University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) Department of Computer & Information Science

September 2006

Crimson: A Data Management System to Support Evaluating Phylogenetic Tree Reconstruction Algorithms

Yifeng Zheng
University of Pennsylvania

Stephen Fisher
University of Pennsylvania

Shirley Cohen
University of Pennsylvania

Sheng Guo
University of Pennsylvania

Junhyong Kim
University of Pennsylvania

See next page for additional authors

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Recommended Citation

Yifeng Zheng, Stephen Fisher, Shirley Cohen, Sheng Guo, Junhyong Kim, and Susan B. Davidson, "Crimson: A Data Management System to Support Evaluating Phylogenetic Tree Reconstruction Algorithms", . September 2006.

Postprint version. Published in *32nd International Conference on Very Large Data Bases*, ACM 2006, September 12-15, 2006, pages 1231-1234. Publisher URL: http://www.vldb.org/dblp/db/conf/vldb/vldb2006.html

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/288
For more information, please contact libraryrepository@pobox.upenn.edu.

Crimson: A Data Management System to Support Evaluating Phylogenetic Tree Reconstruction Algorithms

Abstract
Evolutionary and systems biology increasingly rely on the construction of large phylogenetic trees which represent the relationships between species of interest. As the number and size of such trees increases, so does the need for efficient data storage and query capabilities. Although much attention has been focused on XML as a tree data model, phylogenetic trees differ from document-oriented applications in their size and depth, and their need for structure based queries rather than path-based queries.

This paper focuses on Crimson, a tree storage system for phylogenetic trees used to evaluate phylogenetic tree reconstruction algorithms within the context of the NSF CIPRes project. A goal of the modeling component of the CIPRes project is to construct a huge simulation tree representing a "gold standard" of evolutionary history against which phylogenetic tree reconstruction algorithms can be tested.

In this demonstration, we highlight our storage and indexing strategies and show how Crimson is used for benchmarking phylogenetic tree reconstruction algorithms. We also show how our design can be used to support more general queries over phylogenetic trees.

Keywords
databases, phylogenetics

Comments
Postprint version. Published in *32nd International Conference on Very Large Data Bases*, ACM 2006, September 12-15, 2006, pages 1231-1234. Publisher URL: http://www.vldb.org/dblp/db/conf/vldb/vldb2006.html

Author(s)
Yifeng Zheng, Stephen Fisher, Shirley Cohen, Sheng Guo, Junhyong Kim, and Susan B. Davidson

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/288

Crimson: A Data Management System to Support Evaluating Phylogenetic Tree Reconstruction Algorithms *

Yifeng Zheng, Stephen Fisher, Shirley Cohen,
Sheng Guo, Junhyong Kim, and Susan B. Davidson
University of Pennsylvania
yifeng@cis.upenn.edu, safisher@sas.upenn.edu, shirleyc@cis.upenn.edu,
sheng@mail.med.upenn.edu, junhyong@sas.upenn.edu, susan@cis.upenn.edu

ABSTRACT

Evolutionary and systems biology increasingly rely on the construction of large phylogenetic trees which represent the relationships between species of interest. As the number and size of such trees increases, so does the need for efficient data storage and query capabilities. Although much attention has been focused on XML as a tree data model, phylogenetic trees differ from document-oriented applications in their size and depth, and their need for structure-based queries rather than path-based queries.

This paper focuses on Crimson, a tree storage system for phylogenetic trees used to evaluate phylogenetic tree reconstruction algorithms within the context of the NSF CIPRes project. A goal of the modeling component of the CIPRes project is to construct a huge simulation tree representing a "gold standard" of evolutionary history against which phylogenetic tree reconstruction algorithms can be tested.

In this demonstration, we highlight our storage and indexing strategies and show how Crimson is used for benchmarking phylogenetic tree reconstruction algorithms. We also show how our design can be used to support more general queries over phylogenetic trees.

1. INTRODUCTION

Phylogenetics – the science of identifying and understanding evolutionary relationships between different species – has become increasingly important in biomedical research, and a variety of phylogenetic tree reconstruction algorithms have been proposed [5, 10]. As the use of these algorithms spreads and the number and size of phylogenetic trees that are generated increases, a number of questions arise. First, how do we design efficient data storage and query capabilities for managing phylogenetic trees; and second, how can these algorithms be evaluated? Both of these questions are at the core of the NSF funded Cyberinfrastructure for Phylogenetic Research (CIPRes) effort.

Evaluating phylogenetic tree reconstruction algorithms remains elusive since evolutionary history is not known. Under the standard paradigm for phylogeny algorithm experiments, a branching tree model is generated by some method (usually a stochastic model) and the evolution of a bio-molecular sequence is simulated using the tree as a guide. A typical experiment will consist of many variations of the tree model as well as variations of the bio-molecular sequence evolution model. However, the possible model space for both trees and sequences is extremely large, resulting in poor experimental design by non-evolutionary biologists (e.g., algorithm developers).

The key idea behind the modeling component of the CIPres project is to generate very large tree models and very complex sequence evolution models that are carefully curated by experts [10]. These "gold standards" can then be sampled to evaluate phylogenetic algorithms in a manner analogous to how actual empirical data is collected. The Crimson system focuses on providing data management support for this component of CIPres.

An important concern in phylogenetic tree management, whether trees are constructed by an algorithm or generated as a gold standard, is scalability. Phylogenetic trees contain millions of species. Species may also have species data associated with them. Species data is typically gene sequences representing some phenotypic characteristic (such as eyecolor), and may contain millions of individual sequences each with thousands of characters. Due to the sheer size of the phylogenetic trees, the issue of how to efficiently manage and query this data is important.

Although XML, a standard tree data model, is a natural candidate for representing phylogenetic trees, the data management strategies developed for XML are not suitable due to the size and depth of phylogenetic trees, as well as the type of queries, which are structure-based rather than path-oriented.

According to a study of about 200,000 XML documents [7], most XML documents are relatively shallow: the average depth was reported to be 4, and the deepest document was 135 levels. In contrast, simulation phylogenetic trees have an average depth of greater than 1000, and the deepest tree can be more than 1 million levels.

The queries used with phylogenetic tree are also very different from the path-oriented or restructuring queries supported by XPath and XQuery, and include structure-based queries such as *least common ancestor*, *minimal spanning clade*, *tree pattern match*, and *tree projection* (see [10] for details of these operations).

To benchmark phylogenetic algorithms using the "gold standard" trees, we must also support a variety of *sampling queries*. Since the phylogenetic reconstruction problem is NP-hard [5], current algorithms can only handle a relative small input set (i.e. several

* This work was funded by NSF ITR EF 03-31654 entitled "BUILDING THE TREE OF LIFE: A National Resource for Phyloinformatics and Computational Phylogenetics".

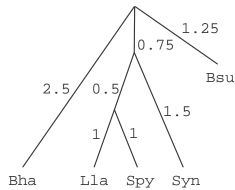

Figure 1: A sample phylogenetic tree

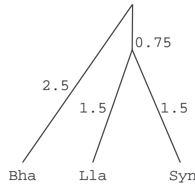

Figure 2: The projection subtree for leaf set $\{Bha, Lla, Syn\}$ from the sample phylogenetic tree

hundred to several thousand species). To benchmark these reconstruction algorithms, we must therefore be able to efficiently sample a subset of species according to various criteria, and *project* the tree pattern induced by the sample in the simulation tree.

To illustrate tree projection, Figure 1 shows an example of a tree in which the leaves have species names, and edges are labeled with the evolutionary time from the parent species to child species (species data is omitted in this figure). Randomly sampling three species from the simulation tree in Figure 1 could yield the set $\{Bha, Lla, Syn\}$. Projecting the tree induced by $\{Bha, Lla, Syn\}$ yields the subtree in Figure 2. Since all nodes in trees produced by phylogenetic tree reconstruction algorithms have outdegree greater than 1, if any such node occurs in the projection tree we merge it with its child and take the new edge weight as the sum of the two edge weights (as is the case with the parent of node Lla).

In this demonstration, we highlight the design of our storage and indexing strategy to efficiently support sampling and structure-based operations on phylogenetic trees, and show how this is used to support benchmarking phylogenetic tree reconstruction algorithms. We also discuss how these strategies can be used for supporting more general queries over phylogenetic trees. The indexing strategy is based on an extension of the Dewey labeling scheme [11] in which the input tree is decomposed into a set of subtrees with bounded depth; each subtree is then labeled using the Dewey labeling scheme.

The rest of paper is organized as follows: Section 2 describes the architecture and each module of Crimson. Section 3 presents the key features of Crimson included in the demonstration.

2. CRIMSON ARCHITECTURE

In this section, we present the main highlights of the Crimson system. As shown in Figure 3, the system consists of three components: a *repository manager*, a *benchmarking manager* and a *GUI manager*. Crimson loads the trees into a relational database via the loading query provided by the repository manager. The benchmarking manager characterizes and evaluates a tree inference algorithm by comparing its output to a set of projection trees from the relational database. The GUI manager provides a user friendly interface.

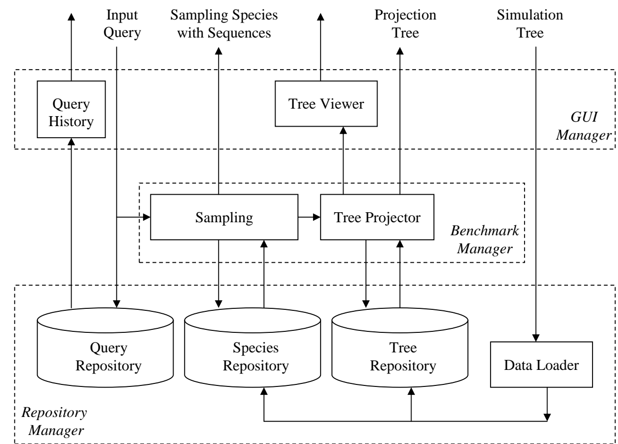

Figure 3: Architecture of Crimson

2.1 Repository Manager

NEXUS [6] is the standard data format for representing phylogenetic data. While it is efficient for exchanging phylogenetic data, NEXUS is not well suited for querying. Crimson therefore stores trees in relational form, and uses indexes based on Dewey labeling [11] to speed up queries.

The idea behind Dewey labels is as follows: For each node n, we randomly order the outgoing edges and use the order as the label of the edge. Since there is a unique path p from the root to a given node n, we concatenate the labels of edges appearing in p, using the resulting string as the Dewey label for node n. For example, the label of the leaf node *Lla* in Figure 1 would be (2.1.1), and that of *Spy* would be (2.1.2). As shown in [10], using Dewey labels for structure-based queries is very efficient. For example, the least common ancestor of *Lla* and *Spy* could be found by computing the longest common prefix of their labels, yielding the (interior) node with label (2.1).

However, since the size of a Dewey label is proportional to the length of the path from the root to the node and phylogenetic trees are very deep, the Dewey labels of nodes may become large enough to hurt query performance. Crimson therefore uses a hierarchical labeling scheme which bounds the size of labels to a constant f. The idea is as follows: Given a phylogenetic tree, we decompose it into a set of subtrees with bounded depth f. We call the set of subtrees "layer 0". If there is more than one subtree in layer 0, we build one or more "layer 1" trees, each of which has bounded depth f; each node in a layer 1 tree corresponds to a subtree in layer 0, and the relationship between nodes in a layer 1 tree is the same as the relationship between the subtrees in layer 0. If layer 1 contains more than one subtree, we recursively build higher layers until we reach a layer with only one subtree. Each node is then given a Dewey label which is local to its tree, and therefore the size of each label is bounded by f.

The LCA l of two nodes m and n inside the same subtree can be calculated as the longest common prefix of the labels of m and n, as illustrated before. If the nodes are not in the same subtree, we go up one layer, find the two nodes r_m and r_n that represent the subtrees in which m and n are located, and compute the LCA of r_m and r_n, denoted l'. (Note that this may imply traversing recursively up through the layers of the tree structure.) Then the LCA l of m

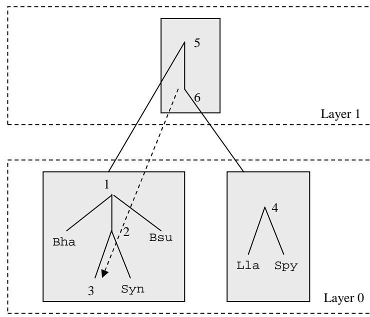

Figure 4: Index structure of the sample tree 1

and n must be located in the subtree that l' represents. To find l in this subtree, we find for each a node which is their ancestor, i.e. nodes m' and n' such that m' is an ancestor-or-self of m and n' is an ancestor-or-self of n. Ancestors are found using source nodes. Then l is the LCA of m' and n'.

For example, Figure 4 shows the index structure of the tree in Figure 1. The subtrees rooted at nodes 1 and 4 are in the partition of the original tree, which is layer 0. The tree rooted at node 5 is in layer 1. The dotted edge from node 6 to node 3 indicates that the tree represented by 6 has been split off from node 3, i.e. that the subtree rooted at node 4 was split off from node 3. We call node 3 the *source node* of node 6.

Continuing our example, suppose we want to find the LCA of nodes *Syn* and *Lla*, which are not in the same subtree. We must therefore go up a layer and find the nodes which represent the trees containing *Syn* and *Lla* (5 and 6). We then compute the LCA of nodes 5 and 6, obtaining node 5 which represents the subtree rooted by node 1. Then the LCA of *Lla* and *Syn* must be located in the subtree rooted by 1. We then find that *Syn* is itself in this subtree, and that node 3 is an ancestor of *Lla* (node 3 is the source of node 6, node 6 is an ancestor-or-self of node 6, and *Lla* is in the tree represented by node 6). Thus the LCA of *Lla* and *Syn* is the LCA of 3 and *Syn*, which is node 1.

Since queries are structure-based, we separate the storage of the tree structure from the species data. In our architecture, the *Tree Repository* and *Species Repository* contain the tree structure and species data, respectively. In addition to these data repositories, the system also records a history of user input queries in the *Query Repository*. Used in conjunction with the Crimson GUI, the Query Repository makes it convenient for users to recall and rerun historical queries.

2.2 Benchmark Manager

The Benchmark Manager tests and evaluates tree inference algorithms against the gold-standard simulation tree. Since reconstructing phylogenetic trees is an inherently hard problem, current tree reconstruction algorithms do not scale to the size of the simulation tree. We therefore provide the ability to sample a subset of species in the simulation tree to create an input set against which the algorithm can be tested. To recreate the tree structure connecting the sampled species in the simulation tree, we must then project the tree over the sampled set.

We now describe two queries that are heavily used within Crimson.

Sampling a set of species with respect to a given time. The target of this sampling method is to guarantee that the sampled results are derived from an evolutionary time period. As an example, a user may want to randomly sample four species with respect to an evolutionary distance of 1 from the sample simulation tree shown in Figure 1. To do this, we use the following strategy: First, we search for all the nodes of the tree (including the leaves) whose total weight from the root of the tree exceeds 1. From our example, there are four nodes which satisfy this condition. They are $\{Bha, x, Syn, BSU\}$, where x is the parent node of Lla and Spy. Then, for each node, we randomly select 4/4=1 leaves from the subtree rooted by the node. The result is $\{Bha, Lla, Syn, BSU\}$ or $\{Bha, Spy, Syn, BSU\}$.

Tree projection: As explained in Section 1, given a tree T and a subset S of its leaves, the tree projection of T over S is a "subtree" T in which each edge is a subpath of a path from the root of T to a node in S and each node has at least two children. To do this, we use the following strategy: We sort the input leaf set according to the pre-order of tree T. Starting with an empty tree T, we insert nodes into the tree in order. In this way, at each point the node being inserted will become the rightmost leaf node in T after insertion. We keep a pointer to the rightmost leaf node in the current tree. To determine the parent of the new node n in T, we check the ancestor-descendant relationship between nodes in the rightmost path in the current tree and the insert node in bottom-up direction. We can check ancestor-descendant relationship by a least common ancestor query: Given two nodes m and n in a tree T, m is an ancestor of n if and only if LCA(m,n) = m.

Our indexing strategy can also be used to support more general structure-based queries over phylogenetic trees, such as least common ancestor, minimal spanning clade and tree pattern match [10]. We have already discussed how to answer least common ancestor, and now describe the other two queries.

Minimal spanning clade: Given a set of input leaf nodes, their minimal spanning clade is the set of nodes in the tree rooted by their least common ancestor. Since the ancestor-descendant relationship between two nodes can be easily checked using LCA, finding the minimal spanning clade can be efficiently implemented by LCA [10].

Tree pattern match: Given an input pattern tree and a tree, tree pattern match determines whether or not the input tree pattern exists in the input tree. As an example, the tree pattern shown in Figure 2 will match the tree shown in Figure 1. However if we exchange the location of species Bha and Lla in the pattern tree, the new pattern will not match the tree.

To answer a tree pattern match query, we first determine the leaves in the tree pattern. Using this set of leaves as input, we then project a subtree from the input tree. We then check whether or not the projected tree and tree pattern are equal (in the case of an exact match) or compute the difference between them as a measure of similarity (in the case of approximate match). Since tree comparison can be done in linear time [10, 1], this can be done very efficiently.

2.3 GUI Manager

Crimson provides two types of user interfaces: a graphical interface and a command-line interface. The graphical interface provides a rich visual environment that is easy to use and understand. The GUI lets the user load sequences and trees into a shared repository

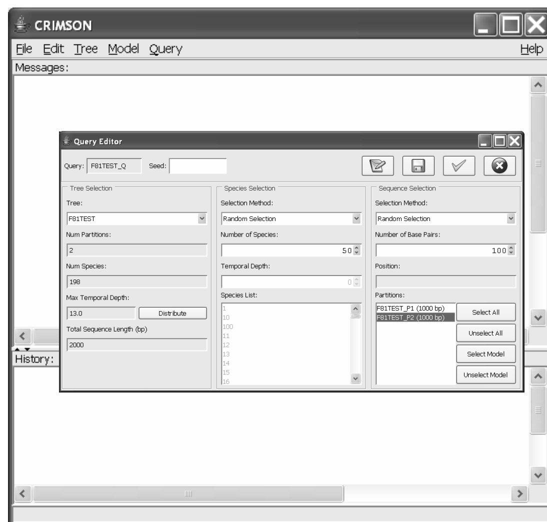

Figure 5: A Snapshot of Crimson System

and run interactive queries on this data. A query wizard helps to construct, execute, and recall queries. The results are displayed in NEXUS or dendrogram format using Walrus [4, 8]. The python scripting based command-line interface further provides users the ability to create their own scripts to automate various tasks. A snapshot of Crimson is shown in Figure 5.

3. DEMONSTRATION

Crimson has been implemented in C++ and Java. The following features of Crimson will be demonstrated:

Loading Data: Users may choose to load a phylogenetic tree with species data, load a phylogenetic tree structure only, or append species data to an existing phylogenetic tree. Messages about the loading status as well as errors are dynamically generated and displayed to the user.

Tree Projection: Crimson supports several ways of selecting species and projecting a tree: random sampling, random sampling with respect to time, and user input. Each selection method may require appropriate input values. If an input value is invalid, a popup window will be generated to show any error messages.

Visualizing the results: Users may display result trees graphically or view them as NEXUS files. The graphical interface is based on Walrus, a Java application developed for 3D viewing of large graphs; Python is used to convert a NEXUS file to a Walrus input file.

Besides demonstrating Crimson, we will describe the index structures and algorithms used in the system.

Why don't we use XML?
Although XML is a tree structured model, existing XML indexing and query evaluation techniques cannot be directly applied to phylogenetic trees for the following reasons:

1. Phylogenetic simulation trees of arbitrary depth, e.g. several thousand nodes. However, existing web and commercial data stored in XML format is typically much shallower, and it is for this type of data that XML indexing and query evaluation techniques have been developed.

2. The queries used in phylogenetic tree benchmarking are not path-oriented or restructuring queries such as those supported by XPath and XQuery. For example, finding a tree pattern induced by a set of nodes is quite different from retrieving a set of nodes following a path pattern.

What are the database challenges in Crimson?
There are several database challenges in Crimson system:

1. Simulation trees are huge, yet the portions retrieved by a single query are relatively small. It is important to support random access based on species names or evolutionary time, which argues against using main memory techniques to implement the operations.

2. Existing indexing techniques to support XPath/XQuery are based on paths, which are not important for phylogenetic trees.

3. Various labeling schemes [3, 2] have been designed to find relationships between nodes, e.g. parent/child, ancestor/descendant. However, only the Dewey labeling scheme [11] facilitates finding the least common ancestor of a set of nodes, which is important when finding the tree pattern induced by a set of nodes.

4. The Dewey labeling scheme suffers, however, in very deep trees since the Dewey number of a node n encodes the path from root to n. How to extend Dewey labeling to answer queries on very deep tree is another issue addressed in this demo.

Why would this demo be interesting for the database community?
Stonebraker recently argued [9] that the traditional database concept of "one size fits all" is no longer applicable in the database market. Nowhere is this more true than with scientific data, whose structure and usage differs dramatically from business data. This demo illustrates the complexity of structure as well as usage of one common form of scientific data.

4. REFERENCES

[1] N. Amenta, F. Clarke, and K. S. John. A linear-time majority tree algorithm. In *Proceedings of WABI*, 2003.

[2] D. DeHaan, D. Toman, M. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD*, 2001.

[3] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of STOC*, 1982.

[4] Y. Hyun. Walrus - graph visualization tool. http://www.caida.org/tools/visualization/walrus/.

[5] J. Kim and T. Warnow. Tutorial on phylogenetic tree estimation. citeseer.nj.nec.com/254275.html.

[6] D. Maddison, D. Swofford, and W. Maddison. NEXUS: an extensible file format for systematic information. *Syst. Biol*, 46:590–621, 1997.

[7] L. Mignet, D. Barbosa, and P. Veltri. The XML web: a first study. In *Proceeding of WWW*, 2003.

[8] T. Munzner. *Interactive Visualization of Large Graphs and Networks*. PhD thesis, Stanford University, 2000.

[9] M. Stonebraker. 'One Size Fits All': An Idea Whose Time Has Come and Gone. In *Proceedings of ICDE*, 2005.

[10] Susan B. Davidson and Junyhong Kim and Yifeng Zheng. Efficiently Supporting Structure Queries on Phylogenetic Trees. In *Proceedings of SSDBM*, 2005.

[11] V. Vesper. Let's do Dewey. http://www.mtsu.edu/ vvesper/dewey.html.