberg. “Navigation in a small world”. In: *Nature* 406.24 August 2000 (2000), page 845 (cited on page 26).

[53] Rainer Kujala et al. “A collection of public transport network data sets for 25 cities”. In: *Scientific Data* 5 (2018) (cited on page 114).

[57] Matthieu Latapy. “Main-memory triangle computations for very large (sparse (power-law)) graphs”. In: *Theor. Comput. Sci.* 407 (2008), pages 458–473 (cited on page 88).

[58] Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. “Stream graphs and link streams for the modeling of interactions over time”. In: *Social Netw. Analys. Mining* 8.1 (2018), 61:1–61:29 (cited on page 109).

[61] Michael Ley. “DBLP - Some Lessons Learned”. In: *Proc. VLDB Endow.* 2.2 (2009), pages 1493–1500 (cited on page 50).

[64] Clémence Magnien, Matthieu Latapy, and Michel Habib. “Fast computation of empirically tight bounds for the diameter of massive graphs”. In: *ACM J. Exp. Algorithmics* 13 (2008) (cited on page 52).

[65] Tyler H. McCormick, Matthew J. Salganik, and Tian Zheng. “How many people do you know? Efficiently estimating personal network size”. In: *Journal of the American Statistical Association* 105.489 (2010), pages 59–70 (cited on page 29).

[66] George B. Mertzios, Othon Michail, and Paul G. Spirakis. “Temporal Network Optimization Subject to Connectivity Constraints”. In: *Algorithmica* 81.4 (2019), pages 1416–1449 (cited on page 116).

[67] Othon Michail. “An Introduction to Temporal Graphs: An Algorithmic Perspective”. In: *Internet Mathematics* 12.4 (2016), pages 239–280 (cited on page 109).

[68] Stanley Milgram. “The Small World Problem”. In: *Psychology Today* 1.1 (1967), pages 61–67 (cited on page 25).

[70] Koji Mizoguchi. “The Evolution of Prestige Good Systems: an Application of Network Analysis to the Transformation of Communication Systems and their Media”. In: *Network Analysis in Archaeology: New Approaches to Regional Interaction*. Edited by Carl Knappett. Oxford University Press, 2013. Chapter 7, pages 151–178 (cited on pages 3, 5).

[73] Karl Pearson. “Notes on regression and inheritance in the case of two parents”. In: *Proceedings of the Royal Society of London* 58 (1895), pages 240–242 (cited on page 69).

[76] Gert Sabidussi. “The centrality index of a graph”. In: *Psychometrika* 31 (1966), pages 581–603 (cited on page 68).

[77] David Schoch. “Centrality without indices: Partial rankings and rank probabilities in networks”. In: *Soc. Networks* 54 (2018), pages 50–60 (cited on page 69).

[79] David Schoch, Thomas W. Valente, and Ulrik Brandes. “Correlations among centrality indices and a class of uniquely ranked graphs”. In: *Soc. Networks* 50 (2017), pages 46–54 (cited on page 69).

[80] M. Sharir. “A strong-connectivity algorithm and its applications in data flow analysis”. In: *Computers & Mathematics with Applications* 7.1 (1981), pages 67–72 (cited on page 93).

[81] Marvin E. Shaw. “Group Structure and the Behavior of Individuals in Small Groups”. In: *The Journal of Psychology* 38.1 (1954), pages 139–149 (cited on page 68).

[83] Jithin Kazuthuveettil Sreedharan et al. “Inferring Temporal Information from a Snapshot of a Dynamic Network”. In: *Scientific Reports* 9 (2019) (cited on page 79).

[84] C. Stark et al. “Biogrid: A General Repository for Interaction Datasets”. In: *Nucleic Acids Res.* 34 (2006), pages D535–9 (cited on page 82).

[88] Duncan J. Watts and Steven H. Strogatz. “Collective dynamics of ‘small-world’ networks”. In: *Nature* 393 (1998), pages 440–442 (cited on page 45).

[95] Virginia Vassilevska Williams and R. Ryan Williams. “Subcubic Equivalences Between Path, Matrix, and Triangle Problems”. In: *J. ACM* 65.5 (2018), 27:1–27:38 (cited on page 64).

[96] Huanhuan Wu et al. “Efficient Algorithms for Temporal Path Computation”. In: *IEEE Transactions on Knowledge and Data Engineering* 28 (2016), pages 2927–2942 (cited on pages 109, 112).

[98] Wayne W. Zachary. “An Information Flow Model for Conflict and Fission in Small Groups”. In: *Journal of Anthropological Research* 33.4 (1977), pages 452–473 (cited on pages 8, 9).

Conference papers

[1] Amir Abboud, Virginia Vassilevska Williams, and Joshua R. Wang. “Approximation and Fixed Parameter Subquadratic Algorithms for Radius and Diameter in Sparse Graphs”. In: *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2016, pages 377–391 (cited on page 64).

[2] Josh Alman and Virginia Vassilevska Williams. “A Refined Laser Method and Faster Matrix Multiplication”. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2021, pages 522–539 (cited on page 49).

[4] Lars Backstrom et al. “Four degrees of separation”. In: *Proceedings of the 4th ACM Conference on Web Science*. ACM, 2012, pages 33–42 (cited on page 44).

[8] Sandeep Bhadra and Afonso Ferreira. “Complexity of Connected Components in Evolving Graphs and the Computation of Multicast Trees in Dynamic Networks”. In: *Proceedings of the 2nd International Conference on Ad-Hoc, Mobile, and Wireless Networks*. Springer, 2003, pages 259–270 (cited on page 108).

[10] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. “HyperANF: approximating the neighbourhood function of very large graphs on a budget”. In: *Proceedings of the 20th International Conference on World Wide Web*. ACM, 2011, pages 625–634 (cited on page 44).

[25] Pierluigi Crescenzi et al. “A Comparison of Three Algorithms for Approximating the Distance Distribution in Real-World Graphs”. In: *Proceedings of First International ICST Conference Theory and Practice of Algorithms in (Computer) Systems*. Volume 6595. Lecture Notes in Computer Science. Springer, 2011, pages 92–103 (cited on page 30).

[26] Pierluigi Crescenzi et al. “On Computing the Diameter of Real-World Directed (Weighted) Graphs”. In: *Proceedings of 11th International Symposium on Experimental Algorithms*. Springer, 2012, pages 99–110 (cited on page 58).

[30] A. Deligkas and I. Potapov. “Optimizing Reachability Sets in Temporal Graphs by Delaying”. In: *AAAI*. 2020, pages 9810–9817 (cited on page 119).

[31] Erik D. Demaine and Morteza Zadimoghaddam. “Minimizing the Diameter of a Network Using Shortcut Edges”. In: *Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*. Springer, 2010, pages 420–431 (cited on page 118).

[54] Jérôme Kunegis. “KONECT: the Koblenz network collection”. In: *Companion Volume 22nd International World Wide Web Conference*. ACM, 2013, pages 1343–1350 (cited on page 87).

[56] Jérôme Kunegis, Andreas Lommatzsch, and Christian Bauckhage. “The slashdot zoo: mining a social network with negative edges”. In: *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009, pages 741–750 (cited on page 32).

[60] Jure Leskovec and Eric Horvitz. “Planetary-scale views on a large instant-messaging network”. In: *Proceedings of the 17th International Conference on World Wide Web*. ACM, 2008, pages 915–924 (cited on page 30).

[69] Alan Mislove et al. “Measurement and analysis of online social networks”. In: *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference*. ACM, 2007, pages 29–42 (cited on page 52).

[75] Tim Roughgarden. “CS167: Reading in Algorithms Counting Triangles”. In: 2014 (cited on page 89).

[85] Siddharth Suri and Sergei Vassilvitskii. “Counting triangles and the curse of the last reducer”. In: *Proceedings of the 20th International Conference on World Wide Web*. ACM, 2011, pages 607–614 (cited on page 88).

[97] Qi Ye, Bin Wu, and Bai Wang. “Distance Distribution and Average Shortest Path Length Estimation in Real-World Networks”. In: *Proceedings of the 6th International Conference on Advanced Data Mining and Applications*. Volume 6440. Lecture Notes in Computer Science. Springer, 2010, pages 322–333 (cited on page 30).

Books

[34] Devdatt Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge: Cambridge University, 2009 (cited on page 37).

[35] G. Waldo Dunnington. *Gauss: Titan of Science*. Cambridge: Mathematical Association of America, 2004 (cited on page 48).

[43] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979 (cited on pages 64, 119).

[45] John Guare. *Six Degrees of Separation*. New York City: Penguin Random House, 1990 (cited on page 26).

[52] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Boston: Addison-Wesley Reading, 1993 (cited on pages 13, 61).

[63] Zvi Lotker. *Analyzing Narratives in Social Networks*. Cham: Springer, 2021 (cited on page 13).

[87] Duncan J. Watts. *Six Degrees: The Science of a Connected Age*. New York, USA: W. W. Norton, 2003 (cited on page 45).

Miscellanea

[9] Smriti Bhagat et al. *Three and a half degrees of separation*. `https://research.facebook.com/blog/2016/02/three-and-a-half-degrees-of-separation/`. Last visited January 18, 2022 (cited on page 44).

[11] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. *Laboratory for Web Algorithmics*. `https://law.di.unimi.it/`. Last visited January 27, 2022 (cited on page 100).

[14] Lorrie Boucher et al. *The Biological General Repository for Interaction Datasets*. `https://thebiogrid.org/`. Last visited January 26, 2022 (cited on page 82).

[18] Filippo Brunelli, Pierluigi Crescenzi, and Laurent Viennot. *On The Complexity of Maximizing Temporal Reachability via Trip Temporalisation*. 2021 (cited on page 119).

[21] L. Ceci. *Hours of video uploaded to YouTube every minute 2007-2020*. https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/. Last visited January 27, 2022 (cited on page 87).

[28] DBLP. *Raw DBL data in a single XML file*. https://dblp.org/xml/. Last visited January 18, 2022 (cited on page 50).

[38] James Fairbanks et al. *JuliaGraphs/Graphs.jl: an optimized graphs package for the Julia programming language*. https://github.com/JuliaGraphs/Graphs.jl/. Last visited December 29, 2021 (cited on page 21).

[40] Wikimedia Foundation. *Wikipedia*. https://www.wikipedia.org/. Last visited January 27, 2022 (cited on page 99).

[47] IMDb. *IMDb Datasets*. http://www.imdb.com/interfaces. Last visited December 29, 2021 (cited on page 5).

[49] JuliaLang.org. *The Julia Programming Language*. https://julialang.org/. Last visited December 29, 2021 (cited on page 21).

[55] Jérôme Kunegis. *The KONECT Project*. http://konect.cc/. Last visited January 27, 2022 (cited on pages 87, 99).

[59] Jure Leskovec. *Stanford Network Analysis Project*. https://snap.stanford.edu/. Last visited January 14, 2022 (cited on page 33).

[62] Google LLC. *YouTube*. https://www.youtube.com/. Last visited January 27, 2022 (cited on page 87).

[71] Fondazione Openpolis. *OpenParlamento*. https://parlamento16.openpolis.it/. Last visited January 27, 2022 (cited on page 92).

[72] Fondazione Openpolis. *Openpolis*. https://www.openpolis.it/. Last visited January 27, 2022 (cited on page 92).

[74] Nicola Prezza. *Algorithms for massive data*. https://nicolaprezza.github.io/pdfs/Algorithms_for_massive_data___notes.pdf. Last visited January 18, 2022 (cited on page 44).

[78] David Schoch. *Periodic Table of Network Centrality*. http://schochastics.net/sna/periodic.html. Last visited January 25, 2022 (cited on page 69).

[82] Jithin K. Sreedharan and Vikram Ravindra. *Human Brain Networks Dataset of 100 Subjects with Node Labels*. https://github.com/jithin-k-sreedharan/data_human_brain_networks. Last visited January 26, 2022 (cited on page 79).

[86] Kamil Ugurbil and David Van Essen. *Connectome Coordination Facility*. https://www.humanconnectome.org/. Last visited January 26, 2022 (cited on page 79).

[89] Wikipedia. *Demographics of the United States*. https://en.wikipedia.org/wiki/Demographics_of_the_United_States. Last visited December 29, 2021 (cited on page 29).

[90] Wikipedia. *Kuopio*. https://en.wikipedia.org/wiki/Kuopio. Last visited January 28, 2022 (cited on page 114).

[91] Wikipedia. *List of Donald Trump 2020 presidential campaign non-political endorsements*. https://en.wikipedia.org/wiki/List_of_Donald_Trump_2020_presidential_campaign_non-political_endorsements. Last visited December 29, 2021 (cited on page 7).

[92] Wikipedia. *List of Joe Biden 2020 presidential campaign celebrity endorsements*. https://en.wikipedia.org/wiki/List_of_Joe_Biden_2020_presidential_campaign_celebrity_endorsements. Last visited December 29, 2021 (cited on page 7).

[93] Wikipedia. *Slashdot*. https://en.wikipedia.org/wiki/Slashdot. Last visited January 14, 2022 (cited on page 32).

[94] Wikipedia. *TOP500*. https://en.wikipedia.org/wiki/TOP500. Last visited December 29, 2021 (cited on page 29).

Index