# ATreeGrep: Approximate Searching in Unordered Trees

Dennis Shasha*
Jason T. L. Wang†
Huiyuan Shan‡
Kaizhong Zhang§

**Abstract**

An unordered labeled tree is a tree in which each node has a string label and the parent-child relationship is significant, but the order among siblings is unimportant. This paper presents an approach to the nearest neighbor search problem for these trees. Given a database $\mathcal{D}$ of unordered labeled trees and a query tree $Q$, the goal is to find those trees in $\mathcal{D}$ that "approximately" contain $Q$. Our approach is based on storing the paths of the trees in a suffix array and then counting the number of mismatching paths between the query tree and a data tree. To speed up a search, we use a hash-based technique to filter out unqualified data trees at an early stage of the search. Experimental results obtained by running our techniques on phylogenetic trees and synthetic data demonstrate the good performance of the proposed approach. We also discuss the use of our work in XML and scientific database management.

**Keywords:** Bioinformatics, hashing, phylogenetic trees, query processing, structure search.

*Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA (shasha@cs.nyu.edu).

†To whom correspondence should be addressed: College of Computing Sciences, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA. Tel: (973) 596-3396, Fax: (973) 596-5777, Email: wangj@oak.njit.edu.

‡Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA (huiyuan@cis.njit.edu).

§Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7 (kzhang@csd.uwo.ca).

# 1 Introduction

Unordered labeled trees represent data in many scientific and commercial disciplines.[1] For example, scientists model phylogenetic relations as unordered labeled trees and develop methods for constructing these trees [3, 5, 15]. A recent workshop report from Yale suggested that more research be undertaken to improve heuristic search strategies using algorithms designed to meet the demands made by increasingly large tree datasets [11]. Recent efforts in Web computing model an XML document as a tree, offering further motivation for the need for efficient tree searching [4, 6]. Other potential applications include information retrieval in linguistic, taxonomic, and neuroanatomical databases, among others.

Many algorithms have been developed for tree searching and matching [1, 12, 18, 24]. Most of these algorithms focus on comparing two trees based on various distance metrics. Chawathe *et al.* [7, 8, 9] studied the tree matching problem in the context of change detection for structured and semistructured data. There are also efforts spent in the development of query languages [2, 14, 16, 19, 20] and query processing techniques [10] for trees, with applications to XML and object-oriented database management.

In this paper we propose a new approach for approximate search among unordered labeled trees. Our problem, denoted the *approximate nearest neighbor search* (ANN) problem for unordered labeled trees, is the following. Given an integer $DIFF$, a query tree $Q$ and a database $\mathcal{D}$ of trees, the ANN problem is to find all the data trees $D$ in $\mathcal{D}$ where $D$ approximately contains $Q$ within distance $DIFF$. That is, $D$ contains a substructure $D'$ and the distance from $Q$ to $D'$ is at most $DIFF$. We proved that this problem was NP complete [27] for edit distance. So we adopt a different method. We measure the distance from $Q$ to $D'$ by the total number of root-to-leaf paths in $Q$ that do not appear in $D'$; the nodes in $D'$ that do not appear in $Q$ can be freely removed. To our knowledge, no previous work has addressed the ANN problem for unordered labeled trees.

To illustrate the distance measure, consider the query tree $Q$ in Figure 1, which has

[1]In this paper we will refer to unordered labeled trees simply as trees when the context is clear.

two paths. The query tree will match data tree $D_1$ with distance of 0, and match data tree $D_2$ with distance of 1. This happens because every path in $Q$ matches a path of the substructure $D_1'$ in $D_1$. ($D_1'$ is enclosed by the dashed line in $D_1$.) On the other hand, there is one path '$a - e$' in $Q$ that cannot be found in $D_2$. Counting mismatching paths is important in, for example, inferring evolutionary history, since that shows a deviation of ancestor-descendant relations. It is also a natural extension of path expressions in XML queries.

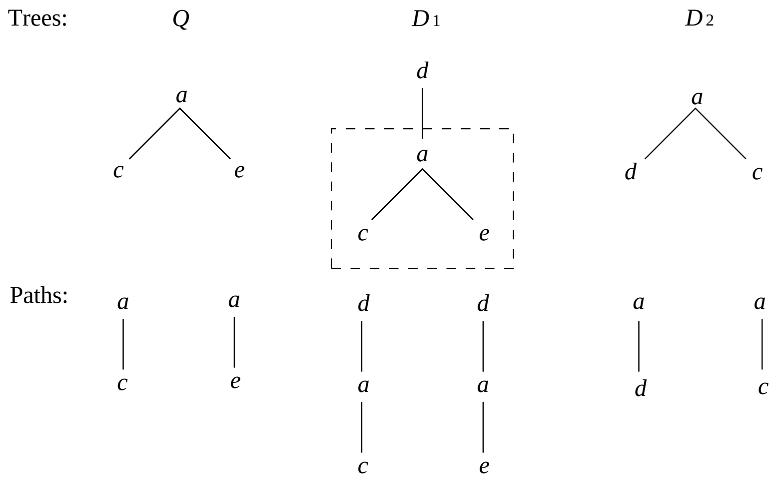


Figure 1: Example trees.

In practice, it's likely that some portion of a query tree is unknown, uninteresting or unimportant. That portion is often represented by a don't care symbol. In general, there are two types of don't care symbols: variable length don't cares (VLDCs) [22, 26] and fixed length don't cares (FLDCs). In string matching, a VLDC, denoted "$*$", in the query string may substitute for zero or more characters in a data string. For example, if "com$*$er" is the query string, then the "$*$" would substitute for the substring "put" when matching with the data string "computer". On the other hand, a FLDC, denoted "?", in the query string substitutes for exactly a single character in a data string. For example, if "com?uter" is the query string, then the "?" would substitute for the character "p" in the match with the data string "computer".

In this paper we discuss generalizations of don't cares to trees and present algorithms for processing them. In these cases, the labels on nodes can be "$*$" or "?".

The rest of the paper is organized as follows. Section 2 describes basic algorithms for processing query trees without don't cares. Section 3 presents algorithms for searching with don't cares. Section 4 describes a filtering technique for speeding up searching. Section 5 presents some experimental results. Section 6 discusses applications of our work. Section 7 concludes the paper.

# 2 Basic Algorithms

Our algorithm, called pathfix, consists of two phases. In the first phase, we build a suffix array database $\mathcal{SD}$ for all the trees in our database $\mathcal{D}$. $\mathcal{D}$ contains strings where each string corresponds to a root-to-leaf path in a data tree. We encode the paths (strings) into a suffix array [17]. In the second phase, which is the on-line search phase, we compare the root-to-leaf paths of the query tree $Q$ with the paths in the suffix array database $\mathcal{SD}$ to locate those substructures approximately matching the query tree. In a later enhancement, we construct a filter to determine which data trees in $\mathcal{D}$ are possible matches.

## 2.1 Database Building Phase

The suffix array is a data structure designed for efficient searching in a large string [17]. This data structure is simply an array containing the pointers to all the string's suffixes sorted in lexicographical order. (A suffix is a substring starting at a certain position in the string and ending at the end of the string.) Searching for a query string can be performed by binary search using the suffix array.

In pathfix, we construct a suffix array for all the paths in a data tree and put it in a global set of suffix arrays for all the data trees. This global set is the database $\mathcal{SD}$. Figure 2 shows the algorithm.

As an example, consider again the data tree $D_1$ in Figure 1. $D_1$ has paths '$d-a-c$' and '$d-a-e$'. We can create a suffix array $SA_1$ for the two paths, separated by a delimiter #, as shown in Figure 3. In this figure, the parenthesized integer in front of each suffix

```
Procedure Build_Suffix_Array
Input: the database 𝒟 of trees.
Output: the suffix array database 𝒮𝒟.

for every tree D in the database 𝒟
    find all the root-to-leaf paths in D;
    concatenate those paths with a delimiter to form a long string S;
    form a suffix array for S and add it to the global set of suffix arrays;
end for;
return the global set of suffix arrays, which is the database 𝒮𝒟;
```

Figure 2: Procedure for building the suffix array database $\mathcal{SD}$.

indicates the position at which the suffix begins in the paths set. This integer, when stored in the suffix array, serves as a pointer to the corresponding suffix in the paths set. Likewise we can create a suffix array $SA_2$ for the paths of the data tree $D_2$ in Figure 1. The suffix array database $\mathcal{SD}$ contains $SA_1$ and $SA_2$.

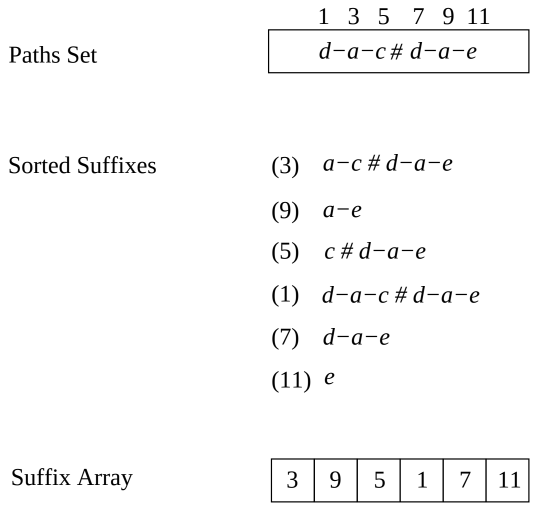


Figure 3: A suffix array.

**Lemma 1.** *Suppose each data tree has at most $N$ nodes, and there are $M$ trees in the database $\mathcal{D}$. The space complexity of the procedure* Build_Suffix_Array *in Figure 2 is $O(MN^2)$. The time complexity is also $O(MN^2)$.*

**Proof.** Each data tree has at most $O(N)$ paths if the tree is very bushy. No path can be longer than $O(N)$. We need at most $N^2$ pointers to the suffixes of the paths. Thus each data tree requires $O(N^2)$ space. There are totally $M$ trees, so the space complexity is $O(MN^2)$. The expected time spent in constructing a suffix array for a string is linearly proportional to the string size [17], so the database $\mathcal{SD}$ can be built in $O(MN^2)$ expected time. $\Box$

In practice, a tree with $N$ nodes either has few paths or short depth, so the above upper bound is very pessimistic. In practice the complexity is linear, i.e. $O(MN)$.

There are two alternative implementations of the suffix arrays. One suffix array can be constructed for all the trees or one suffix array can be constructed for each tree (as in our current implementation). Choosing one or the other depends on the effectiveness of filtering. If filtering yields very few candidate trees, then our current implementation works well. If there are many similar trees in the database, then a single suffix array for all trees is better.

## 2.2 On-Line Search Phase

In the on-line search phase, the query tree $Q$ is compared to each data tree allowing a difference $DIFF$. When comparing $Q$ with a data tree $D$, we take every root-to-leaf path $p$ in $Q$ and find roots of that path in $D$. (As a cutoff optimization, we stop searching $D$ if more than $DIFF$ paths of $Q$ are not found in $D$.) Suppose there are $k$ root-to-leaf paths in $Q$. If a node $n$ in $D$ is the root of the $k$ paths, then the subtree $D'$ rooted at $n$ matches $Q$ with distance 0, provided there are no siblings having the same label. (If there are siblings having the same label, then post-processing can verify the match. Our 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. Figure 4 shows the algorithm. Note that this algorithm can easily be modified to print out the subtree $D'$ rooted at $n$ that matches $Q$. We will refer to this modified algorithm as Basic_Substructure_Search.

**Procedure** Basic_Search
**Input:** the allowed distance threshold $DIFF$, the query tree $Q$, the database $\mathcal{D}$ of trees and the suffix array database $\mathcal{SD}$.
**Output:** the set $\mathcal{R}$ of data trees that approximately contain $Q$ within distance $DIFF$.

1. $\mathcal{R} := \emptyset$;
2. compute all the root-to-leaf paths of the query tree $Q$;
3. let $k$ be the number of paths of $Q$;
4. **for** each data tree $D$ in $\mathcal{D}$ (after filtering);
   /* Suffix array search portion */
5.     **for** each path $p$ in $Q$ from longest to shortest
6.         find the root set $N_p$ in $D$ such that for each $n$ in $N_p$
           there is a node $n'$ and the path from $n'$ to $n$ (ascending in $D$) is $p$;
7.         exit the for loop if the root sets for more than $DIFF$ paths are empty;
8.     **end for**;
   /* Intersection portion */
9.     **for** each $n$ in $N_p$
10.         count the total number of occurrences $T(n)$ of $n$ in all root sets $N_p$'s
            for all paths $p$ in $Q$;
11.         **if** $T(n) \geq k - DIFF$ **then**
12.             $\mathcal{R} := \mathcal{R} \cup \{D\}$;
13.     **end for**;
14. **end for**;

Figure 4: Procedure for finding data trees approximately containing the query tree $Q$.

**Lemma 2.** *Let $q$ be the number of nodes in the query tree $Q$. Suppose that the size of a suffix array is $S$. The time complexity of the suffix array search portion of procedure* Basic_Search *in Figure 4 for such a suffix array is $O(q^2 \log S)$.*

**Proof.** We note that $q$ is an upper bound on the number of paths in $Q$; $q$ is also an upper bound on the lengths of the paths in $Q$. Searching for a path of length $q$ takes time $O(q \log S)$, and hence the result follows. □

The time spent to count the number of occurrences of the entire tree depends on the number of matches of the paths. In the worst case, this can be nearly every node, but in practice is much smaller.

# 3 Query Trees with Don't Cares

When generalizing don't cares to trees, the semantics of the don't cares is as follows. The VLDC "$*$" in the query tree may substitute for a path of length zero or more in a data tree. The FLDC "?" in the query tree may substitute for a single node in the data tree. Figure 5 shows the algorithm for finding data trees exactly containing the query tree $Q$ where the root of $Q$ is not a don't care.

**Procedure** Advanced_Search
**Input:** the query tree $Q$ with don't cares, the database $\mathcal{D}$ of trees and the suffix array database $\mathcal{SD}$.
**Output:** the set of data trees containing $Q$ and for each such data tree $D$,
the substructure $D'$ in $D$ that matches $Q$.

1. partition $Q$ into connected subtrees having no don't cares;
2. match each of those subtrees with data trees in $\mathcal{D}$
   by invoking procedure Basic_Substructure_Search;
3. For the matched substructures that belong to the same data tree, say $D$, determine whether they combine, forming $D'$, to match $Q$ based on the matching semantics of the don't cares explained in the text, and if so, return $D$ and $D'$;

Figure 5: Procedure for finding data trees containing the query tree $Q$ with 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 "$*$" and "?") the path from $r_{sub}$ to $r_{all}$.

To avoid testing the roots of subtrees unnecessarily, the matching makes use of facts 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 $x$.

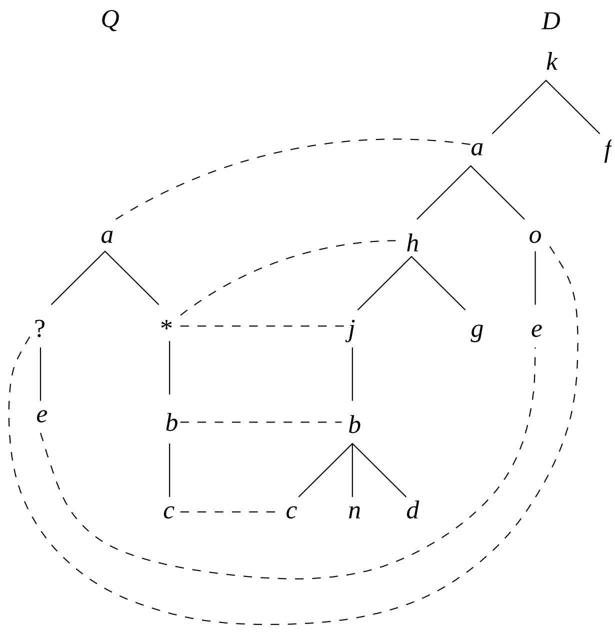


Figure 6: Illustration of matching a query tree having don't cares with a data tree.

Don't cares add to the time because each partition is much more likely to match than the whole tree, so there are many possible combinations to test. The basic time used for checking the suffix arrays, however, is less, since each tree is broken up into smaller trees.

Figure 6 illustrates how to match a query tree $Q$ having don't cares with a data tree $D$. We first partition $Q$ into three subtrees at its don't cares "$*$" and "?". The subtrees of $Q$ match three subtrees in $D$, which are then glued. In the figure, nodes in $D$ that are not touched by a dashed line are to be removed at no cost. The don't care "?" is instantiated into node $o$ in $D$ and the don't care "$*$" is instantiated into nodes $h$, $j$ in $D$ at no cost. The distance between $Q$ and $D$ is 0.

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 $Q$, we find, by invoking procedure Basic_Substructure_Search, all subtrees of data trees that are within distance $DIFF$ of $Q'$. The gluing process involves a testing of whether the glued tree as a whole is indeed within distance $DIFF$ of the entire query tree $Q$.

# 4 Filtering

To speed up the search process, we filter trees in a straightforward way: compute all parent-child pairs in each data tree and establish a count of those pairs. Consider, for example, the data tree $D$ in Figure 7(a). $D$ has the pairs $(a, b)$, $(b, c)$, $(a, a)$, $(a, b)$, $(a, c)$. So, it has $(a, b)$ twice and all the rest once. Store these counts.

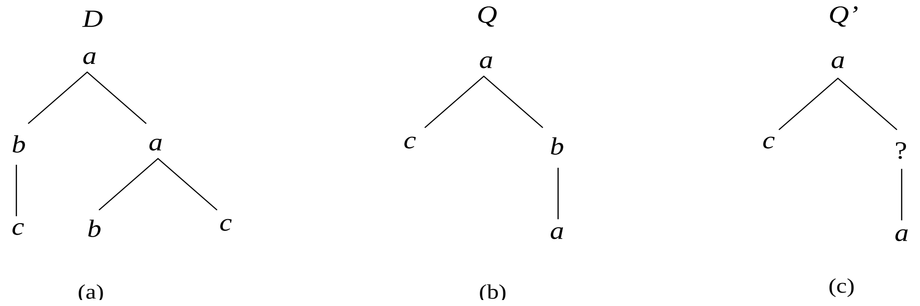


Figure 7: Example trees for illustrating the pathfilter algorithm.

Now suppose a query tree $Q$ is as shown in Figure 7(b). $Q$ has the pairs $(a, c)$, $(a, b)$, $(b, a)$. Since the data tree $D$ doesn't have $(b, a)$, there is no way that it can match $Q$. That is, the filtering sees whether each pair in the query tree is present in the data tree *at least as often*; otherwise the data tree cannot match the query tree. If the query tree has don't cares, simply ignore them in the construction of parent-child pairs. So, for example, the query tree $Q'$ in Figure 7(c) would have only $(a, c)$ as a pair. This heuristic, referred to as pathfilter, eliminates irrelevant trees from consideration in the beginning of a search and gives us a set of candidate trees to look for.

# 5 Experiments and Results

We have conducted a series of experiments to evaluate the performance of our algorithms pathfix and pathfilter, collectively referred to as ATreeGrep (reminiscent of AGrep [25] for approximate string searching and SGrep [13] for structure grep). The programs were written in K (`http://www.kx.com`) and run under Solaris on a Sun Ultra 10 workstation.

One thousand trees were randomly generated, each tree having 100 nodes. The string labels of nodes were randomly chosen from a dictionary. In each run, a tree was selected and modified into the query tree (with or without don't cares) and the other trees were used as data trees. Ten runs were tested and the average was plotted. Figure 8 shows the times spent in running ATreeGrep on the synthetic trees where the dictionary size is 50 and 1000, respectively.

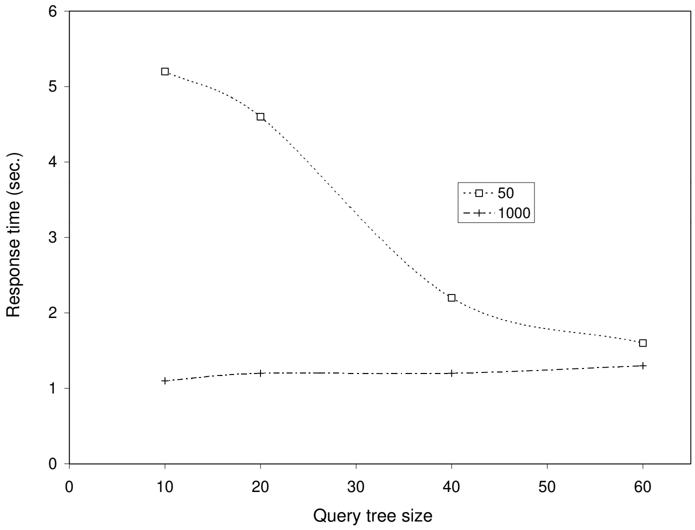


Figure 8: Running times of ATreeGrep on the 1000 synthetic trees with a label dictionary size of 50 and 1000, respectively.

It can be seen from Figure 8 that the dictionary size has a significant impact on the running times of ATreeGrep. When both the dictionary size and query tree are small, pathfilter finds many candidate trees, requiring much time for checking. When the query tree is large, however, the parent-child pairs combined with their counts produce a selective filter and hence there are few trees to look for. Consequently, the total running time decreases.

On the other hand, when the dictionary size is large, very few parent-child pairs are the same regardless of the query tree size. As a consequence, many trees are eliminated by pathfilter.

Figure 9 compares ATreeGrep and pathfix for varying query tree sizes, where the dictionary size was fixed at 1000. The figure shows that pathfilter speeds up ATreeGrep considerably. It can be seen that the running time of pathfix is proportional to the size of a query tree (actually the total number of paths of the query tree). It was also observed that the running time of pathfix is proportional to the number of matches in the database.

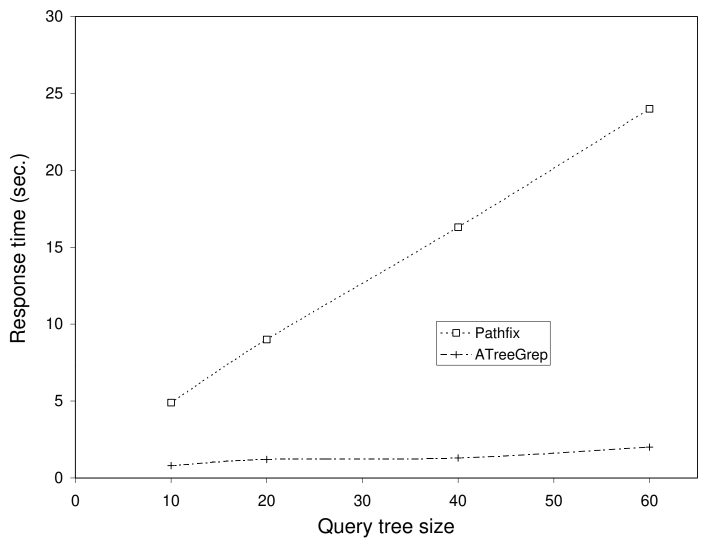


Figure 9: Running times of ATreeGrep and Pathfix on the 1000 synthetic trees with a label dictionary size of 1000.

Finally we ran the algorithms on the phylogenetic trees obtained from TreeBASE maintained at Harvard University Herbaria (`http://www.herbaria.harvard.edu/treebase`). Phylogenetic trees are structures used in biology to study the evolution of various life forms as well as the relationship of a particular life form with other life forms. We considered 1548 phylogenetic trees in TreeBASE. Most non-leaf nodes in these trees have two children. The number of nodes of a tree ranges from 50 to 200, and the dictionary size of node labels is 18870. All the leaf-nodes and some non-leaf nodes have labels. For each non-leaf node without a label, we use "U" as its label. Figure 10 shows the results. The results are promising and consistent with those for the synthetic data.

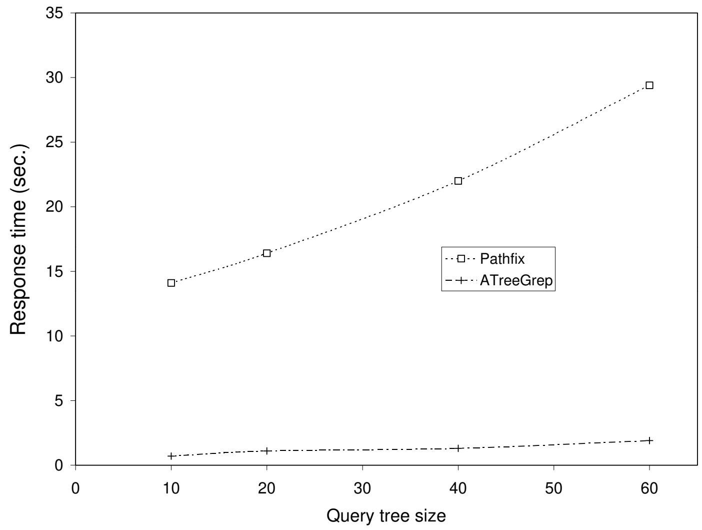


Figure 10: Running times of ATreeGrep and Pathfix on the 1548 phylogenetic trees obtained from TreeBASE.

# 6 Applications

We have incorporated ATreeGrep into two Web-based systems. The first one is a structure-based search engine, accessible at `http://aria.njit.edu/~biotool/`, and now incorporated into TreeBASE's keyword-based search engine, accessible at `http://www.treebase.org/treebase/console.html`. This structure-based search engine is visited a few hundred times a month. Figure 11 shows its 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 containing the query tree.

The second system we have implemented, called XML Query by Example (or XML QBE) and accessible at `http://aria.njit.edu/~mediadb/`, allows the user to input an example XML fragment (query tree) and then finds those XMLs in an XML database that approximately contain the query tree. For example, the query in Figure 12 is to find all the XML documents describing movies in which Robert Redford is the director,

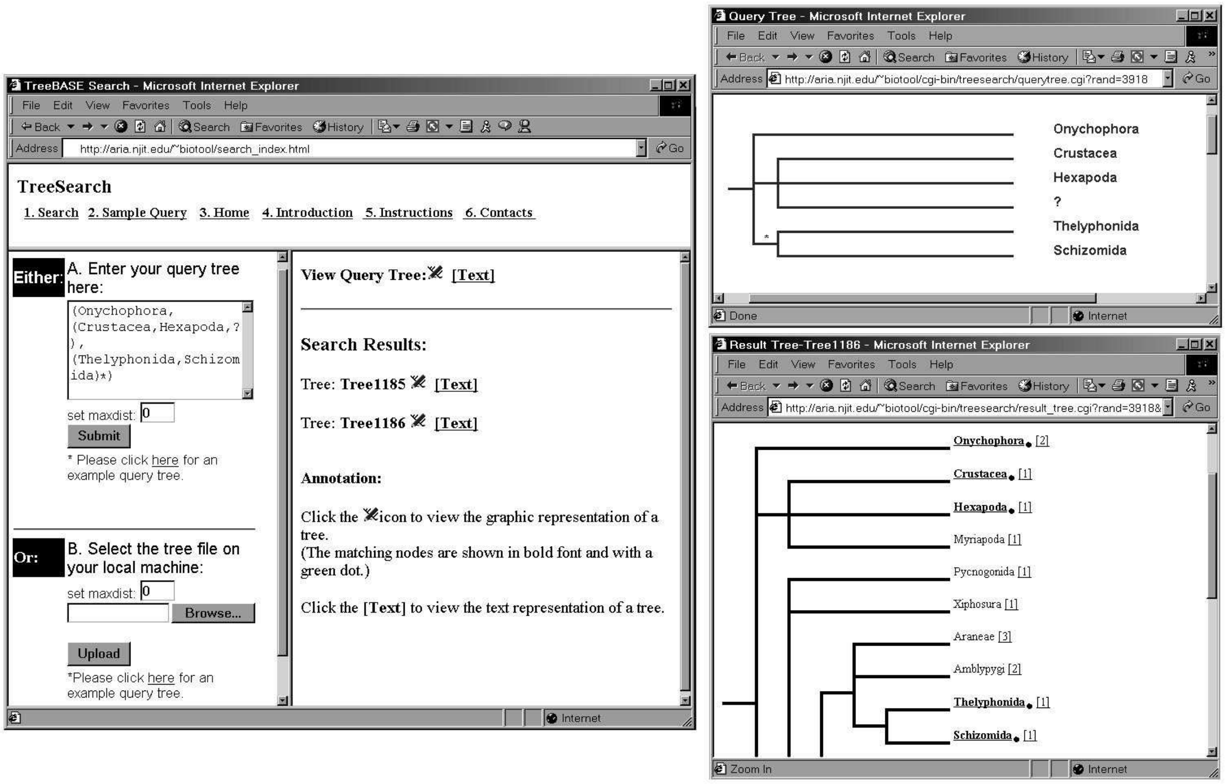


Figure 11: An example query and search results in the structure-based search engine for TreeBASE.

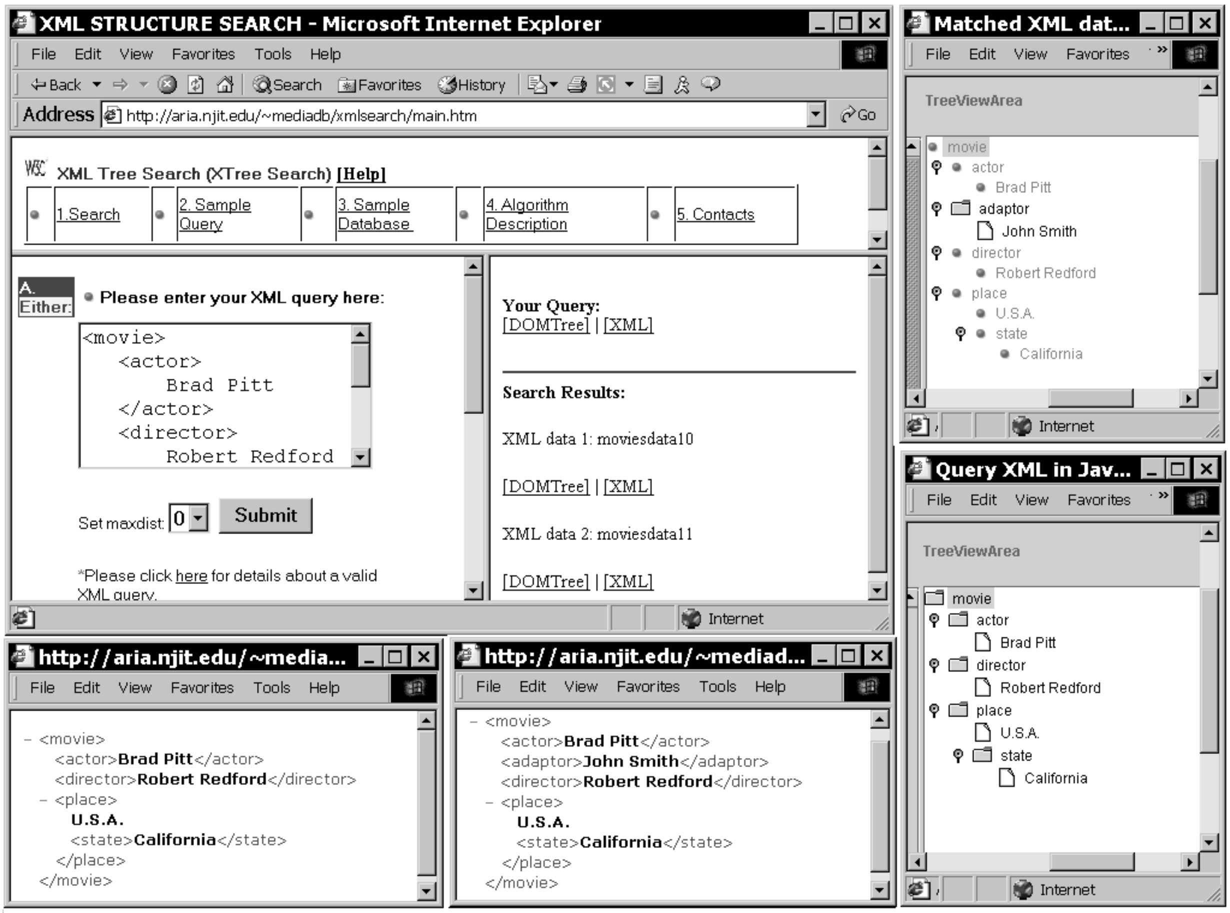

Figure 12: An example query and search results on a movie document database in XML QBE.

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.

In general, when interacting with XML QBE, the user is able to type in his own query, load the query from a file, or use and modify a sample query provided by the system. The user is also able to browse the underlying database and read the XML documents in the database.

# 7 Conclusion

Unordered labeled trees find many applications in computer and natural sciences. In this paper we have presented a new approach, called ATreeGrep, for searching among these trees. Experimental results show that ATreeGrep is fast, particularly when the dictionary size of node labels is large. The algorithm and its code can be obtained at `http://cs.nyu.edu/cs/faculty/shasha/papers/treesearch.html`. We have implemented the algorithm into two Web-based search engines for phylogenetic databases and XML repositories. Future work includes extending our algorithm for ordered labeled trees, for searching other types of scientific databases, and for finding patterns in these databases [21, 23].


## Acknowledgment

We would like to thank William Piel, the designer of TreeBASE, for his continuing collaboration. We also thank Ken Abe for performing experiments to determine the best filter parameters, and thank Greg Heil for a very compact K language implementation of suffix arrays. Finally we thank Sen Zhang, who implemented XML Query by Example and provided Figure 12. This work was supported in part by U.S. NSF grants IIS-9988345, IIS-9988636, and by the Natural Sciences and Engineering Research Council of Canada under Grant No. OGP0046373.


## References


[1] E. N. Adams. Consensus techniques and the comparison of taxonomic trees. *Systematic Zoology*, 21:390–397, 1972.

[2] 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.

[3] V. Berry and D. Bryant. Faster reliable phylogenetic analysis. In *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology*, pages 59–68, 1999.

[4] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the 6th International Conference on Database Theory*, pages 336–350, 1997.

[5] J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:311–326, 1965.

[6] G. Chang, M. J. Healey, J. A. M. McHugh, and J. T. L. Wang. *Mining the World Wide Web: An Information Search Approach*. Kluwer Academic Publishers, Norwell, Massachusetts, 2001.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J. Cracraft and M. Donoghue. Assembling the tree of life: Research needs in phylogenetics and phyloinformatics. Report from NSF Workshop, Yale University, July 2000.

[12] 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.

[13] 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`.

[14] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages*, 2001.

[15] S. Kannan, E. Lawler, and T. Warnow. Determining the evolutionary tree. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 475–484, 1990.

[16] 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 Data Bases and Programming Languages*, pages 157–175, 1993.

[17] 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.

[18] A. S. Noetzel and S. M. Selkow. An analysis of the general tree-editing problem. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 237–252. Addison-Wesley, Reading, MA, 1983.

[19] 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.

[20] 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 Data Bases and Programming Languages*, pages 115–135, 1993.

[21] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 115–125, 1994.

[22] 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.

[23] 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.

[24] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and K. M. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):889–895, 1998.

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

[26] 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.

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