ASP at Work: An ASP Implementation of PhyloWS*

Tiep Le, Hieu Nguyen, Enrico Pontelli, and Tran Cao Son

Department of Computer Science
New Mexico State University
(tile,nhieu,epontell,tson)@cs.nmsu.edu

Abstract

This paper continues the exploration started in [3], aimed at demonstrating the use of logic programming technology to support a large scale deployment and analysis of phylogenetic data from biological studies. This application paper illustrates the use of ASP technology in implementing the PhyloWS web service API—a recently proposed and community-agreed standard API to enable uniform access and inter-operation among phylogenetic applications and repositories. To date, only very incomplete implementations of PhyloWS have been realized; this paper demonstrates how ASP provides an ideal technology to support a more comprehensive realization of PhyloWS on a repository of semantically-described phylogenetic studies. The paper also presents a challenge for the developers of ASP-solvers.

1998 ACM Subject Classification J.3 Life and Medical Sciences, I.2.3 Logic programming

Keywords and phrases Answer sets, phylogenetic inference, systems, applications

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.359

1 Introduction

Phylogenetic inference is the task of constructing a phylogenetic tree that accurately characterizes the evolutionary lineages among a set of given species or genes. Phylogenetic trees allow us to understand the lineages of various species and how various functions evolved, to inform multiple alignments, and to identify what is the most conserved or important in some class of sequences. As such, phylogenetic trees have gained a central role in modern biology. They have become fundamental tools for building new knowledge, thanks to their explanatory and comparative-based predictive capabilities. In [9], 20 uses of evolutionary trees are discussed. Phylogenies are also used with increased frequency in several fields, e.g., genomics [5] and ecology [22]. Indeed, an ambitious goal in system biology is the construction of the *Tree of Life*, a phylogeny representing the evolutionary history of all species [2].

The explosive growth of phylogenetic data and the central role of phylogenetic knowledge in system biology led to the development of a database of phylogenies, called TreeBASE (e.g., [14, 18], `www.treebase.org`). The database contains phylogenetic trees and data matrices, together with information about the relevant publication, taxa, morphological and sequence-based characters, and published analyses. The trees are stored as text field strings structured in the Newick format [8]. The database provides retrieval capabilities via web interface, allowing users to locate phylogenies and to obtain datasets for different studies. Users can also retrieve data via a web service interface API (`sourceforge.net/`

* This work was partially supported by NSF grant IIS-0812267.

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 359–369

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

apps/mediawiki/treebase/index.php?title=API). This interface can deliver data in several different formats, including Newick, NEXUS [12], JSON, NeXML [21].

The creation of TreeBASE is a significant step towards the goal of creating the Tree of Life. Yet, it has been recognized that the lack of interoperability and standards in data and services between tools for the inference of phylogenies prevent large-scale and integrative analyses. To address these shortcomings, several efforts have been made. One of such efforts led to the development of an *interoperation stack* (*EvoIO Stack*) for the encoding and exchange of evolutionary structures. EvoIO comprises of *(i)* an ontology for data description (*Comparative Data Analysis Ontology (CDAO)*) [17], *(ii)* an exchange format (*NeXML*) [21], and *(iii)* a web service interface (*PhyloWS*) [11].

The PhyloWS interface specification [11] identifies several classes of queries specifically tied to phylogenies. The interface is comprehensive and represents the most extensive collection of queries and transformations for biological phylogenies ever proposed—in particular, it largely subsumes previous attempts to characterize access to phylogenetic databases (e.g., the approach of [15], implemented in Prolog by [3]). The implementation of these queries on an RDF representation of phylogenies proved to be challenging; in particular, traditional languages for RDF (e.g., SPARQL) do not provide the power to perform the type of computations on phylogenies required by PhyloWS—e.g., they lack the expressive power to capture recursive computations and transitive closures (which are essential, e.g., to determine ancestors and lineage in a phylogeny).

In this paper, we propose a new and modular implementation of PhyloWS using answer set programming (ASP) [13, 16]. We present the encoding of PhyloWS and evaluate the performance of the implementation using the data extracted from TreeBASE which shows that ASP is sufficiently expressive for answering various types queries of specified in the PhyloWS specification. The experimental results show that ASP-solver is efficient but pre-processing is needed for such a data intensive application.

2 Background—PhyloWS: Phylogenetic Web Service API

PhyloWS [11] is a web-services standard for accessing phylogenetic trees, data matrices, and their associated metadata from online phylogenetic data. Together with NeXML and CDAO, PhyloWS is a part of the platform, called *EvoIO Stack* [19], that combines support for exchange of data and their semantics and predictable programmatic access. PhyloWS is proposed to address the lack of a web-service API that allows for the integration of phylogenetic data and tools into new services for large-scale analysis. To date, PhyloWS only exists as a specification. The implementation proposed in this paper is its first implementation[1]. PhyloWS contains specification for a variety of tasks necessary for the creation, maintaining, retrieving, and manipulating of phylogenetic data (e.g., trees, matrices). In this paper, we will focus on the services for retrieving phylogenetic data. Based on the specification of PhyloWS [11], queries can be grouped into the following four categories:

- **Node-oriented queries**: Queries of this type ask for nodes satisfying certain conditions, e.g., the most recent common ancestor of two (or more) terminal nodes in the specified tree; or the nodes which have the distance to the root greater than a given distance; or determine the relationships among nodes, e.g., obtain the patristic distance between two nodes tn707506 and tn444001; etc.

[1] The implementation of PhyloWS at sourceforge.net/apps/mediawiki/treebase/index.php?title=API is tailored to TreeBASE and limited to simple retrievals.

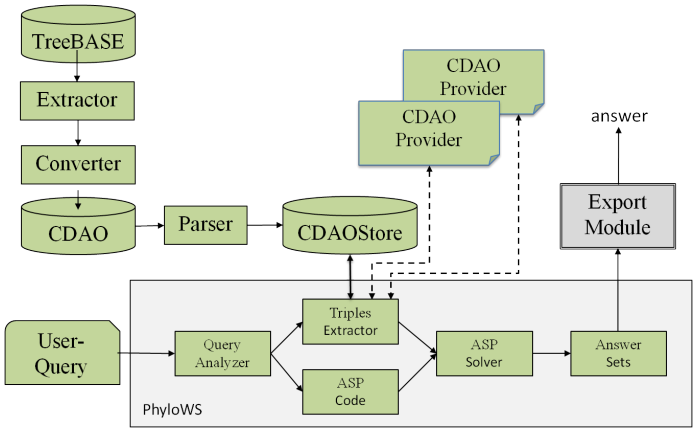

Figure 1 Overall Structure: System Implementation.

- **Clade-oriented queries**: Queries of this type are related to clades with some properties—e.g., determine the clades consisting of a species and all its descendants; or find the minimum spanning clade in a tree that contains the TUs (taxonomic units) *Ilexanomala* and *Ilex glabra*.
- **Tree-oriented queries**: These queries ask for trees satisfying some criteria, e.g., the trees created by *Knapp S.* no later than *2003-08-20*; or determine relationships among trees, e.g., the *Robinson-Foulds* distance between two trees.
- **Data-oriented queries**: These queries extract the metadata for different phylogenetic data, e.g., the *taxonVariant id* or *ncbi id* of a node; the creator of a tree; or the character matrix from all matrices containing a given set of OTUs.

3 System Organization

The overall implementation of PhyloWS is depicted in Figure 1. The top part shows the components of the system necessary to populate CDAOStore; CDAOStore [3] is triple store, build using CDAO, used for our experiments. First, data from current phylogenetic tree repositories (e.g., TreeBASE) is extracted into NeXML data files (*Extractor*) and converted to CDAO representation (*Converter*). This process is executed only once to populate the repository of phylogenetic data in CDAO representation. A standard XML-parser is used to generate triples from NeXML and import them into the CDAOStore.

The main contribution of this paper is the PhyloWS box. User queries are analyzed by a query analyzer that determines the actual ASP-code and the necessary data type from the CDAOStore repository. This information is passed on to the *Triples Extractor* module. The ASP program (facts and code) is sent to the ASP solver to compute its answer sets. The export module obtains the answer sets from the solver and generates answer for the user. Let us emphasize that the PhyloWS implementation in this paper is fully modular, and can be applied to any data source that can provide phylogenetic data as CDAO RDF triples.

Observe that in the construction of the CDAOStore, we were able to reuse the prototype described in [3]. The present system includes several improvements over the reported prototype—e.g., it is able to extract *all* studies from TreeBASE, the conversion to CDAO is more precise. The key differences between the current paper and the system described in [3] lie in (*i*) our focus here is on the full implementation of PhyloWS, instead of the simple querying covered in [3]; and (*ii*) the complete use of ASP technology in the implementation of the PhyloWS, instead of a mixture of Prolog and SPARQL.

4 PhyloWS in ASP

In this section, we will present the ASP implementation for the web services described in Section 2. For efficiency purpose, we develop a front-end that analyzes the queries and determines the type of phylogenetic data that needs to be extracted from the CDAO repository. Presently, the type of data corresponds to the type of queries. For example, for a node-oriented query, information about the trees (nodes and edges) will be extracted. The *Triples Extractor* module is responsible for extracting the necessary information from the CDAO representation and generating ASP facts for use to answer the query. Observe that this task can be combined and executed via ASP extensions such as `dlvhex` [4]. We will discuss the reason behind our design choice in the discussion part of the paper.

Representing CDAO in ASP. The information about phylogenies can be easily encoded as ASP facts. Following are some sample facts generated by the *Triples Extractor* module:

```
tree(t_id).                      % t_id is a tree
tree_label(t_id,lab).            % t_id has label "lab"
tree_is_defined_by(t_id,s_id).   % t_id is studied in s_id
tree_ntax(t_id,n_Taxa).          % t_id has n_Taxa taxa
edge(t_id,n1,n2).                % t_id contains an edge  from n1 to n2
edge_length(t_id,n1,n2,l).       % l is the length of the edge (n1,n2) in t_id
represents_TU(t_id,n1,tu_id).    % node n1 of t-id represents tu_id
taxon_id(tu_id,taxon_id).        % tu_id represents taxon_id
matrix_type(m_id,m_type).        % matrix m_id is of the type "m_type"
belongs_to_TU(m_id,cell,tu_id). % cell in matrix m_id belongs to tu_id
```

ASP Encoding of PhyloWS. The encoding of the PhyloWS in ASP starts with the definition of a set of rules that will be frequently used in several types of queries. The following code (syntax of `clingo` [10]) defines the predicates `node, leaf, root, parent`, and `ancestor` within a tree. It also defines the predicate `common_ancestor` of a set of taxa, identified by the predicate `set_of_taxa`, whose elements will be specified by `member/2`.

```
node(T,N):- edge(T,N,_).                    node(T,N):- edge(T,_,N).
leaf(T,N):- node(T,N),{edge(T,N,_)}0.       root(T,N):- node(T,N),{parent(T,_,N)}0.
parent(T,N1,N2):- tree(T), edge(T,N1,N2).
ancestor(T,N1,N2):- parent(T,N1,N2). ancestor(T,N1,N2):- ancestor(T,N1,Nb), parent(T,Nb,N2).
common_ancestor(T,N,S):- tree(T),node(T,N),set_of_taxa(S),{member(E,S):not ancestor(T,N,E)}0.
```

Most of the above rules are simple. The last rule states that node N in the tree T is a common ancestor of a set of taxa S if N is an ancestor of every member of S. We will next present the detailed encodings for the different types of queries.

Node-oriented Queries. We currently consider four frequently used node-oriented queries.

- *Query N1*: *compute the most recent common ancestor of two or more leaf nodes in a specified tree.* The input consists of a tree t and a set s of leaf nodes in t. The output should be the most recent common ancestor n of elements in s, denoted by $mrca(n, s)$, determined using the ASP rule:

  ```
  mrca(N, S):- tree(T), node(T,N), set_of_taxa(S), common_ancestor(T,N,S),
               {common_ancestor(T,Nb,S): ancestor(T,N,Nb)}0.
  ```

  The rule elegantly encodes that the most recent common ancestor of a set S is a common ancestor of S that does not have a descendant which is also a common ancestor of S.
- *Query N2*: *compute the patristic distance between two taxa in a given tree.* The patristic distance between taxa n_1 and n_2 of a tree t is defined as the sum of the distances from the most recent common ancestor to each node:

```
set_of_taxa(s). member(n1,s). member(n2, s).
distance_to_ancestor(T,N1,N2,L):- parent(T,N1,N2),edge_length(T,N1,N2,L).
distance_to_ancestor(T,N1,N2,D):- parent(T,Nb,N2),edge_length(T,Nb,N2,L),
        distance_to_ancestor(T,N1,Nb,L2), D=L+L2.
patristic_distance(T,N1,N2,D):- mrca(M, s), D=L1+L2,
        distance_to_ancestor(T,M,n1,L1), distance_to_ancestor(T,M,n2,L2).
```

The rules are simple thanks to the definition of the most recent common ancestor of a set in *Query N1*. Rules for computing the distance are standard.

- *Query N3*: *identify the set of* matching nodes *of a tree whose distance to the root is greater than a predefined distance.* Given a tree t (e.g., by identifier) and a distance c, output the matching nodes whose distance to the root is greater than c. This is implemented by the following rule:

```
matching_nodes(T,N):- root(T,R), distance_to_ancestor (T,R,N,L), L>=c.
```

- *Query N4*: *output the lineage of ancestors of a given node.* Given a tree t, a node n, the lineage of ancestors for n can be determine by the following rule, built using the facts $has\_Ancestor(t, n, x)$ representing that x is an ancestor of n in the tree t.

```
lineage_node(t,n,Ancestor_node_id) :- has_Ancestor(t,n,Ancestor_node_id).
```

Clade-oriented Queries. Two typical clade-oriented queries are implemented.

- *Query C1*: *find the minimum spanning clade and the TUs of the clade for a set of taxa in a specified tree.* Given a set of taxa s, determine the minimum spanning clade of s and its TUs. Nodes belong to the minimum spanning clade are represented by the atoms of the form $minimum\_clade(s, n)$. Atoms of the form $label(x, y)$ represent the label associated to the nodes in the clade. Since the answer is the tree whose root is the `mrca` n of s and all n's descendants, this can be implemented as follows.

```
minimum_clade(S,N):- tree(T), node(T,N), mrca(N, S).
minimum_clade(S,D):- tree(T), node(T,N), mrca(N, S), ancestor(T,N,D).
label(D,TU_Label):- tree(T), minimum_clade(_,D),
                    represents_TU(T,D,TU),tu_label(T,TU,TU_Label).
```

The first rule states that the given most recent common ancestor belongs to the minimum clade. The next rule obtains all of its descendants.

- *Query C2*: *find a clade in a tree whose taxa has a given character*, i.e., given a tree t and a character c, find a clade (or all) s of t s.t. every taxon in s has the character c.

```
clade(s).              {in_clade(s,N) : leaf(t,N)}.   member(N,s):- in_clade(s,N).
:- minimum_clade(s, N),not in_clade(s, N).
:- in_clade(s,N), represents_TU(T,N,TU),
        belongs_to_TU(M,Cell,TU), not belongs_to_Character(M,Cell,c).
```

The fact $clade(s)$ specifies the name s of the clade produced. The choice rule states that a leaf might or might not belong to the clade. The third rule defines the membership of the node in the set of taxa s that has been selected for use to determine the minimum clade (Query C1). The first constraint ensures that the elements of the clade are only those that are selected. The second removes clades that contain taxa that do not have the specified character.

Tree-oriented Queries. We consider eight types of tree-oriented queries.

- *Query T1*: *find trees matching a topology.* The topology can be given by (*i*) the range of the numbers of taxa count of the tree, i.e., between $n - c$ and $n + c$ for two constants n and c; (*ii*) the range of the width of the tree; etc.
  Most of the above queries can be straightforwardly encoded in ASP. For example, given a constant c and the tree $tr1386$, the following rule determines all trees with their taxa count in the range $[n - c, n + c]$ where n is the number of taxa of $tr1386$. The rule makes use of the predicate $tree\_ntax(t, n)$ that represents the number of taxa of a tree.

```
matching_ntax(T,Cnt):- tree_ntax(tr1386,N),tree_ntax(T,Cnt),Cnt <= N+c,Cnt>=N-c.
```

- *Query T2*: *find trees whose length is shorter (or longer) than the length of a given tree (or a constant)* where the tree length is defined as the maximal distance from the root of the tree to its taxa (leaves). This type of queries can be answered with the definition of the tree length, implemented as follows.

```
distance_to_root(T,N,L):- root(T,R),leaf(T,N),distance_to_ancestor(T,R,N,L).
tree_length(T,L):- root(T,R), leaf(T,N), distance_to_root(T,N,L),
                   {distance_to_root(T,X,L1): L1>L}0.
```

- *Query T3*: *find trees with the shortest distance from the root to a given node n.* This can be easily implemented using the predicate *distance_to_root*.
- *Query T4*: *given a set s of OTUs (or taxa), find a tree containing this set.* We present the rules for identifying trees with a set of OTU, specified by the atom $otu\_set(s)$ and the membership atoms $member(x, s)$.

```
connect_tu(TU,S,T):- tree(T),otu_set(S),member(TU,S),represents_TU(T,_,TU).
tree_otus(T,S):- tree(T),otu_set(S),{member(TU,S):not connect_tu(TU,S,T)}0.
```

- *Query T5*: *find trees based on tree metadata.* For example, trees that were *(i)* created by some author; *(ii)* created before a given date; *(iii)* built with a given type of data; etc. Since the data included in each study contains information such as *has_creator*, *has_creationDate*, *matrix_type*, etc. this type of queries can be easily encoded in ASP. Again, we omit the detail ASP rules for brevity.
- *Query T6*: *identify trees by the parsimony tree length* which is defined by the total number of characters of its taxa. This can be implemented as follows.

```
parsimony_length(T,L):- tree(T),
        L =  #count {belongs_to_Character(_,Cell_id,Character):
              belongs_to_TU(_,Cell_id,TU_id):represents_TU(T,_,TU_id)}.
```

- *Query T7*: *determine trees with size greater (or smaller) than a given constant c, or a certain ratio r of internal to external nodes.* Again, using the aggregate function *#count*, this type of query can be implemented as follows[2].

```
matching_tree_size(T,S):- tree(T), S = #count {node(T,_)}, S>=c.
internal_node(T,N):- node(T,N), not leaf(T,N).
matching_tree_ratio(T):- tree(T),R=R1/R2, R>=r,
        R1 = #count {internal_node(T,_)}, R2 = #count {leaf(T,_)}.
```

[2] The code assumes that the ration is an integer. Using the scripting feature available for `clingo`, this assumption can be removed.

- *Query T8*: *computing the Robinson-Foulds distance [1] between two trees.* The Robinson-Foulds distance is frequently used to compare phylogenetic trees. It measures the number of clusters of descendant leaves that are not shared by the two trees. The Robinson-Foulds distance of two trees T_1 and T_2 can be computed using the following algorithm:
  - Compute the multi-set of clusters of each tree, where each cluster is a set of taxa (leaves) that are descendants of an internal node n. Let us denote the set of clusters of T_1 and T_2 by $C(T_1)$ and $C(T_2)$, respectively.
  - Compute D_1 (resp. D_2), the number of clusters which belong $C(T_1) \setminus C(T_2)$ (resp. $C(T_2) \setminus C(T_1)$).

  The Robinson-Foulds distance is then defined by $(D_1 + D_2)/2$. Given two trees T_1 and T_2, we define the predicate $rf\_distance(T_1, T_2, D_1, D_2)$ that encodes the Robinson-Foulds distance. This can be implemented using the following set of ASP rules.

```
in_cluster(T,N,L):- internal(T,N), leaf(T,L), ancestor(T,N,L).
neq_cluster(X,Y):- internal(T1,X),internal(T2,Y),T1!=T2,in_cluster(T1,X,L),
                                                    not in_cluster(T2,Y,L).
neq_cluster(X,Y):- internal(T1,X),internal(T2,Y),T1!=T2,not in_cluster(T1,X,L),
                                                    in_cluster(T2,Y,L).
eq_cluster(X,Y,T1,T2):- internal(T1,X),internal(T2,Y),T1!=T2,not neq_cluster(X,Y).
{matched(X,Y,T1,T2) : eq_cluster(X,Y,T1,T2)}.
2{used(T1,X), used(T2,Y)}:- matched(X,Y,T1,T2).
matched(X,Y,T1,T2):- matched(Y,X,T2,T1).
:-matched(X,Y,T1,T2),matched(X,Z,T1,T2),Y!=Z.
:-matched(Y,X,T1,T2),matched(Z,X,T1,T2),Y!=Z.
:-eq_cluster(X,Y,T1,T2), not used(T1,X), not used(T2,Y).
not_matched(T,N):- internal(T,N),not used(T,N).
rf_distance(T1,T2,D1,D2):- tree(T1), tree(T2), T1!=T2,
        D1 = #count {not_matched(T1,N)}, D2 = #count{not_matched(T2,N)}.
```

  The clusters are named by the internal nodes. The first rule defines the elements of a cluster. Next two rules state that two clusters from different trees are different when their sets of taxa are different. The third rule defines when two clusters are identical. The choice rule defines the predicate $matched(X, Y, T_1, T_2)$ among identical clusters of the trees. The next two rules define the predicates $matched(X, Y, T_1, T_2)$ and $used(X, T_1)$ ($used(Y, T_2)$) which say that the cluster X of tree T_1 is identical to the cluster Y of tree T_2 and will not be counted towards D_1 and D_2 respectively. The constraints ensure that each cluster is used to match with at most one cluster and the matching should be done as long as it is possible. $not\_matched(T, N)$ indicates the cluster that is not matched with any cluster of another tree. The last rule encodes the Robinson-Foulds distance.

Data-oriented Queries. We consider four types of data-oriented queries.

- *Query D1*: *list metadata for a given taxon n or a given tree t.* Such information can be obtained from the facts associated to the tree. Some of the rules are:

```
metadata_belongs_to(n,T,Study_id):- node(T,n), tree_is_defined_by(T,Study_id).
metadata_represents(n,TU_id,TU_label,Taxon_id,TaxonVariant_id,Ncbi_id,Ubio_id):-
        represents_TU(T,n,TU_id), tu_label(T,TU_id,TU_label),
        taxon_id(TU_id,Taxon_id),taxonVariant_id(TU_id,TaxonVariant_id),
        ncbi_id(TU_id,Ncbi_id), ubio_id(TU_id,Ubio_id).
metadata_character(n,Character_id):-
        represents_TU(_,n,TU_id),belongs_to_TU(_,Cell_id,Tu_id),
        belongs_to_Character(_,Cell_id,Character_id).
```

- *Query D2*: *identify all matrices containing a given OTU (tu_id); or determine all characters in a matrix (m_id) that have data for an OTU.* This query is encoded as follows.

```
matrices_with_otu(M,tu_id):- has_TU(M,tu_id).
character_has_otu(C,m_id,tu_id):- belongs_to_TU(m_id,Cell,tu_id),
                                  belongs_to_Character(m_id,Cell,C).
```

- *Query D3*: *identify all OTUs in a matrix which have a given set of characters*:

```
has_character(Tu,C):- belongs_to_TU(M,Cell,Tu), belongs_to_Character(M,Cell,C).
obtain_otus_having_characters(Tu,S):- has_TU(M,Tu), set_characters(S),
        {member(C,S): not has_character(Tu,C)}0.
obtain_otus_having_characters_belonging_matrix(matrix_id,Tu,S):-
        has_TU(M,Tu), set_characters(S),{member(C,S): not has_character(Tu,C)}0.
```

- *Query D4*: *identify the character that appears in all matrices containing data for a given set of OTUs*. The ASP rules for this query are:

```
matching_matrices(M,S):- otu_set(S), has_TU(M,_),{member(E,S): not has_TU(M,E)}0.
has_character(M,C):- belongs_to_Character(M,_,C).
character_in_all_matrices(C):- matching_matrices(M,S), has_character(M,C),
        {matching_matrices(M1,S): not has_character(M1,C)}0.
```

  The first rule identifies the matrix that contains all the given OTUs. The other rules search for the characters that appear in all those matrices.

5 Evaluation

We have successfully extracted all data from the TreeBASE and created various types of NeXML files for studies (2989 files, 2.55GB), matrices (5794 files, 1.96GB), and trees (8621 files, 500MB). So far, we have converted them into 9558 CDAO files and stored them in study (861), matrix (76), and tree (8621) files. The space requirement for CDAO study, matrix, and tree files is 18GB, 3410MB and 2214MB, respectively. We observe that the conversion is fairly time consuming and space demanding. However, most of the time is spent in the conversion of `Character State Data Matrix`, e.g., the program takes 3.25 hours to convert the `Character State Data Matrix` of the study `S715`, stored in a 3.36MB file, that has 44 OTUs and each OTU has 2721 characters. On the other hand, the conversion of the largest tree (identifier `Tr47158`), stored in a 3MB file, into CDAO took less than 5 minutes. From this data, we have built around 4GB fact data from study and matrix files and 172MB fact data from tree files and populated CDAOStore with both sets of data. The huge size of the data is the main reason for the design decisions discussed in the next section.

Table 1 contains sample results of the system for several queries discussed in the previous section. The experiment is conducted using 261 CDAO files. The ASP solver used in the experiment is `clingo` version 3.0.3. The machine used in the experiment uses Linux OS with a Genuine Intel(R) CPU T2400 @ 1.83GHz and 1015 MB. Because the *Triples Extractor* is fairly efficient (it takes usually less than 15 seconds to extract the necessary data from the CDAOStore), we only report the time (in ms.) used by the ASP solver.

6 Related Work and Discussion

Related Work. ASP has been used in the construction of phylogenetic network such as the evolutionary history of Indo-European languages [7]. The method was later applied to the analysis of parasite-host systems [6]. Our use of ASP in this paper is different in that we use ASP in the development of phylogenetic web services.

The present work is most closely related to our previous work [3]. As we have indicated in Section 3, the present work is much advanced comparing to the early work. In particular,

Table 1 Evaluation of queries.

Query	Data size	Execution time	Query	Data size	Execution time
$N1$	644.5 KB	2.360	$N2$	698.8 KB	2.840
$N3$	685.2 KB	1.060			
$C1$	1.5 MB	2.940	$C2$	31.9 MB	13.420
$T1$	698.8 KB	1.210	$T2$	698.8 KB	1.000
$T3$	698.8 KB	0.810	$T4$	1.2 MB	0.450
$T5$	65.6 KB	0.020	$T6$	31.9 MB	1913.820
$T7$	644.5 KB	0.830	$T8$	6.2 KB	0.820
$D1$	44.0 MB	15.270	$D2$	34.9 MB	10.310
$D3$	34.9 MB	14.020	$D4$	19.1 MB	5.940

the set of queries implemented in this paper—as indicated in the PhyloWS specification and agreed by the community—is broader and addresses the need of the community. Furthermore, the implementation described in this paper employs only ASP for the query evaluation.

Design Choices. ASP technologies have been extended to allow ASP programs interact with ontologies such as the system `dlvhex` [4]. As such, it is natural to ask the question of whether PhyloWS could be implemented using `dlvhex` and how would the system perform. To answer these questions, we have experimented with the web interface at the URL `http://asptut.gibbi.com/`. With a few changes in the syntax to conform with the `dlvhex` syntax, most queries can be executed with sample data. The difficulty arises when we attempt to run with the real data. As it turns out, converting everything into triples using `dlvhex` using the command: `triple(X,Y,Z):-&rdf[file_URI](X,Y,Z)` and then defining necessary predicates such as `has_TU, belongs_to_TU` using standard LP rules such as[3]

```
has_TU(X,Z) :- triple(X,"<http://___/cdao.owl#has_TU>",Z).
belongs_to_TU(X1,Y1,Z1) :-
    triple(X1,"<http://___/cdao.owl#has_Character>",Z2),
    triple(Y1,"<http://___/cdao.owl#belongs_to_Character>",Z2),
    triple(Y1,"<http://___/cdao.owl#belongs_to_TU>", Z1).
```

does not provide the desired efficiency. For example, our parser took 30 minutes to process the study `S261` (12 MB in CDAO representation); the web-interface does not return the result after 1.5 hours. This indicates that a straightforward application of `dlvhex` features to simplify the amount of programming will not yield an acceptable result. We are planning to further experiment with `dlvhex` without using the web-interface.

The huge size of the CDAO files and the lack of an efficient interface between ASP and ontologies led to the use of the parser (using JAVA and the Jena framework) to generate facts from CDAO and store them in the CDAOStore.

As noted, the current size of the CDAOStore is about 5GB. Intuitively, any query listed in Section 4 could have been processed using this data. However, `clingo` cannot deal with file of 70 MB. We observed this during our experiment: whenever the amount of data is more than 70MB, a *killed* message is displayed and the computation is aborted. The *Query Analyzer* and *Triple Extractor* modules are developed to deal with this issue.

Limitations and Challenges. The previous discussion details some limitations of the current system. While it would be interesting whether the use of `dlvhex` will help us to

[3] `___` stands for `www.evolutionaryontology.org/cdao/1.0`.

eliminate the intermediate steps of the *Query Analyzer* and `Triple Extractor` modules, the critical limitation lies in the scalability of ASP-solver. As we have mentioned, `clingo` cannot yet deal with input larger than 70MB. Considering that in the current experiment, we only use data from 261 studies (around 1/10 of the total number of studies) and the necessary data could go up to 44MB, a full fledged implementation of PhyloWS using ASP will require additional techniques and/or better ASP-solvers. This also raises the question of whether other ASP extensions (e.g., DLVDB [20]) will provide a more scalable implementation.

7 Conclusion and Future Work

We described an ASP based implementation of PhyloWS, a web services API for phylogenetic applications. The implementation focuses on retrieval services, expressed by four different types of queries. We discussed the ASP implementation of the queries and evaluated with data from 261 studies extracted from TreeBASE. We detailed the design choices and discussed the limitation of the implementation that presents a challenge to the ASP community.

To continue with the development of PhyloWS, we plan to exploit the strengths of ASP to enrich PhyloWS with (*i*) constraints over the answers; and (*ii*) preferences between answers. We envision that this can be achieved via a web-interface that not only allows users to specify their queries but also the additional constraints and preferences. We plan to experiment with other ASP-extensions such as `dlvhex` or DLVDB to identify a more scalable system. In addition, we will also investigate whether different methods of computing answer sets (e.g., using reactive answer set solver) could be useful. Finally, we plan to complete the import of data from the 9558 CDAO files to CDAOStore.

References

1 T. Asano, J. Jansson, K. Sadakane, R. Uehara, and G. Valiente. Faster computation of the Robinson-Foulds distance between phylogenetic networks, 2010.

2 O. R. P. Bininda-Emonds. *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life,* Computational Biology Series, Vol. 4. Kluwer Academic Publisher, 2004.

3 B. Chisham, E. Pontelli, T. C. Son, and B. Wright. Cdaostore: A phylogenetic repository using logic programming and web services. In *Technical Communications of the 27th ICLP*, Vol. 11, *LIPIcs*, 209–219. 2011.

4 T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. ACM international conference on web intelligence. In *Web Intelligence*, pages 1073–1074. IEEE Computer Society, 2006.

5 H. Ellegren. Comparative genomics and the study of evolution by natural selection. *Molecular Ecology*, 17(21):4586–4596, 2008.

6 E. Erdem. PHYLO-ASP: Phylogenetic Systematics with Answer Set Programming. In *LPNMR*, 567–572. Springer, 2009.

7 E. Erdem, V. Lifschitz, and D. Ringe. Temporal phylogenetic networks and logic programming. *TPLP*, 6(5):539–558, 2006.

8 J. Felsenstein. The newick tree format, 1986. `http://evolution.genetics.washington.edu/phylip/newicktree.html`.

9 W. M. Fitch. Uses for evolutionary tree. *Phi. Trans. R. Soc. Lond. B*, 349:93–102, 1995.

10 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *LPNMR*, 260–265. Springer-Verlag, 2007.

11 H. Lapp and R. Vos. Phyloinformatics Web Services API: Overview. `https://www.nescent.org/wg/evoinfo/index.php?title=PhyloWS`, NESCE, 2009.

12 D. Maddison, D. Swofford, and W. Maddison. NEXUS: an Extensible File Format for Systematic Information. *Syst. Biol.*, 46(4):590–621, 1997.

13 V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, 375–398, 1999.

14 V. Morell. TreeBASE: the roots of phylogeny. *Science*, pages 273–569, 1996.

15 L. Nakhleh, D. Miranker, F. Barbancon, W. Piel, and M. Donoghue. Requirements of phylogenetic databases. In *3rd IEEE Symposium on Bioinf. and Bioeng.*, 141–148, 2003.

16 I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

17 F. Prosdocimi, B. Chisham, E. Pontelli, J.D. Thompson, and A. Stoltzfus. Initial implementation of a comparative data analysis ontology. *Evol. Bioinfor.*, 5:47–66, 2009.

18 M. Sanderson, B. G. Baldwin, G. Bharathan, C. S. Campbell, D. Ferguson, J. M. Porter, C. VonDohlen, M. F. Wojciechowski, and M. J. Donoghue. The growth of phylogenetic information and the need for a phylogenetic database. *Syst. Biol.*, 42:562–568, 1993.

19 A. Stoltzfus, N. Cellinese, K. Cranston, H. Lapp, S. McKay, E. Pontelli, and R. Vos. The evoio interop project. `http://www.evoio.org/wiki/Main_Page`, NESCE, 2009.

20 G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP*, 8(2):129–165, 2008.

21 R. Vos. `nexml`: Phylogenetic data in xml. `http://www.nexml.org`, 2008.

22 C. Webb, D. Ackerly, M. McPeek, and M. Donoghue. Phylogenies and communtiy ecology. *Annu. Rev. Ecol. Syst.*, 33(1), 2002.