Chapter 5
Web Search

Paolo Ferragina and Rossano Venturini

Abstract Faced with the massive amount of information on the Web, which includes not only texts but nowadays any kind of file (audio, video, images, etc.), Web users tend to lose their way when browsing the Web, falling into what psychologists call "getting lost in hyperspace". Search engines alleviate this by presenting the most relevant pages that better match the user's information needs. Collecting a large part of the pages in the Web, extrapolating a user information need expressed by means of often ambiguous queries, establishing the importance of Web pages and their relevance for a query, are just a few examples of the difficult problems that search engines address every day to achieve their ambitious goal. In this chapter, we introduce the concepts and the algorithms that lie at the core of modern search engines by providing running examples that simplify understanding, and we comment on some recent and powerful tools and functionalities that should increase the ability of users to match in the Web their information needs.

5.1 The Prologue

Just 10 years ago, major search engines were indexing about one billion Web pages; this number has today exploded to about one trillion as reported in Google's blog by Alpert et al. [5]. Such growth is proportional to three orders of magnitude, thus leading everyone to talk about the *exponential* growth of the Web. But this number denotes only the amount of pages that are *indexed* by search engines and thus are available to users via their Web searches; the *real* number of Web pages is much larger, and in some sense unbounded, as many researchers observed in the past. This is due to the existence of pages which are dynamic, and thus are generated on-the-fly when users request them, or pages which are hidden in private archives,

P. Ferragina (✉) · R. Venturini
Dipartimento di Informatica, Università di Pisa, largo B. Pontecorvo 3, 56123 Pisa, Italy
e-mail: ferragina@di.unipi.it; rossano@di.unipi.it

G. Ausiello and R. Petreschi (eds.), *The Power of Algorithms*,
DOI 10.1007/978-3-642-39652-6_5, © Springer-Verlag Berlin Heidelberg 2013

and thus can be accessed only through proper credentials (the so-called *deep Web*). At the extreme, we could argue that the number of (dynamic) pages in the Web is infinite, just take sites generating calendars.

Faced with this massive amount of information, which includes not only text but nowadays any kind of file (audio, video, images, etc.), Web users tend to lose their way when browsing the Web, falling into what psychologists call “getting lost in hyperspace”. In order to avoid this critical issue, computer scientists designed in the recent past some sophisticated software systems, called *search engines*, that allow users to specify some keywords and then retrieve in a few milliseconds the collection of pages containing them. The impressive feature of these systems is that the retrieved pages could be located in Web servers spread all around the world, possibly unreachable even by expert users that have clear ideas of their information needs. In this chapter we will review the historical evolution and the main algorithmic features of search engines. We will describe some of the algorithms they hinge on, with the goal of providing the basic principles and the difficulties that algorithm designers and software engineers found in their development. This will offer a picture of the complexity of those systems that are considered as the most complex tools that humans have ever built. A commented literature concludes the chapter by providing pointers to several fundamental and accessible publications that can help readers to satisfy their curiosity and understanding of search engines and, more specifically, the wide and challenging research field known as Information Retrieval.

The following sections will cover all those issues, starting with an analysis of two networks that are distinct, live in symbiosis, and are the ground where search engines work: the Internet and the Web.

5.2 Internet and Web Graphs

As we saw in Chap. 2, a *graph* is a mathematical object formed by a set of *nodes* and a set of *edges* which represent relationships between pairs of nodes. Founded in the eighteenth century, Graph Theory has gained great importance in the last century as a means to represent entities in relation to each other. Nowadays, Graph Theory has further increased its importance being essential for the study of the Internet and its applications (such as the Web).

A network of computers (hereinafter simply referred to as a *network*) is a set of devices that send messages through electronic connections by means of cables, fiber optics, radio or infrared links. In our abstraction of a network, connections are represented with the edges of a graph whose nodes represent the devices. The Internet is actually a network of networks: institutions may own many computers which are connected to each other, but each institution enters the Internet as a single unit. These units are called *autonomous systems* (ASs) and constitute the nodes of the Internet graph. Each of these ASs could be either a user owning a single computer, or a set of computers connected together within the same building, or even very complex networks whose computers may be geographically far from

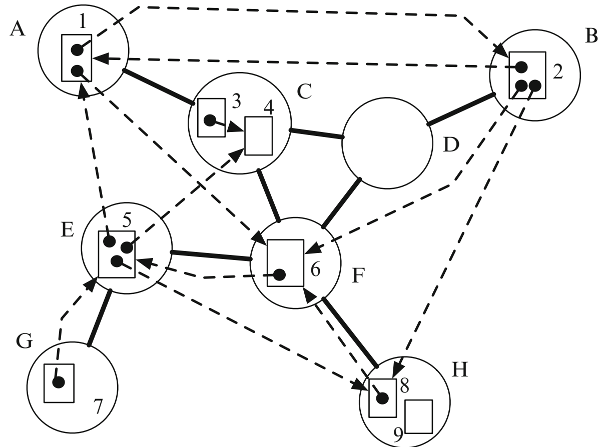

Fig. 5.1 The structure of the Internet and the Web graphs. *Circles* represent the autonomous systems, and edges marked with *solid lines* represent the connections between them. *Rectangles* represent Web pages, and edges marked with *dashed arrows* represent the links between them

each other. *Internet Service Providers* (ISPs) are examples of the latter typology of ASs. ISPs are companies which regulate the traffic of messages on the Internet by selling their services to other users.

Figure 5.1 shows a possible fragment of the Internet graph: nodes are drawn as circles connected by edges which are represented by continuous lines. Notice that edges can be traversed in both directions. A direct message from G to B, e.g., an e-mail, could follow the path $G - E - F - D - B$. User G is the sender who pays her service provider E, which regulates its costs with subsequent nodes. However, the real scenario is much more complicated. The Internet is huge: even if we do not know how many computers are connected (this question is indeed misplaced because the network topology is constantly changing), we can estimate that this number exceeds one billion, considering all the computers within the ASs. The path followed by a message is not determined beforehand and may even change during the transmission. This is a consequence of the enormous size and *anarchistic* structure of the Internet, which grows and evolves without any centralized control, and the inevitable continuous changes in the connections deriving from technical or maintenance issues. This behavior has characterized the network since its birth in the late 1960s and distinguishes the Internet from both telephone and electrical networks.

Traditional telephone networks work by using a methodology called *circuit switching*: two people talking on the phone are using the "channel" that has been reserved for their communication. The two network nodes establish a dedicated communication channel through the network before they start communicating. This channel works as if the nodes were physically connected with an electrical circuit. However, if something goes wrong, the communication is interrupted. The initial idea for the Internet was to resort to a mechanism called *message switching*: the routing of messages in the network is established node by node depending on the location of the recipient and the current level of traffic. At each step, the current node is responsible for choosing the next node in the path. As a side effect of this

mechanism, a message sent to a close user may be sent through a path longer than necessary. This method was soon replaced by a close-relative method called *packet switching*, which is currently still in use. A (binary) message is divided into *packets* of a fixed length; this length is reported at the beginning of each packet together with its destination address. Sometimes it happens that packets of the same message are routed thought different paths and reach the destination in a order that differs from the original one. The recipient is then responsible for reordering the packets to (re-)obtain the original message. This mechanism guarantees that a message is always accepted by the network, even if it may take time before all of its packets reach their final destination and the message can be reconstructed. In a telephone network, instead, a phone number can be "busy" even if the recipient's phone is free, due to the saturation of the chain of connections which link the two locations involved in the call because of other calls.[1]

Now let us see what we mean by Web (or *www*, which is the acronym for the World Wide Web). Born in 1989 at CERN in Geneva and based on the known concept of *hypertext*, the Web is a set of documents called *pages* that refer to each other to refine or improve the same subject, or to draw a new subject in some relation to the first one. The luck of the Web is inextricably linked to the Internet. Pages of the Web are stored in the memories of computers on the Internet network, so that users can freely consult this collection of pages by moving from one page to another one via (hyper-)links, quickly and without leaving their PCs, regardless of the location of the requested pages. Seen in this way, the Web is represented as a graph whose nodes are pages and whose edges are the (hyper-)links between pairs of pages. Edges in this graph are *directed*, meaning that each edge can be traversed only in one predetermined direction. Figure 5.1 shows a portion of the Web graph in which the orientation of an edge is specified by means of an arrow.

We should immediately establish some properties of this graph being very important for search engines which collect, inspect and make Web pages available to users. First, the graph is literally huge, and its topology varies continuously due to the continuous creation or deletion of pages. Regarding the size of the graph, sometimes we read unfounded and fanciful stories: it is reasonably sure that search engines provide access to roughly hundreds of billions of pages. Nonetheless, the number of existing pages is larger, though many of them may be not directly available as we commented at the beginning of this chapter.[2] Another important observation is that, although the Web pages are stored in computers of the Internet,

---

[1]This phenomenon happened frequently years ago. Nowadays, it is rare due to the improvement of transmission techniques. However, it can still happen if we call foreign countries or if we try to make national calls in particular periods of the year (e.g., New Year's Eve).

[2]We usually refer with *indexable Web* to the set of pages that could be reached by search engines. The other part of the Web, called the *deep Web*, includes a larger amount of information contained in pages not indexed by search engines, or organized into local databases, or obtainable using special software. Nowadays an important area of research studies the possibility of extending the functionalities of search engines to the information stored in the deep Web. In this direction can be classified the initiative *open data*, see, for example, `linkeddata.org`.

	A	B	C	D	E	F	G	H
A	0	0	1	0	0	0	0	0
B	0	0	0	1	0	0	0	0
C	1	0	0	1	0	1	0	0
D	0	1	1	0	0	1	0	0
E	0	0	0	0	0	1	1	0
F	0	0	1	1	1	0	0	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	0	1	0	0

I

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

W

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

W^2

Fig. 5.2 Adjacency matrices I and W for the Internet and the Web graphs of Fig. 5.1, and the square matrix W^2

these two graphs do not have any other relationship between each other. Referring to the example of Fig. 5.1, page 1 has a link to page 2 but there is no direct link between the two nodes A and B of the Internet (i.e., the computers containing these pages). On the other hand, the two nodes C and F are connected but there are no links between the pages contained therein. Another difference between these two graphs is that the edges of the Web are directed, while those of the Internet are not. Moreover, the graph of the Internet is *strongly connected*, i.e., there always exists a path that connects any two nodes, unless a temporary interruption of some connection does occur. This, however, does not happen in the Web graph where there are nodes that cannot be reached by others. This may happen for several reasons: a node cannot be reached by any other node since it has only outgoing links (nodes 3 and 7 in the figure), a node cannot reach other nodes since it has only incoming links (node 4), or a node is completely disconnected since it has no links at all (node 9). In the box below we present a deeper algorithmic elaboration about these important characteristics.

Adjacency Matrix and Paths of a Graph

A graph G with n nodes can be represented in a computer through an *adjacency matrix* M having n rows and n columns. Rows and columns are in correspondence with nodes of G. The cell of M at row i and column j, denoted with $M[i, j]$, corresponds to the pair of nodes i and j. We set $M[i, j]$ equal to 1 whenever G has an edge from node i to node j; $M[i, j]$ is 0 otherwise. The Internet graph and the Web graph of Fig. 5.1 have, respectively, the adjacency matrices I of size 8×8 and W of size 9×9. These matrices are shown in Fig. 5.2.

Notice that any undirected graph induces an adjacency matrix which is symmetric with respect to its main diagonal, because if the edge (i, j) exists then so does the edge (j, i), hence $M[i, j] = M[j, i] = 1$. We observe that the Internet graph is undirected and, thus, matrix I is symmetric.

(continued)

(continued)

For an example, edge $A - C$ can be traversed in both directions and, thus, $I[A, C] = I[C, A] = 1$. In directed graphs, as the Web graph, the matrix may be asymmetric and so there exist entries such that $M[i, j] \neq M[j, i]$. In our example we have $W[1, 2] = 1$ and $W[2, 1] = 1$ because there are two distinct edges going in both directions. However, we also have cases in which only one edge between two nodes is present (e.g., we have $W[1, 6] = 1$ and $W[6, 1] = 0$).

In mathematics the square of an $n \times n$ matrix M is an $n \times n$ matrix M^2 whose cells $M^2[i, j]$ are computed by a way different than the standard product of numbers. Each cell $M^2[i, j]$ is, indeed, obtained by multiplying the ith row and the jth column of M according to the following formula:

$$M^2[i, j] = M[i, 1] \times M[1, j] + M[i, 2] \times M[2, j] + \ldots + M[i, n] \times M[n, j]. \tag{5.1}$$

This formula has a deep meaning that we will illustrate through our example matrix W of Fig. 5.2. Take $W^2[6, 4]$, which is equal to 1 because the sixth row and the fourth column of W are, respectively, equal to [0 0 0 0 1 0 0 0 0] and [0 0 1 0 1 0 0 0 0]. So that the formula above returns the value 1 since there is a pair of multiplied entries ($W[6, 5]$ in row 6 and $W[5, 4]$ in column 4) that are equal to 1.

$$\begin{aligned} W^2[6, 4] &= W[6, 1] \times W[1, 4] + W[6, 2] \times W[2, 4] + \ldots + W[6, 9] \\ &\quad \times W[9, 4] \\ &= 0 \times 0 + 0 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 + 0 \times 0 + 0 \times 0 \\ &\quad + 0 \times 0 + 0 \times 0 \\ &= 1 \end{aligned}$$

Interestingly, there is a deeper reason behind the value 1 obtained by multiplying $W[6, 5]$ and $W[5, 4]$. Since these two cells indicate that in the Web graph there is one edge from node 6 to node 5 and one edge from node 5 to node 4, we conclude that there exists a path that goes from node 6 to node 4 traversing exactly two edges. Therefore, each cell $W^2[i, j]$ indicates the number of distinct paths from node i to node j that traverse exactly two edges. We can understand this statement better by further examining other entries of the matrix W^2. We have $W[1, 2] = 1$ but $W^2[1, 2] = 0$ because there is an edge from node 1 to node 2 but there does not exist any path of length 2 connecting these two nodes (see Fig. 5.1). Furthermore, we have

(continued)

(continued)

$W^2[2,6] = 2$ because the second row of W is [1 0 0 0 0 1 0 1 0] and its sixth column is [1 1 0 0 0 0 0 1 0]:

$$\begin{aligned} W^2[2,6] =& W[2,1] \times W[1,6] + W[2,2] \times W[2,6] + \ldots + W[2,9] \\ & \times W[9,6] \\ =& 1 \times 1 + 0 \times 1 + 0 \times 0 + 0 \times 0 + 0 \times 0 + 1 \times 0 + 0 \times 0 \\ & + 1 \times 1 + 0 \times 0 \\ =& 2 \end{aligned}$$

Here we find two pairs of 1s: $W[2,1] = W[1,6] = 1$ and $W[2,8] = W[8,6] = 1$, meaning that the Web graph has two paths of length 2 connecting node 2 to node 6 (i.e., $2-1-6$ and $2-8-6$).

Following the same rule we can proceed in the calculation of successive powers W^3, W^4, ... of the matrix W. The entries of these matrices indicate the number of paths of length 3, 4, ... between pairs of nodes, and thus the number of links that we must follow in order to move from one Web page to another. For example, the elements of W^3 are obtained by resorting to Eq. (5.1), as the product of a row of W^2 and a column of W (or vice versa). We have thus:

$$W^3[7,6] = 1\times1+0\times1+0\times0+1\times0+0\times0+0\times0+0\times0+1\times1+0\times0 = 2$$

which corresponds to the two paths of length 3 from node 7 to node 6 (i.e., $7-5-1-6$ and $7-5-8-6$).

Classical algorithms compute a power of a matrix by taking two powers with smaller exponents and by applying Eq. (5.1) to one row and one column of each of them. Since the number of these pairs is n^2, and since the calculation of Eq. (5.1) requires a time that is proportional to n (in fact, the expression contains n multiplications and $n-1$ additions), the total computation time required by one matrix multiplication is proportional to (i.e., *is order of*) n^3. Of course, the actual time required by an algorithm also depends on the computer and on the programming language in use. n any case, a *cubic* behavior like this is undesirable: it means, for example, that if the number of nodes of the graph doubles from n to $2n$, the time grows from n^3 to $(2n)^3 = 8n^3$, i.e., it becomes eight times larger. As we shall see in a following section, computing powers of matrices is one of the main ingredients to establish an order of importance (*ranking*) among Web pages. However, considering that the Web graph has hundreds of billions of nodes, it is unfeasible to perform

(continued)

(continued)

this computation on a single computer. The solution usually adopted consists of dividing the work among many computers by resorting to techniques of *distributed computation*, which are, however, much too complex to be discussed here.

5.3 Browsers and a Difficult Problem

A *Web site* is a group of related Web pages whose content may include text, video, music, audio, images, and so on. A Web page is a document, typically written in plain text, that, by using a special language, specifies the components of that page (e.g., text and multimedia content) and the way in which they have to be combined, displayed and manipulated on a computer screen. A Web site is hosted on at least one *Web server* and can be reached by specifying the address of one of its pages. Each page has, indeed, its own numeric address which is, for convenience, associated to a *textual name* which is easier to remember (called URL). For example, www.unipi.it/index.htm is the name of the main Web page of the University of Pisa, while its numeric address of the hosting Web server is `131.114.77.238`. A Web site is characterized by its domain name, e.g., `unipi.it`, and by its main page, which is the starting point to visit the other secondary pages of the site. The URLs of these pages are refinements of the domain name through a hierarchical structure expressed by means of a path. For example, www.unipi.it/research/dottorati is the address of the page that contains the list of PhD courses of the University of Pisa.

A *browser* is a software that allows Web users to visualize a page on the screen of a computer by specifying its URL. After the first Web browser developed at CERN for the original Web, many commercial products were developed, e.g., Netscape Navigator, Internet Explorer, Firefox, Chrome, and many others. In order to speed up the access to Web pages by browsers and other software/applications, network engineers have designed special nodes that collect large groups of page copies. These nodes are called *caches*, and *caching* refers to the activity of storing pages in them. Figure 5.3 shows a typical organization of connections and caches in the Internet graph.

A key role in the network is played by *proxies*, which are computers serving as a local cache within an AS. These computers keep copies of the most frequently requested Web pages by users of the AS. These pages are typically news sites, popular social networks, search engines, and so on. The cached pages are either from outside the AS, or from inside the AS. In the example of Fig. 5.3, users A and B reside inside the same AS, so if they have recently requested the same page, then this page is probably stored in proxy 1 of their AS. In addition, proxies cache Web pages which are stored in the AS and are requested frequently from computers

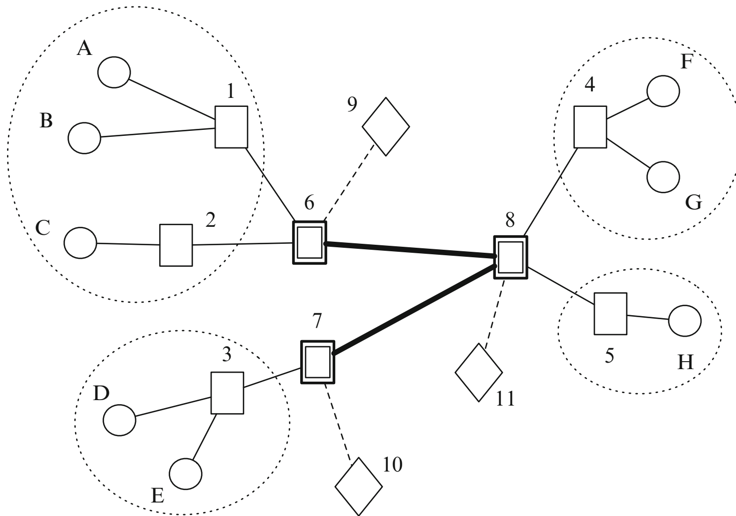

Fig. 5.3 Several levels of caching for Web pages are shown. *Circles* labeled with $A, B, \ldots, H$ are computers with a browser which store in their memory the most recently accessed Web pages. Regions enclosed with *dotted lines* are ASs, while *rectangles* represent their corresponding proxies. *Double rectangles* represent routers, which are devices that route the messages flowing through the network. Finally, *diamonds* represent large CDN subnetworks

outside that AS. In this way, these frequent requests are served immediately without the need of traversing the whole AS. In the example of Fig. 5.3, if pages stored in the computers A and B are requested frequently from other users of the network, copies of them may be kept in proxy 1, which is responsible for answering requests coming from outside. Caching can be structured in a hierarchical way, by introducing the so-called *Content Delivery Networks* (CDN), which are subnetworks of computers that provide caching at geographic level. It goes without saying that the same Web page can be replicated in many proxies and CDNs, provided that this improves its delivery to the requesting browsers and Web applications.

An interesting problem is that of distributing Web pages in the caches of the network with the aim of minimizing the overall amount of time required to access the Web pages that are more likely to be requested by users in a certain period of time. As we will see, from an algorithmic point of view, this problem is practically insolvable because all known solutions have exponential time complexity. In fact, this problem belongs to a class of hard problems for which we can compute efficiently only approximated solutions.[3] We will support this claim by introducing the well-known Knapsack Problem, discussed in Sect. 2.3.2. What is nice in this

[3]This class of problems, called *NP-hard* problems, is extensively discussed in Chap. 3.

Object	1	2	3	4	5	6	7
Weight	15	45	11	21	8	33	16
Value	13	25	14	15	6	20	13
	1	0	1	1	0	0	1

Fig. 5.4 An example of the Knapsack Problem: we have to choose a subset of objects that maximizes their total value and satisfies a constraint on their total weight, which should be at most $C = 70$. The last row represents the optimal subset $\{1, 3, 4, 7\}$, which is encoded by indicating with a 1 an object included in the subset and with a 0 an object not included in it. The total weight is $15 + 11 + 21 + 16 = 63 < 70$, while the total value is $13 + 14 + 15 + 13 = 55$. This is optimal in that no other subset of weight smaller than 70 has a larger value

discussion is that, although these two problems appear very different, they turn out to be highly related in terms of the time required to compute their solution.

Let us consider Fig. 5.4 in which we have seven objects, numbered from 1 through 7, and a knapsack of maximum capacity $C = 70$. For each object we report its weight and its value (e.g., object 1 has weight 15 and value 13, object 2 has weight 45 and value 25, and so on). In the last row of the table we represent a subset of objects by using a bit that indicates whether the corresponding object has been selected (value 1) or not (value 0). We recall that the Knapsack Problem identifies the subset of objects which maximizes the total value, provided that its total weight is at most C. As already observed in Chap. 2, it is strongly believed that this problem does not admit any solution which is more efficient than the one that considers all possible subsets of objects and discards the ones having total weight larger than C. Since the number of possible subsets of n objects is 2^n (including the empty one), this solution requires exponential time and thus it is unfeasible even for a small number of objects.[4]

In the box below we will show the existing relation between the Knapsack Problem and the problem of distributing Web pages in a cache (Cache Problem). We observe that a more general formulation of the latter problem is more complicated than the one presented here. For example, in real applications it is typical to consider also the probability of accessing a certain page, the frequency of requests arriving from each AS, the time variation of these quantities, and so on. However, it suffices to show that the simplified problem is still hard in order to prove the hardness of any more general formulation. In our case, it can be proved that the Cache Problem is *at least as hard as* the Knapsack Problem. This proof is called *reduction* and, specifically, consists of showing that whenever there exists an algorithm that solves the Cache Problem in less than exponential time, the same algorithm can be applied with simple changes to the Knapsack Problem and solve it in less than exponential time too. Since this event is considered to be impossible,

[4]For example, for $n = 20$ objects we have $2^{20} > 1$ millions of subsets.

one can reasonably assume that such a "surprisingly efficient algorithm" for the Cache Problem does not exist; so this problem can be addressed only by means of exponential solutions or, efficiently, by returning approximate solutions. Space limitations and the scope of this chapter do not allow us to detail the reduction that links the Cache Problem to the Knapsack Problem; rather we content ourselves by showing a simpler result, namely, that an approximate solution for the Cache Problem can be derived from an approximate solution to the Knapsack Problem.

The Cache Problem

Let us assume that the network has k ASs, denoted with $AS_1, AS_2, \ldots, AS_k$, n pages, denoted with $p_1, p_2, \ldots, p_n$, and just one cache that stores copies of Web pages for a total size of at most B bytes. Furthermore we assume that each page is stored in its AS, that all pages are requested with the same probability, and that all ASs access the same number of pages. As in the Knapsack Problem, we define an array W of page weights so that $W[j]$ is the size in bytes of the file representing page p_j. It is a little bit harder to define an array of "values" for the Cache Problem that mimics the array of values in the Knapsack Problem. For this aim we use A^j to indicate the AS owner of page p_j; we use $d(i, j)$ to denote the distance, expressed in number of hops, that separate the generic AS_i from A^j; and we use $c(i, j)$ to indicate the cost that AS_i has to pay, in terms of number of hops, to obtain the page p_j. This cost may depend on the choice between placing or not placing the page p_j in the single cache we assumed to exist in the network. Indeed, if p_j is in the cache and the distance between AS_i and the cache is smaller than $d(i, j)$, we have $c(i, j) < d(i, j)$. In any other case, the value of $c(i, j)$ is equal to $d(i, j)$. We can then introduce the value $u(i, j) = d(i, j) - c(i, j)$, which expresses the gain for AS_i of having the page p_j copied in cache. At this point we are ready to define the value $V[j]$ of p_j as the total gain of placing p_j in the cache, summing over all ASs: $V[j] = \sum_{i=1 \ldots k} u(i, j)$. Finally, the "reduction" to the Knapsack Problem can be concluded by taking an auxiliary array A to indicate the subset of pages to be cached.

At this point, we could solve the "synthetic" Knapsack Problem either exactly in exponential time (by enumerating all subsets), or approximately in polynomial time. In this second case, we could choose the objects (pages) in order of decreasing ratio value (gain) versus weight (byte size) until the capacity C of the knapsack (cache) is saturated. The behavior of this simple algorithm is shown in Fig. 5.5. However there do exist solutions to the Cache Problem that do not pass through this "algorithmic reduction": the simplest one consists of caching the most popular Web pages. Clearly, there do also exist more sophisticated approximation algorithms that exploit knowledge about the topology of the network and the frequency of distribution of the

(continued)

Object	3	1	7	5	4	6	2
Weight	11	15	16	8	21	33	45
Value	14	13	13	6	15	20	25
Value/Weight	1.27	0.87	0.81	0.75	0.71	0.61	0.55

Fig. 5.5 The objects of Fig. 5.4 are sorted by decreasing values of the ratio $V[i]/W[i]$. The heuristically chosen subset of objects is $\{3, 1, 7, 5\}$, its total weight is 50 and its total value is 46. Recall that the optimal subset has weight 63 and value 55, as shown in Fig. 5.4

(continued)

page requests. For example, the cache in a CDN C_i may give a larger priority to a page p_i having a high value of the product $\pi_j \times d(i, j)$, where π_j is the frequency of request for page p_j.

5.4 Search Engines

Browsers are fundamental tools for navigating the Web, but their effective use imposes that users clearly know their information needs and where the Web pages matching them are located in the Web. However, it is common that a user has only a partial knowledge of her information need, and wants to find pages through their content without necessarily knowing their URL. Search engines are designed to address this goal and are indispensable for finding information in the huge graph of available Web pages. Bing, Google and Yahoo! are the three most famous search engines available to Web users to match their information needs. They are based on similar algorithmic principles, which are nevertheless implemented differently enough to show, in response to the same user query, a different set of result pages. Here we will neither provide a comparison of different search engines, nor we will discuss how to implement effective user queries; rather we will limit ourselves to the more instructive description of the algorithmic structure of any modern search engine, detailing some of its components.

One of the key features of the search task is that it must be executed fast over a huge set of indexed pages. To this aim, each search engine resorts to a large number of computers grouped in different data-centers distributed around the world. Although many companies are reluctant to reveal precise information about their data-centers, it is estimated that each search engine deploys hundreds of thousands of computers organized into subnetworks, each of which provides different functions. We can distinguish these functions into two main categories (detailed in the following pages): those intended for building a huge index of the

Web pages, and those intended for resolving in the best and the fastest possible way the queries submitted by the users.

In the first category, we can identify several important algorithmic steps: the *crawling* of the Web, the analysis of the Web graph and the parsing of the crawled pages, and finally the construction of an *index* containing all relevant information to match efficiently the user queries. All these tasks are repeated at regular time intervals in order to keep the index (and therefore the results provided to the user queries) updated with respect to the continuous variations of the Web.

In the second category, we can also identify several important algorithmic steps which are executed at each user query and mainly consist of consulting the current index in order to discover the *relevant* pages for that query, ranking these pages in order of relevance, and possibly applying some classification or clustering tools that aim at offering different (and eventually more meaningful) views on the returned results. All these steps have as ultimate goal the one of satisfying in the best and the fastest way the information need hidden within the user queries.

A user query is typically formed by a sequence of *keyword*s. The process that leads the user to choose a specific set of keywords is critical since it significantly influences the quality of the results reported by the search engine. It is clear that an information need may be correctly settled by different groups of results. However, the actual relevance of these groups depends on the user submitting the query and her current information need, which may change, even drastically, from one user to another, and indeed it can change even for the same user depending on her specific interests at the time the query is issued. For example, the query Lufthansa for a user may have a *navigational* goal because she wants to find the homepage of the airline, a *transactional* goal because she might wish to buy a ticket, or an *informational* goal because she is interested in gathering some information regarding the company. And of course, the same user could issue queries at different times having different goals.

If the user is not completely satisfied by the results returned by the search engine, she could refine her query by adding keywords, or she could more clearly specify her intention by reformulating the query itself. However, this rarely happens: statistics show that more than 80 % of the queries are formed by only two keywords and their average number is around 2.5. Add to this that most users look at only the first page of results. This behavior is driven by not only user laziness in composing selective queries and browsing the returned results, but also in the intrinsic difficulty for users to model their information needs by means of appropriate keywords.

Despite all these problems, modern search engines perform their tasks very efficiently and provide very relevant results. Moreover, they are improving every day, thanks to intensive academic and industrial research in the field. In the following we will describe the salient operations performed by search engines noticing that, not surprisingly, many algorithms usually employed are not publicly known.

5.4.1 Crawling

In the slang of Internet, the term *crawling* refers to the (un-)focused retrieval of a collection of Web pages with the purpose of making them available for subsequent analysis, cataloguing and indexing of a search engine. A *crawler*, also named *spider* or *robot*, is an algorithm that automatically discovers and collects pages according to a properly-designed traversal of the Web graph. The reader should not confuse browsers with crawlers: the former retrieve and visualize specific pages indicated by a user via their URL; the latter retrieve and collect pages which are automatically identified via proper Web-graph visits. The following box details the algorithmic structure of a crawler.

Crawling Algorithm

A crawler makes use of two data structures called *queue* and *dictionary*, whose functionalities are close to the ones these terms assume in the context of transports (a queue of cars) and linguistics (a dictionary of terms), respectively. A queue Q is a list of elements waiting to be served. The element that is placed at the head of Q is the next that will be served and, when this happens, the element will be removed. In this way, the second element will become the new head of the queue. This operation is denoted by $Q \rightarrow e$ and indicates that element e is released outside. A new element is always inserted at the end of Q, and it will be served after all elements currently in Q. This operation is denoted by $e \rightarrow Q$.

A dictionary D is a set of elements (not necessarily words in some natural language) waiting to be examined. In this case, however, there is more flexibility than in the queue regarding the operations that can be supported. What concerns us here is, indeed, that there is an efficient way to determine whether a particular element e is already in the dictionary D and possibly remove it (denoted by $D \rightarrow e$). The dictionary is built by means of insertions of new elements (denoted by $e \rightarrow D$). Notice that, since we wish to perform fast searches in the dictionary D, we have to carefully insert and organize the elements into it.

Now we will introduce a (necessarily simplified) crawling algorithm whose fundamental steps are reported in Fig. 5.6. We first notice that the owner of a site could forbid the crawling of some Web pages, by adding a special file called `robots.txt`, which specifies which pages can be downloaded and which cannot.

The crawling algorithm deploys a queue Q containing addresses of Web pages to be processed, two dictionaries D_{urls} and D_{pages} containing, respectively, the addresses of the Web pages already processed and an archive of

(continued)

(continued)

information extracted from those pages. Initially, both D_{urls} and D_{pages} are empty, while Q contains a set of addresses that are the starting seeds of the crawling process. Not surprisingly, the choice of these initial seed pages is fundamental to quickly reach the most relevant part of the Web. Seed pages are usually Web portals (e.g., DMOZ, Yahoo!), educational sites (e.g., Wikipedia and universities), news and social-network sites, since they contain pages that point to important and popular resources of the Web.

The algorithm of Fig. 5.6 is not trivial. When a new link U' is found in a crawled page, its address is inserted in Q ready to be processed in the future. This is done only if the link U' is not already present in D_{urls}, which means that its text $T(U')$ has not been downloaded yet. Notice that the same link U' may be contained in several pages which are discovered by the crawling algorithm before that U''s content is downloaded. Thus, this check ensures that U' will be downloaded and inserted in D_{pages} only once.

Obviously, state-of-the-art crawling algorithms are more complex than the one presented here, and have to include sophisticated functionalities and optimizations. One of the most important issues regards the fact that the Web changes at such high rate that, as estimated, we have a 30 % renewal every year. So the crawlers must usually be trained to follow the more rapid variations (think, e.g., news sites), and designed to be as fast as possible in making "one scan of the Web" in order to keep the search engine index as fresh as possible. Moreover, the crawler should reduce the interactions with the crawled sites as mush as possible, in order to not congest them with continuous requests, and it should make use of advanced algorithmic techniques in *distributed computing* and *fault tolerance*, in order to ensure that it will never stop its operations. Therefore, the design and development of an efficient and effective crawler is much more complicated than what a reader could deduct from the algorithm reported in Fig. 5.6. The reader interested in those algorithmic details may look at the literature reported in Sect. 5.6.

We conclude this section by observing that the design of a crawler has to be optimized with respect to three parameters: the maximum number N of Web pages that can be managed before its algorithms and data structures are "overwhelmed" by the size of D_{url}; the speed S with which the crawler is able to process pages from the Web (nowadays crawlers reach peaks of thousands of pages per second); and, finally, the amount of computational resources (CPU, memory and disk space) used to complete its task. Clearly, the larger are N and S, the higher is the cost of maintaining the queue Q and the dictionaries D_{url} and D_{pages}. On the other hand, the more efficient is the management of Q, D_{url} and D_{pages}, the lower is the amount of computational resources used and the consumption of energy. The latter is nowadays an extremely important issue, given the high number of computers used to implement the modern search engines.

Crawling Algorithm

- Input: $\{u_1, \ldots, u_k\}$ an initial set of addresses of Web pages;
- Output: D_{urls} and D_{pages}.

---

1. Insert $u_1, \ldots, u_k$ into the queue Q;
2. Repeat until Q is non-empty
3. Extract $Q \rightarrow u$ the next page-address u from Q;
4. If $u \notin D_{\mathsf{urls}}$, then
5. Request the file `robots.txt` from the site of u;
6. If this file allows to access page u, then
7. Request the text $T(u)$ of the page u
8. Insert $u \rightarrow D_{\mathsf{urls}}$
9. Insert $T(u) \rightarrow D_{\mathsf{pages}}$
10. Parse $T(u)$, and for any link u' in $T(u)$
11. if $u' \notin D_{\mathsf{pages}}$, add $u' \rightarrow Q$

Fig. 5.6 A simple crawler using the urls $\{u_1, \ldots, u_k\}$ as initial seed set

5.4.2 *The Web Graph in More Detail*

At this point it is natural to ask how large the Web graph is and what is the structure of its interconnections, since the effectiveness of the crawling process is highly dependent on these characteristics.

We have already noticed that the Web is huge and rapidly changing. There is, therefore, no hope that a crawler can collect in D_{pages} all the existing Web pages; so it has necessarily to be resigned to obtaining only a subset of the Web which, hopefully, is as broader and more relevant as possible. In order to optimize the quality of the collected pages, the crawler has to perform a visit of the graph which is inevitably more focused and complex than the one used in Fig. 5.6. For this purpose, the crawler uses a more sophisticated data structure, called *priority queue* (see Chap. 4), that replaces the simple queue Q and extracts its elements depending on a priority assigned to each of them. In our case the elements are Web pages and the priorities are values related to the relevance of those pages. The higher the priority, the sooner the crawler will process the page and download its neighbors. The objective is that of assigning low priority to pages with a lower relevance or that have been already seen, or to pages that are part of a site which is too large to be collected in its entirety. To model this last case, we take into account the *depth* of a page in its site as a measure of its importance. The depth is measured as the number of forward slashes in its URL address (for example, the page http://www.unipi.it/ricerca/index.htm is less deep than the page http://www.unipi.it/ricerca/dottorati/index.htm, and thus is assumed to be more important in that site and hence crawled first). Recent studies have shown that the depth and the number and quality of the links incoming and outgoing from a page are effective indicators for the assignment of these priorities.

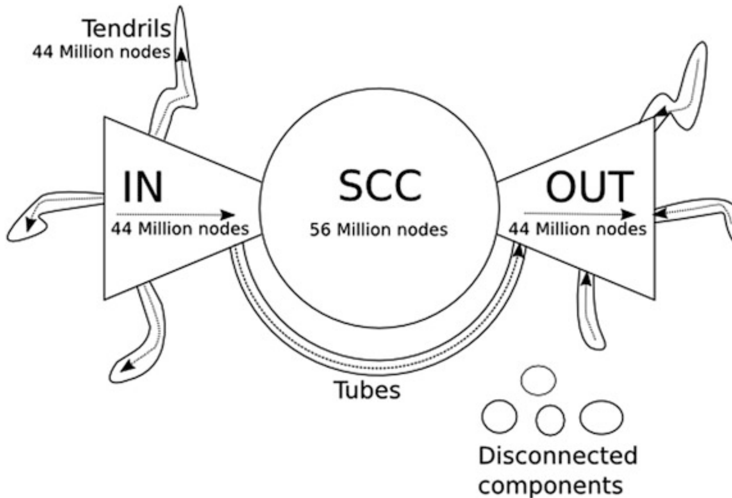

Fig. 5.7 The characteristic "bow" shape of the Web graph (1999). The subgraphs SCC, IN, OUT, tendrils and tubes, each consist of about one quarter of the nodes in the total graph

The structure of the graph significantly influences the behavior of the crawler. We consider two extreme examples that make this fact more evident. If the graph was made up of many disconnected components, the seed set of the crawler should contain at least one address for each of them; if the graph was instead made up of one (long) chain of pages, the seed set should contain the heading pages of this chain in order to guarantee that the graph visit traverses most of the Web. In November 1999 a study, now classic, analyzed the structure of a subgraph of the Web of that time formed by about 200 million pages. It turned out that this portion of the Web did not recall the two previous extreme examples, but it consisted of four main components shown in Fig. 5.7, all having about the same size: a strongly connected component, denoted SCC and called *core*,[5] a subgraph IN with paths that end up in pages of SCC, a subgraph OUT with paths that start at SCC, and a number of tendrils and tubes, namely pages linked in chains that do not pass through SCC or are completely isolated. These findings were later confirmed by studies carried out on larger and more recent samples of the Web: they actually showed not only that the sampled graph always has the form indicated in Fig. 5.7 and its components have about the same sizes, but also that the graph structure has some *fractal property* which leads any sufficiently large subgraph to have the same structure as its original containing graph. These surprising results are nowadays justified by mathematical studies on the laws that regulate the growth of the networks.

[5]Recall that a graph is strongly connected if and only if there exists a path that connects any pair of its nodes.

This structure of the Web graph suggests to insert in the seed set of the crawler pages chosen from IN or SCC. The problem is how to efficiently determine candidates from these two sets of nodes before a visit of the Web graph is performed. If candidates are chosen randomly from the Web, then we would have a probability $1/2$ that each selected page is either in IN or in SCC, given that each of these sets is $1/4$ of the total graph. Thus, it would be enough to choose a small number of candidates to be reasonably sure to start the crawling from IN or SCC. Unfortunately, this strategy is difficult to implement because there are neither available list of all the addresses of existing pages nor it is clear how to perform uniform sampling without this list. So the typical choice of existing crawlers is the simple one sketched above, and consists of taking as seeds the portals and any other sites that can lead to good pages of the Web and probably lie in IN or SCC.

5.4.3 Indexing and Searching

The pages collected by the crawler are subsequently processed by an impressive number of algorithms that extract from them a varied set of information that is stored in proper data structures, called *index*es, allowing it to efficiently answer the user queries. Entire subnets of computers are dedicated to this significant task in order to perform it in a reasonable amount of time.

Hereafter we will talk about *document*s, instead of pages, and with this term we will refer to the content of a Web page p plus some other information about p itself collected during the crawling process. An example of such additional information is the so-called *anchor text*, which corresponds to a portion of text surrounding a link to p in another page. Given that page p may have many incoming links, page p may have many anchor texts written by authors who are possibly different from the author of p. An anchor text may be therefore a particularly reliable and important piece of information because, presumably, it describes succinctly the content of p. Not surprisingly, search engines give great importance to anchor texts and use them to extend the content of Web pages, because they typically use a different set of words to describe their content and thus allow search engines to extend the results of a query.

For example, let us assume that a page p contains pictures of various species of insects but does not contain the word "insect(s)" in its body, or even does not contain any text at all, consisting just of that picture. Nevertheless it is possible that a passionate entomologist wrote his own Web page with a link to page p and annotated this link with the phrase "beautiful images of insects". This piece of text is an anchor for p, so the words in "beautiful images of insects" are added to those found in p and are considered highly characteristic for this page. Therefore a query "insects" would find p, even if it does not contain that word.

Unfortunately, as often happens in the Web, a valuable use of information is followed by a malicious one of it. In 1999 the first result reported by Google for the

query “more evil than Satan” was the homepage of Microsoft. This phenomenon was eventually the result of the creation of many pages containing links to the homepage of Microsoft with anchor text “more evil than Satan”. This awkward situation was resolved by Google in a few days, but a similar accident happened again in November 2003 with the query “miserable failure” and the page returned by Google as the first result was the homepage of the former U.S. President George W. Bush. This type of attack is nowadays called *Google bombing*, and it was repeated many other times in languages other than English.

Once the search engine has crawled a huge set of documents, it analyzes them to extract the *term*s contained therein and inserts these terms in a dictionary. Checking the presence of query terms in this dictionary is the first step performed during a query resolution. We observe that a query term may be not just a word, but any sequence of alphanumeric characters and punctuations because it may represent telephone numbers (911 or 1-800-237-0027), abbreviations (e-ticket), models of our favorite devices (N95, B52, Z4), codes of university courses (AA006), and so on. We cannot go into the details of efficient implementations of dictionary data structures, but we observe here that it is unfeasible to implement keyword searches through a linear scan of the dictionary, because this would take too much time.[6] It is thus crucial to have appropriate algorithms and data structures to manage efficiently, both in time and space, these numerous and long sequences of characters. One approach could be the dichotomous search algorithm presented in Fig. 2.24; more efficient solutions are known, mainly based on *tries* and *hash tables*, so we refer the reader to the literature mentioned at the end of this chapter.

The dictionary is just a part of the mass of information extracted by a search engine from the crawled documents, during the so-called *indexing* phase. The overall result of this phase is the construction of a data structure, called an *inverted list*, which is the backbone of the algorithms answering the user queries. An inverted list is formed by three main parts: the above dictionary of terms, one list of occurrences per term (called *posting list*), plus some additional information indicating the importance of each of these occurrences (to be deployed in the subsequent ranking phase). The word “inverted” refers to the fact that term occurrences are not sorted according to their position in the text, but according to the alphabetic ordering of the terms to which they refer. So inverted lists remind the classic *glossary* present at the end of a book, here extended to represent occurrences of all terms present into a collection of documents.

The posting lists are stored concatenated in a single big array kept in memory. The URLs of the indexed documents are placed in another table and are

[6] Several experimental results have shown that the number n of distinct terms in a text T follows a mathematical law that has the form $n = k|T|^{\alpha}$, with k equal to a few tens, $|T|$ being the number of words of the text, and α approximately equal to $1/2$. The actual size of the Web indexed by search engines is hundreds of billions of pages, each with at least a few thousand terms, from which we derive $n > 10 \times 10^6 = 10^7$. Thus, the dictionary can contain hundreds of millions of distinct terms, each having an arbitrary length.

D (Dictionary)

Term	Post
…	
foot	…
football	90
footing	…
footnote	…
…	

U (Urls)

docID	Web Address
…	
50	www.nfl.com
…	
100	www.bbc.co.uk/football
…	
500	www.afl.com.au
…	

P (Posting Lists)

i	1 2 3 . . . . . 90 91 92 93 94 . . . .
P	. . . . 50 1 5 100 3 15 17 500 2 15 20 # . . . .

Fig. 5.8 Indexing by using inverted lists. From the dictionary D we know that the list of documents containing the term "football" starts at position 90 in the array P of posting lists. The term "football" is contained in the Web pages whose docID is 50, 100, 500, and whose URL address is reported in table U

succinctly identified by integers, called *docID*s, which have been assigned during the crawling process. The dictionary of terms is also stored in a table which contains some satellite information and the pointers to the posting lists. Of course, the storage of all terms in the documents impacts the total space required by the index. Previously, software engineers preferred to restrict the indexed terms to only the most significant ones; nowadays, search engines index essentially all terms extracted from the parsed documents because advances in data compression allowed engineers to squeeze terms and docIDs in reduced space and still guarantee fast query responses. Actually, search engines also store the *positions* of all term occurrences in each indexed document because this information is used to support phrase searches and to estimate the relevance of a document with respect to a query, based on the distance between the query terms in that document. It is evident that such a refined set of information has huge size and thus necessitates sophisticated compression techniques. The literature reports several studies about this issue, nonetheless the actual compressors adopted by commercial search engines are mostly unknown.

Figure 5.8 illustrates the structure of an inverted list. Each term t ("football" in the example) has associated a subarray of P which stores, in order, the docID of a document d containing term t (the first document in the example is 50), the number of times that t occurs in d (1 in the example), the position in d of each of these term occurrences (position 5 in the example). The posting list of t ends with a terminator #. From the figure we notice that the term t = `football` is contained in document 50 at one single position, i.e., 5; in document 100 the term occurs in three positions (15, 17 and 25); in document 500 it occurs in two positions (15 and 20). It is convenient to store the docIDs of each posting list in increasing order (50, 100, 500 in the example) because this reduces the space occupancy and the time required to solve future queries. In this case each docID can be stored as the *difference* with respect to its preceding docID. The same method can be used to succinctly store the positions of the occurrences of term t in document d. So, in the posting list of Fig. 5.8, we can represent the sequence of docIDs 50 100 500 as 50 50 400: the first 50 is exactly represented, since it has no preceding docID, whereas we have $100 - 50 = 50$ and $500 - 100 = 400$ for the next two docIDs. By also inserting the

occurrences of the term, encoded similarly, the compressed posting-list of the term "football" becomes:

$$50\ 1\ 5\ 50\ 3\ 15\ 2\ 8\ 400\ 2\ 15\ 5\ \#.$$

The original posting list is easily reobtained from the compressed one by a simple sequence of additions. We are speaking about "compression" because the obtained numbers are smaller than the original ones, so we can squeeze the total space usage by means of appropriate integer coders that produce short bit sequences for small integers.

As we anticipated above, the order of the docIDs in the posting lists is important also to efficiently answering queries that consist of more than one keyword. Imagine that a user has formulated a query with two keywords, t_1 and t_2 (the extension to more keywords is immediate). Solving this query consists of retrieving the posting lists L_1 and L_2 of docIDs referring to t_1 and t_2, respectively. As an example, take $L_1 = 10\ 15\ 25\ 35\ 50\ \ldots\#$ and $L_2 = 15\ 16\ 31\ 35\ 70\ldots\#$, where we are assuming to have already decompressed the lists. Now the problem is to identify the documents that contain both t_1 and t_2 (namely, the elements in common to both the two posting lists). The algorithm is deceptively simple and elegant; it consists of scanning L_1 and L_2 from left to right comparing at each step a pair of docIDs from the two lists. Say $L_1[i]$ and $L_2[j]$ are the two docIDs currently compared, initially $i = j = 1$. If $L_1[i] < L_2[j]$ the iterator i is incremented, if $L_1[i] > L_2[j]$ the iterator j is incremented, otherwise $L_1[i] = L_2[j]$ and thus a common docID is found and both iterators are incremented. If we let n_1 and n_2 be the number of elements in the two lists, we can realize that this algorithm requires time proportional to $n_1 + n_2$. A each step, indeed, we execute one comparison and advance at least one iterator. This cost is significantly smaller than the one required to compare each element of L_1 against all elements of L_2 (which is $n_1 \times n_2$), as would happen if the lists were not sorted. Since the values of n_1 and n_2 are on the order of some hundreds of thousands (or even more for the common terms), the latter algorithm would be too slow to be adopted in the context of a search engine answering millions of queries per day.

5.4.4 Evaluating the Relevance of a Page

Since user queries consist of a few keywords, the number of pages containing these keywords is usually huge. It is thus vital that a search engine sorts these pages and reports the most "relevant" ones in the top positions to ease their browsing by the user. However, an accurate characterization of what is the "relevance" of a Web page has a high degree of arbitrariness. Nonetheless, this is probably the most important step in modern search engines, so that a bunch of sophisticated algorithms have been proposed to efficiently quantify that relevance. This step goes under the name of *ranking*, and its solution represents the main point of distinction between

the major known search engines. It is indeed not exaggerated to affirm that one of the key ingredients that enabled Google to achieve its enormous popularity was its algorithm for ranking the pages shown to the user, called *PageRank* (see below for details).[7]

Nowadays the relevance of a page p is measured by combining several parameters: the type and the distribution of the occurrences of the query-terms in p, the *position* of p in the Web graph and its interconnections with other pages, the frequency with which Web users visit p as a result of a user query, and many other factors, not all revealed by search engines. In particular, it is known that Google uses about 100 parameters! We present below the two most important measures of relevance for Web pages known in the literature, prefacing them with some considerations that will allow us to understand their inspiring motivations.

It is natural to think that the relevance of a term t for a document d depends on the frequency $TF[t,d]$ of occurrence of t in d (called *Term Frequency*), thus, on the *weight* that the author of d has assigned to t by repeating this term several times in the document. However, considering only the frequency may be misleading because, for example, the articles and prepositions of a language are frequent in texts without characterizing them in any way. Thus, it is necessary to introduce a correction factor which also takes into account the *discriminative power* of a term which is very low for secondary linguistic elements. However, the situation is more complicated, since a term like "insect" may be discriminant or not, depending on the collection of documents in its entirety: "insect" is unusual and probably relevant for a collection of computer-science texts, whereas it is obvious and therefore irrelevant for a collection of entomology texts. It is therefore crucial to consider the rarity of a term in the collection by measuring the ratio between the number ND of documents in the collection and the number $N[t]$ of documents containing the term t. The rarer the term t, the larger the ratio $ND/N[t]$, and thus t is potentially more discriminative for the documents in which it is contained. Usually this ratio is not directly used to estimate the discrimination level of t, but it is mitigated by applying the logarithmic function. This defines the parameter $IDF[t] = \log_2(ND/N[t])$, which is called the *Inverse Document Frequency*. In this way, the measure is being not too sensitive to small variations in the value of $N[t]$. On the other hand, it is not always true that a rare word is very relevant for the document d. For example, the presence of the word may be caused by a typing error. Therefore, term frequency and inverse document frequency are combined to form the so-called TF-IDF measure of relevance of a term in a document. This combination was proposed in the late 1960s and is given by the formula $W[t,d] = TF[t,d] \times IDF[t]$. Notice that if t is, say, an article, it appears frequently in almost all the documents in the collection. Thus, its ratio $ND/N[t]$ is very close to the value 1 and its logarithm is close to the value 0, thus forcing a small value of $W[t,d]$. Similarly, a term typed incorrectly will have a

[7] Google trusts so much in its ranking algorithm that it still shows in its homepage the button "I'm feeling lucky" that immediately sends the user to the first ranked page among the results of her query.

small value for *TF*, thus forcing again a small value of $W[t,d]$. In both cases then the term relevance will be correctly evaluated as not significant. Numerous linguistic studies have corroborated the empirical validity of the TF-IDF weight which is now at the basis of any information retrieval system.

The first generation of search engines, such as Altavista, adopted the TF-IDF weight as a primary parameter to establish the importance of a Web page and sorted the query results accordingly. This approach was effective at the time in which the Web content was mainly restricted to government agencies and universities with authoritative pages. In the mid-1990s the Web was opened to the entire world and started to become a huge "shopping bazaar", with pages composed without any control in their content. All this led some companies to build "rigged" pages, namely pages that contained, in addition to their commercial offerings, also the set of properly concealed keywords that frequently appeared in user's queries. The net aim was to promote the relevance of these pages, even in other contexts. Thus, it was evident that the textual TF-IDF weight alone could not be used to assess the importance of a page, but it was necessary to take into account other factors specific to the Web graph.

Since the mid-1990s several proposals came from both academia and industry with the goal of exploiting the links between pages as a *vote* expressed by the author of a page p to the pages linked by p. Among these proposals, two ranking techniques gave rise to the so-called *second generation search engines*: the first technique, called *PageRank*, was introduced by the founders of Google, Larry Page and Sergey Brin, and the second technique, called *HITS* (Hyperlink Induced Topic Search), was introduced by Jon Kleinberg when he was at IBM. In PageRank each page is assigned with a relevance score which is independent of its textual content, and thus of the user query, but depends on the Web-graph topology. In HITS each page is assigned with two relevance scores which depend on the topology of a subgraph selected according to the user query. Although very different, PageRank and HITS have two common features: they are defined recursively, so the relevance of a page is computed from the relevance of the pages that are linked to it; they involve computations on very large matrices derived from the structure of the Web graph, so they need sophisticated mathematical tools (readers less familiar with linear algebra can jump to Sect. 5.4.6).

5.4.5 Two Ranking Algorithms: PageRank and HITS

PageRank measures the relevance of a page p according to its "popularity" in the Web graph, which in turn is computed as a function of the number and origin of links that point to p. In mathematical terms, the popularity $R(p)$ is computed as the probability that a user will reach page p by randomly walking over the Web graph, traversing at each step one of the links outgoing from the currently visited page, each selected with equal probability. Let $p_1, \ldots, p_k$ be the pages having at least one link to p, and let $N(p_i)$ be the number of pages linked by p_i (i.e., the number of

outgoing links from p_i in the Web graph). The basic formula for the calculation of $R(p)$ is the following:

$$R(p) = \sum_{i=1...k} (R(p_i)/N(p_i)). \tag{5.2}$$

Notice that only the pages having a link to p directly contribute to the value of $R(p)$, and moreover, this contribution is proportional to the relevance of these pages scaled by their number of outgoing links. The ratio underlying this formula is that if a page p_i with a certain relevance $R(p_i)$ points to p, it increases the popularity of p, but this increment should be equally shared among all the pages to which p_i points.

It is evident that the formula is recursive, and its computation presents some technical problems because it requires us to specify the initial value of $R(p)$, for all pages p, and to indicate how to deal with pages that do not have either incoming or outgoing edges. To address these two issues, we consider a slightly different formula that introduces a correction factor taking into account the possibility that a user leaves the link-traversal and jumps to a randomly chosen page in the Web graph. This change allows the random walker to not remain stacked into a page that has no outgoing links, or to reach a page even if it has no incoming links. Therefore, the formula becomes:

$$R(p) = d \sum_{i=1...k} (R(p_i)/N(p_i)) + (1-d)/n, \tag{5.3}$$

where n is the number of pages collected by the crawler and indexed by the search engine, and d is the probability of continuing in the link-traversals, whereas $(1 - d)$ is the complement probability of jumping to a randomly chosen page in the crawled graph. In the extreme case that $d = 0$, all pages would obtain the same relevance $R(p) = 1/n$, while in the case of $d = 1$ the relevance $R(p)$ would entirely depend on the structure of the Web graph and it would show the problems mentioned above. Experiments have suggested taking $d = 0.85$, which actually attributes more importance to the relevance that emerges from the structure of the Web.

The real computation of $R(p)$, over all pages p, is performed by resorting to matrix operations. We have already seen that the Web graph may be represented by a matrix W whose powers W^k indicate the number of paths of length k between pairs of nodes in the graph (see box *Adjacency matrix and paths of a graph*). For this reason, we introduce a matrix Z of size $n \times n$, whose elements have value $Z[i, j] = d \times W[i, j] + (1 - d)/n$. The value $Z[i, j]$ represents the probability that a random walker traverses the link from p_i to p_j, while the matrix-powers Z^k represent the probability that paths of length k are traversed by that random walker.

We can also represent the relevance of the pages in vector form: $R[i]$ is the relevance of page p_i; and use the notation $R_k[i]$ to denote the relevance of page p_i

after k iterations of the algorithm. At this point we can compute the configurations for all R_k as:

$$R_1 = R_0 \times S, R_2 = R_1 \times Z = R_0 \times Z^2, \ldots, R_k = R_0 \times Z^k. \quad (5.4)$$

This formula is related to a deep mathematical theory known as *Markov's chains*, which is, however, too difficult to be discussed here. We note that this theory guarantees that the limit value for $R_k[i]$, when $k \to \infty$, equals the probability that a random walker traverses page p_i, and it also guarantees that this limit value does not depend on the initial conditions R_0, which can then be assigned arbitrarily.

Obviously, the calculation indicated in Eq. (5.4) is not trivial due to the size of the matrices involved, which are indeed huge since they consist nowadays of hundreds of billions of pages and hence have a total size of at least 25×10^{20} elements! However, the computation of R_k is simplified by the fact that we do not need to care about the exact values of its elements, since it suffices to determine only their order: if $R(p_1) > R(p_2)$, then page p_1 is more important than page p_2. Therefore, we can stop the above computation whenever the values of R_k's components are sufficiently stable and their order can be determined with some certainty. Experimental tests showed that about 100 iterations suffice.

We conclude the discussion on PageRank by recalling that it induces an ordering among pages which is a function only of the graph structure and thus it is independent of the user query and page content. Therefore, PageRank can be calculated at the indexing phase, and deployed at query time in order to sort the result pages returned for a user query. Many details on the current version of PageRank are unknown, but numerous anecdotes suggest that Google combines this method with TF-IDF and a 100 other minor parameters extracted automatically or manually from the Web.[8]

Let us study now the HITS algorithm, which is potentially more interesting than PageRank because it is *query dependent*. For a given query q, it retrieves first the set P of Web pages that contain all query terms, and then it adds those pages that point to or are pointed to by pages in P. The resulting collection is called the *base set* and contains pages that are related to q either directly (because they contain the query terms) or indirectly (because they are connected to a page in P). The situation is shown in Fig. 5.9a.

A (sub-)graph is then built by setting the pages in the base set as nodes and the links between these pages as edges of the (sub-)graph. For each node, we calculate two measures of relevance, called *authority* and *hubness* scores. The first score, denoted with $A(p)$, measures the authoritativeness of page p relative to the query q.

[8] In a recent interview, Udi Manber (VP Engineering at Google) revealed that some of these parameters depend on the language (ability to handle synonyms, diacritics, typos, etc.), time (some pages are interesting for a query only if they are fresh), and templates (extracted from the "history" of the queries raised in the past by the same user or by her navigation of the Web).

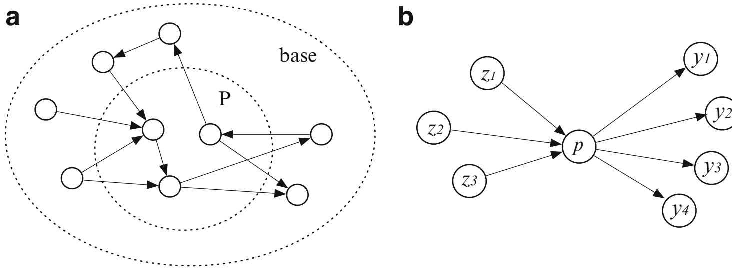

Fig. 5.9 (**a**) The set P of pages that contain the terms of a query q, and its base set; (**b**) the pages z_i and y_i that contribute to determine respectively the *authority* and the *hubness* score of a page p

The second score, denoted with $H(p)$, measures how much the p's content is a good survey for the query q (i.e., a directory that points to many authoritative pages on the subject). This way, a page p having a large value of $A(p)$ is an *authority* for q, while a large value of $H(p)$ implies that p is a *hub* for q. Computing these two measures follows their intuitive meanings: a page p is a good hub (and, thus, the value $H(p)$ is large) as p points to many authoritative pages; a page p is a good authority (and, hence, the value of $A(p)$ is large) as more good hubs point to p. We can formalize the previous insights with the following two formulas:

$$\begin{aligned} A(p) &= \textstyle\sum_{i=1\ldots k} H(z_i); \\ H(p) &= \textstyle\sum_{i=1\ldots k} A(y_i), \end{aligned} \tag{5.5}$$

where $z_1, \ldots, z_k$ denote the pages that point to p, and $y_1, \ldots, y_h$ denote the pages pointed to by p (see Fig. 5.9b). Similarly to what was done for the PageRank, we can compute the two scores by resorting matrix computations. We then define the adjacency matrix B of the graph induced by the base set, and we compute the vectors A and H with the formulas (5.5) via matrix computations involving B. These computations are similar to the ones performed for PageRank with two essential differences. The first concerns the size of the matrices, which is now moderate since B usually consists of only a few thousands of nodes. The second is that the calculation has to be executed on-the-fly at query time because B is not known in advance, and thus the values of A and H cannot be precalculated. This represents a strong limitation for the application of this method on the Web, and in fact HITS was originally proposed for search engines operating on small collections of documents and for a limited number of users (e.g., on a company intranet). Another limitation of this approach resides in its small robustness to *spam* (see Sect. 5.4.6); for this reason in the literature this issue got some attention with many interesting proposals.

5.4.6 On Other Search Engine Functionalities

Among the other operations that a search engine is called to perform, the presentation to a user of the results of her query has great importance. Search engines show, for each result page, a short textual fragment, known as a *snippet*, which represents the context surrounding the query terms in that page. Other search engines offer also the possibility of viewing a copy of a result page as it was retrieved by the crawler. This is particularly useful whenever the link returned for a result page is broken, due to the fact that this page was removed from the Web since it was crawled. The copy of this page can thus be useful to retrieve the indexed page, even if it is no longer present in the Web.

We emphasize that the results of search engines are sometimes corrupted with sophisticated techniques, which fraudulently increase the relevance of certain pages to let them appear among the first results for a user query. This is known as *spamming* and consists of constructing proper subgraphs of the Web that artificially increase the relevance of the fraudulent pages. Other forms of spamming are more subtle. One technique, known as *cloaking*, is adopted by fraudulent servers to mask the real content of their pages and thus make them the result of queries which are not related to their content. The cloaking idea makes servers return to the search engine some good content taken, for example, from Wikipedia at each crawling request. If the artifact is relevant for a user query, the search engine will then display a snippet appropriate and interesting for the user and referring to a Wikipedia page. However, the page that appears to the user after clicking on the link shown in the snippet is totally different and possibly contains irrelevant (if not offensive) content.

Certainly, an exhaustive discussion on spamming methods cannot be addressed in this short chapter. However, it is interesting to note that spamming has a large reach, as it is estimated that more than 20 % of Web pages consist of artifacts that endanger the reputation and usefulness of search engines. In all their phases, consequently, search engines adopt sophisticated anti-spam algorithms to avoid the gathering, the indexing and the reporting of these artifacts. As for all the previous problems, the solutions currently employed by commercial search engines are only partially revealed, in part to make the job of spammers more difficult.

We finally remark that the goal of search engines is moving toward the identification of user intentions hidden behind the purely syntactic composition of their query. This explains the proliferation of different methods that cluster the query results on the screen (started by the search engine Vivisimo.com), that integrate different sources of information (news, Wikipedia, images, video, blogs, shopping products, and so on), and that possibly provide suggestions for the composition of refined queries (Google Suggest and Yahoo! Search Suggest are the most notable examples). In addition, search engines have to tackle the fact that users are moving from being *active agents* in the search process to becoming more and more *passive spectators*: advertising, suggestions, weather forecasts, friends on-line, news, and so on, are all information that we probably set as interesting in some personal record or alerts, or that the search engines have in some way inferred as interesting for us

given our Web life. All of this information already appears, or will appear more and more frequently in the near future, on our screens as a result of a query, or on our email readers, our personal pages on iGoogle or MyYahoo!, or even on applications in our smartphones.

Many other features of search engines deserve to be carefully studied and discussed. An interested reader can find many inspiring references in the review of the literature at the end of this chapter.

5.5 Towards Semantic Searches

Although Web search engines are fairly new, researchers and software engineers achieved during the last two decades significant improvements in their performance. These achievements identified many other interesting avenues of further research which should lead in the near future to implementing more efficient and effective Information Retrieval (IR) tools. In fact, although the algorithms underlying the modern search engines are much sophisticated, their use is pretty much restricted to retrieving documents by keywords. But, clearly, users aim for much more!

Keyword-based searches force users to abstract their needs via a (usually short) sequence of terms; this process is difficult and thus error prone for most of Web users, who are unskilled. It would surely be more powerful to let users specify their needs via natural-language queries: such as "Will it rain in Rome within the next three hours?", and get more precise answers than just an ordered list of pages about Rome, or a Web page about the weather in Rome, such as "yes, it will rain on the coast". Interestingly enough, this is not just a matter of ranking; we are actually asking the search engine to understand the *semantics* underlying the user query and the content of the indexed pages. Some interesting research is actually going on, trying to address these issues by adding metadata to pages in order to better describe their content (known as the *Semantic Web*, and as the *Resource Description Framework*), or by adding some structure to pages in order to simplify the automatic extraction of useful information from them (known as *Open Data*), or by developing powerful Natural Language Processing techniques that better interpret short phrases up to long documents. This last research line has led to some interesting results that we will sketch briefly in this final part of the chapter.

The typical IR approach to indexing, clustering, classification, mining and retrieval of Web pages is the one based on the so-called *bag-of-words paradigm*. It eventually transforms a document into an array of terms, possibly weighted with TF-IDF scores (see above), and then represents that array via a highly dimensional point in a Euclidean space. This representation is purely syntactical and unstructured, in the sense that different terms lead to different and independent dimensions. Co-occurrence detection and other processing steps have been thus proposed to identify the existence of synonymy relations, but everyone is aware of the limitations of this approach especially in the expanding context of short (and thus

poorly composed) documents, such as the snippets of search-engine results, the tweets of a Twitter channel, the items of a news feed, the posts on a blog, etc.

A good deal of recent work attempts to go beyond this paradigm by enriching the input document with additional *structured annotations* whose goal is to provide a contextualization of the document in order to improve its subsequent "automatic interpretation" by means of algorithms. This general idea has been declined in the literature by identifying in the document short and meaningful sequences of terms, also known as *entities*, which are then connected to unambiguous topics drawn from a catalog. The catalog can be formed by either a small set of specifically recognized types, most often People and Locations (also known as *named entities*), or it can consists of millions of generic entities drawn from a large knowledge base, such as Wikipedia. This latter catalog is ever-expanding and currently offers the best trade-off between a catalog with a rigorous structure but with low coverage (like `WordNet` or `CYC`), and a larger catalog with wide coverage but unstructured and noisy content (like the whole Web).

To understand how this annotation works, let us consider the following short news: "Diego Maradona won against Mexico". The goal of the annotation is to detect "Diego Maradona" and "Mexico" as significant entities, and then to hyperlink them with the Wikipedia pages which deal with the two topics: the former Argentinean coach and the Mexican football team. The annotator uses as entities the anchor texts which occur in Wikipedia pages, and as topics for an entity the (possibly many) pages pointed in Wikipedia by it, e.g., "Mexico" points to 154 different pages in Wikipedia. The annotator then selects from the potentially many available mappings (entity-to-topic) the most pertinent one by finding a collective agreement among all entities in the input text via proper scoring functions. There exist nowadays several such tools[9] that implement these algorithmic ideas and have been successfully used to enhance the performance of classic IR-tools in classification and clustering applications. Current annotators use about eight million entities and three million topics.

We believe that this novel entity-annotation technology has implications which go far beyond the enrichment of a document with explanatory links. Its most interesting benefit is the structured knowledge attached to textual fragments that leverages not only a *bag of topics* but also the powerful *semantic network* defined by the Wikipedia links among them. This automatic *tagging* of texts mimics and automates what Web users have done with the advent of Web 2.0 over various kinds of digital objects such as pages, images, music and videos, thus creating a new parallel language, named "*folksonomy*". This practice has made famous several sites, such as Flickr, Technorati, Del.icio.us, Panoramio, CiteULike, Last.fm, etc. Topic annotators could bring this tagging process to the scale of the Web, thus improving the classification, clustering, mining and search of Web pages, which then could be driven by topics rather than keywords. The advantage would be

[9] See, for example, TAGME (available at tagme.di.unipi.it), and WIKIPEDIA MINER (available at http://wikipedia-miner.cms.waikato.ac.nz/).

the efficient and effective resolution of ambiguity and polysemy issues which often occur when operating with the purely syntactic bag-of-words paradigm.

Another approach to enhance the mining of Web pages and queries consists of extracting information from *query-logs*, namely the massive source of queries executed by Web users and their selected results (hence, pages). Let us assume that two queries q_1 and q_2 have been issued by some users and that they have then clicked on the same result page p. This probably means that p's content has to do with those two queries, so that they can be deployed to extend p's content, much as we did with anchor texts of links pointing to p in the previous sections. Similarly, we deduce that queries q_1 and q_2 are probably correlated and thus one of them could be *suggested* to a user as a refinement of the other query. As an example, Google returns for the two queries "iTunes" and "iPod" the page http://www.apple.com/itunes/ as the first result. So we expect that many users will click on that link, thus inferring a semantic relation between these two queries.

There are also cases in which the same query q might lead users to click on many different page results; this might be an indication that either those pages are similar or that q is polysemous. This second case is particularly important to be detected by search engines because they can then choose to adopt different visualization forms for the query results in order to highlight the various *facets* of the issued query and/or *diversify* the top answers with samples of pertinent topics. As an example, let us consider the query "eclipse", which could be issued by a user interested in astronomical events, or in the software development framework, or in a plane model (Eclipse 500), or in a Mitsubishi car. So the query-log will contain many pages which are semantically connected to the query "eclipse", all of them pertinent with its various meanings.

It is therefore clear at this point that the analysis of all queries issued to a search engine and of the clicks performed by their issuing users can lead us to construct a huge graph, called the *query-log graph*, which contains an impressive amount of semantic information about both queries and pages. The mining of the structure and content of this graph allows us to extract impressive amounts of useful knowledge about the "folksonomy" of Web searches, about the community of Web users and their interests, about the relevant pages frequently visited by those users and their semantic relations. Of course, a few clicks and searches are error prone, but the massive amounts of issued queries and user clicks made every day by the Web community make the information extractable from this graph pretty robust and scalable to an impressive number of topics and Web pages. It is evident that we are not yet at a full understanding of the content of a page, neither are we always able to disambiguate a query or fully understand the user intent behind it, but we are fast approaching those issues!

5.6 Bibliographic Notes

Web search is a complex topic which was worth thousands of scientific papers in the last three decades. In this section we report the books and some scientific articles that have advanced the history of this fascinating and rapidly evolving field of research. These references offer a good and accessible treatment of the various topics dealt with by this chapter.

The book written by Witten et al. [113] contains a general study of the characteristics of search engines and implications on their use. This very clear text represents a great source of knowledge that does not enter much into algorithmic details. Two recent books by Manning et al. [75], and by Baeza-Yates and Ribeiro-Neto [7], describe the basics of Information Retrieval, whereas the book by Chakrabarti [17] provides a complete introduction on the gathering and analysis of collections of Web pages. Finally, the book by Witten et al. [112] is a fundamental reference for what concerns the organization and the processing of massive data collections (not necessarily formed by Web pages).

As far as papers are concerned, we mention the publications by Hawking [57,58] and Lopez-Ortiz [74], which offer two ample and clearly written surveys on the algorithmic structure of search engines. On the other hand, the paper by Zobel and Moffat [116] concentrates on indexing data structures, describing in much detail Inverted Lists. Two historical papers are the ones published by Broder et al. [12] on the analysis of the Web graph, and by Brin and Page [11] on PageRank, thus laying down the seeds for Google's epoch.

As far as Web tools are concerned, we point out the papers by Fetterly [42], who deals with spamming and adversarial techniques to cheat crawlers, by Baeza-Yates et al. [8], who describe the scenario of semantic search engines, and by Silvestri [101], who surveys the use of query-logs in many IR applications. Finally, we mention the papers by Ferragina and Scaiella [39] and Scaiella et al. [98], which review the literature about "topic annotators" and their use in search, classification and clustering of documents. We conclude this chapter by pointing out to the readers Google's blog by Alpert et al. [5], in which these engineers claimed that Google crawled and indexed in July 2008 about one trillion pages.

Acknowledgements We would like to thank Fabrizio Luccio, who contributed to the writing of the Italian version of this chapter for Mondadori.