Algorithmics and Applications of Tree and Graph Searching[*]

Dennis Shasha
Courant Institute
New York University
shasha@cs.nyu.edu

Jason T. L. Wang
Computer Science Dept.
New Jersey Inst. Tech.
wangj@njit.edu

Rosalba Giugno
Math & Computer Science
University of Catania
giugno@dmi.unict.it

ABSTRACT

Modern search engines answer keyword-based queries extremely efficiently. The impressive speed is due to clever inverted index structures, caching, a domain-independent knowledge of strings, and thousands of machines. Several research efforts have attempted to generalize keyword search to keytree and keygraph searching, because trees and graphs have many applications in next-generation database systems. This paper surveys both algorithms and applications, giving some emphasis to our own work.

1. INTRODUCTION

Next-generation database systems dealing with XML, Web, network directories and structured documents often model the data as trees and graphs. These data modeling efforts include Lorel [3], StruQL [38], and UnQL [17, 19], for semi-structured data, XQuery [15], XML-QL [34], XPath [72] and XSL [67], for XML data, and [45] for structured documents. There have been several proposed approaches for querying trees [8, 9, 40, 45, 53, 78] and for querying graphs [2, 3, 16, 29, 46, 47, 70, 71]. Besides applications over XML data, these algorithms have applications to scientific databases where data are naturally represented by trees (such as phylogeny) and graphs (such as molecular databases).

In Section 2 we present motivating query examples on trees and survey algorithms for processing these queries. Section 3 describes algorithms for searching in graphs. Section 4 concludes the paper and suggests avenues for future work.

2. SEARCHING IN TREES

2.1 Approximate Containment Queries

Just as keyword searching matches words against sequences, keytree searching matches tree patterns against underlying data trees. The following two examples come from the literature.

[*]Work supported in part by U.S. NSF grants IIS-9988345 and IIS-9988636.

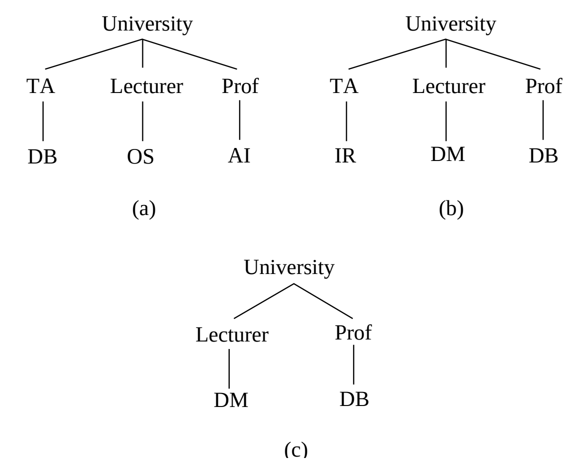


Figure 1: (a) & (b) XML trees. (c) A query tree.

XQuery [15]. Figure 1(a) and (b) show two XML trees describing universities maintained in a XML database. Consider the query [8]: *Find the universities that have a lecturer teaching a data mining (DM) course and that have a professor teaching a database (DB) course.* This query could be expressed by a tree pattern, as shown in Figure 1(c). The tree pattern is contained in the tree in Figure 1(b) and hence the university in Figure 1(b) would be returned as an answer to the query.

AQUA Query [62, 86, 87]. AQUA was an object-oriented data model developed at Brown University for supporting bulk types such as trees, sets, bags, etc. Consider, for example, the family tree in Figure 2(a) [86]. Each node represents a person object. Each edge stands for the relationship "a child of" and a path in the tree stands for the relationship "a descendent of". Now consider the query supported by the "select" operator in AQUA [86]: *Find all nodes (persons) who are ancestors of Alex and also descendants of Mary.* This query could be expressed by a tree pattern, as shown in Figure 2(b). The node "*" in the tree pattern is a variable length don't care (VLDC) [93, 106], which would be instantiated into (matched with) a path of nodes of a data tree at no cost. In our example, the nodes in the family tree matched by the VLDC "*" (here, Bill and Adam) would be returned as answers to the query.

The preceding queries share some characteristics.

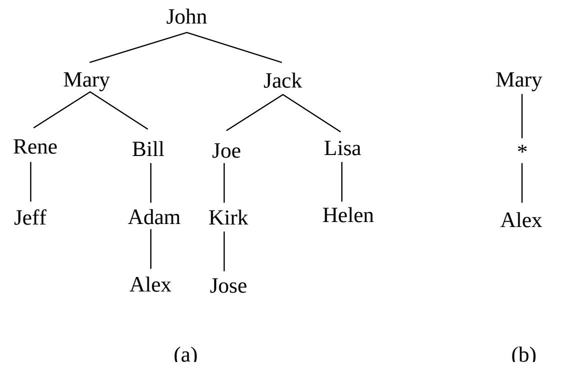


Figure 2: (a) A family tree. (b) A query tree with a VLDC.

- The query could be expressed as a tree pattern, termed "query tree". The database can be represented as a single tree (as in AQUA) or as a set of trees (as in XQuery).
- Each tree could be *ordered* in which the order among siblings is significant (as in the XML data model) or could be *unordered* as in hereditary trees.
- The queries are often concerned with the "parent-child", "ancestor-descendant", or "path" relationship among the nodes of the trees.
- The queries can be expressed by a *containment* mapping. That is, one asks whether the query tree appears, or approximately appears, in a data tree. Here, the "approximation" is measured by the number of paths in the query tree that do not appear in the data tree [84], or by some other distance functions [22, 23, 24, 85, 96, 104, 107].
- The query tree may contain *don't cares* or wildcards [72]. There are fixed length don't cares (FLDCs), "?", that may match a single node and variable length don't cares (VLDCs), "$*$" [84].

We shall refer to this class of queries as *approximate containment* (AC) queries.

In general, a query tree may contain redundant nodes, removal of which would not affect answers to the query. Amer-Yahia, Srivastava, and colleagues [8, 9] developed algorithms for minimizing a query tree, both in the absence and in the presence of integrity constraints. Their algorithms are useful for query optimization since the size of a query tree affects the efficiency of tree pattern matching.

2.2 Path-Only Searches

Many AC queries are concerned with paths only [4, 18, 73], e.g. *find the descendants of Mary who is a child of John.* Since paths are represented as strings, existing algorithms and tools such as AGrep for string searching are applicable to processing these queries (see [10, 11, 12, 41] for a review).

XISS [64] is a XML indexing and querying system designed to support regular path expressions. In each XML tree, each node is associated with a pair of integers enabling the determination of ancestor-descendant relationships among nodes. Each node in the tree is indexed according to its value and the document to which it belongs. Further, all parent-child edges are stored in an index. Processing a query consists of the following steps: (1) decompose the query into parent-child edges or ancestor-descendent paths; (2) for each query edge, use the index to find the corresponding edges in the data trees; (3) for a query path having a Kleene star ($*$) (d1/$*$/d2), locate data node pairs (d1, d2) corresponding to the ancestor-descendent of the query path and determine whether d1 is an ancestor of d2 by using the pair of integers associated with d1 and d2; and (4) combine the results to determine whether there is a match to the query path. Additional techniques can be found in [7, 13].

2.3 Extension to Trees

When extending path-only searching to tree searching, one has to combine path matches into tree matches. We describe our algorithm, called pathfix, for processing AC queries on trees as an illustration.

2.3.1 A Suffix Array Based Algorithm

The pathfix algorithm works in two phases. In the first phase, the database building phase, the algorithm encodes each root-to-leaf path of every data tree into a suffix array database [66]. In the second phase, the on-line search phase in which the query tree Q is given, the algorithm compares Q with each data tree D in the database $\mathcal{D}$ allowing a difference $DIFF$, i.e. at most $DIFF$ paths in Q are allowed to be absent in D in order to consider D to be a match. When comparing Q with D, pathfix takes every root-to-leaf path in Q and finds roots of that path in D by searching in the suffix array database. (As a cutoff optimization, the algorithm stops searching D if more than $DIFF$ paths of Q are missing from D.) Suppose there are k root-to-leaf paths in Q. If a node n in D is the root of all k paths, then the subtree D' rooted at n matches Q with distance 0, provided no siblings have the same label in either the data or query tree. (If there are siblings having the same label, then post-processing can verify the match. The technique will never miss a match.) If n is the root of $k-1$ paths, then D' matches Q with distance 1 and D approximately contains Q with distance 1. If the sibling order in the query tree Q must be preserved in a putative match with D, then the order among the paths in Q must be checked against the order among the paths in D (using the postorder number of the leaves of the paths in D, for example).

2.3.2 Techniques for Queries with Don't Cares

If the query tree Q contains don't cares, pathfix works in three steps:

- partition Q into connected subtrees having no don't cares;
- match each of those don't care free subtrees with data trees in $\mathcal{D}$;
- for the matched substructures that belong to the same data tree, determine whether they combine to match Q based on the matching semantics of the don't cares.

In general, for a query tree Q with don't cares, a node x in a data tree D is the root of a subtree that matches Q if all of the following hold:

1. The partition of Q containing the root r_{all} of Q (call that the root partition of Q) matches D at x.

2. Consider the path p from the root r_{sub} of a subtree in Q to r_{all}. Suppose that r_{sub} matches D at possibly many nodes $x_1, x_2, \ldots$. The path from at least one such node in D, say x_j, has the property that the ascending path from x_j to x matches (with appropriate substitutions for "$*$" and "?") the path from r_{sub} to r_{all}.

To avoid testing the roots of subtrees unnecessarily, the matching uses heuristics like the following: if Q is to match the data tree D at x, then the only relevant matches of a subtree of Q rooted at r_{sub} are nodes that are descendants of x.

When a distance $DIFF$ is allowed in matching a query tree Q with a data tree, for each don't-care-free subtree Q' of Q, pathfix finds all subtrees of data trees that are within distance $DIFF$ of Q'. The gluing process involves a test of whether the glued tree as a whole is indeed within distance $DIFF$ of the entire query tree Q.

2.3.3 Filtering

The above search process can be heuristically improved by using a hashing technique that works as follows on the non-wildcard portion of data and query trees. Compute and store all individual node labels and all parent-child label pairs in each data tree into a hash table, associating each parent-child pair with the set of data trees that contain the parent-child pair. Now suppose a query tree Q is given with a certain distance allowed in searching, $DIFF$. Take the multiset of labels from Q and see which data trees have a super-multiset of those possibly with $DIFF$ missing labels. Take the multiset of parent-child pairs from Q and see which data trees have a super-multiset of those parent-child pairs, again possibly with $DIFF$ missing pairs. This heuristic eliminates irrelevant trees from consideration at the beginning of a search and yields a set of candidate trees to look for.

2.3.4 Implementation

The search and filtering algorithms just described are collectively referred to as ATreeGrep (whose name is shamelessly adapted from AGrep [103] for approximate string searching and SGrep [51] for structure grep). We have implemented ATreeGrep in a XML search engine, called XML Query by Example (XML QBE), which takes an example XML fragment (query tree) and finds the XMLs in a XML database that approximately contain the query tree. For example, the query in Figure 3 is to find all the XML documents describing movies in which Robert Redford is the director, Brad Pitt is an actor, and the movies are made in California, U.S.A. Shown in the figure are (counterclockwise, starting from upper left) the main menu, the querying window, the example XML (query) displayed via a Microsoft IE browser, a matching XML containing the query displayed via the IE browser, the query tree displayed via Java tree show applets, and the matching XML tree displayed via Java tree show applets. The matched portions in the matching XML tree are highlighted and marked with a bullet.

Figure 4 shows an application of ATreeGrep to searching phylogenetic trees maintained in Harvard's TreeBASE [80], accessible from http://www.herbaria.harvard.edu/treebase. The figure shows the search engine's querying interface (in the left window), a query tree (in the right, top window) and a data tree (in the right, bottom window). In the figure, the query tree matches the data tree with distance 0, "?" matches "Myriapoda" and "$*$" matches a path of the data tree. This query finds all the phylogenetic trees in TreeBASE that contain the query tree.

2.4 Related Approaches

2.4.1 Approximate Embedding Queries

Hoffman and O'Donnell [49], and later Ramesh and Ramakrishnan [81], and Cole *et al.* [28] presented algorithms for finding the occurrences of a wildcard-free ordered query tree Q in an ordered data tree D. (In an ordered tree, the order among siblings matters.) Both Q and D are ordered and the occurrences of Q in D refer to those subtrees of D that can be obtained from Q by attaching new subtrees to the leaves of Q. This pattern-matching problem is also known as the *exact ordered tree embedding* problem [83].

While trees for XML and structured documents are ordered, interesting queries are often based on unordered trees, because ordering in the data might not matter to the user. That is why XQuery, for example, supports unordered queries. Schlieder and Naumann [83] extended the exact embedding problem and studied the *approximate embedding* (AE) problem for unordered trees. Consider, for example, the query: *Find all books whose editor is John and that contain a chapter with the title XML.* Figure 5 shows the query tree and a data tree in which the query tree can be approximately embedded. Here, we allow a matching data tree to have nodes between a parent-child pair in the query tree, provided the ancestor-descendant relationship is preserved in the data tree. Figure 5 shows this: the matching data tree has a "Name" node that is missing from the query tree. This type of embedding is also known as *tree inclusion* as defined in [57, 58, 59, 60], where Kilpelainen and Mannila showed the problem to be NP-complete. The AE queries complement the AC queries described in Section 2.1.

The notion of "approximation" can be further generalized by introducing a cost function that assigns a low cost to embeddings in which none or only a small number of nodes are skipped. Schlieder and Naumann [83] presented algorithms to retrieve and rank search results using this cost function. Their algorithm is based on dynamic programming and processes the query tree Q in a bottom up fashion. For each node q of Q, each embedding of the query subtree Q_q rooted at q is computed from the embeddings of the query subtrees rooted at the child nodes of q. Among the valid embeddings of Q_q in a data subtree D_d, the algorithm only maintains the one with the minimal cost. Repeating the above steps for each matching data node of q yields a set of embeddings of Q_q. At the end of the algorithm, the embeddings of Q are sorted by increasing cost and presented to the user. The

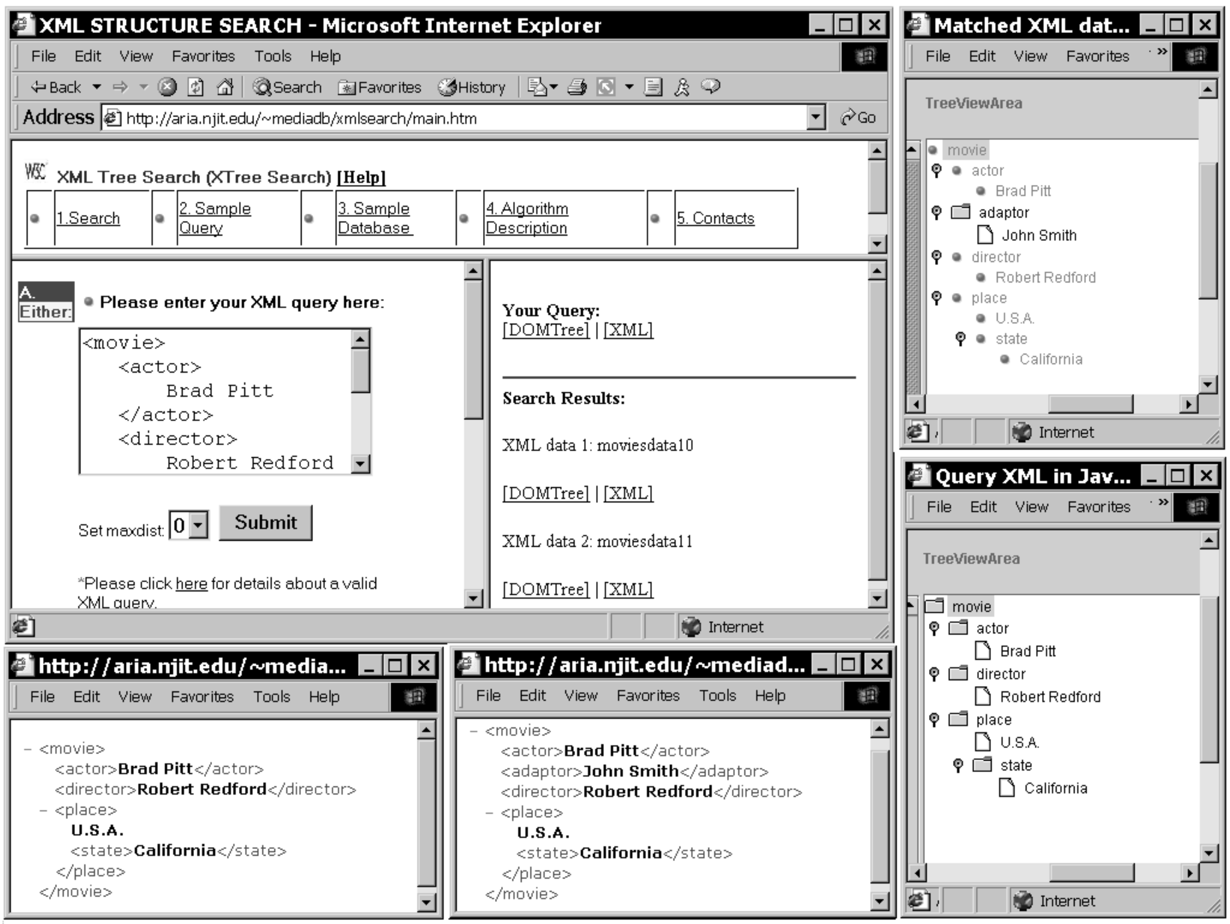


Figure 3: An example query and search results on a movie document database in XML QBE.

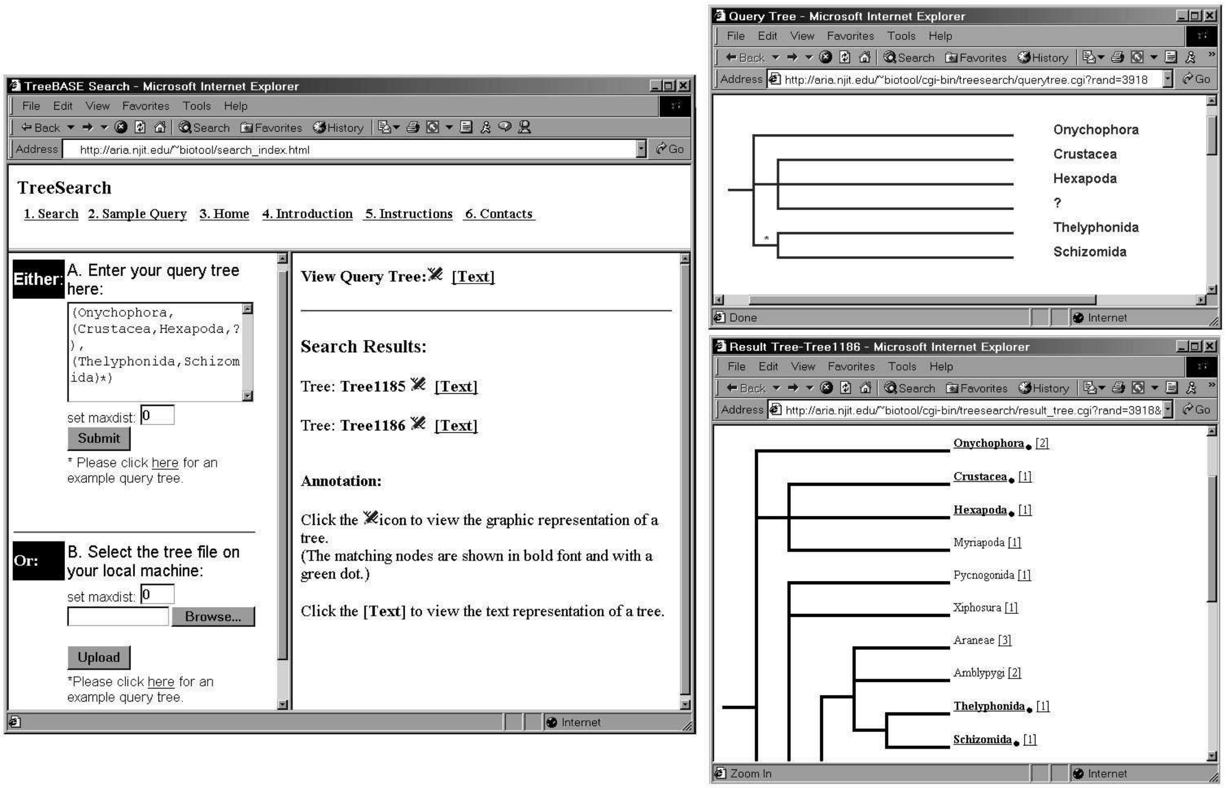


Figure 4: An example query and search results in the structure-based search engine for TreeBASE.

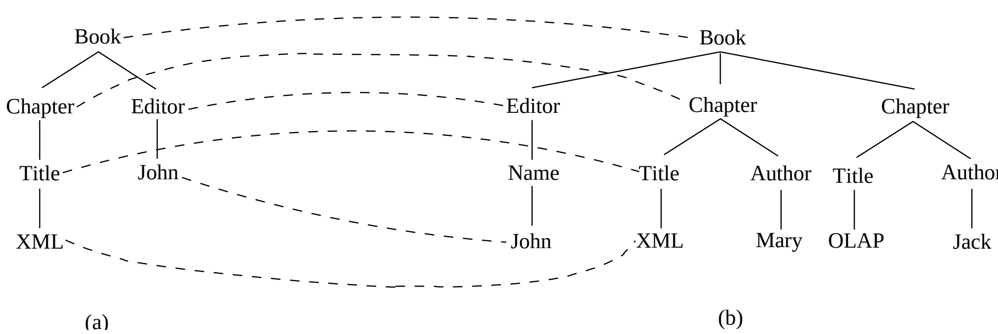


Figure 5: Unordered inclusion of a query tree in a data tree. (a) A query tree. (b) Part of a XML data tree.

complexity of the algorithm is exponential though the algorithm may run much faster depending on the data.

2.4.2 Selectivity Estimation

One technique for filtering trees out faster is to use selectivity estimation. In [69] McHugh and Widom describe Lorel's cost-based query optimizer, which maintains statistics about subpaths of length $\leq k$, and uses it to infer selectivity estimates of longer path queries. Krishnan *et al.* [61] described similar techniques for processing query strings containing wildcards, i.e. estimating the number of strings in a database that contain a given query string with wildcards. Other relevant work can be found in [26, 52, 54, 99].

Chen *et al.* [25] generalized the selectivity estimation problem for unordered trees. Specifically, given a data tree D and a wildcard-free query tree Q, which the authors called a twig, they estimate the total number of twig matches of Q in D. The authors represent frequency information about small twigs, called twiglets, in D using a correlated subpath tree (CST). Processing a query consists of the following steps: (1) parse each root-to-leaf path in Q into a set of subpaths that have matches in CST; (2) for each twig node that is a branch node, consider all subpaths obtained in (1) that are rooted at the same node and that pass through the branch node, and call each subtree induced by these subpaths a query twiglet; and (3) estimate the number of matches of Q by piecing together count estimates for the twiglets obtained in (2) based on an independence assumption about the occurrences of subpaths, and using probabilistic estimation formulae. For each path of length n, finding all its subpaths that have matches in CST takes $O(n^2)$ time. Thus the time complexity of the algorithm is $O(h^2 l)$ where h is the height of Q and l is the number of leaves of Q.

3. SEARCHING IN GRAPHS

A graph database can be viewed as either a single (large) labeled graph (e.g. www) or a collection of labeled graphs (e.g. chemical molecules). By *keygraph searching* we refer to graph or subgraph matching in data graphs. The complexity of the (sub)graph-to-graph matching problem and a review of certain algorithms with potential applications to keygraph searching in databases are discussed in Section 3.3. Although (sub)graph-to-graph matching algorithms can be used for keygraph searching, efficiency considerations suggest the use of indexing techniques to reduce the search space and the time complexity especially in large databases.

The keygraph search problem in a database consists of three basic steps just as for keytree searching.

1. *Reduce the search space by filtering.* For a database of graphs we find the most relevant graphs; for a single-graph database we identify the most relevant subgraphs. We confine ourself to filtering techniques based on the structure of the labeled graphs (paths, subgraphs). Since looking for subgraph structures is quite difficult, most algorithms choose to locate paths of node labels.

2. *Formulate query into simple structures.* The keygraph can be given directly as a set of nodes and edges or as the intersection of a set of paths. Furthermore the query can contain wildcards (representing nodes or paths) to allow for more general searches. This step normally reduces the query graph to a collection of small paths.

3. *Match.* Matching is implemented either by traditional (sub)graph-to-graph matching techniques, or by combining the set of paths that result from processing the path expressions in the query through the database.

Several systems for querying and indexing graph databases have been implemented—both general-purpose [30, 46] and application-specific [44, 55, 73]. The underlying techniques are described in the next section.

3.1 Keygraph Searching in Graph Databases

Cook *et al.* [30, 35] applied an improvement of the inexact graph matching method (algorithm A^*) described by Nilsson [79] based on an inexact graph matching algorithm proposed in [21] to find similar repetitive subgraphs in a single-graph database. Thus, their methods are primarily of interest for the third step above. Their system, SUBDUE, has been applied to discovery and search for subgraphs in protein databases, image databases, Chinese character databases, CAD circuit data and software source code. Furthermore an extension of SUBDUE (WebSUBDUE [68]) has been applied to hypertext data.

Guting [46] proposed a general purpose object-oriented data model and query language (GraphDB) for graph databases. Nodes in a graph are classes representing data (objects) and edges are classes linking two nodes. GraphDB contains classes to store several paths in the database. Path classes and indexing data structures (e.g. B-tree, LSD) are used to index nodes, paths and subgraphs in the graph database. Graph queries are specified using regular expressions and they may restrict the search space to a subgraph of the whole graph. GraphDB provides graph search operations to find the shortest paths between two nodes or to find subgraphs from a starting node within a distance range. The implementation is based on A^*.

Daylight [55] is a system used to retrieve substructures in databases of molecules where each molecule is represented by a graph. Daylight uses fingerprinting to find relevant graphs from a database (step 1). Each graph (of the database) is associated with a fixed-size bit vector— called the fingerprint of the graph. Given a graph G, its fingerprint bits are set in the following way: all the paths in G of length zero and up to a limit length are computed; each path is used as a seed to compute a random number; and the bit representation of this number is added to the fingerprint. The fingerprint represents structural features of the graph. The similarity of two graphs is computed by comparing their fingerprints. Some similarity measures are: the Tanamoto Coefficient (the number of bits in common divided by the total number); the Euclidean distance (geometric distance); and the Tversky similarity—used to measure the similarity of a query graph with a subgraph of a data graph. The search space is filtered by comparing the fingerprint of the query graph with the fingerprint of each graph in the database. Queries can include wildcards. For most queries, the matching is implemented using application specific techniques. However queries including wildcards may require exhaustive graph traversals.

Goldman, Widom [44] and colleagues [77] proposed a system, called Lore, to store and query a semistructured database (which is modeled as a large rooted labeled directed graph; see [1, 88, 92] for a survey). Lore uses four kinds of indices to accelerate (regular) path expression searching. For each edge label l in the graph, a value index (Vindex) is used to index all the nodes that have incoming edges labeled with l and with atomic values that satisfy some condition. A text index (Tindex) is used for all nodes with incoming l-labeled edges and with string atomic values containing specific words. A link index (Lindex) indexes the nodes with outgoing l-labeled edges. A path index (Pindex–DataGuide) indexes all the nodes reachable from the root through a labeled path. Each path query that starts at the root uses the DataGuide. All other path queries use the other three indexes in which case they find a set of candidates and then traverse the graph to prune away paths that do not match the query path. Because the other indexes are unselective, there are potentially many more candidates than matching paths.

Milo and Suciu [73] proposed a data structure, called T-index, to index semistructured database nodes that are reachable from several regular path expressions. A T-index is a non-deterministic automaton whose states represent (roughly) the equivalence classes produced by the Rabin-Scott algorithm and whose transitions correspond to edges between objects in those classes. By relaxing the determinism requirement imposed on DataGuides, a T-index can be constructed and stored more efficiently. They may represent a more efficient DataGuide in both time and space. For example the authors reported that in a graph of 1500 nodes, the T-index size is 13% of the size of the graph database.

GraphGrep, presented in the next section, is a new hash-based AC algorithm for finding *all* the occurrences of a query graph in a database of graphs. A set of intersecting regular path expressions is deduced by the query graph. GraphGrep uses variable length paths (that may contain cycles) to index the database: this allows efficient filtering by directly selecting the most relevant subgraphs of the most relevant graphs.

Table 1 summarizes the features of several keytree and keygraph searching techniques for tree and graph databases, respectively.

3.2 GraphGrep: A Variable Length Path Index Approach

For illustration purposes we focus on undirected graphs in which edges do not have labels. The techniques generalize to directed graphs with labeled edges. GraphGrep assumes that the nodes of the data graphs have an identification number (*id-node*) and a label (*label-node*). We define an *id-path* of length n to be a list of n id-nodes with an edge between any two consecutive nodes. A *label-path* of length n is a list of n label-nodes. For example, in Figure 6(a), (C,A) is a label path, and (3,1) is the id-path corresponding to it.

There are three basic components of GraphGrep: (1) build the index to represent the database of graphs as sets of paths (this step is done only once), (2) filter the database based on the submitted query and the index to reduce the search space, and (3) perform exact matching. We discuss these components in turn.

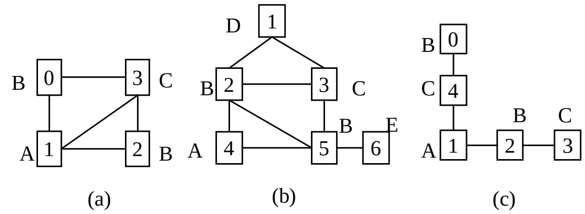


Figure 6: A database containing 3 graphs. (a) Graph g_1. (b) Graph g_2. (c) Graph g_3. The labels can be strings of arbitrary length.

3.2.1 Index Construction

For each graph and for each node, we find all paths that start at this node and have length one (single node) up to a (small, e.g. 4) constant value l_p (l_p nodes). We use the same l_p for all graphs in the database. Because several paths may contain the same label sequence, we group the id-paths associated with the same label-path in a set. The "path-representation" of a graph is the set of label-paths in the graph, where each label-path has a set of id-paths (see Figure 7(a)). The keys of the hash table are the hash values of the label paths. Each row contains the number of id-paths associated with a key (hash value) in each graph. We will refer to the hash table as the fingerprint of the database (see Figure 7(b)). Let $|D|$ be the number of graphs in a database D. Let n and m be the number of nodes and the maximum valence (degree) of the nodes in a data graph, respectively. The worst case complexity of building the index and the path representation for the database is $O(\sum_i^{|D|}(n_i m_i^{l_p}))$, whereas the memory cost is $O(\sum_i^{|D|}(l_p n_i m_i^{l_p}))$.

A={(1)} AB={(1, 0), (1,2)} AC ={(1, 3)} ACBA={…}
ABCA={(1 ,0 ,3 ,1),(1, 2, 3, 1)} CB={(3,0),(3,2)} C={(3)}
CBAB={(3, 0, 1 ,2),(3 ,2 ,1 ,0)} B={(0),(2)} BA={(0,1),(2,1)}
BAB={(0,1,2), (2,1,0)} ABC ={(1, 3, 0), (1,3,2)} ACB={…}
ABCB={…} BC={…} BAC={…} BCB={…} CBA={…}
BABC={…}CBAC={…} CABC={…} CAB={(3,1,0), (3,1,2)}
BACB={…}BCBA={…}BCAB={…} BCA={…} CA={(3,1)}

(a)

Key	g_1	g_2	g_3
h(CA)	1	0	1
……			
h(ABCB)	2	2	0

(b)

Figure 7: (a) The path representation of the graph in Figure 6(a) with $l_p = 4$. (b) The fingerprint of the database showing only part of rows.

3.2.2 Filtering the Database

The query graph is parsed to build its fingerprint (hashed set of paths). We filter the database by comparing the fingerprint of the query with the fingerprint of the database.

System	Application	Database Model	Path-Indexing		Filter-out		Wild-cards	Matching
			From	Length	Item	Sub-Item		
ATreeGrep	General	Many Trees	Every Leaf	Full	Yes	No	Yes	PC
XISS	XML	Many Trees	Every Node	1	Yes	Yes	Yes	PC
Daylight	Molecule	Many Graphs	Every Node	Parameter	Yes	No	Yes	AD
GraphDB	General	One Graph	Some Nodes	Parameter	Yes	Yes	Yes	A^*, PC-T
GraphGrep	General	Many Graphs	Every Node	Parameter	Yes	Yes	Yes	PC
Lore	Semistructured-Data	One Rooted-Graph	Root Every Node	Full 1	No	Yes	Yes	PC-T
SUBDUE	General	One Graph	No	No	No	No	No	A^*
T-index	Semistructured-Data	One Rooted-Graph	Root Some Nodes	Full Variable	No	Yes	Yes	PC-T

Table 1: Comparison of tree and graph searching systems. Filter-out is divided into Item (whether an entire tree or graph can be removed from consideration when matching a given query) and Sub-Item (whether relevant portions of selected trees and graphs are identified by the filtering steps). We use AD for an "Application Dependent" matching algorithm (e.g. tailored for molecules), PC for "Path Combination" (e.g. intersecting paths), and PC-T for a "Path Combination" matching algorithm which requires tree or graph "Traversal". Different systems have different expressive power using wildcards.

A graph, for which at least one value in its fingerprint is less than the corresponding value in the fingerprint of the query, is discarded when looking for an exact subgraph match. For example, in the query graph in Figure 8 with $l_p = 4$, the graphs (b) and (c) in Figure 6 are filtered out because they do not contain the label-path ABCA. Filtering the database takes linear time in the size of the database. The remaining graphs *may* contain one or more subgraphs matching the query.

3.2.3 *Finding Subgraphs Matching with Queries*

After filtering, we look for all the matching subgraphs in the remaining graphs. The branches of a depth-first traversal tree of the query are decomposed into sequences of overlapping label-paths, which we also call *patterns*, of length l_p or less (see Figure 8).

Overlaps may occur in the following cases:

1. For consecutive label-paths, the last node of a pattern coincides with the first node of the next pattern (e.g. ABCB, with $l_p = 3$, is decomposed into two patterns: ABC and CB).
2. If a node has branches it is included in the first pattern of every branch (see node C in Figure 8(c)).
3. The first node visited in a cycle appears twice: in the beginning of the first pattern of the cycle and at the end of the last pattern of the cycle (the first and last pattern can be identical, as in Figure 8(c)).

We use the path representation of the graphs to look for occurrences of the query. Only the parts of each (candidate) graph whose id-path sets correspond to the patterns of the query are selected and compared with the query. After the id-path sets are selected, we identify overlapping id-path lists and concatenate them (removing overlaps) to build a matching subgraph. For overlapping cases (1) and (2) a pair of lists is combined if the two lists contain the *same*

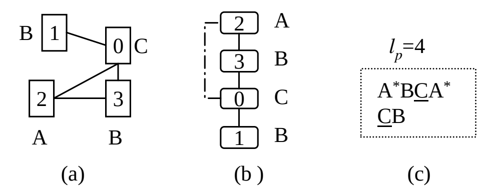


Figure 8: (a) A query graph. (b) The depth first tree of the graph in (a). (c) A set of patterns obtained with $l_p = 4$. In this example overlapping labels are marked with asterisks or underlining. Labels with the same mark represent the same node.

id-node in the overlapping position. In overlapping case (3), a list is removed if it does not contain the same id-node in the overlapping positions; finally, lists are removed if equal id-nodes are not found in overlapping positions.

Example. Let us consider the steps to match the query in Figure 8(a) with the graph g_1 in Figure 6(a).

1. Select the set of paths in g_1 (Figure 7(a)) matching the patterns of the query (Figure 8(c)): ABCA= $\{(1,0,3,1), (1,2,3,1)\}$, CB= $\{(3,0),(3,2)\}$.
2. Combine any list l_1 from ABCA with any list l_2 of CB if the third id-node in l_1 is equal to the first id-node of l_2 and the first id-node in l_1 is equal to the fourth id-node of l_1: ABCACB= $\{((1,0,3,1),\ (3,0)), ((1,0,3,1),(3,2)), ((1,2,3,1),(3,0)), ((1,2,3,1),(3,2))\}$.
3. Remove lists from ABCACB if they contain equal id-nodes in non-overlapping positions (the positions in each list not involved above). The two substructures in g_1 whose composition yields ABCACB are $((1,0,3,1), (3,2))$ and $((1,2,3,1),(3,0))$.

The matching algorithm depends on the number of query graph patterns p that need to be combined; p is somewhat difficult to determine for the average case. Roughly speaking, it is directly proportional to the query size and to the maximum valence of the nodes in the query. The larger l_p, the smaller p, though this relationship is data-dependent. In general if $\tilde{n}$ is the maximum number of nodes having the same label, the worst case time complexity for the matching is $O(\sum_i^{|D_f|}((\tilde{n}_i m_i^{l_p})^p))$ with $|D_f|$ being the size of the database after the filtering.

3.2.4 Techniques for Queries with Wildcards

Query graphs with wildcards are handled by considering the parts of the query graph between wildcards as disconnected components, just as we do for pathfix. For example, the disconnected components of the graph in Figure 9 are the path ABC and the single node D.

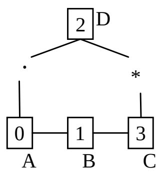


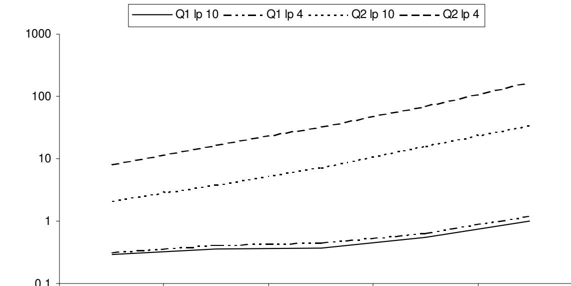
Figure 9: The query graph matches a graph with these properties: (1) a path between a C-labeled node and a D-labeled node may exist; (2) there is a two-edge path between an A-labeled and the D node; (3) there is an edge between the A node and a B-labeled node; and (4) there is an edge between the B and C nodes.

The matching algorithm described above is done for each component. The Cartesian product of the sets that match each component constitute the candidate matches. An entry in the Cartesian product is a valid match if there is a path (of length equal to the wildcards' values in the query) between nodes that are connected with wildcards. The paths in the candidate graph are checked using a depth first search traversal of the graph. This step may be optimized by maintaining the transitive closure matrices of the database graphs and searching in a candidate graph only if the wildcard's value is greater than or equal to the shortest path between the nodes.

3.2.5 Experimental Results

To evaluate the performance of GraphGrep we conducted numerical experiments on NCI [76] databases containing up to 16,000 molecules. We used a Linux workstation equipped with a 1GHz pentium III processor. The NCI database graphs have an average number of 20 nodes; several graphs have up to 270 nodes. We report wall-clock querying time for varied query sizes (13, to 189 nodes), database sizes (1,000 - 16,000 graphs), and l_p values (4, and 10) (see Figure 10).

Different values of l_p influence the query running time: for the $Q2$ query, the matching algorithm performs better when $l_p = 10$ compared with $l_p = 4$, which is consistent with the time complexity analysis. In addition, in these examples we verify that the querying time is linear in the size of the database, and exponential in $p \times l_p$. Recall that p (the number of paths within size l_p that have to be tested)



	1000	2000	4000	8000	16000
Q1 lp 10	0.29	0.35	0.37	0.56	1.02
Q1 lp 4	0.32	0.41	0.45	0.64	1.2
Q2 lp 10	2.12	3.91	7.21	15.93	33.6
Q2 lp 4	8.21	16.78	33.48	70	167.1

Figure 10: The horizontal axis gives the size of the database and the vertical axis the wall-clock time measured in seconds (in logarithmic scale). Q1 is a molecule with 13 nodes and 14 undirected edges. Q2 is a molecule with 189 nodes and 210 undirected edges. For the queries Q1 and Q2, 99% of the database is discarded during filtering for both values of l_p. For the 16,000 molecules database, 640 subgraphs are found for Q1 and 612 for Q2.

is proportional to the query size. As expected, p decreases substantially with larger l_p, but not always.

3.3 Subgraph Matching

In [105] Yannakakis surveyed traditional graph searching problems with applications to data management, including computing transitive closures, recursive queries, and the complexity of path searching in databases. We review here some classical subgraph matching techniques, mostly having to do with step 3 of our query processing framework.

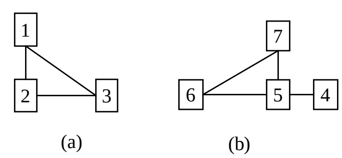


Figure 11: (a) A query graph G_a. (b) A data graph G_b.

A simple theoretical enumeration algorithm to find the occurrences of a query graph G_a in a data graph G_b (Figure 11), is to generate all possible maps between the nodes of the two graphs and to check whether each generated map is a match. All the maps can be represented using a *state-space representation* tree: a node represents a pair of matched vertices; a path from the root down to a leaf represents a map between the two graphs. A path from a node at the k^{th} level in the tree up to the root represents a *partial matching* between the graphs; only a subset (k) of vertices have been matched. Only certain leaves correspond to a subisomorphism between G_a and G_b (Figure 12). The complexity of such an algorithm is exponential, but it is the best known algorithm—the problem of subgraph isomorphism is proven to be NP-complete [42].

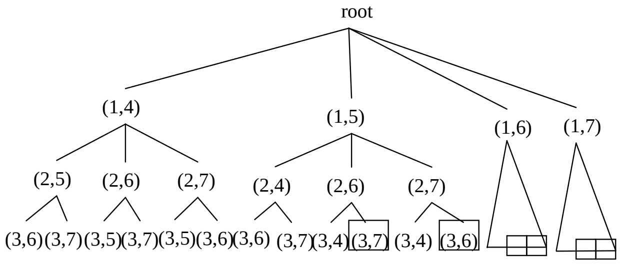


Figure 12: All the maps between G_a and G_b. The leaves in the rectangular frames correspond to subisomorphisms between G_a and G_b.

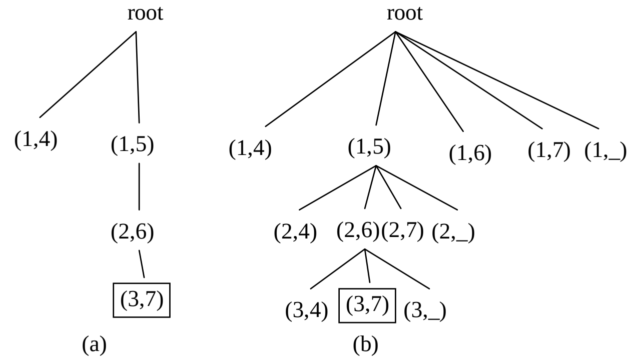


Figure 13: The tree search-space to reach the first isomorphism. (a) The tree-search space pruned by Ullmann's algorithm. (b) The tree-search space pruned by Nilsson's algorithm. The match (1,_) represents a tree-search node deletion. Here we assume that an underlying evaluation function is used to guide the state expansions. Different evaluation functions prune the tree-search differently.

There have been many attempts to reduce the combinatorial cost of AC query processing in graphs or keygraph searching. They can be classified as approximate, inexact, and exact algorithms. Approximate algorithms [6, 27, 37, 43, 91, 101] have polynomial complexity but they are not guaranteed to find a correct solution. Exact and inexact algorithms do find correct answers and therefore have exponential worst-case complexity [14, 48, 50, 63, 74, 79, 82, 102]. Inexact algorithms employ error correction techniques for a noisy data graph. These algorithms employ a cost function to measure the similarity of the graphs. For example, a cost function may be defined based on semantic or syntactic transformations to transform one graph into another. (Of course, approximate algorithms can also be used for noisy data graphs.) Relevant work can be found in [20, 21, 31, 36, 39, 65, 75, 89, 97, 98]. The most popular exact (and inexact) subgraph matching algorithms are based on heuristics on the state-space representation tree that corresponds to a subisomorphism.

Ullmann's Algorithm. Ullmann [90] presented an algorithm for an *exact* subgraph matching based on the *state space search with backtracking* algorithm in [32]. A depth-first search on the state space tree representation depicts the algorithm's progress. When a node (a pair of matching vertices) is added to the tree, the isomorphism conditions are checked in the partial matching. If the isomorphism condition is not satisfied the algorithm *backtracks* (i.e. the tree-search that would correspond to a full enumeration is pruned). Upon termination only the paths with length equal to the number of nodes in G_a (corresponding to unpruned leaves) represent a subisomorphism.

The performance of the above state-space representation algorithm is improved by a refinement procedure called *forward checking*: in order to insert a node in the tree not only must the subisomorphism conditions hold, but, in addition, a possible mapping must exist for *all* the unmatched vertices. As a result, the algorithm prunes the tree-search more efficiently at a higher level (see Figure 13(a)).

Nilsson's Algorithm ($\mathbf{A^*}$). Nilsson [79] presented an *inexact* subgraph matching algorithm. This time, a breath-first search on the state-space representation tree depicts the algorithm's progress. Each node in the tree-search represents a vertex in G_a that has been either matched with a vertex in G_b or deleted. If a vertex in G_a has to be deleted, it is matched to a null vertex in G_b. A cost is assigned to the matching between two vertices. The cost of a partial matching is the sum of the costs of the matched vertices. A function evaluates the partial matching by summing its cost to a lower bound estimation of the cost to match the remaining vertices in G_a. The tree search is expanded to states for which the evaluation function attains the minimum value (among all possible expansion states). The leaves of the tree (that have not been pruned) represent final states, i.e, states where all the vertices of G_a have been matched (see Figure 13(b)).

4. CONCLUSIONS AND FUTURE WORK

We have focused primarily on pattern-matching based algorithms for fast searching in trees and graphs. These algorithms could be used for direct support of queries on the data types, or could be used as a preprocessor for join-like algorithms [5]. Future work in this field includes:

- Improve the performance of existing keytree searching algorithms (e.g. ATreeGrep) and keygraph searching algorithms (e.g. GraphGrep) so that they can be as fast as keyword searching engines like google. Many of these algorithms are embarrassingly parallelizable so will scale well.
- Develop indexes that trade time for space optimally (storing all paths may be more than is needed, but storing just parent-child pairs may be too little).
- Develop practically meaningful distance measures on trees and graphs and approximate query processing algorithms to support inexact matching.
- Develop a framework for selectivity estimation for queries on trees and graphs with wildcards.
- Develop a framework for turning searching to pattern discovery in trees and graphs [33, 94, 95, 100].

- Develop support for semantic extensions: semi-flexible or flexible queries [56] in which parent-child relationships in queries may become ancestor-descendant or even descendant-ancestor relationships in data graphs.

5. ACKNOWLEDGMENTS

We thank Ken Abe, Greg Heil, Katherine Herbert, Huiyuan Shan and Sen Zhang for their contributions in the various phases of the ATreeGrep project.

6. REFERENCES

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, 1997.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[4] S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences*, 58(3):428–452, 1999.

[5] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the IEEE International Conference on Data Engineering*, 2002.

[6] H. Almohamad and S. O. Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5), 1993.

[7] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Data Bases*, pages 53–64, 2000.

[8] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 497–508, 2001.

[9] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *Proceedings of the International Conference on Extending Database Technology*, 2002.

[10] R. Baeza-Yates. Algorithms for string searching: A survey. *SIGIR Forum*, 23(3-4):34–58, 1989.

[11] R. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[12] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[13] D. Barbosa, A. Barta, A. O. Mendelzon, G. A. Mihaila, F. Rizzolo, and P. Rodriguez-Guianolli. ToX - The Toronto XML engine. In *Proceedings of the Workshop on Information Integration on the Web*, pages 66–73, 2001.

[14] H. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4:83–84, 1976.

[15] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language; available at http://www.w3.org/TR/xquery/.

[16] P. Buneman. Semistructured data. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 117–121, 1997.

[17] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[18] P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured and structured databases. In *Proceedings of the ACM SIGMOD SIGACT SIGART Symposium on Principles of Database Systems*, pages 129–138, 1998.

[19] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[20] H. Bunke. Error correcting graph matching: On the influence of the underlying cost function. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9):917–922, 1999.

[21] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[22] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 4–13, 1998.

[23] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, 1997.

[24] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[25] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 595–604, 2001.

[26] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 216–225, 2000.

[27] W. J. Christmas, J. Kittler, and M. Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):749–764, 1995.

[28] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 245–254, 1999.

[29] M. P. Consens and A. O. Mendelzon, GraphLog: A visual formalism for real life recursion. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416, 1990.

[30] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[31] L. Cordella, P. Foggia, C. Sansone, and M. Vento. An efficient algorithm for the inexact matching of arg graphs using a contextual transformational model. In *Proceedings of the International Conference on Pattern Recognition*, volume 3, pages 180–184, 1996.

[32] D. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51–64, 1970.

[33] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36, 1998.

[34] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for XML. *Computer Networks*, 31(11-16):1155–1169, 1999.

[35] S. Djoko, D. J. Cook, and L. B. Holder. An empirical study of domain knowledge and its benefits to substructure discovery. *IEEE Transactions on Knowledge and Data Engineering*, 9(4), 1997.

[36] M. A. Eshera and K.-S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 14(3):353–363, 1984.

[37] J. Feng, M. Laumy, and M. Dhome. Inexact matching using neural networks. In E. Gelsema and L. N. Kanal, editors, *Pattern Recognition in Practice IV: Multiple Paradigms, Comparative Studies and Hybrid Systems*. North–Holland, 1994.

[38] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a Web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 414–425, 1998.

[39] M.-L. Fernandez and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 753–758, 2001.

[40] A. Ferro, G. Gallo, R. Giugno, and A. Pulvirenti. Best-match retrieval for structured images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(7):707–718, 2001.

[41] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, 1992.

[42] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company, 1979.

[43] S. Gold and A. Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(4):377–388, 1996.

[44] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 436–445, 1997.

[45] G. H. Gonnet and F. W. Tompa. Mind your grammar: A new approach to modeling text. In *Proceedings of the International Conference on Very Large Data Bases*, pages 339–346, 1987.

[46] R. H. Guting. GraphDB: Modeling and querying graphs in databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 297–308, 1994.

[47] M. Gyssens, J. Paredaens, and D. V. Gucht. A graph-oriented object database model. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 417–424, 1990.

[48] T. C. Henderson. *Discrete Relaxation Techniques.* Oxford University Press, 1990.

[49] C. M. Hoffman and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[50] R. Horaud and T. Skordas. Stereo correspondence through feature grouping and maximal cliques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1168–1180, 1989.

[51] J. Jaakkola and P. Kilpelainen. Using sgrep for querying structured text files. University of Helsinki, Department of Computer Science, Report C-1996-83, November 1996; available at http://www.cs.helsinki.fi/u/jjaakkol/sgrep.html.

[52] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multi-Dimensional substring selectivity estimation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 387–398, 1999.

[53] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–144, 1999.

[54] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 249–260, 1999.

[55] C. A. James, D. Weininger, and J. Delany. *Daylight Theory Manual-Daylight 4.71.* Daylight Chemical Information Systems, www.daylight.com, 2000.

[56] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2001.

[57] P. Kilpelainen. Tree matching problems with applications to structured text databases. Report A-1992-6, University of Helsinki, Finland, 1992.

[58] P. Kilpelainen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, 1993.

[59] P. Kilpelainen and H. Mannila. Query primitives for tree-structured data. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, pages 213–225, 1994.

[60] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356, 1995.

[61] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 282–293, 1996.

[62] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik. The AQUA data model and algebra. In *Proceedings of the 4th Workshop on Database Programming Languages*, pages 157–175, 1993.

[63] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcols*, 9:341–354, 1972.

[64] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 361–370, 2001.

[65] J. Llados, E. Marti, and J. Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1137–1143, 2001.

[66] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[67] S. Maneth and F. Neven. Structured document transformations based on XSL. In *Proceedings of the 7th Workshop on Database Programming Languages*, pages 80–98, 1999.

[68] N. Manocha, D. J. Cook, and L. B. Holder. Structural Web search using a graph-based discovery system. In *Proceedings of the Florida Artificial Intelligence Research Symposium*, 2001.

[69] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the International Conference on Very Large Data Bases*, pages 315–326, 1999.

[70] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[71] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[72] G. Miklau and D. Suciu. Containment and equivalence of XPath expressions. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2002.

[73] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, pages 277–295, 1999.

[74] S. H. Myaeng and A. Lopez-Lopez. Conceptual graph matching: A flexible algorithm and experiments. *Journal of Experimental and Theoretical Artificial Intelligence*, 4:107–126, 1992.

[75] R. Myers, R. Wilson, and E. R. Hancock. Bayesian graph edit distance. In *Proceedings of the 10th International Conference on Image Analysis and Processing*, 1998.

[76] National Cancer Institute. http://www.nci.nih.gov/.

[77] S. Nestorov, J. D. Ullman, J. L. Wiener, and S. S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the International Conference on Data Engineering*, pages 79–90, 1997.

[78] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 145–156, 2000.

[79] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, California, 1980.

[80] W. H. Piel, M. J. Donoghue, and M. J. Sanderson. TreeBASE: A database of phylogenetic information. In *Proceedings of the 2nd International Workshop of Species 2000*, Tsukuba, Japan, 2000.

[81] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the ACM*, 39(2):295–316, 1992.

[82] A. Sanfeliu and K. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):353–362, 1983.

[83] T. Schlieder and F. Naumann, Approximate tree embedding for querying XML data. In *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval*, Athens, Greece, July 2000.

[84] D. Shasha, J. T. L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate searching in unordered trees. Submitted, 2002.

[85] H. Su, H. Kuno, and E. Rundensteiner. Automating the transformation of XML documents. In *Proceedings of the Workshop on Web Information and Data Management*, 2001.

[86] B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik. The AQUA approach to querying lists and trees in object-oriented databases. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 80–89, 1995.

[87] B. Subramanian, S. B. Zdonik, T. W. Leung, and S. L. Vandenberg. Ordered types in the AQUA data model. In *Proceedings of the 4th Workshop on Database Programming Languages*, pages 115–135, 1993.

[88] D. Suciu. An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29, 1998.

[89] W. H. Tsai and K. S. Fu. Error-correcting isomorphism of attributed relational graphs for pattern analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:757–768, 1979.

[90] J. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42, 1976.

[91] S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):695–703, 1988.

[92] V. Vianu. A web odyssey: From Codd to XML. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2001.

[93] J. T. L. Wang, K. Jeong, K. Zhang, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, 1994.

[94] J. T. L. Wang, B. A. Shapiro, and D. Shasha (eds). *Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications*. Oxford University Press, New York, 1999.

[95] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and C.-Y. Chang. Automated discovery of active motifs in multiple RNA secondary structures. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 70–75, 1996.

[96] J. T. L. Wang, D. Shasha, G. J.-S. Chang, L. Relihan, K. Zhang, and G. Patel. Structural matching and discovery in document databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 560–563, 1997.

[97] J. T. L. Wang, K. Zhang, G. Chang, and D. Shasha. Finding approximate patterns in undirected acyclic graphs. *Pattern Recognition*, 35(2):473–483, 2002.

[98] J. T. L. Wang, K. Zhang, and G.-W. Chirn. The approximate graph matching problem. In *Proceedings of the International Conference on Pattern Recognition*, volume 2, pages 284–288, 1994.

[99] M. Wang, J. S. Vitter, and B. R. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 169–180, 1997.

[100] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, S. Dikshitulu, I. Rigoutsos, and K. Zhang. Automated discovery of active motifs in three dimensional molecules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 89–95, 1997.

[101] R. Wilson and E. Hancock. Bayesian compatibility model for graph matching. *Pattern Recognition Letters*, 17:263–276, 1996.

[102] A. Wong and M. You. Entropy and distance of random graphs with application to structural pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):599–609, 1985.

[103] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[104] XML TreeDiff. http://www.alphaworks.ibm.com/aw.nsf/FAQs/xmltreediff.

[105] M. Yannakakis. Graph theoretic methods in database theory. In *Proceedings of the ACM SIGMOD SIGACT SIGART Symposium on Principles of Database Systems*, pages 230–242, 1990.

[106] K. Zhang, D. Shasha, and J. T. L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, 16(1):33–66, 1994.

[107] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.