J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

SEARCHING

Searching is the process of seeking a desired element in a set of related elements. The task of searching is one of the most frequent operations in computer science. There exist several basic variations of the theme of searching, and many different approaches, data structures, and algorithms have been developed on this subject.

Figure 1 describes graphically the basic components of searching in the natural and the digital world. In our natural world, a human having an information need is *searching* a data set. In the digital world, an information need (or query) of a user or application is pursued over a data set stored in computer memory. An *algorithm* is used to carry out the task of searching. As searching is a very frequent operation, in many cases the data are structured and stored in such a way so that it facilitates that task. However, the efficiency of searching is not the only factor that determines the structuring of data. Other factors are the storage space required, the efficiency of updating, and so on. As a consequence, in many cases auxiliary data structures are created and maintained in order to speed up the task of searching. These auxiliary data structures store data that are derived from the original data set and are exploited by the searching algorithm.

Outline

We can categorize the searching approaches and techniques according to three basic questions: *where*, *what*, and *how*.

The first question, *where*, concerns the type and the structure of the data set over which searching takes place. Roughly, we can distinguish the following kinds of data set:

- Sequences of Records For example, the data set can be a (sorted) table (or a file) of integers (or records).
- Sequences of Characters For example, the data set can be a string stored in main memory, or a text file stored in secondary memory.
- Graphs For example, the data set can be the map of a metro network. A special case of graphs is *trees*; for example, the data set can be a taxonomy of classes, a set of geographical names structured by spatial inclusion relation, or a game tree.
- Tables For example, the data set can be a relational database.
- *k*-Dimensional Spaces For example, the data set can be a set of points in a three-dimensional space or a two-dimensional array of pixels (a digital image).

We can also distinguish data sets that consist of *composite* data elements. For instance, the data set can be:

- A set of *documents*, where a document can be seen as a sequence of strings and images, and furthermore, a document may be structured in sections, subsections etc.

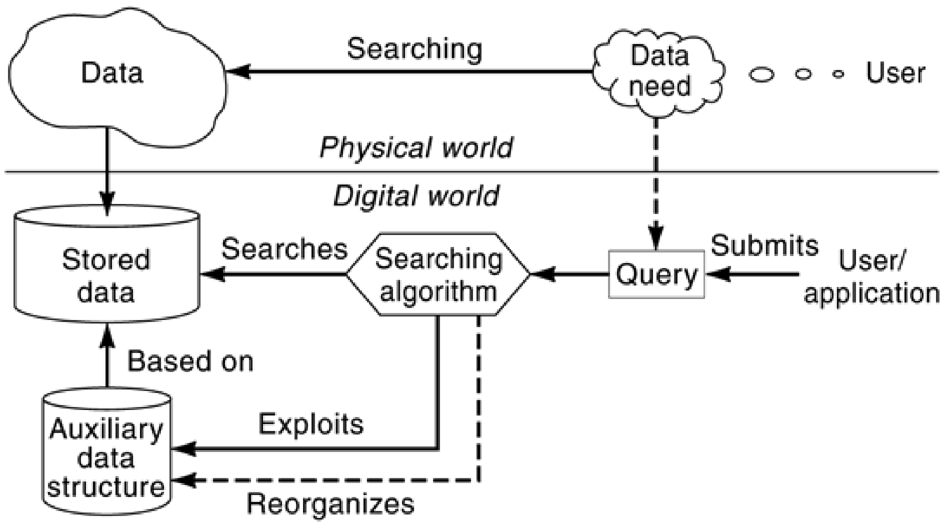

Fig. 1. The basic components of searching in the natural and the digital world.

- A set of Web pages, where we can view the Web as a distributed stored graph where each node is a document

Searching in this kind of data sets is more sophisticated.

The second question, *what*, concerns the objective of searching and the method used for specifying this objective. In particular, the objective of searching may be an element or a set of elements. The desired element(s) can be specified by some key, by other information related to the key, or by specifying the conditions that the desired elements must fulfil. Furthermore, in some cases the specification of the objective is done gradually, that is, it is the outcome of a process (e.g. the navigation process in a hierarchy of subjects, or the query reformulation through relevance feedback in information retrieval systems).

The third question, *how*, concerns the algorithm, that is, the sequence of steps, for finding the desired element(s) in the data set. As mentioned earlier, for speeding up searching, auxiliary data structures are usually created and maintained along with the original data set.

For the same search problem there may be several algorithms, and this raises the question of how to decide which of them is preferable. There are two different approaches for answering this question. The *empirical* (or *a posteriori*) approach consists in programming the competing algorithms and trying them on different instances with the help of a computer. The *theoretical* (or *a priori*) approach consists in determining mathematically the quantity of resources (execution time, memory space, etc.) needed by each algorithm as a function of the size of the instances considered (this is also referred to as *computational complexity*).

It is natural to ask at this point what *unit* should be used to express the theoretical efficiency of an algorithm. There can be no question of expressing this efficiency in seconds, say, since we do not have a standard computer to which all measurements might refer. An answer to this problem is given by the *principle of invariance*, according to which two different implementations of the same algorithm will not differ in efficiency by more than some multiplicative constant. More precisely, if two implementations take $t_1(n)$ and $t_2(n)$ seconds, respectively, to solve an instance of size n, then there always exists a positive constant c such that $t_1(n) \le ct_2(n)$ whenever n is sufficiently large. This principle remains true regardless of the computer used (provided it is of a conventional design), of the programming language used, and of the skill of the programmer (provided that he or she does not actually modify the algorithm).

Coming back to the question of the unit to be used to express the theoretical efficiency of an algorithm, there will be no such unit: we express this efficiency only to within a multiplicative constant. We say that an algorithm takes time *on the order of* $t(n)$, for a given function t, if there exist a positive constant c and an implementation of the algorithm capable of solving every instance of the problem in time bounded above by $ct(n)$ seconds, where n is the size of the instance considered. We also say than the algorithm has $O(t(n))$ running time. By the principle of invariance, any other implementation of the algorithm will have the same property, although the multiplicative constant may change from one implementation to another.

Certain orders occur so frequently that it is worthwhile giving them names. For example, if an algorithm takes time on the order of the size n of the instance to be solved, we say that it takes *linear* time. In this case we also talk about a *linear algorithm*. Similarly, an algorithm is *quadratic, cubic, polynomial*, or *exponential* if it takes a time in the order of n^2, n^3, n^k, or c^n, respectively, where k and c are appropriate constants.

For more about computational complexity, see Ref. 1.

Below we discuss searching according to the type and structure of the data set.

Searching a Sequence of Records

Assume that the data set is an array (or file) of integers (or records), and suppose that we are searching for a desired element. The simplest method for searching involves testing elements in sequential order, starting at the beginning and stopping when the desired element is found or proved to be missing. This is a *serial search*; it takes linear time, and the average time for finding an element in a data set of n elements in $n/2$.

As mentioned earlier, the data set can be stored so as to speed the searching. This raises the question: given a set of elements characterized by a key (upon which an ordering relation is defined), how is the set to be organized so that the retrieval (seeking) of an element with a given key involves as little effort as possible?

Hashing is one answer to this question. Because in a computer each element is ultimately accessed by specifying a storage address, hashing essentially consists of finding an appropriate mapping, or *hash function* h, of keys (K) into addresses (A), that is, a function $h : K \rightarrow A$ (or $h : K \rightarrow \{1,\ldots, N\}$). If we assume an array structure, then h is a mapping transforming keys into array indices. The fundamental difficulty in using a key transformation is that the set of possible key values is much larger than the set of available store addresses (array indices). The function h is therefore obviously a many-to-one function. For example, if K is a set of integers, then modulo N is a hash function. A hash function is a good choice if it efficiently disperses all the possible elements, that is, if $h(x) \neq h(y)$ for most of the pairs $x \neq y$ that are likely to be found. When $x \neq y$ but $h(x) = h(y)$, we say that there is a *collision* between x and y. The *hash table* is an array $T[1,\ldots, N]$ of lists in which $T[i]$ is the list of those items such that $h(x) = i$. Thus searching for an element with key q requires searching sequentially the list $T[h(q)]$. The ratio $a = n/N$, where n is the number of distinct elements in the data set, is called the *load factor* of the table. Increasing the value of N reduces the average search time but increases the space occupied by the table. However, hashing is not efficient in *range queries*, that is, queries of the form: find all elements x such that $q_{\min} \leq x \leq q_{\max}$. Range queries can be evaluated efficiently if instead of creating a hash table we sort the data set, or create a search tree.

As mentioned earlier, an alternative approach to hashing is to build a *search tree* (which is discussed in a subsequent section) or to *sort* the elements. Assume that the data set is a sorted array, a sorted list, or a sorted file of records. In this case algorithms more sophisticated than serial search can be employed. A widely used algorithm is *binary search*.

Binary searching predates computing. In essence, it is the algorithm used to look up a word in a dictionary or a name in a telephone directory. It is probably the simplest application of divide-and-conquer. To speed up the search, divide-and-conquer suggests that we should look for the desired element x either in the first half of the array or in the second half. To find out which of these searches is appropriate, we compare x with an element in the middle of the array: if x is less than this element, then the search for x can be confined to the first half; otherwise it is sufficient to search the second half. Binary search runs in $O(\log n)$. This is significantly better than serial search for large data sets. However, binary search requires the data set to be random-access. Arrays are random-access, but sorted lists or files are not.

If the data set is not random-access (e.g. a sorted list or a file of ordered records), then there is no obvious way to select the middle of the list, which would correspond to the first step of binary search. In this case, probabilistic algorithms, such as the Sherwood algorithm [whose expected execution time is $O(\sqrt{n})$] can be employed.

For more about these algorithms see Ref. 2.

Searching a Sequence of Characters

Assume that the data set is a sequence of characters (that is, a *string*) stored in main memory, or a text file stored in secondary memory. The search problem is formulated as follows:

find the first occurrence (or all occurrences)
of a string (or pattern) p of length m in a string s of length n

Commonly, n is much larger than m. The simplest algorithm is *brute-force* (*BF*), or *sequential*, text searching. It consists of merely trying all possible positions in the text. For each position it verifies whether the pattern matches at that position. Since there are $O(n)$ text positions and each one is examined at $O(m)$ worst-case cost, the worst-case time for brute-force searching is $O(nm)$.

There are several more efficient algorithms (e.g. the Knuth–Morris–Pratt or the Boyer–Moore algorithm), which use a modification of this scheme. They employ a *window* of length m, which is slid over the text. It is *checked* whether the text in the window is equal to the pattern (if it is, the window position is reported as a match). Then, the window is *shifted* forward. The algorithms mainly differ in the way they check and shift the window. For instance, the Knuth–Morris–Pratt (*KMP*) algorithm does not try all window positions as BF does. Instead, it reuses information from previous checks. For doing this, the pattern p is preprocessed to build a table called "next". A prefix of a string s is any substring of s that starts from the first character of s, while a suffix of s is any substring of s that ends at the last character of s. The "next" table at position j says which is the longest proper prefix of $p[1,\ldots,j-1]$ that is also a suffix and is such that the characters following prefix and suffix are different. Hence $j - \text{next}[j] - 1$ window positions can be safely skipped if the characters up to $j-1$ matched and the jth did not. For example, when searching for the word "abracadabra," if a text window matched up to "abracab", five positions can be safely skipped, since next[7] = 1, as shown in Table 1:

Since at each text comparison the window or the pointer advances by at least one position, the algorithm performs at most $2n$ comparisons (and at least n).

Sequential searching is appropriate when the text is small (a few megabytes). If the text is big, then we can build data structures over the text, called *indices*, to speed up search. The most widely used techniques for indexing are *inverted files*, *suffix trees* (and *suffix arrays*), and *signature files*. Examples of these techniques are given in Fig. 2. Inverted files are currently the best choice for most applications.

The *inverted file* structure is composed of two parts: the *vocabulary* and the *occurrences*. The vocabulary is the set of all different words in the text. For each such word a list of all the text positions where the word appears is stored. The search algorithm on an inverted index follows three general steps. First, the words (or patterns) in the query are sought in the vocabulary. Then the lists of occurrences of all the words found are retrieved. Finally the occurrences are processed with regard to phrases, proximity, or Boolean operations.

Table 1. Statistical versus Knowledge-Based Information Retrieval

Statistical Approaches	Knowledge-Based Approaches
Text representation on counts of single words	Text representation based on syntactic and semantic analysis
Retrieval based on a similarity function	Retrieval based on inference
Domain-independent	Potentially large domain knowledge bases
Feedback based on statistical models	Learning based on symbolic and connectionist models
User-independent	Models of individuals and classes of users
Efficient	Currently expensive to implement
Effective	Effective in narrow, specific domains

Queries such as phrases are expensive to answer using inverted indices, as inverted indices actually view the text as a sequence of words. A type of index that allows answering efficiently more complex queries (e.g. phrases) is the *suffix tree*. These indices view the text as one long string. Each position in the text is considered as a text *suffix*, that is, a string that goes from that text position to the end of the text. Each suffix is uniquely identified by its position. In essence, a *suffix tree* is a *trie* data structure built over all the suffixes of the text. A trie is a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes. In a suffix tree the pointers to the suffixes are stored at the leaf nodes of the trie.

A problem with this data structure is its size. However, not all text positions need to be indexed. Index points can be selected from the text, which point to the beginning of text positions that will be retrievable. *Suffix arrays* provide essentially the same functionality as suffix trees with much less space requirements. Their main drawbacks are their costly construction process, the fact that the text must be readily available at query time, and the fact that the results are not delivered in text position order.

Signature files are word-oriented index structures based on hashing. A signature file uses a hash function (or signature) that maps words to bit masks of B bits. It divides the text into blocks of b words each. To each text block of size b, a bit mask of size B will be assigned. This mask is obtained by bitwise ORing the signatures of all the words in the text block. The main idea is that if a word is present in a text block, then all the bits set in its signature are also set in the bit mask of the text block. Hence, whenever a bit is set in the mask of the query word and not in the mask of the text block, the word is not present in the block. Signature files require only a low storage space overhead, at the cost of forcing a sequential search over the index. However, we may encounter *false drops*, that is, it is possible that all the corresponding bits are set even though the word is not there. The most delicate part of the design of a signature file is to ensure that the probability of a false drop is low enough while keeping the signature file as short as possible.

Inverted files outperform signature files for most applications.

For more about inverted files, suffix trees, and signature files see Chap. 8 of Ref. 3.

Searching in Graphs

Many important searching problems can be formulated in terms of graphs. A graph is a pair $G = \langle N, A \rangle$ where N is a set of *nodes* and $A \subseteq N \times N$ is a set of *edges*. We can distinguish *directed* and *undirected* graphs. An edge

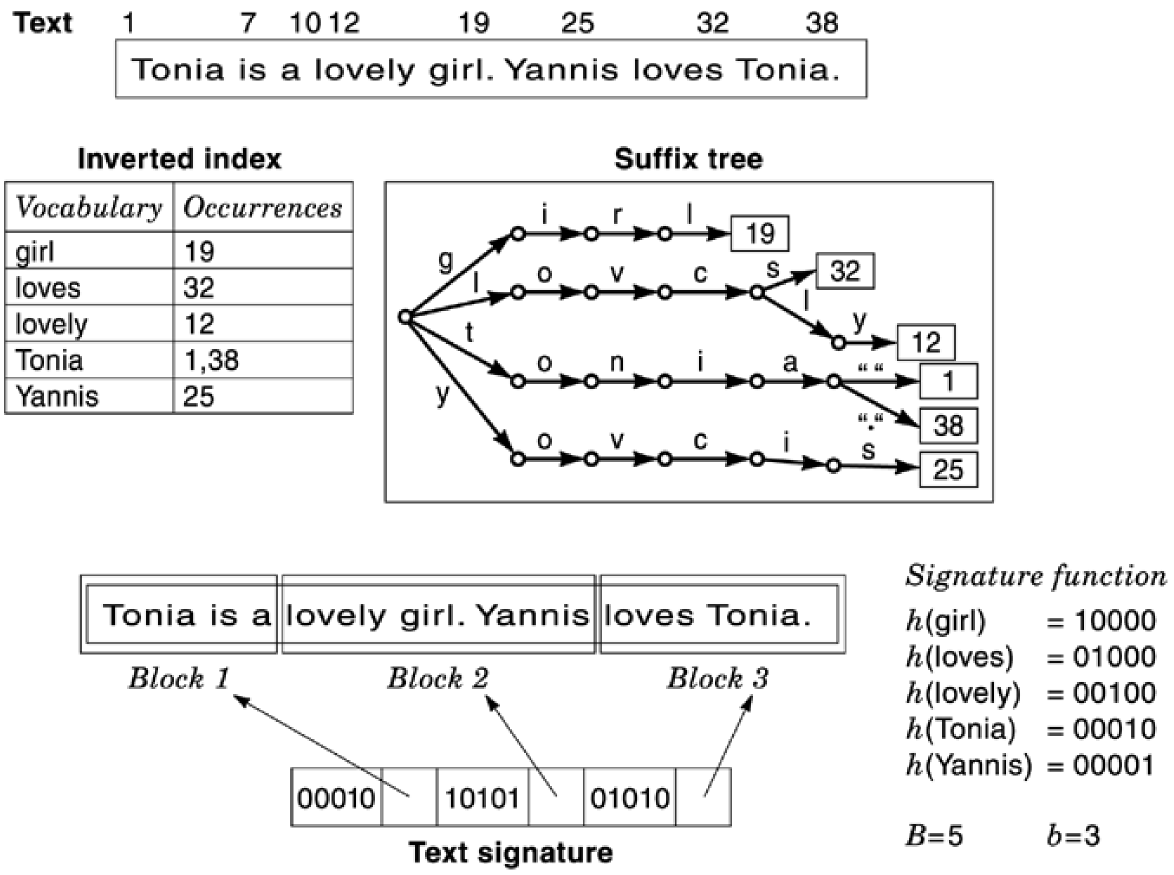

Fig. 2. An inverted index, a suffix tree, and a signature file for the same text.

from node n to node n' of a directed graph is denoted by the ordered pair (n, n'), whereas an edge joining nodes n and n' in an undirected graph is denoted by the set $\{n, n'\}$.

In an undirected graph a sequence of nodes $n_1,\ldots, n_k$ where $\{n_i, n_{i+1}\} \in A$ for $i = 1,\ldots, k - 1$ is called a *path* of length k from node n_1 to node n_k. Often it is convenient to assign positive *costs* to edges; for example, $c(n, n')$ may denote the distance between two cities denoted by n an n'. The cost of a path is the sum of the costs of all edges in the path. In certain search problems, given two nodes n and n', we want to find a path of minimal cost among all paths from n to n'.

In a directed graph, if an edge is directed from a node n to a node n', then we say that n is the *parent* (or ancestor) of n' and that n' is a *child* (or successor) of n. A sequence of nodes $n_1,\ldots, n_k$ with each n_{i+1} a successor of n_i, for $i = 1,\ldots, k - 1$, is called a path of length k from node n_1 to node n_k. If $c(n, n')$ denotes the cost of an edge directed from n to successor node n', then the cost of a path is the sum of the costs of all edges connecting in the path. In certain search problems, given two nodes n and n', we want to find a path of minimal cost among all paths from n to n'. Such paths are called *optimal paths*. Figure 3 shows a directed graph consisting of four nodes and five edges.

A special case of directed graph is the *directed acyclic graph* (*DAG*). A DAG is a directed graph with no cycles, where a *cycle* is defined to be a path whose initial and final node coincide. For example the directed graph shown in Figure 3 is not a DAG, as there is the cycle (1,2,4,1). DAGs are used in many applications.

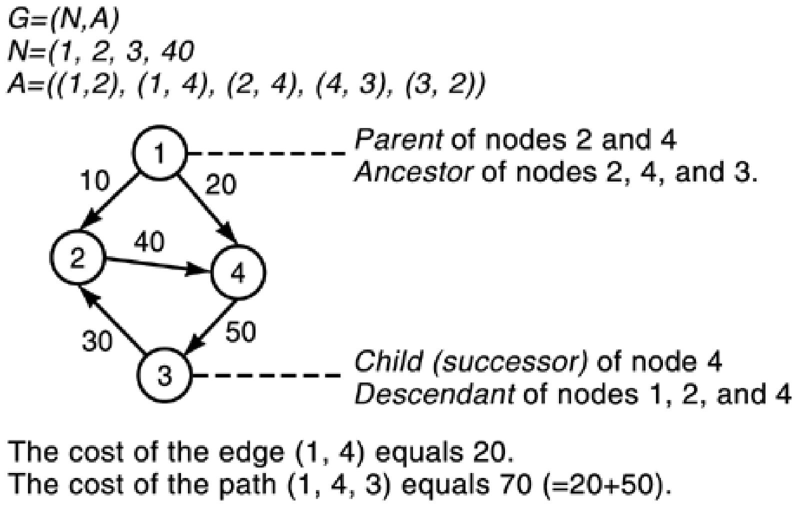
G=(N,A)
N=(1, 2, 3, 40
A=((1,2), (1, 4), (2, 4), (4, 3), (3, 2))
10
20
40
30
50
Parent of nodes 2 and 4
Ancestor of nodes 2, 4, and 3.
Child (successor) of node 4
Descendant of nodes 1, 2, and 4
The cost of the edge (1, 4) equals 20.
The cost of the path (1, 4, 3) equals 70 (=20+50).

Fig. 3. Graph notation.

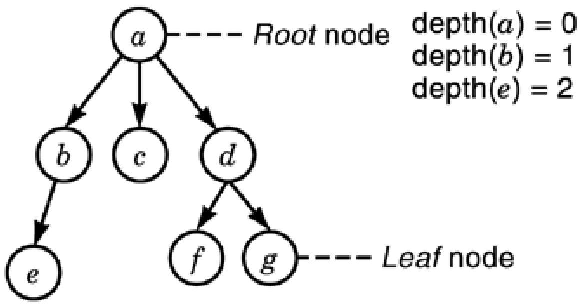

Fig. 4. Tree notation.

For example, they are used to represent the structure of an arithmetic expression that includes repeated subexpressions; or the different stages of a complex project: the nodes represent the different states of the project, from the initial state to the final completion state of the project, and the edges correspond to activities that have to be completed to pass from one state to another. Moreover, DAGs offer a natural representation for partial orderings.

Another special case of directed graph is the directed *tree*, in which each node has exactly one parent, except for a single node that has no parent and is called the *root* of the tree. A node in the tree having no successors (children) is called a tip node or a *leaf*. Two nodes are *siblings* if they have the same parent. The *depth* of the root node is 0, and the depth of any other node in the tree is defined to be the depth of its parent plus 1. Certain trees have the property that all nodes except the leaves have the same number of successors, say b. In this case, b is called the *branching factor* of the tree. Figure 4 shows an example of a tree.

The following three subsections describe, respectively, an application of trees for speeding up searching in a sequences of records; searching in explicit graphs, that is, in graphs that are stored explicitly in the computer memory; and searching in implicit graphs, that is, in graphs for which we have available a description of the nodes and edges, but that are not stored explicitly in the computer memory.

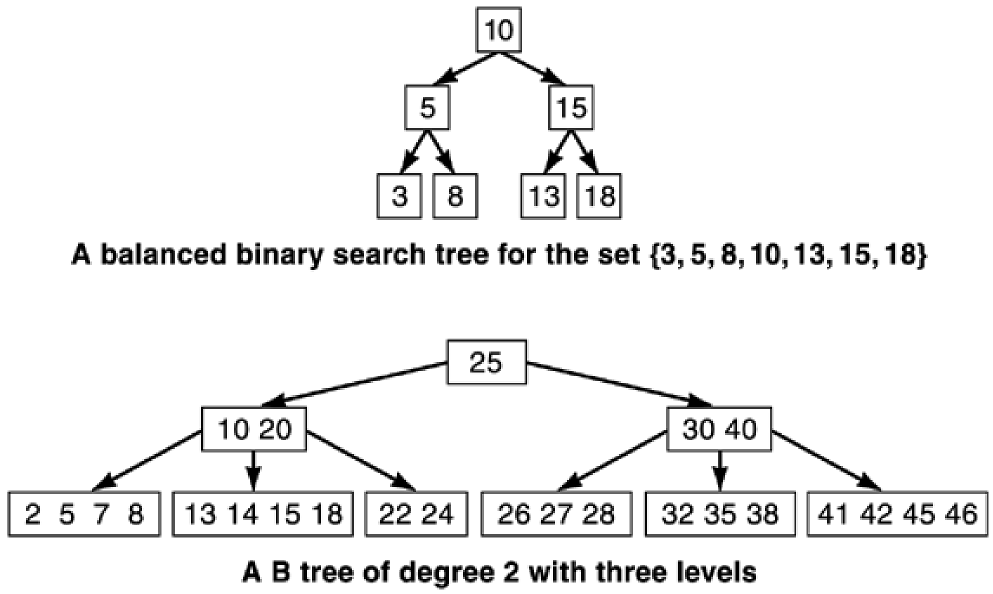
10
5
15
3
8
13
18
A balanced binary search tree for the set {3, 5, 8, 10, 13, 15, 18}
25
10 20
30 40
2 5 7 8
13 14 15 18
22 24
26 27 28
32 35 38
41 42 45 46
A B tree of degree 2 with three levels

Fig. 5. Search trees.

Search Trees. An *ordered tree* is a tree in which the branches of each node are ordered. The number of successors of a node is called its *degree*. The maximum degree over all nodes is the degree of the tree. *A binary tree* is an ordered tree of degree 2. Trees with degree greater than 2 are called *multiway trees*. Binary trees are frequently used to represent a set of data whose elements are to be retrievable through a unique key. If a tree is organized in such a way that for each node n all keys in the left subtree are (numerically or lexicographically) less than the key of n, and those in the right subtree are greater than the key of n, then this tree is called a *search tree*. The upper part of Fig. 5 shows a binary search tree. Note that all keys in the left subtree of node 10 (i.e. the keys $\{5,3,8\}$) are less than 10, and that all keys in the right subtree (i.e. the keys $\{15,13,18\}$) are greater than 10. In a search tree it is possible to locate an arbitrary key by starting at the root and proceeding along a search path switching to a node's left or right subtree by a decision based on inspection of the node's key only. As n elements may be organized in a binary tree of a height as small as log n, a search among n elements may be performed with as few as log n comparisons if the tree is perfectly balanced.

An *AVL tree* is a binary search tree in which for every node the heights of its two subtrees differ by at most 1.

An application of multiway trees is the construction of large-scale search trees where the primary store of a computer is not large enough (or is too costly) to be used for long-term storage. An example of such tree is the *B tree*. A B tree of degree 2 and depth 3 is shown in Fig. 5.

If information about the probabilities of access to individual keys is available, then we can structure the search tree so as to minimize the average path length. For example suppose we have an ordered set $c_1 < \ldots < c_n$ of n distinct keys, and let p_i be the probability that a request refers to key c_i, $i = 1,2,\ldots, n$. If some key c_i is held in a node at depth d_i, then $d_i + 1$ comparisons are needed to find it. Thus for a given tree the average number of comparisons needed for finding a key is

$$C = \sum_{i=1}^{n} p_i(d_i + 1)$$

Trees whose structure minimizes the quantity C are called *optimal search trees*.

For more about search trees and optimal search trees, see Ref. 4.

Explicit Graphs. Suppose the objective of searching is to see whether a node (or the information assigned to a node) exists in a graph. This requires traversing the graph. We can distinguish the *depth-first* traversal (or search) and the *breadth-first* traversal (or search).

To carry out a depth-first search (*dfs* for short) of a graph, we choose any node of G, say n, as the starting node and mark it to show that it has been visited. Next, if there are nodes adjacent to n (or successors of n, if G is a directed graph) that have not been visited yet, we choose one as the new starting node and call the dfs procedure recursively. When all nodes adjacent to n have been marked, the search starting at n is finished. If there remain any nodes of G that have not been visited yet, we choose one of them as a new starting node, and continue in this way until all nodes of G have been marked. Dfs can be also used for detecting whether a given directed graph is acyclic.

In breadth-first search (*bfs* for short), when we arrive at some node n, we first visit all the nodes adjacent to n (or all siblings of n, if G is a directed graph), and not until this has been done do we go on to look at nodes farther away. Unlike dfs, bfs is not naturally recursive. For example, a dfs traversal of the tree shown in Fig. 4 will visit the nodes of the tree in the following order: a, b, e, c, d, f, g, while a bfs traversal will visit the nodes in the order a, b, c, d, e, f, g.

Given a graph, there are many cases where we need to search for paths, or for subgraphs, that satisfy certain conditions. For example, given an explicit graph, we might want to find a path from a node n to each of the other nodes in the graph. Such a collection of paths constitutes a *spanning tree* of the graph—a tree rooted at n—and a famous problem is finding a *minimal spanning tree*. Other famous graph-searching problems of this kind include the *shortest-route problem* and the *topological sort*. A topological sort of the nodes of a directed acyclic graph is the operation of arranging the nodes in a linear order in such a way that if there exists an edge (n, n'), then n precedes n' in the linear order. The necessary modification to the procedure depth-first search to make it into a topological sort is immediate. For more on the computational aspects of graphs and on graph algorithms see Chap. 6 of Ref. 4 and Chap. 6 of Ref. 5.

In artificial intelligence (*AI*), directed graphs are used to model the world of an agent and the effects of its actions on the world model. These graphs are called *state-space graphs*. The nodes are labeled by representations of the individual worlds, and the edges by operators, that is, the actions that an agent can take. If the number of different distinguishable world situations is sufficiently small, a graph representing all of the possible actions and situations is stored explicitly. To find a set of actions that will achieve a specified goal (a world situation), an agent needs to find a path in the graph from a node representing its initial world state (the *start node*) to a node representing a specified goal state, the *goal node*. The actions that will achieve the goal can then be read out as the labels on the edges of this path. The operators labeling the edges along a path to a goal can be assembled into a sequence called a *plan*, and searching for such a sequence is called *planning*. Searching such graphs involves propagating "markers" over the nodes of the graph. We start by labeling the start node with a 0, and then we propagate successively larger integers out in waves along the edges until an integer hits the goal node. Then, we trace a path back from the goal to the start along a decreasing sequence of numbers. The actions along this path, from start to goal, are the actions that should be taken to achieve the goal (if there is a single goal node, the process could also be implemented in the reverse direction—starting with the goal node and ending when an integer hits the start node). The stages of marker propagation correspond to a bfs traversal. The process of marking the successors of a node is called *expansion*.

Implicit Graphs. In certain situations a graph exists only implicitly. For instance, we often use abstract graphs to represent games, such as chess: each node corresponds to a particular position of the pieces on the board, and the fact that an edge exists between two nodes means that it is possible to get from the first to the second of these positions by making a single legal move. Often the original problem translates to searching for a specific node, path, or pattern in the associated graph. If the graph contains a large number of nodes, it may be wasteful or infeasible to build it explicitly in computer memory before applying one of the search

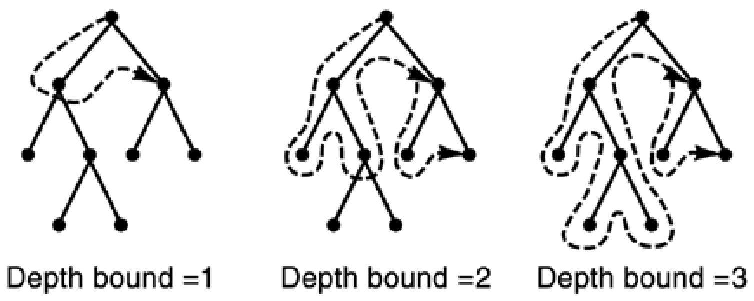

Fig. 6. Iterative-deepening search.

techniques that we have encountered so far. For example, the number of nodes in the state-space graph of chess is approximately 10^{40}. Most of the time, all we have is a representation of the current position. An implicit graph is one for which we have available a description of its nodes and edges. Relevant portions of the graph can thus be built as the search progresses. Therefore computing time is saved whenever the search succeeds before the entire graph has been constructed.

We can distinguish two broad classes of search processes in implicit graphs. In the first, called *uninformed*, we have no problem-specific reason to prefer one part of the search space to another, as far as finding a path to the goal is concerned. In the second class, called *heuristic*, we do have problem-specific information to help focus on a specific search. (The word heuristic comes from the Greek word *ευρίσκειν* (heuriskein), meaning to "discover.")

Uninformed Search. The simplest uninformed search procedure is *breadth-first search*. The basic property of this search is that when a goal node is found, we have found a path of minimal length to that goal. It has the disadvantage, however, that it requires the generation and storage of a tree whose size is exponential in the depth of the shallowest goal node.

Uniform-cost search is the analog of bfs for graphs that have costs assigned to their edges. In uniform-cost search, nodes are expanded outward from the start node along contours of equal cost rather than equal depth.

Another method is dfs, or *backtracking*. To prevent the search process from running away towards nodes of unbounded depth from the start node, a *depth bound* is used. No successor is generated whose depth is greater than the depth bound. The memory requirements of dfs are linear in the depth bound. A disadvantage of dfs is that when a goal is found, we are not guaranteed to have found a path to it having minimal length.

A technique called *iterative deepening* offers the linear memory requirements of dfs while guaranteeing that a goal node of minimal length will be found. In iterative deepening, successive depth-first searches are conducted—each with depth bounds increasing by 1—until a goal node is found. Figure 6 shows an example of iterative-deepening search.

Heuristic Search. Suppose the search proceeds preferentially through nodes that problem-specific information indicates as being on the best path to a goal. We call such processes *best-first* or *heuristic search*. For doing this we define a *heuristic* (evaluation) function f to help decide which node is the best one to expand next. It is a real-valued function of state descriptions, and its definition is based on information specific to the problem domain. During searching we expand next the node n satisfying a certain condition expressed by f—for example, we may expand next the node n resulting in the smallest value of $f(n)$—and we terminate when the node to be expanded is a goal node. Usually we also take into account the depth of node n, that is, f has the form $f(n) = g(n) + h(n)$, where $g(n)$ is an estimate of the depth of n in the graph (i.e., the length of the shortest path from the start to n), and $h(n)$ is a heuristic evaluation of node n. There are conditions on graphs and on h that guarantee that the best-first algorithm applied to these graphs is *admissible*, that is, it is guaranteed to find an optimal path to the goal (for more see Chap. 9 of Ref. 6.

The classic book on heuristic search is Ref. 7.

Approximate Search. Relaxing the requirement of producing optimal paths often reduces the computational cost of finding a plan. This reduction can be done either by searching for a complete path to a goal node without requiring that this path be optimal, or by searching for a partial path that does not take us all the way to a goal node.

A best-first search can be used in both cases. In the first, we use a nonadmissible heuristic function, and in the second, we quit searching before reaching the goal—using either an admissible or a nonadmissible heuristic function. Examples of this kind of algorithms include the *island-driven search*, the *hierarchical search*, and the *limited-horizon search*. For instance, in the island-driven search, heuristic knowledge from the problem domain is used to establish a sequence of *island nodes* in the search space, through which it is suspected that goal paths pass. Suppose that n_0 is the start node, n_g is the goal node, and $(n_1,\ldots, n_k)$ is a sequence of such islands. In this case a heuristic search is initiated with n_0 as the start node and with n_1 as the goal node, using a heuristic function appropriate for that goal. When the search finds a path to n_1, another search starts with n_1 as the start node and n_2 as the goal node, and so on, until a path to n_g is found.

Rewards Instead of Goals. In the previous discussions we assumed that the objective of searching is a goal node. In many problems the common task cannot be so simply stated. Instead, the task may be an ongoing one. The user expresses his or her satisfaction and dissatisfaction with task performance by occasionally giving the agent positive or negative *rewards*. The task for the agent is to maximize the amount of reward it receives. The special case of a simple goal-achieving task can be cast in this framework by rewarding the agent positively (just once) when it achieves the goal, and negatively (by the amount of an action's cost) every time it takes an action. In this sort of task environment, we seek to describe an action policy that maximizes reward. One problem for ongoing, nonterminating tasks is that the future reward might be infinite, so it is difficult to decide how to maximize it. A way of proceeding is to discount future rewards by some factor. That is, the agent prefers rewards in the immediate future to those in the distant future.

Constraint Satisfaction and Constraint Propagation. There are applications of search techniques beyond the problem of selecting actions for an agent. These applications include finding solutions to problems of assigning values to variables subject to constraints and solving optimization problems. When the goal node is defined not by a specific data structure but by conditions or constraints, it might be that the problem is to exhibit some data structure satisfying those conditions; the steps that produce it using the above graph-search methods might be irrelevant. We call these problems *constraint-satisfaction problems*. A prominent subclass of this class involves assigning values to variables subject to constraints. Such problems are called *assignment problems*.

We can solve constraint-satisfaction problems by graph-search methods. A goal node is a node labeled by a data structure (or state description) that satisfies the constraints. Operators change one data structure to another. The start node is some initial data structure. A good example of an assignment problem is the eight-queens problem: to place (assign) eight queens on a chess board in such a way that there is a queen in every row and column but with the additional constraint that only one queen can be in any single row, column, or diagonal. We call this an assignment problem because it can be posed as a problem of assigning values from the set {row 1,…, row 8} to variables from the set {position of queen in column 1,…, position of queen in column 8}). In assignment problems, since the path to the goal is not the important thing, we often have many choices about what the start state and operators can be.

Constructive methods try assigning a value to each variable in turn (e.g. the position of a queen). A computational technique called *constraint propagation* often helps markedly in reducing the size of the search space. It is used in combination with a constructive solution technique—assigning a value to each variable in turn. Constraints are represented in a directed graph called a *constraint graph*. Each node in this graph is labeled by a variable name together with a set of possible values for that variable. A directed constraint edge, say (i, j), connects a pair of nodes i and j if the value of the variable labeling i is constrained by the value of the variable labeling j. We say that a directed edge (i,j) is *consistent* if each value of the variable at the tail of the arc has at least one value of the variable at the head of the arc that violates no constraints. After assigning values to one or more variables, we can use the concept of edge consistency to rule out some of the values of other variables. The process of constraint propagation iterates over the edges in the graph and eliminates values of variables at the tails of arcs in an attempt to enforce edge consistency. The process halts when no more values can be eliminated. Figure 7 shows a constraint graph for the four-queens problem. In this problem, each variable constrains all of the others. The lower part of the figure shows the constraint propagation assuming $q1 = 2$. At each step, the arrow that determines the elimination of values is in boldface.

Constraint propagation has been applied to a variety of interesting problems, including the problem of labeling lines in visual scene analysis and that of *propositional satisfiability*. For a survey of the method, its extensions, and its applications, see Ref. 4

Function Optimization (Hill Climbing). In some problems, instead of having an explicit goal condition, we may have some function ν over data structures and seek a structure having a maximum (or minimum) value of that function. If we view the data structures as points in a space, this function can be thought of as a landscape over the space. One class of methods consists of those that traverse the landscape, looking for points of high elevation. Since we may not know the value of the global maximum, we may never know for sure if we have reached a point having maximal height. Among the techniques for traversing a space are the *hill-climbing* methods, which traverse by moving from one point to that adjacent point having the highest elevation. Hill-climbing methods typically terminate when there is no adjacent point having a higher elevation than the current point—thus, they can get stuck on local maxima. We can use graph-searching methods to do hill climbing. As in assignment problems, the path to the goal is not important in this kind of problems. Hill climbing follows a single path (much like a dfs without backup), never descending to a lower point.

Game Trees. Most games of strategy can be represented in the form of directed graphs. A node of the graph corresponds to a particular position in the game, and an edge corresponds to a legal move between two positions. The graph is infinite if there is no *a priori* limit on the number of positions possible in the game. For simplicity we assume that two players, A and B, play the game, each of whom moves in turn, that the game is symmetric (the rules are the same for both players), and that chance plays no part in the outcome (the game is deterministic).

To determine a winning strategy for a game of this kind, we need only attach to each node in the graph a label chosen from the set {win, lose, draw}. The label corresponds to the situation in which neither player will make an error.

The purpose of a *game tree search* is to find the best possible move given a game position—ideally, to find a move for which even if the opponent plays perfectly, victory is still guaranteed. However, most games are sufficiently complex that it is impossible to follow the tree all the way to the end condition (win, lose, or draw). Thus an *evaluation function* is needed to evaluate the quality of a given position. This function attributes some value to each possible position. Ideally, say in a chess game, we want the value of eval(u) to increase as the position u becomes more favorable to A. It is customary to give values not far from zero to positions where neither side has a marked advantage, and large negative values to positions that favor B. When applied to a *terminal* position (i.e., a position that does not offer any legal move), the evaluation function should return $+\infty$ if B has been mated, $-\infty$ if A has been mated, and 0 if the game is a draw.

Figure 8 shows the part of the graph corresponding to some game. In this example we suppose that player A is trying to maximize the evaluation function and that player B is trying to minimize it. The values attached to the leaves are obtained by applying the function eval to the corresponding positions. The values for the other nodes can be calculated using the *minimax rule*. We see why the technique is called *minimax*: B tries to minimize the advantage of A, and A tries to maximize the advantage he obtains from each move.

The basic minimax technique can be improved in a number of ways. According to *alpha–beta pruning*, the exploration of certain branches are abandoned early if the information we have about them is already sufficient to show that they cannot possibly influence the values of nodes farther up the tree.

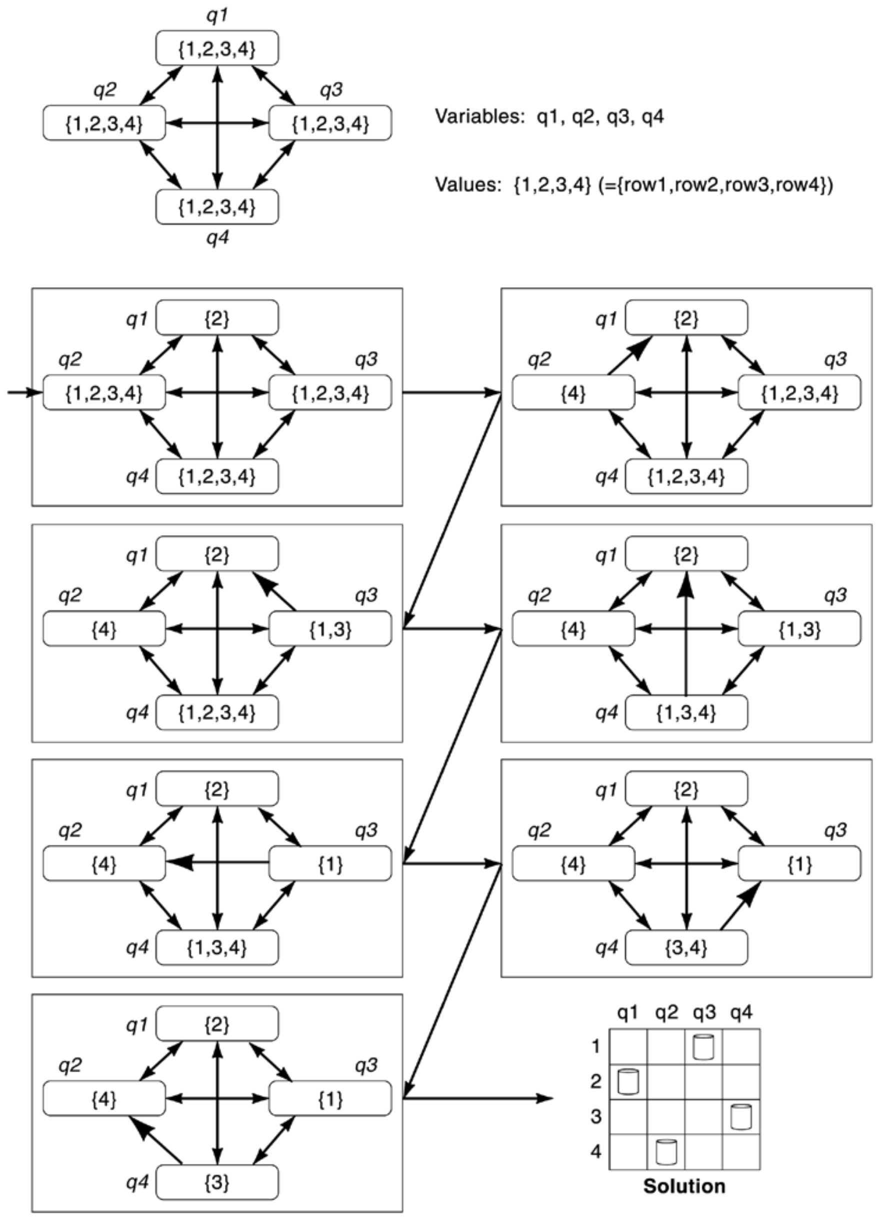
q1
q2
q3
q4
{1,2,3,4}
Variables: q1, q2, q3, q4
Values: {1,2,3,4} (={row1,row2,row3,row4})
{2}
{4}
{1,3}
{1,3,4}
{1}
{3,4}
{3}
1
2
3
4
Solution

Fig. 7. Constraint propagation for solving the four-queens problem.

Branch-and-bound is another technique for exploring an implicit directed graph. At each node we calculate a bound on the possible value of any solutions that might happen to be farther on in the graph. If the bound shows that any such solution must necessarily be worse than the best solution we have found so far, then we do not need to go on exploring this part of the graph. In the simplest version, calculation of these bounds is combined with a bfs or dfs, and serves only, as we have just explained, to prune certain branches of a tree or to

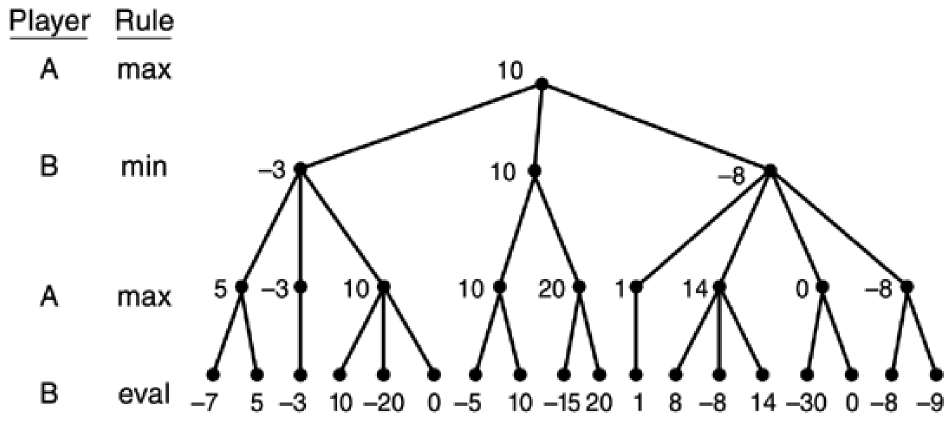

Fig. 8. The minimax principle: player A is trying to maximize, and player B to minimize, the evaluation function.

close certain paths in a graph. More often, however, the calculated bound is used not only to close off certain paths, but also to choose which of the open paths looks the most promising, so that it can be explored first.

Some games, such as backgammon, involve an element of chance. For example, the moves that one is allowed to make may depend on the outcome of a throw of the dice. We can use game trees in such games too. A's and B's turns now each involve a throw of a die. We might imagine that at each throw, a fictitious third player, DICE, makes a move. That move is determined by chance. In the case of throwing a die, the six outcomes are all equally probable, but the chance element could also involve an arbitrary probability distribution. Values can be backed up in game trees involving chance moves also, except that when backing up values to nodes at which there was a chance move, we might back up the *expected* (average) values of the successors instead of a maximum or minimum.

A good overview of graph-searching algorithms and their application in AI is Ref. 6; a good overview of game tree searching algorithms can be found in Ref. 9.

Searching in Tables

Here the data set is a table or a set of tables. A table T consists of a *scheme* denoted sch(T) and an *instance*. The scheme is a set of column headings, also called *attributes*, and the instance is a set of rows over these attributes. A *row* is a mapping that associates each attribute with a value from the set of values, or *domain*, of that attribute. For example, Fig. 9(a) shows a table whose scheme consists of the attributes A, B, and C and whose instance consists of four rows. In the example, we assume that all attributes have the same domain, namely the set of characters.

Searching a given table means asking for a new table to be constructed by keeping some of the columns and/or some of the rows of the given one. The specification of the new table is done using two operations, *projection* and *selection*.

The projection operation takes as input a table T and a subset X of the attributes of T and returns a table, denoted by $\pi_X(T)$, whose scheme is X and whose instance consists of the restrictions of all rows of T over the attributes of X. For example, if T is the table of Fig. 9(a) and $X = \{A, B\}$, then $\pi_X(T)$ is the table shown in Fig. 9(b).

The selection operation takes as input a table T and a *selection condition* C and returns a table, denoted by $\sigma_C(T)$, whose scheme is that of T and whose instance consists of those rows of T that satisfy C. Elementary

T

A	B	C
a	b	c
a	b'	c'
a'	b'	c
a'	b	c'

(a)

$\Pi_{\{A,B\}}(T)$

A	B
a	b
a	b'
a'	b'
a'	b

(b)

$\sigma_{(A=a)\ \&\ (C=c')}(T)$

A	B	C
a	b'	c'

(c)

Fig. 9. Searching in tables using projection and selection.

conditions of the form $\langle attribute = value \rangle$, where *value* is taken from the domain of *attribute*, are the building blocks of selection conditions. A selection condition is formed by combining elementary conditions using the traditional logical connectives (&, not, or). For example, if T is the table of Fig. 9(a) and $C = \langle A = a \rangle \& \langle C = c' \rangle$, then $\sigma_C(T)$ is the table shown in Fig. 9(c).

One important generalization of searching a table is that of searching a relational database. A *relational database* is a set of tables, and users search it by submitting queries. A *query* is a well-formed expression whose operands are tables and whose operations are projection and selection (for searching one table, as we have seen earlier) plus a few binary operations for combining two or more tables. For more on relational databases and their query languages see Ref. 10.

Data Mining. Data mining (also known as *knowledge discovery in databases—KDD*) is a new kind of searching. Here, the objective is to discover prefiously unknown relationships among the data. These relationships, or *rules*, can lead to reasonable predictions about the future. For example, marketers use data mining in trying to distill useful consumer data from Web sites. Let $I = \{i_1, \ldots, i_m\}$ be a set of items, and let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. An *association rule* is an implication of the form $X \Rightarrow Y$, where X and Y are subsets of I, and $X \cap Y = \emptyset$. A rule $X \Rightarrow Y$ holds in the transaction set D with *confidence c* if $c\%$ of transactions in D that contain X also contain Y. A rule $X \Rightarrow Y$ has *support s* in the transaction set D if $s\%$ of transactions in D contain $X \cup Y$.

Specifically, high-level languages are used where the user specifies two parameters, namely the minimum support and the minimum confidence, and the system searches for and finds all association rules whose support and confidence are greater than the given values.

Other kinds of mining include classification and clustering. For more see Ref. 11.

Searching in Semistructured Data

In order to represent data with loosely defined or irregular structure, the *semistructured data model* has emerged. At the same time the document community has developed XML as a format in which *more* structure is added to documents in order to simplify and standardize the transmission of data via documents. It turns out that these two representations are essentially identical.

From a database point of view, semistructured data are often described as "schemaless" or "self-describing," because there is no separate description of the type or structure of the data (as in the relational data model). One of the main strengths of semistructured data in comparison with other data model is its ability to accommodate

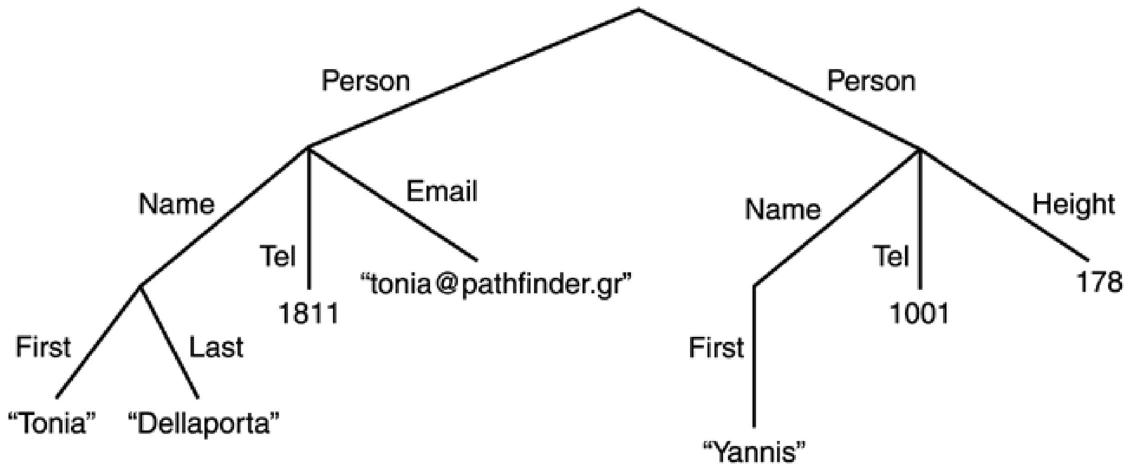

Fig. 10. Semistructured data.

variations in structure. The variations typically consist of missing data and duplicated fields. An example is given in Fig. 10.

The semistructured data model is actually a directed acyclic graph (N,A) with a labeling function $L : A \rightarrow L_A$, where L_A is the domain of edge labels. The model usually also includes node labels, though those are typically confined to leaf nodes. The semistructured data model can be used to represent several kinds of databases, including the relational ones.

The objective of searching is that of database query languages, that is, to select and transform information from large sources. Query languages for semistructured data emphasize on restructuring. However there is no accepted notion of *completeness* for semistructured data restructuring. The building blocks of any query language are path expressions. A path expression is a simple query whose result, for a given data graph, is a set of nodes. For example the result of the path expression "person.name.first" is the set of nodes {"Tonia", "Yannis"}. Path expressions may contain regular expressions allowing the specification of paths that have some properties. For example, the regular expression "first | last" matches either a first name or a last name. In addition, there are *wildcards* that match any label, and symbols that specify arbitrary repeats of a regular expression. Roughly, a high-level query language for semistructured data supports path expressions and patterns to extract data, variables to which the data are bound, and templates that determine the construction of the output.

An XML document is actually a labeled ordered tree whose leaves are text. There are only some slight differences from the semistructured data model: XML trees are ordered, and the labels are not attached to edges but to nodes. XML has already become a standard for knowledge representation on the Web. For more about semistructured data XML see Ref. 12.

Searching in Metric Spaces (Multimedia)

Assume that the data set is a set of images, fingerprints, or audio or video segments. These data sets cannot be meaningfully searched in the classical sense. Not only can they not be ordered, but also they cannot even be compared for equality. There is no interest in an application for searching an audio segment exactly equal to a given one. The probability that two different images are pixelwise equal is negligible unless they are digital copies of the same source. In multimedia applications, all the queries ask for objects *similar to* a given one. This is called *similarity searching* or *proximity searching*. Some example applications are face recognition, fingerprint matching, voice recognition, or similarity searching in general multimedia databases.

In similarity searching the data set is a *metric space*, that is, a set of elements X equipped with a metric function d. The function $d : X \times X \rightarrow \Re$ denotes a measure of "distance" between objects (i.e., the smaller the distance, the more similar are the objects). Distance functions have the following properties, for any x, y, z in X:

(1) *Positiveness*: $d(x,y) \geq 0$
(2) *Symmetry*: $d(x,y) = d(y,x)$
(3) *Reflexivity*: $d(x,x) = 0$
(4) *Triangle inequality*: $d(x,y) \leq d(x,z) + d(z,y)$

If the elements of the metric space (X,d) are tuples of real numbers (actually tuples of any field), then the pair is a *finite-dimensional vector space*. A k-dimensional vector space is a metric space where the objects are identified with k real-valued coordinates $(x_1,\ldots, x_k)$. There are a number of options for the distance function to use, but the most widely used is the family of L_s distances defined as

$$L_s((x_1, \ldots, x_k), (y_1, \ldots, y_k)) = \left(\sum_{i=1}^{k} |x_i - y_i| \right)^{1/s}$$

For instance, the L_1 distance equals the sum of the differences along the coordinates. It is also called "block" or "Manhattan" distance, since in two dimensions it corresponds to the distance to walk between two points in a city of rectangular blocks. The L_2 distance is better known as "Euclidean" distance, as it corresponds to our notion of spatial distance. The other most used member of the family is L_∞, which corresponds to taking the limit of L_s when s goes to infinity.

Thus the data set is a finite subset U of X. Possible types of searches (objectives) are *range queries* (retrieve all elements that are within distance r of the query q) and *k-nearest-neighbor queries* (retrieve the k closest elements to q). For evaluating these kinds of queries (similarity or proximity queries), auxiliary data structures, called *indexes*, are used in order to reduce the number of distance evaluations at query time. Indexes can be distinguished to (a) tree indexes for *discrete* distance functions, (functions that deliver a small set of values); (b) tree indexes for *continuous* distance functions (functions where the set of alternatives is infinite or very large); and (c) non-tree-based indexes.

Examples of tree indexes for discrete distance functions include the Burkhard–Keller tree (BKT) (13), the fixed query tree (FQT) (14), and many others. In the BKT an arbitrary element p in U is selected as the root of the tree. For each distance $i > 0$, we define $U_i = \{u \in U |\ d(u,p) = i\}$ as the set of all the elements at distance i to the root p. Then, for any nonempty U_i, we build a child of p (labeled i), and we recursively build the BKT for U_i. This process can be repeated until there is only one element to process, or until there are no more than b elements (and we store a *bucket* of size b). All the elements selected as roots of subtrees are called *pivots*. The left part of Figure 11 shows the division of the elements of a metric space when u_3 is taken as a pivot, while the right part shows the first level of BKT with u_3 as root. The distances have been discretized and return integer values.

An FQT (which is a variation of the BKT) built over n elements has $O(\log n)$ height on average. It is built using $O(n \log n)$ distance evaluations, and the average number of distance computations is $O(n^a)$, where $0 < a < 1$ is a number that depends on the range of the search and the structure of the space.

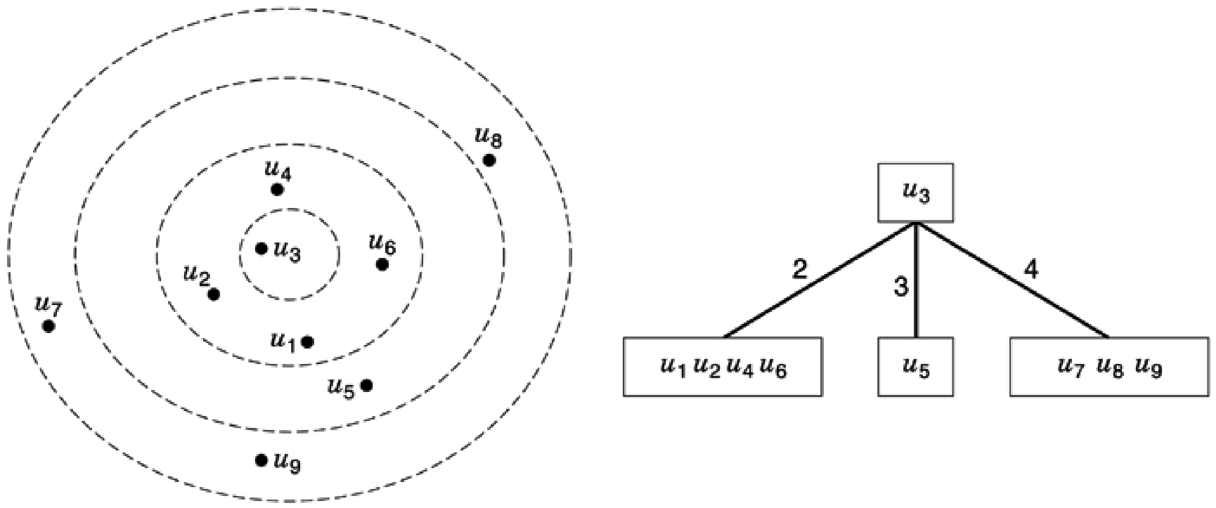

Fig. 11. A BKT for a data set consisting of eight elements.

If we have a continuous distance or if the distance function gives too many different values, it is not possible to have a child of the root for any such value. However the indexes for discrete functions can be adapted to a continuous distance by assigning a range of distances to each branch of the tree. Other examples indexes for continuous distance functions include vantage-point trees (*VPTs*), multi-vantage-point trees (*MVTs*), Voronoi trees (*VTs*), and M-trees (*MTs*). Vantage-point trees, also called metric trees, are actually binary trees. One builds them by taking any element p as the root and taking the *median* of the set of all distances, M = median($d(p,u) \mid u \in U\}$. Those elements u such that $d(p,u) \leq M$ are inserted into the left subtree, while those such that $d(p,u) > M$ are inserted into the right subtree. The VTP takes $O(n)$ space and is built in $O(n \log n)$ worst-case time. The query complexity is $O(\log n)$.

Other, non-tree-based indexes include the approximating eliminating search algorithm (*AESA*) and the linear AESA (*LAESA*).

Search structures for vector spaces are called *spatial access methods* (*SAMs*). Among the most popular are *kd trees*, *R trees*, *quadtrees*, and *X trees*. These techniques make extensive use of coordinate information to group and classify points in the space. For example, kd trees divide the space along different coordinates, and R trees group points in hyperrectangles.

All these techniques are very sensitive to the vector-space dimensions. Closest-point and range search algorithms have an exponential dependence on the dimension of the space (this is called the *curse of dimensionality*). Vector spaces may suffer from large differences between their representational dimension (k) and their intrinsic dimension, that is, the real number of dimensions in which the points can be embedded while keeping the distances among them. For example, a plane embedded in a 50-dimensional space has intrinsic dimension 2 and representational dimension 50. This is, in general, the case of real applications, where the data are clustered, and it has led to attempts to measure the intrinsic dimension, such as the concept of fractal dimension. Despite the fact that no technique can cope with intrinsic dimensions higher than 20, much higher representational dimensions can be handled by dimensionality reduction techniques. For search techniques for vector spaces see Ref. 15.

This kind of searching has several applications: querying by content of multimedia objects, text retrieval, computational biology, pattern recognition, function approximation, and audio and video compression.

A comprehensive survey of searching in metric spaces can be found in Ref. 16.

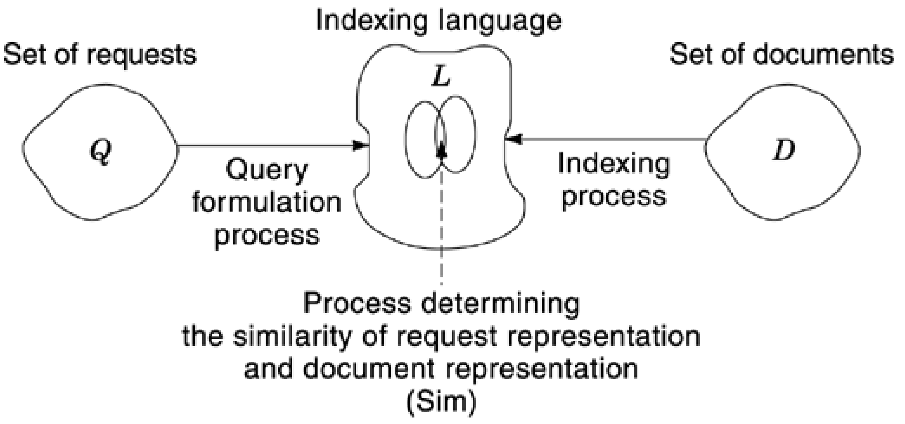
Indexing language
Set of requests
Set of documents
L
Q
D
Query formulation process
Indexing process
Process determining the similarity of request representation and document representation (Sim)

Fig. 12. Functional overview of information retrieval.

Searching in Documents (Information Retrieval)

Suppose, the data set is a collection of documents with natural language text stored in digital form. Users want to search such collections in order to find information about a subject, or topic. This kind of searching is usually referred to as *information retrieval* (*IR*). Whereas searching for data (or *data retrieval*) aims at retrieving elements that satisfy clearly defined conditions, IR aims at retrieving elements that satisfy conditions that are not always defined clearly. Whereas a data retrieval system (such as a relational database) deals with data that have a well-defined structure and semantics, an IR system deals with natural language text, which is not always well structured and may be semantically ambiguous. Data retrieval, while providing a solution to the user of a database system, does not solve the problem of retrieving information about a topic. To be effective in its attempt to satisfy the user information need, the IR system must somehow interpret the contents of the information items (documents) in a collection and *rank* them according to a degree of *relevance* to the user query. This interpretation of a document's content involves extracting syntactic and semantic information from the document text and using this information to match the user information need. The difficulty is not only knowing how to extract this information but also knowing how to use it to decide relevance. Thus, the notion of relevance is a central notion of IR. In fact, the primary goal of an IR system is to retrieve all documents that are relevant to a user query while retrieving as few nonrelevant documents as possible.

An IR system can be described as consisting of a set of documents D, a set of requests Q, and some mechanism Sim for determining which of the documents meets the requirements of, or is relevant to, the requests. In practice, the relevance of specific documents to particular requests is not determined directly. Rather, the objects are first converted to a specific form using a classification or *indexing language* (L). The requests are also converted into a representation consisting of elements of this language. The mapping of objects to the indexing language is known as the *indexing process*, while the mapping of the information requests to the indexing language is known as the *query formulation process*. The procedures for determining which objects should be retrieved in response to a query are based on the representations of the objects and the requests in the indexing language. Figure 12 shows a functional overview of information retrieval.

The set of indexing terms may be *controlled*, that is, limited to a predefined set of index terms, or *uncontrolled,* that is, allowing use of any term that fits some broad criteria. If the indexing language is uncontrolled, then automatic indexing techniques are usually employed. These techniques are based on text analysis (e.g., see Ref. 17). If the indexing language is controlled, then the indexing of the documents is usually done manually. Manual indexing involves some intellectual effort to identify and describe the content of a document. However there are techniques that allow the automatic indexing of objects under a controlled vocabulary.

We use the term *statistical information retrieval* to refer to the case where the indexing language is free and the indexing of documents is done automatically. On the other hand, we use the term *knowledge-based retrieval* to refer to the case where the indexing language is controlled.

Much of the research in information retrieval has used the statistical approach. The retrieval strategies and indexing techniques used in this approach are simple, easily implemented, and reasonably effective. The knowledge-based approach is less well understood. Loosely speaking, this approach is concerned more with the cognitive than with the engineering aspects of information retrieval. By trying to understand more about how people retrieve information and by emphasizing representation and reasoning using domain knowledge, researchers pursuing a knowledge-based approach hope to build systems that achieve significantly better retrieval effectiveness than those based on statistical techniques. Table 1 lists the basic differences between the statistical and knowledge-based approaches (for more see Ref. 18).

The statistical approach provides techniques that can deal with very large databases in a variety of domains and languages, whereas the knowledge-based approach promises to provide techniques for retrieving passages and extracting facts more accurately. Knowledge-based approaches include natural-language-processing techniques for analyzing the text of documents and queries; inference and domain knowledge used in the retrieval process; learning techniques used to improve performance; and user modeling for responding to individual needs. The challenge is to discover how these approaches can be merged into a single theoretical framework and combined in efficient, effective system implementations.

Statistical Information Retrieval. Here the indexing language consists of those words that appear in the documents of the collections. The three classic models in IR are called Boolean, vector, and probabilistic. In the *Boolean model*, documents and queries are represented as sets of index terms (this model can be called set-theoretic). In the *vector model*, documents and queries are represented as vectors in a finite-dimensional space (this model can be called algebraic). In the *probabilistic model*, the framework for modeling document and query representations is based on probability theory (this model can be called probabilistic). Several alternative modeling paradigms have been proposed. Regarding alternative set-theoretic models, we distinguish the fuzzy and the extended Boolean model. Regarding alternative algebraic models, we distinguish the generalized vector, the latent semantic indexing, and the neural network models. Regarding alternative probabilistic models, we distinguish the inference network and the belief network models. In all these models, users specify their information need by a query that can be a phrase (or a document).

Consider a collection D of documents, which together contain n different words. It is common practice to exclude words that do not carry any information when isolated, such as "and," "or," "has," or "why" (this is called elimination of *stopwords*). Sometimes words are *stemmed* before weight vectors are calculated. This reduces distinct words to their common grammatical root.

The Boolean model is a simple retrieval model based on set theory and Boolean algebra. According to this model, keywords are either present or absent in a document, and the queries are specified as Boolean expressions, that is, a query is composed of keywords linked by the three connectives *and, or, not*. The Boolean model predicts that each document is either *relevant* or *nonrelevant* to the query. The main advantages of the Boolean model are the clean formalism behind the model and its simplicity; the main disadvantage is that exact matching may lead to retrieval of too few or too many documents. This problem does not occur in the vector model and in the probabilistic model, as these models take into account the frequencies of the words that appear in documents and queries.

In the vector model, documents are represented as vectors of keywords. Each query or document is represented as an n-dimensional vector where each component corresponds to one keyword in the collection. The weight of term t_i in document d_j, denoted w_{ij}, is the product $w_{ij} = \mathrm{tf}_{ij}\cdot\mathrm{idf}_i$, where tf_{ij} is the *term frequency* of t_i in d_j, and idf_i is the *inverted document frequency* of t_i in the collection D. Specifically, the term frequency tf_{ij} of a term t_i in the document d_j is the number of occurrences of t_i in d_j; while the inverted document frequency

idf_i of a term t_i in a collection D of documents is defined as

$$\text{idf}_i = \log \frac{\text{number of documents in } D}{\text{number of documents in } D \text{ that contain the term } t_i}$$

The motivation for this representation is that there is a natural means of comparing two vectors: the angle, or inverse cosine of the dot product, between the two vectors. The smaller the angle, the more similar the two vectors are—and so, therefore, are the documents they represent. If d_1 and d_2 are document vectors, the similarity is expressed as the cosine of the angle between the two document vectors:

$$\text{sim}\ (d_1, d_2) = \frac{d_1 d_2}{\|d_1\| \cdot \|d_2\|} = \frac{\sum_{i=1}^{n} w_{i1} \times w_{i2}}{\sqrt{\sum_{i=1}^{n} w_{i1}^2} \times \sqrt{\sum_{i=1}^{n} w_{i2}^2}}$$

For a description of the probabilistic model, see Ref. 19.

The evaluation of an IR system commonly concerns the retrieval *efficiency* and *effectiveness.* Retrieval efficiency concerns issues like the user effort and the time needed for retrieving the desired information. The evaluation of retrieval effectiveness is based on a test reference collection and on an evaluation measure. The test reference collection consists of a collection of documents, a set of example requests, and a set of relevant documents (provided by specialists) for each example information request. Given a system S, the evaluation measure quantifies (for each example request) the similarity between the set of documents retrieved by S and the set of relevant documents provided by the specialists. This provides an estimation of the goodness of the system S. The most widely used evaluation measures are *recall* and *precision*, defined as follows. Let R be the set of relevant documents for a given information request, and assume that the system being evaluated processes the information request and generates a document answer set A. The recall and precision are defined as follows:

$$\text{recall} = \frac{|A \cap R|}{|R|}, \qquad \text{precision} = \frac{|A \cap R|}{|A|}$$

Alternative measures that have been proposed include the *harmonic mean* and the *E measure*. The *E* measure allows the user to specify whether he or she is more interested in recall or precision, and it is defined as follows:

$$E(j) = 1 - \frac{1 + b^2}{\frac{b^2}{\text{recall}(j)} + \frac{1}{\text{precision}(j)}}$$

where recall(j) is the recall for the jth document in the ranking, precision(j) is the precision for the jth document in the ranking, $E(j)$ is the evaluation measure relative to recall(j) and precision(j), and b is a user-specified parameter that reflects the relative importance of recall and precision. Other user-oriented evaluation measures that have been proposed include *coverage* and *novelty*.

Frequently, the initial query yields an answer that does not satisfy the user's information need. In such cases, the user can reformulate the query. *Relevance feedback* is the most popular query reformulation strategy. In a relevance feedback cycle, the user is presented with a list of retrieved documents and, after examining them, marks those that are relevant. In practice only the top-ranked documents need to be examined. The main idea consists of selecting important terms, or expressions, attached to the documents that have been identified as relevant by the user, and enhancing the importance of these terms in a new query formulation. The expected effect is that the new answer will be moved towards the relevant documents and away from the nonrelevant ones.

For more on information retrieval see Refs. 17, 4, and 20.

Knowledge-Based Retrieval. Here the indexing language is a controlled vocabulary, which may contain terms that do not appear in the documents of the collection. Usually, these vocabularies are structured, by a small set of relations such as subsumption and equivalence. The so-called *thesauri* (21) constitute an important example of such indexing languages. They capture an adequate body of real world (domain) knowledge, which is exploited through some form of reasoning for improving the effectiveness of retrieval. The adoption of thesauri has proved its usefulness in improving the effectiveness of retrieval and in assisting the query formulation process by expanding queries with synonyms, hyponyms, and related terms.

However, when the indexing process is done manually, indexing of objects can also be done with respect to more expressive conceptual models, usually called *ontologies*. These models represent domain knowledge in a more detailed and more precise manner, using logic-based formalisms and their reasoning mechanisms for retrieving the objects. Recently, several works have followed this approach to IR (e.g. relevance terminological logics or four-valued logics). Even ontologies that have no clear semantic interpretation, such as some linguistic ontologies, can nevertheless be made useful in IR by applying techniques such as spreading activation (22) or by representing objects and queries by lexical conceptual graphs (23).

Searching in the Web

Here, the data set is the World Wide Web. Roughly, we can view the Web as a distributed stored directed graph where each node is a Web page, that is, an HTML page. Each page contains text (and probably other media such as images, audio, and videos) and *hyperlinks* that originate from specific positions in the page and point to other pages. The distinguishing characteristics of the Web are that it is very big, it is not stored in a single machine, and it is subject to continuous change. However, the basic objective of searching in the Web is identical to the objective of searching in documents (IR, viz., to find information that is relevant to a topic.

There are basically three different ways of searching the Web. The first is to use *search engines* that index (a portion of) the Web as a full-text database. In this case, the objective of searching can be specified by a set of words, a phrase, or a pattern (using proximity operators or wildcards), or by a page that is similar to the desired ones. The second is to use *Web catalogues*, which classify selected Web pages by subject. Here the objective is specified gradually, by browsing a hierarchy of subject terms until the area of interest has been reached. The corresponding node then provides the user with links to related pages. The third is to search the Web by exploiting its *hyperlink structure*. Here the objective is related to the connectivity of the graph (e.g., find all pages that have links pointing to a specific page).

Search Engines. Search engines are usually based on IR techniques. However, in the case of the Web, the data set is not stored in a single machine, so the IR techniques for searching cannot be applied directly. In order to apply them, most search engines use a centralized crawler-indexer architecture as shown in Fig. 13. A specialized program called a *crawler* traverses the Web and sends pages to a main server, where they are indexed. Crawlers (also called robots, spiders, or knobots) start from a given set of (popular) pages and traverse the Web in breadth-first or depth-first fashion. One problem here is how to avoid visiting the same page more than once. Moreover, as the Web is subject to continuous change, efficient techniques are needed for keeping up to date the indexes stored at the server.

For the indexing of the gathered pages, search engines use variants of the inverted file approach. Moreover, in order to give the user some idea about each page retrieved, the index is complemented with a short description

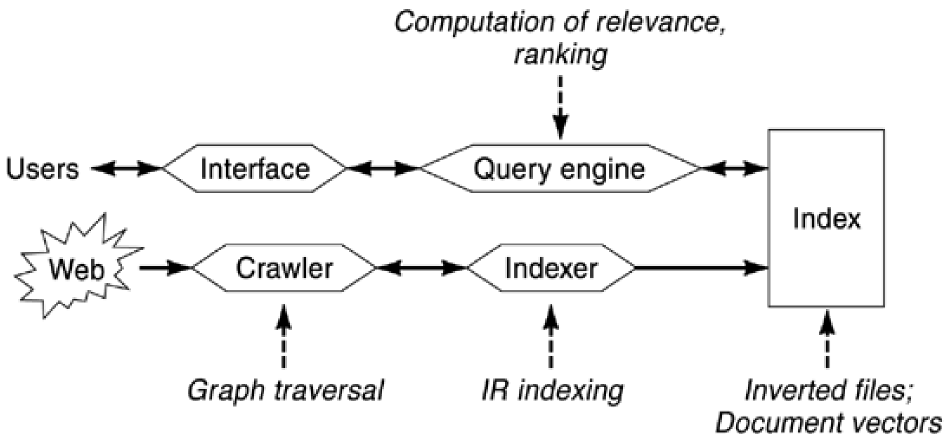

Fig. 13. The crawler-indexer architecture for searching the Web.

of each Web page. A query is answered by doing binary search on the sorted list of words (vocabulary) of the inverted file.

Concerning relevance, most search engines use variants of the Boolean or vector model to do ranking. As the Web is very big, the link structure is exploited in order to deduce the pages that contain valuable information. This is an important difference between the Web and normal IR databases. The number of hyperlinks that point to a page provides a measure of its popularity and quality. Also, many links in common between pages, or many pages referenced by the same page, often indicate a relationship between those pages. A popular ranking scheme is hypertext-induced topic search (HITS) (24). It considers the set S of pages that point to or are pointed to by pages in the answer. Pages that have many links pointing to them in S are called *authorities* (they should have relevant content). Pages that have many outgoing links to pages in S are called *hubs* (they should point to similar content). A positive two-way feedback exists: better authority pages come from incoming links from good hubs, and better hub pages come from outgoing links to good authorities. Let $H(p)$ and $A(p)$ be the hub and the authority value of a page p. These values are defined so that the following equations are satisfied for all pages p:

$$H(p) = \sum_{u \in S | p \to u} A(u), \qquad A(p) = \sum_{u \in S | u \to p} H(u)$$

where $p \to u$ means that page p has a link pointing to page u.

Another ranking scheme is *page rank*, which is part of the ranking algorithm used by the popular search engine Google (25). This scheme simulates a user navigating randomly in the Web who jumps to a random page with probability q or follows a random hyperlink (on the current page) with probability $1 - q$. This process can be modeled with a Markov chain, from which the stationary probability of being in each page can be computed. This value is then used as part of the ranking mechanism. Let $C(a)$ be the number of outgoing links of page a, and suppose that page a is pointed to by pages p_1 to p_n. Then the page rank of a, PR(a), is defined as

$$\mathrm{PR}(a) = q + (1-q) \sum_{i=1}^{n} \frac{\mathrm{PR}(p_i)}{C(p_i)}$$

where q must be set by the system (a typical value is 0.15). The page rank can be computed using an interactive algorithm, and corresponds to the principal eigenvector of the normalized link martrix of the Web (which is the transition matrix of the Markov chain).

Page rank is a global ranking scheme that can be used to rank search results, while the HITS algorithm identifies, for a given search query, a set of authority pages and a set of hub pages. A comparison of the performance of these link-based ranking techniques can be found in Ref. 26.

There are several variants of the crawler-indexer architecture. Among them, the most popular is Harvest, which uses a distributed architecture to gather and distribute data.

Web Catalogs. Web catalogs, such as Yahoo! (www.yahoo.com) or Open Directory (http://dmoz.org), use structured and controlled indexing languages for indexing the pages of the Web. These catalogs turn out to be very useful for browsing and querying. Although they index only a fraction of the pages that are indexed by the search engines using statistical methods, they are hand-crafted by domain experts and are therefore of high quality. Recently, the search engines have started to exploit these catalogs in order to enhance the quality of retrieval and to offer new functionalities. Specifically, the search engines now employ catalogs for computing "better" degrees of relevance, and for determining and presenting to the user a set of relevant pages for each page in the answer set. In addition, some search engines now employ taxonomies in order to enable limiting the scope (or defining the context) of search.

For example, one can first select a category (e.g. Sciences/CS/DataStructures) from the taxonomy of a catalog and then submit a natural language query (e.g. "Tree"). The search engine will compute the degree of relevance with respect to the natural language query "Tree" only of those pages that fall in the category Sciences/CS/DataStructures of the catalogue. Clearly, this enhances the precision of retrieval and reduces the computational cost.

Searching Using Hyperlinks. There are paradigms for searching the Web that are based on exploiting its hyperlinks. For example, one might like to search for all Web pages that contain at least one image and that are reachable from a given site following at most three links. To pose this kind of query, the Web is viewed as a *labeled graph*. Examples of this kind of approach are the *Web query languages* and *dynamic searching*. The existing Web query and manipulation languages provide access to the structure of Web pages and allow the creation of new structures as a result of a query.

Dynamic search in the Web is equivalent to sequential text searching. The idea is to use an online search to discover relevant information by following links. The main advantage is that searching is carried out in the current structure of the Web, and not in what is stored in the index of a search engine.

Metasearching. Metasearchers are also employed for searching the Web. A metasearcher is actually a mediator, that is, a secondary information source aiming at providing a uniform interface to a number of underlying sources (which may be primary or secondary). Users submit queries to the mediator. Upon receiving a user query, the mediator queries the underlying sources. This involves selecting the sources to be queried and formulating the query to be sent to each source. Finally, the mediator appropriately combines the returned results and delivers the final answer to the user.

The main advantages of metasearchers are that (a) they combine the results of many sources and (b) they allow the user to avoid posing the same query to multiple sources, by providing a single common interface. Metasearchers differ mainly in the way ranking is performed in the result, and in how well they translate the user query to the query language of each search engine.

For more details see Chap. 13 of Ref. 3, and for an overview of current Web search engine design see Ref. 27.

BIBLIOGRAPHY

1. M. R. Garey D. S. Johnson *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W. H. Freeman, 1979.
2. D. Knuth *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Reading, MA: Addison-Wesley, 1973.
3. R. Baeza-Yates B. Ribeiro-Neto *Modern Information Retrieval*, New York: ACM Press, Addison-Wesley, 1999.
4. G. Brassard P. Bratley *Algorithmics: Theory and Practice*, Upper Saddle River, NJ: Prentice-Hall, 1988.
5. T. Cormen A. Thomas *Introduction to Algorithms*, Cambidge, MA and New York: MIT Press and McGraw-Hill, 1990.
6. N. J. Nilsson *Artificial Intelligence—A New Synthesis*, San Francisco: Morgan Kaufmann, 1998.
7. J. Pearl *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Boston: Addison-Wesley, 1984.
8. V. Kumar Algorithms for constraint-satisfaction problems: A survey, *Artif. Intell. Mag.*, **13** (1): 32–44, 1992.
9. T. A. Marsland *Tree Searching Algorithms, Computer, Chess and Cognition*, 1991, pp 133–158.
10. R. Ramakrishnan *Database Management Systems*, New York: WCB/McGraw-Hill, 1998.
11. D. J. Hand *et al. Principles of Data Mining*, Cambridge, MA: MIT Press, 2001.
12. S. Abiteboul *et al. Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann, 1999.
13. W. Buthard R. Keller Some approaches to best-match file searching, *Comm. ACM*, **16** (4): 230–236, 1973.
14. R. Baeza-Yates *et al.* Proximity matching using fixed-queries trees, *Proc. 5th Combinatorial Pattern Matching Conf. (CPM'94)*, Lecture Notes in Computer Science, Vol. 807, 1994, pp. 198–212.
15. C. Bohn *et al.* Searching in multidimensional spaces, *ACM Comput. Surveys*, **33** (3): 322–373, 2001.
16. E. Chavez *et al.* Searching in metric spaces, *ACM Comput. Surveys*, **33** (3): 273–321, 2001.
17. G. Salton M. J. McGill *Introduction to Modern Information Retrieval*, New York: McGill, 1983.
18. B. Croft Knowledge-based and statistical approaches to text retrieval, *IEEE Expert*, **9**: 8–12, 1993.
19. S. E. Robertson K. Sparck Jones Relevance weighting of search terms, *J. Amer. Soc. Inf. Sci.*, **27** (3): 129–146, 1976.
20. R. R. Korfhage *Information Storage and Retrieval*, New York: Wiley, 1997.
21. International Organization for Standardization, *Documentation—Guidelines for the Establishment and Development of Monolingual Thesauri*, Ref. No. ISO 2788-1986, 1988.
22. C. Paice A thesaural model of information retrieval, *Inf. Process. Manage.*, **27** (5): 433–447, 1991.
23. N. Guarino *et al.* OntoSeek: Content-based access to the Web, *IEEE Intell. Syst.*, **14** (3): 70–80, 1999.
24. J. Kleinberg Authoritative sources in a hyperlinked environment, *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, San Francisco, 1998.
25. S. Brin L. Page The anatomy of a large-scale hypertextual Web search engine, *Proc. 7th Int. WWW Conf.*, Brisbane, Australia, 1998.
26. B. Amento *et al.* Does authority mean quality? Predicting expert quality ratings of Web documents, *Proc. 23rd ACM SIGIR Conf.*, 2000.
27. A. Arasu *et al.* Searching the Web, *ACM Trans. Internet Technol.*, **1** (1): 2001.

NICOLAS SPYRATOS
Université de Paris-Sud
YANNIS TZITZIKAS
University of Crete