Search Algorithms

CHAPTER 1: FUNDAMENTALS OF SEARCH ALGORITHMS

1. Introduction

In the last few decades, the field of relation database has emerged as a powerful technological tool for manipulation and transfer of data. Recent years have witnessed rapid advances in data science and technology which have significantly impacted the ways of data representation (Abiteboul et al., 1995; 1997; 1999). A new challenge has surfaced in the database technology which limits the efficient representation of data via classical tables. Several database applications present the data as trees and graphs. On the other hand, there are some applications which require the particular database system to manipulate uncertainty and time. Extensive research work is being carried out to combat the challenges encountered in the database systems (Adalı & Pigaty, 2003; Shaw et al., 2016).

Presently, the researchers are exploring the prospects of formulating the models for the "next generation database systems", i.e. the databases capable of representing novel data types along with providing novel manipulation abilities while effectively supporting standard search operations. Modeling of the data is carried out in the form of graphs and trees in next-generation databases which efficiently deal with Web, XML, structured documents and network directories (Altınel & Franklin, 2000; Almohamad & Duffuaa, 1993). Additionally, new database systems also utilize graph or tree models for data representation for dealing with a range of application areas like image databases, commercial databases or molecular databases. Currently, the development of various algorithms for graph and tree graph querying is being carried out due to the immense significance of graph and tree database systems (Altman, 1968; Amer-Yahia et al., 2001; 2002). Apart from the necessity for querying and searching graphs and trees, several applications also need temporal uncertainty linked with database objects, e.g., databases for commercial package deliveries, weather databases, and financial databases (Atkinson et al., 1990; Andries & Engels, 1994).

Efficient sorting and searching are considered as the fundamental and broadly encountered challenges in the field of computer science. For instance, for a collection of certain objects, the objective of the search algorithm is to find and distinguish a particular object from the others. On the other hand, the search algorithm can perform the identification analysis of a particular object which is not present in the system (Baeza-Yates, 1989; Baeza-Yates et al., 1994). The database objects often possess some key values which form the basis for a particular search. Additionally, there are some data values which represent the information intended to be retrieved after an object has been found (Baeza-Yates & Gonnet, 1996; Baeza-Yates & Ribeiro-Neto, 1999). For instance, a telephone book contains a collection of contacts with different names and telephone numbers. The search algorithm is used to locate a particular name or number after incorporation of certain search inputs. It is a common understanding that some data is associated with these key values, i.e., name, number, etc. Let's consider a search case

which involves searching for a single key value (e.g., name). The group of objects is usually stored in an array or a list. Given the collection of n (number of) objects in a particular array A (i.e., A [1 . . . n]), the i^{th} element (i.e., A[i]) typically corresponds to the key value for the i^{th} object present in the collection (Barrow & Burstall, 1976; Barbosa et al., 2001; Boag et al., 2002).

The objects are often sorted by using key values (e.g., a phone book), however, this is not essential in all the cases. Different search algorithms may be required depending on the sorting condition of the data (i.e., either sorted or not). The inputs for a particular search algorithm include the number of objects (i.e., n), an array of objects (i.e., A) and a key value being required (i.e., x). Different types of search algorithms are discussed in the following sections (Boncz et al., 1998; Bomze et al., 1999).

2. Unordered Linear Search

Let's assume that any given array is not essentially sorted. This might correspond to unsorted collection exams which lack alphabetical sorting. For instance, if a student wants to obtain his/her exam results, how could she/he do so? He/she would search through the whole collection of exams sequentially until her/his exam is found (Boole, 1916; Bowersox et al., 2002). This search is associated with the unordered linear search algorithms. A typical example of an Unordered Linear Search is illustrated below:

Input: A-objects array; n-the number of objects; x-key value being determined

Output: return point *i*, if not, return note "x not found"

i. Execute the comparison of x with each array (A) element A from the very start.

ii. If x = i^{th} element in array A. Return point *i* after termination of the search.

iii. If not, continue searching for the succeeding element until the array ends.

iv. If any proper element is not found in the array (A), return note "x not found".

It is essential to search the whole collection to determine the existence of a particular object. Let's consider the array illustrated below:

A	35	17	26	34	8	23	49	9
i	I	II	III	IV	V	VI	VII	VIII

If we need to search for x = 34 in this array (A). We require the comparison of x with (35, 17, 26, 34), each element once. The desired number (34) is present in position 4. Hence, we return 4 after execution of 4 comparisons.

If the search for x = 19 is required in this particular array. We require the comparison of x with (35, 17, 26, 34, 8, 23, 49, 9), each element once. We do not find 19 after searching all the objects in this array. Therefore, we return "18 not found". We have executed a sum of 8 comparisons in this case. Generally, it is essential to carry out the determination (search) of x in an array of unordered objects with n

elements. Sometimes, it is mandatory to search through the whole array to get the desired answer. It implies the execution of n comparisons. An equation representing the number (n) of executed comparisons can be formulated as T (n) = n (brin, 1995; Bozkaya & Ozsoyoglu, 1999).

2. Ordered Linear Search

Let's assume that a given array is sorted. Such cases do not necessitate the search to be executed through the whole list to locate a particular object or inquire about its existence in the collection of objects. For instance, if the sorting of a collection of exam scores is carried out by name, it is not essential to search beyond the "J"s to determine whether the exam score for "Jacob" exists in the collection or not. A simple amendment of the above algorithm results in the ordered linear search algorithms (Brusoni et al., 1995; Console et al., 1995; Buneman et al., 1998). A typical model of the Ordered Linear Search is illustrated below:

Input: B-objects array; n-number of objects; x-key value being determined

Output: return point *i*, if not, return note "x not found"

i. From the start of the array B, carry out the comparison of x with the element B[i] in A to check their equality.

ii. If x = B[i], stop the search and return location i.

iii. If not, execute the comparison of x with that element once again too if the value of x is greater than B[i].

iv. If x > B[i], continue searching for the next object in array B.

v. If not (i.e., x < B[i]), stop the search and return a message "x not found".

Let's consider the following sorted version of the previously used array:

B	8	9	17	23	26	34	35	49
i	I	II	III	IV	V	VI	VII	VIII

For searching x = 34 in the above-mentioned array, the comparison of x with each element (8, 9, 17, 23, 26) is carried out twice (i.e., one for "="and another one for ">"). After that, the comparison of x with (34) is carried out only once for "=". Hence, we will find 34 in position 6 which will return position 6. Total number of comparisons = 2*5 + 1 = 11.

If x = 19 is searched in this array, the comparison of x with each element (8, 9, 17, 23, 26) is carried out twice (i.e., one for "=" and another one for ">"). And in the final comparison (i.e., if x > 25), we get a response "NO" which means all the objects after 26 in this array are essentially greater than x. Hence, we return the message "x not found". A total number of comparisons are 2*4 = 8.

Generally, it is essential to carry out the determination (search) of x in an array of ordered objects with n elements. Sometimes (when the value of x is greater than all values of the array), it is mandatory to search through the whole array to get the desired answer. It implies the execution of n x 2 comparisons (i.e., n for "=" and another n for ">"). An equation representing the number (n) of executed comparisons can be formulated as T (n) = 2n (Bunke & Allermann, 1983; Bunke, 1999; 2000).

3. Chunk Search

In case of an ordered list, there is no need to search through the whole collection sequentially. Let's consider the search for a name in the phone book or locating a particular exam in a sorted collection: one might instinctively grab 40 or more pages at once from the phone book or 20 or more exams at once from the collection to swiftly determine the 40 page (or 20 exams) chunk (pile) in which the desired information lies. A careful search needs to be carried out through this pile (chunk) via an ordered linear search algorithm (Bunke & Keller, 1973; Bunke, 1997; Bunke et al., 2002). Let's assume that c is the chunk size utilized for 40 pages or 20 exams. Moreover, we can assume that a considerably generalized algorithm is accessible to us for ordered linear search. These assumptions in conjunction with the above-mentioned ideas can result in the formulation of the chunk search algorithm (Burns & Riseman, 1992). A typical Chunk Search algorithm is illustrated below:

Input: c-chunk size, an A-ordered array of objects, n-the number of elements, the x-key value being determined.

Output: if found, return position i, if not, return the message "x not found". Disintegrate array A into c-sized chunks.

i. Perform the comparison of x with the each chunk's last elements, excluding the last chunk.
ii. Determine if the value of x is greater than that particular element.
iii. If yes, continue to check the next chunks
iv. If no, it means x lies in that particular chunk
v. Execute Ordered Linear Search algorithm inside the chunk

Consider the array as described follows:

B	8	9	17	23	26	34	35	49
i	I	II	III	IV	V	VI	VII	VIII

Let's select the size of the chunk as 2 and search for x = 34. Initially, we divide this array into four chunks of size 2 followed by a one-time comparison of x with the each element's last chunk (9, 23, 34). We perform the comparison to see if the value of x is greater than that particular element. We received an answer "NO" when the comparison is run for 34 which implies that x should lie in the third chunk.

Finally, the execution of an ordered liner search is carried out in the third chunk followed by the location of 33 in position 6. In this particular scenario, we perform three comparisons to determine the proper chunk followed by execution of 3 comparisons inside the chunk (i.e., 2 for 26 and 1 for 34). Six comparisons are performed in total. Generally, it is essential to carry out the search of x in an array of ordered objects with chunk size c and n elements (Chase, 1987; Chawathe et al., 1997; 1998). In the worst case scenario, execution of n/c - 1 comparisons are carried out to determine the proper chunk while execution of 2c comparisons is carried out to perform the linear search. An equation representing the number (n) of executed comparisons can be formulated as: T(n) = n/c +2c – 1. Typically, we just ignore the constant number, because the constant number has no effect when n becomes higher. Finally, we get T(n) = n/c +2c (Cai et al., 1992; Chawathe et al., 1996; Caouette et al., 1998).

4. Binary Search

Let's consider the below discussed concept for a search algorithm employing the phone book example. Suppose, we choose a page randomly from the middle of a phonebook. If the particular name being determined is on this page, we are successful. If the particular name being determined occurs alphabetically prior to this page, the processes are repeated on the phonebook's first half; otherwise, the process is repeated on the phonebook's second half. It may be noted that each iteration involves the division of the remaining chunk of the phonebook to be sought into two halves; the algorithm involving such strategies is known as binary search (Chiueh, 1994; Christmas et al., 1995; Ciaccia et al., 1997). This algorithm may not sound like the most suitable algorithm for searching the phonebook (or an ordered list), this is possibly the quickest. This is true for many computer algorithms, i.e., the most natural (suitable) algorithm is not essentially the best (Cole & Hariharan, 1997; Clark & DeRose, 1999). A typical model for Binary Search algorithm is illustrated below:

Input: an A-ordered array of objects, n-the number of elements, the x-key value being determined.

Output: if found, return position i, if not, return the message "x not found".

i. Divide the array into two equal halves.
ii. Perform the comparison of x with the first half's last element to check the equality of x with that particular element.
iii. If yes, stop search and return the position.
iv. If no, perform the comparison of x with the first half's last element once again to see if the value of x is greater than that element.
v. If yes, it means x should lie in the second half. Now, the second half must be treated as a new array for performing the Binary Search on it.
vi. If no, it means x should lie in the first half. Now, the first half must be treated as a new array and Binary Search is performed on this array.
vii. If x is not found, return a message "x not found".

Again consider the below illustrated array:

B	8	9	17	23	26	34	35	49
i	I	II	III	IV	V	VI	VII	VIII

If we need to search for x = 26. We will perform the comparison of x with each element (23, 34) twice (i.e., one for "=" and another for ">") followed by comparison with the element (26) once for "=". Finally, we find x in 5^{th} position. Total number of comparison are 2 x 2 + 1 = 5. In general, the worst case scenario gives: $T(n) = 2\log_2 n$ (Cook & Holder, 1993; Cole, et al., 1999)

5. Searching in Graphs

Graphs are widely used in several applications to characterize the data because of their general, powerful and flexible structure. A graph consists of a set of edges and vertices between the pairs of vertices. Typically, the edges are used to represent the relations between different data variables while vertices are used to represent the data (i.e. anything which needs a description). Depending on the extent of abstraction employed to represent the information (data), a graph offers either a semantic or a syntactic illustration of the data (Cordela et al., 1996; 1998; 2001).

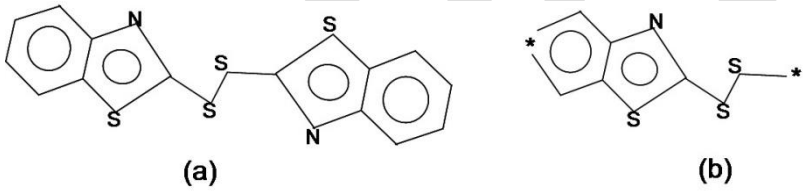

Fig. 1. (a) Formula of a chemical compound. (b) A query consisting of wildcards. Graphs are naturally used to describe their structures

[Source: https://www.semanticscholar.org/paper/Searching-Algorithms-and-Data-Structures-for-%2C-and-Giugno/4378f3b5d0495f4164c4ac74f01ecef414951974]

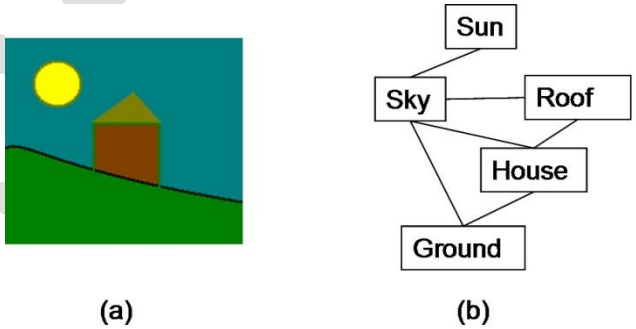

Fig. 2. (a)Illustration of an image (b) Demonstration of a Region Adjacent Graph- RAG of the image

[Source: https://www.semanticscholar.org/paper/Searching-Algorithms-and-Data-Structures-for-%2C-and-Giugno/4378f3b5d0495f4164c4ac74f01ecef414951974]

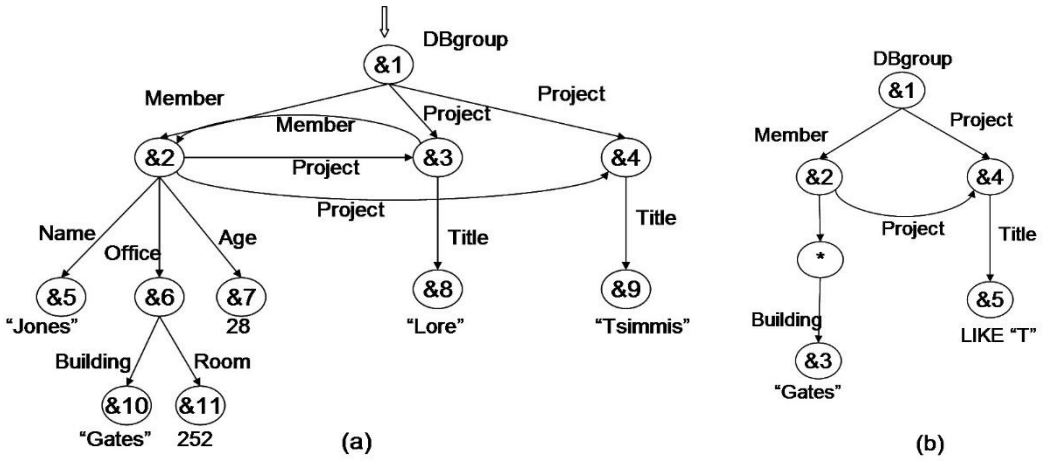

Fig. 3. Illustration of (a) A structured database tree and (b) A query consisting of wildcards

[Source: https://www.semanticscholar.org/paper/Searching-Algorithms-and-Data-Structures-for-%2C-and-Giugno/4378f3b5d0495f4164c4ac74f01ecef414951974]

Let's discuss a biochemical (e.g., proteins) database system [97, 18, 146]. Proteins are typically characterized using labeled graphs, i.e., the edges represent the linkages between different atoms while vertices represent various atoms. The classification of proteins is usually carried out on the basis of their common structural features. One of the applications of such categorizations is the estimation of the functionality for a new protein fragment (i.e., synthesized or discovered). The deduction of this functionality can be carried out by locating the common structural features between known proteins and the new fragments. Finally, the wildcards may be present in the queries which exhibit matching characteristics with vertex or paths in the data (Corneil & Gotlieb, 1970; Day et al., 1995).

Visual languages contain graphs which serve as the underlying data models. These languages are employed in software engineering and computer science to specify tools and projects for integrated environments, in Computer Integrated Manufacturing systems exhibiting process modeling and in visual database systems for describing the query language semantics (Bancilhon et al., 1992; Dehaspe et al., 1998; Dekhtyar et al., 2001).

Computer vision graphs are used to represent various kinds of images in different abstraction levels (DeWitt et al., 1994; Deutsch et al., 1999). In a representation with the low-level description, the graph vertices correspond to edges and pixels to spatial associations between pixels (Djoko et al., 1997; Dinitz et al., 1999). In higher description levels, the image represented by a graph (i.e., RAG) is carried out in such a way that the image decomposition in regions takes place. In this scenario, the regions signify the graph vertices and the edges represent the spatial linkages between different regions (Dutta, 1989; Dubiner et al., 1994).

The network directories, semi-structured database systems, and Web usually model the data in the form of graphs (Dyreson & Snodgrass, 1998; Eiter et al., 2001). Such database systems typically consist of directed labeled graphs possessing the vertices with complex objects while the edges are an illustration of the linkages between the objects. The large sizes of such database systems necessitate the specification of the queries using wildcards. This phenomenon allows for the retrieval of subgraphs utilizing a partial information of the whole graph (Eshera & Fu, 1984; Engels et al., 1992; Cesarini et al., 2001).

Apart from data storage in graphs, most of the above-mentioned applications necessitate the use of tools for comparing different graphs, recognizing different regions of graphs, retrieving the graphical data and classifying the data. In case of the non-structured data (e.g., strings), an efficient response is received from the modern search engines for keyword-based queries. This excellent speed is possible due to various factors which include caching, inverted (clever) index structures and the use of distributed and parallel computing (Ferro et al., 1999; 2001' Foggia et al., 2001). Similar to the mechanism of word matching in keyword searching, the query graphs are matched in keygraph searching against the fundamental data graphs. Significant efforts have been carried out to generalize keyword searches for keygraph searching (Frakes & Baeza-Yates, 1992; Fortin, 1996). However, such generalizations are not so natural, i.e., keyword searching exhibits *polynomial* complexity on the size of the database. On the other hand, keygraph searching exhibits *exponential complexity which makes it an* entirely distinct class of problems. There are different types of problems associated with keygraph searching, which are discussed in the following sections (Gadia et al., 1992; Fernandez et al., 1998; Fernández & Valiente, 2001).

5.1.Exact Match or Isomorphisms

Given a data graph G_b and a query graph G_a, we can determine if these two graphs are identical. Isomorphic nature of the graphs Ga and Gb are determined along with the mapping of the Ga vertices and Gb vertices while maintaining the conforming edges in Gb and Ga. This issue is recognized to be in NP (nondeterministic polynomial time) but it is still unknown whether it lies in NP-complete or P (polynomial time) (Garey & Johnson, 1979; Gold & Rangarajan, 1996; Goldman & Widom, 1997).

5.2.Subgraph Exact Matching or Subgraph Isomorphism

Given a data graph G_b and a query graph G_a, we say conveniently assume that G_a is a subgraph isomorphic to G_b provided that G_a is also isomorphic to the subgraph of G_b. It may be noted that G_a has the potential to be subgraph isomorphic for various subgraphs of G_b. This issue is considered to lie in NP-complete. Additionally, it is significantly expensive to find all the subgraphs which exhibit similarity with the query graph G_a (e.g., exhibiting the maximum occurrences of G_a graph in G_b) rather than finding a single occurrence (Gonnet & Tompa, 1987; Grossi, 1991; 1993).

5.3. Matching of Subgraph in a Graphs' Database

Given a data graphs *D* and a query graph G_a, we want to discover all the G_a occurrences in each graph of *D*. Although the algorithms with graph to graph matching can be employed, but the use of special techniques has proved to be efficient in order to diminish the time complexity and search space in the database systems. This problem also corresponds to NP-complete (Hirata & Kato, 1992; Güting, 1994; Gupta & Nishimura, 1998).

A simple listing algorithm to discover the occurrence of the query graph G_a in the data graph G_b involves the generation of all potential maps between the vertices of the two graphs followed by checking the matching characteristics of the map. The algorithms with such graphs exhibit exponential complexity (Hirschberg & Wong, 1976; Kannan, 1980; Goodman & O'Rourke, 1997).

Many attempts have been made to lessen the combinatorial costs involved in graph searching. The research efforts in this field have taken following three directions:

i. The first effort involves the study of matching algorithms for specific graph structures, e.g., planar graphs, associated graphs and bounded valence graphs (Umeyama, S. (1988; Cour et al., 2007);
ii. The second effort include the mechanism to figure out tricks for reducing the number (quantity) of generated maps (Wilson & Hancock, 1997; Luo & Hancock, 2001) ;
iii. Finally, the third research effort is to offer approximate algorithms with polynomial complexity; however, they do not guarantee a correct solution (Leordeanu & Hebert, 2005; Leordeanu et al., 2009).

Various algorithms have been established for keygraph searching to deal with the cases in which precise matches are difficult to find (Milo & Suciu, 1999; Conte et al., 2004). Such algorithms are quite useful in applications involving noisy graphs. These algorithms typically utilize a cost function to predict the graphs' similarity and perform the transformation of graphs into one another (McHugh et al., 1997; Al-Khalifa et al., 2002; Chung et al., 2002). For instance, *semantic transformations can be used to define* a cost function which mainly depends on the specific application domains and permit the vertices' matching with discordant values. *Semantic transformations* are also dependent on *syntactic transformations* (i.e., branch insertion and deletion) which are responsible for matching structurally distinct parts of the graphs. Alternatively, the approximate algorithms are also employed for noisy data graphs (Ciaccia et al., 1997; Cooper et al., 2001; Kaushik et al., 2002).

In case of query graphs present in the database of graphs, most of the contemporary methods are intended for specific applications (Gyssens et al., 1989; Salminen & Tompa, 1992). Several querying approaches for semi-structured database systems have been proposed by various researchers. Moreover, numerous commercial products and academic projects have also been carried out which involve subgraph searching in different biochemical database systems (Macleod, 1991; Kilpeläinen & Mannila,

1993). These two distinct examples possess different fundamental data models (i.e., initially the database is observed as a large graph in case of commercial products while in the database is seen as a collection of graphs in case of academic projects). However, the above-mentioned techniques exhibit following common approaches:

i. Regular path expressions ([193, 46])
ii. Regular indexing methods

These techniques are used during query time to trace the substructures in the database in addition to avoiding the unnecessary database traversals (Tague et al., 1991; Navarro & Baeza-Yates, 1995).

In comparison with the application-specific approaches, only a small number of application-independent techniques are present for querying graph database systems. The query graphs with same sizes are typically employed in the database systems, however, there are some techniques which exempt the same-size restriction (Dublish, 1990; Burkowski, 1992; Clark et al., 1995). A general idea in the above-mentioned algorithms is to develop an index of similarities between the subgraphs and graphs of the database followed by their organization in suitable data structures. A method was proposed by Bunke (2000), which indexes the labeled database graphs in exponential time scale and computes the isomorphism of a subgraph in the polynomial time. Both matching and indexing are based on all potential permutations of the neighboring matrices of the graphs. The above algorithm can display improved performance if only a set of probable permutation is maintained (Verma & Reyner, 1989; Nishimura et al., 2000). Cook and Holder (1993) suggested a different approach (i.e., not based on any indexing technique) for searching of a subgraph in a database. They witnessed similar repetitive subgraphs present in a single-graph database system after the application of typical graph matching algorithms (Luccio & Pagli, 1991; Fukagawa & Akutsu, 2004).