EasyChair Preprint

№ 367

Phylotastic: An Experiment in Creating, Manipulating, and Evolving Phylogenetic Biology Workflows Using Logic Programming

Thanh Nguyen, Enrico Pontelli and Tran Cao Son

July 22, 2018

Phylotastic: An Experiment in Creating, Manipulating, and Evolving Phylogenetic Biology Workflows Using Logic Programming

THANH HAI NGUYEN, ENRICO PONTELLI, TRAN CAO SON

Department of Computer Science, New Mexico State University

(*e-mail:* thanhnh | epontell | @nmsu.edu, tson@cs.nmsu.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Evolutionary Biologists have long struggled with the challenge of developing analysis workflows in a flexible manner, thus facilitating the reuse of phylogenetic knowledge. An evolutionary biology workflow can be viewed as a plan which composes web services that can retrieve, manipulate, and produce phylogenetic trees. The Phylotastic project was launched two years ago as a collaboration between evolutionary biologists and computer scientists, with the goal of developing an open architecture to facilitate the creation of such analysis workflows. While composition of web services is a problem that has been extensively explored in the literature, including within the logic programming domain, the incarnation of the problem in Phylotastic provides a number of additional challenges. Along with the need to integrate preferences and formal ontologies in the description of the desired workflow, evolutionary biologists tend to construct workflows in an incremental manner, by successively refining the workflow, by indicating desired changes (e.g., exclusion of certain services, modifications of the desired output). This leads to the need of successive iterations of incremental replanning, to develop a new workflow that integrates the requested changes while minimizing the changes to the original workflow. This paper illustrates how Phylotastic has addressed the challenges of creating and refining phylogenetic analysis workflows using logic programming technology and how such solutions have been used within the general framework of the Phylotastic project.

KEYWORDS: Bioinformatics, workflows, web services, planning, composition, re-composition, similarity, quality of service

1 Introduction

A *phylogeny* (phylogenetic tree) is a representation of the evolutionary history of a set of entities—in the context of this work, we focus on phylogenies describing biological entities (e.g., organisms). Typically, a phylogeny is a branching diagram showing relationships between species, but phylogenies can be drawn for individual genes, for populations, or for other entities (e.g., non-biological applications of phylogenies include the study of evolution of languages). Phylogenies are built by analyzing specific properties of the species (i.e., *characters*), such as morphological traits (e.g., body shape, placement of bristles or shapes of cell walls), biochemical, behavioral or molecular features of species or other groups. In building a tree, species are organized into nested groups based on shared derived traits (traits different from those of the group's ancestor). Closely related species typically have fewer differences among the value of their characters, while less related species tend to have more. Currently, phylogenetic trees can be

either explicitly constructed (e.g., from a collection of descriptions of species), or extracted from repositories phylogenies, such as `OpenTree` [1] and `TreeBASE` [2]. In a phylogeny, the topology is the branching structure of the tree. It is of biological significance, because it indicates patterns of relatedness among `taxa`, meaning that trees with the same topology provide the same biological interpretation. Branches show the path of transmission from one generation to the next. Branch lengths indicate genetic change, i.e., the longer the branch, the more genetic change (or divergence) has occurred. A variety of methods have been devised to estimate a phylogeny from the traits of the taxa (e.g., (Penny 2004)).

Phylogenies are useful in all areas of biology, to provide a hierarchical framework for classification and for process-based models that allow scientists to make robust inferences from comparisons of evolved things. A standing dream in the field of evolutionary biology is the assembling a *Tree of Life (ToL),* a phylogeny broadly covering some 10^7 species (Mora et al. 2011; Cracraft et al. 2002). The first draft of a grand phylogenetic synthesis—a single tree with perhaps 2.5×10^6 species—is emerging from the Open Tree of Life (OpenTree) project. Yet, the current state of the ToL is a collection of trees, and there are various reasons to expect that this situation will persist. When we refer to "the ToL" here, we mean the dispersed set of available species trees, with a strong focus on larger trees from reputable published sources (e.g., (Bininda-Emonds et al. 2007; Jetz et al. 2012; Smith et al. 2011)).

While experts continue expanding, consolidating, and improving the ToL, our motivation is to put this expert knowledge into the hands of everyone: ordinary researchers, educators, students, and the public. To achieve this, we launched the *Phylotastic* project, aimed at building a community-sustainable architecture to support flexible on-the-fly delivery of expert phylogenetic knowledge. The premise of disseminating knowledge is that it will be re-used. How do trees get re-used? On a per-tree basis, re-use is rare—most trees are inferred *de novo* for a specific study, rendered as images in a published report, stored on someone's hard drive, and (apparently) not used again (Stoltzfus et al. 2012). Yet, large species trees are re-used in ways that small species trees (and sequence-family trees) are not. In a sample of 40 phylogeny articles, Stoltzfus et al. (2012) found 6 cases in which scientists obtained a custom tree by extraction from a larger species tree. These studies implicate diverse uses: functional analyses of leaf traits or lactation traits; phylogenetic diversity of forest patches; analyzing niche-diversity correlations, spatial distribution of wood traits, and spatial patterns of diversity. The implicated trees include those covering $4,510$ extant mammals, $55,473$ angiosperm species, and $1,566$ angiosperm taxa.

The Phylotastic project offers a solution to the reuse of phylogenetic knowledge problem, by adopting a web services composition approach. The *phyloinformatic* community has been very active in the development of a diversity of data repositories and software tools, to collect and analyze artifacts relevant to evolutionary analysis. These tools are sufficient to realize all of the studies in the previously mentioned papers, when properly integrated in a coherent workflow. Furthermore, the analysis protocols adopted in such studies are often the result of an iterative process, where the protocol is successively refined to better suit the available data and produce results of adequate quality (Stoltzfus et al. 2012).

In this paper, we describe the infrastructure used by Phylotastic to achieve the following goals: **(1)** Allow a user to provide the available knowledge about the desired protocol (e.g., input, type of output, selected classes of operations that should be used); **(2)** Derive a collection of workflows

[1] `http://tree.opentreeoflife.org`
[2] `https://treebase.org`

that satisfy the desired conditions, through a web service discovery and composition process; **(3)** Allow the user to suggest manipulations of a chosen workflow (e.g., exclusion of a service, addition of another output); **(4)** Determine new workflows that satisfy the requested manipulations while maintaining maximum similarity with the chosen workflow. All the manipulations of the workflows—i.e., composition of web services, modification of workflows, computation of similarity between workflows—are realized in Answer Set Programming (ASP). ASP provides a clear advantage, allowing the simple integration of different forms of knowledge (e.g., ontologies describing services and artifacts, user preferences) and facilitating the encoding of the composition and re-composition problems as ASP planning problems. We present the overall structure of the Phylotastic architecture, describe how web service composition is encoded in ASP, and analyze how workflow refinements are achieved. We also demonstrate various aforementioned features through a use-case.

2 Background: The Phylotastic Project and Its Implementation

2.1 Architecture

The Phylotastic project proposes a flexible system for on-the-fly delivery of custom phylogenetic trees that would support many kinds of tree re-use, and be open for both users and data providers. Phylotastic proposes an open architecture, composed of a collection of *web services* relevant to creation, storage, and reuse of phylogenetic knowledge, that can be assembled in user-defined workflows through a Web portal and mobile applications (Fig. 1).

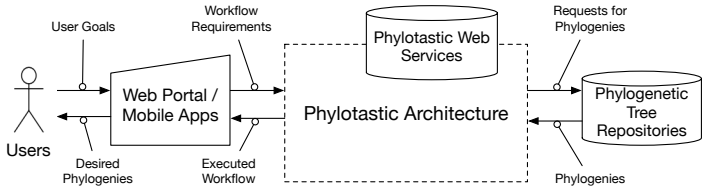

Fig. 1: Overall Phylotastic Structure

Figure 2 shows an overview of our web service composition framework for Phylotastic. It consists of a web service registry, an ontology, a planning engine, a web service execution monitoring system, and a workflow description tool. The flow of execution of the architecture starts with the *workflow description tool*—a graphical user interface that allows the user to provide information about the desired requirements of the phylogenetic trees generation or extraction process. The information collected from the user interface are mapped to the components of a planning problem instance, that will drive the web service composition process. The planning problem instance representing the web service composition problem is obtained by integrating the user goals with the description of web services, obtained from the service registry and the ontology. The planning engine is responsible for deriving an executable workflow, which will be enacted and monitored by a web service execution system. The final outcome of the service composition and execution is presented to the user using the same workflow description tool.

Thus, the components of the Phylotastic web service composition framework are:

- *The Phylotastic Ontology*: this ontology is composed of two parts: an ontology that describes the artifacts manipulated by the services (e.g., alignment matrices, phylogenetic trees, species names) (the CDAO ontology (Prosdocimi et al. 2009)) and an ontology that describes the

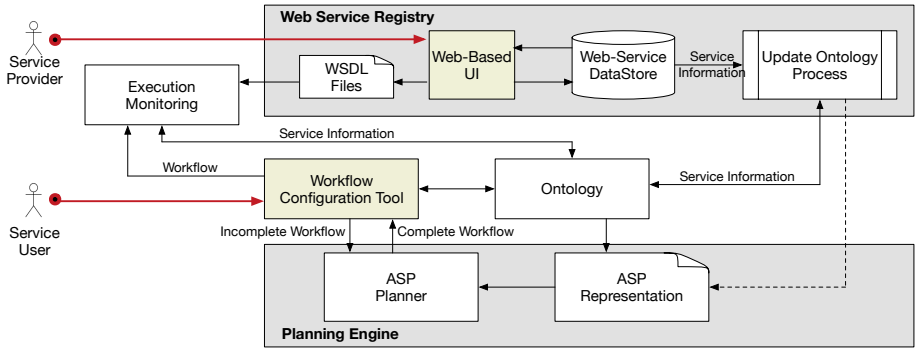

Fig. 2: The Phylotastic Web Service Composition Framework

actual operations and transformations performed by the services. Each class of services is associated with a name, inputs, parameters, and outputs. Instances of the services will also be associated with the data formats of their inputs, outputs, and parameters. For example, a service in the class *taxon_based_ext* takes *bio_taxa* as an input and produces outputs *species_tree* and *http_code*. An instance of this class is *get_PhyloTree_OT_V1* whose input (*bio_taxa*) has the *list_of_strings* format and its outputs (*species_tree*, *http_code*) have the *newickTree* and *integer* format, respectively. This information can be encoded in ASP as follows:

```
op_cl(tree_ext). op_cl(taxon_based_ext). op(get_PhyloTree_OT_V1).
subcl(taxon_based_ext,tree_ext). cl(bio_taxa). cl(species_tree).
t_of(get_PhyloTree_OT_V1,taxon_based_ext).
has_input(taxon_based_ext,set_of_names_1,bio_taxa).
has_output(taxon_based_ext,phylo_tree_1,species_tree).
has_output(taxon_based_ext,http_code_1,http_code).
has_inp_df(get_PhyloTree_OT_V1,bio_taxa,list_of_strings).
has_out_df(get_PhyloTree_OT_V1,species_tree,newickTree).
has_out_df(get_PhyloTree_OT_V1,http_code,integer).
```

- *Web Service Composition as Planning*: In Phylotastic, we adopt the view, advocated by several researchers, of mapping the web service composition problem to a *planning problem* (Carman et al. 2004; McIlraith et al. 2001; Peer 2005). In this perspective, available web services are viewed as *actions* (or *operations*) that can be performed by an agent, and the problem of determining the overall workflow can be reduced to a planning problem. In general, a planning problem can be described as a five-tuple (S, S_0, G, A, I'), where S is set of all possible states of the world, $S_0 \subseteq S$ denotes the initial state(s) of the world, $G \subseteq S$ denotes the goal states the planning system attempts to reach, A is the set of actions to change one state of the world to another state, and the transition relation $I' \subseteq S \times A \times S$ defines the precondition and effects for the execution of each action. In term of web services, S_0 and G are the initial state and the goal state specified in the requirement of web service requesters (e.g., the available input and the desired output of the workflow). The set A is a set of available services; I' describes the effect of the execution of each service (e.g., data produced).
- *Web Service Engine*: The planning engine implemented in the Phylotastic project employs Answer Set Planning (ASP) (Lifschitz 2002) and is responsible for creating an executable workflow from the incomplete workflow and/or from the user specifications. The basic planning algorithm has been discussed by Nguyen et al. (2018). This engine differs from the usual ASP-planning system in that it uses a two-stage process in computing solutions. In the first

stage, planning is done at the abstract-level and the engine considers only service classes, matching their inputs and outputs. The result is a workflow whose elements are web services described at the abstract level. The second stage instantiates the abstract web services from the first stage with concrete services. In the process, it might need to solve another planning problem, to address the issues of mismatches between formats of different services; for example, there are several concrete services to recognize gene names and their outputs are sets of scientific names of the genes; however, they are saved in different formats. This program consists of following ASP-rules encoding the operations and the initial state. The key rules are

- `ext(I,0) :- initially(I,DF`$_I$`).` This rule states that the resource `I` with data format `DF`$_I$ exists at the time moment 0.
- The rule encoding the operations and their executability are as follows:

```
{executable(A,T)} :- op_cl(A).
:- executable(A,T), has_input(A,N,I), not match(A, I, T).
p_m(A,I,T,O,T1) :- op_cl(A), has_input(A,N,I), T1≤T,
                   ext(O,T1),subcl(O,I).
1 {map(A,I,T,O,T1) : p_m(A,I,T,O,T1)}1.
match(A,I,T) :- map(A,I,T,_,_).
```

- The next rules are used to generation operation occurrences:

```
1 { occ(A,T): op_cl(A) } 1.
:- occ(A,T), not executable(A,T).
ext(O,T+1) :- occ(A,T), has_output(A,N,O).
```

In all the rules, `T` or `T1` denotes a time step. From now on, we will denote with Π_L the ASP-program for web service composition in the Phylotastic project.

3 Selecting The Most Preferred Workflow: A Qualitative Approach

Currently, Π_L receives from the Workflow Configuration Tool a description of the desired workflow, e.g., desired input, desired output, specific classes of operations that should occur in the workflow. Using this information, it generates a concrete workflow meeting the desired requirements, sends it to the execution monitoring component which will execute the workflow and output the desired phylogenetic tree. Frequently, there are several ways to construct a tree given the input, i.e., there are several solutions that Π_L could return. Due to the differences in web services, not all solutions will produce the same result at execution (e.g., produce the same phylogeny), e.g., because of lack of agreement on the evolutionary relationships between certain species or the ambiguity of certain species names. In this paper, we present two enhancements of the system. In both enhancements, the notion of a *most preferred workflow* is defined and users can interact with the system to select their most preferred workflow(s).

One way to compare the workflows is to rely on the notion of *quality of service (QoS)* of web services. QoS of a web service can be used as a discriminating factor that differentiates functionally similar web services. In general, QoS of a web service is characterized by several attributes, such as performance, reliability, scalability, accuracy, integrity, availability, and accessibility (Rajendran and Balasubramanie 2009). For the web services used in the Phylotastic project, we collect the following attributes that influence the performance of a web service: **(1)** *response time*, **(2)** *throughput*, **(3)** *availability*, **(4)** *reliability*. Specifically,

- *Response time*: Given a service s, the response time $q_{rt}(s)$ measures the delay in seconds between the moment when a request is sent and the moment when the results are received.

- *Throughput*: $q_{tp}(s)$ is the average number of successful responses for a give period of time.
- *Availability*: The availability $q_{av}(s)$ of a service s is the probability that the service is accessible for a given period of time. The value of the availability of a service s is computed using the following expression $q_{av}(s) = T_a(s)/\theta$, where T_a is the total amount of time (in seconds) in which service s is available during the last θ seconds (θ is a constant set by an administrator of the service community).
- *Reliability*: The reliability $q_{re}(s)$ is the average operation time of service s in which service s is accessible and processes clients requests successfully. It is measured by total operation time of service s divided by the number of failures.

The quality vector of a service s is denoted by the tuple $q(s) = (q_{rt}(s), q_{tp}(s), q_{av}(s), q_{re}(s))$. For the web services in the Phylotastic project, this information is maintained in the Service Registry (along with the ontology-based description of each service). We next define the QoS of a workflow based on the QoS of the web services.

3.1 QoS of Workflows

Let $p = (s_1, s_2, ..., s_n)$ be a workflow of web services. The quality of services of p, denoted by $q(p)$, is defined by $q(p) = (q_{rt}(p), q_{tp}(p), q_{av}(p), q_{re}(p))$ where

- *Response time*: The response time $q_{rt}(p)$ is defined as total response time of all services in the workflow p: $q_{rt}(p) = \sum_{i=1}^{n} q_{rt}(s_i)$.
- *Throughput*: The throughput $q_{tp}(p)$ of plan p is the average of the throughputs of the services that participate in p: $q_{tp}(p) = \frac{\sum_{i=1}^{n} q_{tp}(s_i)}{n}$.
- *Availability*: In general, the availability of the services for p should be defined as $q_{av}(p) = \prod_{i=1}^{n} Pr(s_i \mid s_1, \ldots, s_{i-1})$ where $Pr(s_i \mid s_1, \ldots, s_{i-1})$ is the conditional probability of s_i is available given that $s_1, \ldots, s_{i-1}$ have been successfully executed. For simplicity of representation, we assume that the services are mutually independent, then $q_{av}(p) = \prod_{i=1}^{n} q_{av}(s_i)$. In our current implementation, we use $q_{av}(p) = \min\{q_{av}(s_i) \mid i = 1, \ldots, n\}$, as an approximation which avoids extensive floating point operations.
- *Reliability*: The reliability $q_{re}(p)$ is calculated as the average of reliability values of element services in p: $q_{re}(p) = \frac{\sum_{i=1}^{n} q_{re}(s_i)}{n}$.

3.2 ASP Encoding of QoS

The QoS of a workflow can be computed using ASP as follows. As with the actions used in the web composition process, we extract the QoS information of services and represent it as ASP-facts of the forms: $has\_qos\_rt(s,v), has\_qos\_av(s,v), has\_qos\_tp(s,v), has\_qos\_re(s,v)$ where s is the service identifier, and v is the QoS value of the corresponding attribute. The computations of QoS attributes for a workflow are encoded as following:

$$
\begin{array}{lll}
qos\_rt\_wf(RT_W) & :- & RT_W = \#sum\{RT, T, X : occ\_concrete(X,T), has\_qos\_rt(X,RT)\}. \\
qos\_tp\_wf(TP_S/nsteps) & :- & TP_S = \#sum\{TP, T, X : occ\_concrete(X,T), has\_qos\_tp(X,TP)\}. \\
qos\_re\_wf(RE_S/nsteps) & :- & RE_S = \#sum\{RE, T, X : occ\_concrete(X,T), has\_qos\_re(X,RE)\}. \\
qos\_av\_wf(AV_W) & :- & AV_W = \#min\{AV, T, X : occ\_concrete(X,T), has\_qos\_av(X,AV)\}.
\end{array}
$$

In the above rules, $nsteps$ is the number of services in the plan that is computed by the planning module. Since the QoS of a service (or a workflow) is a tuple of values representing different attributes, there are different ways for comparing services (or workflows). Different users might

have different preferences over these attributes (e.g., response time is the most important factor, or reliability is the most important factor, etc.). We discuss two possibilities:

- *Weighted QoS*: A user specifies the weights W_{rt}, W_{tp}, W_{av}, and W_{re} that he/she would like to assign for the response time, the throughput, the availability, and the reliability, respectively. The weighted QoS of a plan p is then computed by $w(p) = q_{rt}(p) * W_{rt} + q_{tp}(p) * W_{tp} + q_{av}(p) * W_{av} + q_{re}(p) * W_{re}$. Under this view, $w(p)$ can be computed as follows:

  $$\begin{aligned} score\_qos\_wf(Sc) :- \; & qos\_rt\_wf(RT_W), wei\_rt(W_{rt}), qos\_tp\_wf(TP_W), wei\_tp(W_{tp}), \\ & qos\_re\_wf(RE_W), wei\_re(W_{re}), qos\_av\_wf(AV_W), wei\_av(W_{av}), \\ & Sc = RT_W * W_{rt} + TP_W * W_{tp} + RE_W * W_{re} + AV_W * W_{av}. \end{aligned}$$

  To select workflows with the best QoS, we will only need to add the statement

  $$\#maximize\{ScoreQoS : score\_qos\_wf(ScoreQoS)\}.$$

- *Specified Preferences QoS*: An alternative to the weighted QoS is to allow users to specify a partial ordering over the set of attributes that will be used in identifying most preferred workflows by a lexical ordering in accordance to the preferences. For example, assume that the preference ordering is $x_1 > x_2 > x_3 > x_4$ where $x_i > x_j$ means that attribute x_i is preferred to the attribute $x_i \neq x_j$ and $x_i \in \{rt, av, tp, re\}$. As the values in the QoS of a service behave differently, we write $q_x(s) \prec q_x(s')$ to denote that s is better than s' w.r.t. the attribute x. The most preferred workflow is defined via a lexicographic ordering: $p \prec p'$ if there is $1 \leq i \leq 4$ such that $q_{x_j}(p) = q_{x_j}(p')$ for $j < i$ and $q_{x_i}(p) \prec q_{x_i}(p')$. This can easily be implemented using the priority level in **clingo**.

  $\#maximize\ \{q_{x_1}@4\}.\ \ \#maximize\ \{q_{x_2}@3\}.\ \ \#maximize\ \{q_{x_3}@2\}.\ \ \#maximize\ \{q_{x_4}@1\}.$

  For a concrete example, assume that the preference ordering is $rt > re > tp > av$, the corresponding ASP encoding is as follow:

  $$\begin{aligned} & \#maximize\{RT_W@4 : qos\_rt\_wf(RT_W)\}. \quad \#maximize\{RE_W@3 : qos\_re\_wf(RE_W)\}. \\ & \#maximize\{TP_W@2 : qos\_tp\_wf(TP_W)\}. \quad \#maximize\{AV_W@1 : qos\_av\_wf(AV_W)\}. \end{aligned}$$

The above feature is implemented in the Phylotastic system. In either case, we can display a set of workflows for the users to decide which workflow should be executed.

4 Refining a Workflow

Evidence that emerged in the Phylotastic project as described by A. Stoltzfus et al. (2013) shows that evolutionary biologists tend to develop their analysis protocols in an incremental manner, through successive refinements, often driven by the specific properties of the dataset being processed and the opportunities revealed by intermediate results. Users of the Phylotastic system also often have strong preferences about certain type of services that they want (or do not want) to use. For this reason, the Workflow Configuration Tool allows a user to update a given workflow and resubmit it to the ASP-planner. Presently, the ASP-planner considers this as a new request and restarts the computation. This approach is simple but also has some drawbacks. First, the new workflow and the original workflow can be very different in the services that they use, which is often unexpected to the user (and undesired, since changing services might lead to a different phylogeny). Second, this approach can be computational expensive as it cannot reuse the original workflow. We propose an approach to address the changes requested by a user that can preserve *as much as possible the original workflow*. We focus on the following four categories of changes:

- *IO request*: Request to change input and/or output.
- *Avoidance request*: Avoid using one class of services.
- *Inclusion request*: Request that a particular service to be used in the workflow.
- *Insertion request*: Request that a service is inserted at a particular position.

4.1 Similarity Between Workflows: Formalization

Given two workflow p and p', we define the concept of *similarity* between p to p'. In the following, we view a workflow as a directed acyclic graph $G = (V,E)$, where V is the set of nodes and each node is a service; E is the set of edges and each edge is an exchange of resources between two services in the workflow. Observe that each service v will have a set of inputs, a set of outputs, and some description. For each service v, $input(v)$ and $output(v)$ denote the set of inputs and outputs of v, respectively. The *similarity between two workflows* is defined as a combination of *nodes similarity*, *edges similarity*, and *contextual and topology similarity* (Becker and Laue 2012; Antunes et al. 2015). Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. The similarity between G_1 and G_2, denoted by `sim_workflows` (G_1, G_2), is defined next.

- **Node Similarity.** Let $v_1 \in V_1$ and $v_2 \in V_2$. We fist define the similarity between two nodes and then use this measure to define the similarity between nodes of workflows. As a node represent a web service, the similarity of two nodes can be determined based on their mutual position in the ontology (note that the Phylotastic ontology classifies services based on a taxonomy of classes of services). Thus, two nodes can be considered to be similar if they are close in the ontology, share the same inputs, outputs, or have similar descriptions. These features are considered in the following definitions.
  - `sim_nodes_onto` (v_1, v_2): measures the similarity between v_1 and v_2 by considering them as nodes in the ontology:
    $sim\_nodes\_onto(v_1, v_2) = \frac{1}{1+d\_nodes\_onto(v_1,v_2)}$, where
    $d\_nodes\_onto(v_1, v_2) = path\_len(lca(v_1, v_2), v_1) + path\_len(lca(v_1, v_2), v_2)$. Intuitively, the similarity between two nodes in an ontology is disproportional to the distance ($path\_len(.)$) between their lowest common ancestors ($lca(.)$) of them.
  - `sim_nodes_inp` (v_1, v_2): measures the similarity the nodes by considering their inputs, the more inputs they share the more similiar they are. Thus,

$$sim\_nodes\_inp(v_1, v_2) = 2 * \frac{|input(v_1) \cap input(v_2)|}{|input(v_1)| + |input(v_2)|}.$$

  - `sim_nodes_oup` (v_1, v_2): measures the similarity between two nodes by considering their outputs, computed similarly to `sim_nodes_oup` (v_1, v_2). So,

$$sim\_nodes\_oup(v_1, v_2) = 2 * \frac{|ouput(v_1) \cap output(v_2)|}{|output(v_1)| + |output(v_2)|}.$$

  - `sim_nodes_des` (v_1, v_2): measures the similarity between the English descriptions of the two nodes. We use off-the-shelf libraries `Stanford CoreNLP` [3], `NLTK`[4], and `Scikit-Learn`[5] to process the descriptions and transform these text descriptions to a ma-

[3] `https://stanfordnlp.github.io/CoreNLP`
[4] `http://www.nltk.org/`
[5] `http://scikit-learn.org/`

trix of TF-IDF (term frequency-inverse document frequency) of the two documents. From each document we derive a real-value TF-IDF vector; the similarity index between two text descriptions is computed based on cosine similarity between their TF-IDF vectors.

The similarity between two nodes, denoted by **sim_nodes(v_1, v_2)**, is then defined as the weighted sum between their four similarity measures

$$\begin{aligned} sim\_nodes(v_1,v_2) = \quad & w_{onto} * sim\_nodes\_onto(v_1,v_2) + w_{inp} * sim\_nodes\_inp(v_1,v_2) + \\ & w_{oup} * sim\_nodes\_oup(v_1,v_2) + w_{des} * sim\_nodes\_des(v_1,v_2) \end{aligned}$$

where $w_{onto}, w_{inp}, w_{oup}, w_{des}$ are weight values assigned to each attribute, $w_x \in [0,1]$ ($x \in \{onto, inp, oup, des\}$) and $w_{onto} + w_{inp} + w_{oup} + w_{des} = 1$. In our implementation, the values of $w_{onto}, w_{inp}, w_{oup}, w_{des}$ are $0.6, 0.15, 0.15$ and 0.1 respectively. The intuition behind these values lies in the fact that within an ontology, the similarity between objects depends heavily on their relative position to their lowest common ancestor; thus w_{onto} should play the deciding factor. $w_{inp} = w_{oup}$ because of the symmetry between inputs and outputs. w_{des} is smaller than w_{inp} or w_{oup} because our current ontology has different levels of detail in the English description of services.

Finally, **sim_nodes_workflows(G_1, G_2)** is defined as follows:

$$sim\_nodes\_workflows(G_1,G_2) = 2 * \frac{\sum_{v_1 \in V_1} \sum_{v_2 \in V_2} sim\_nodes(v_1,v_2)}{|V_1| + |V_2|} \tag{1}$$

- **Edge Similarity.** Let $e_1 \in E_1$ and $e_2 \in E_2$. As an edge connected two services (nodes) in a workflow and denotes an exchange between two nodes (output of one is input of the other). For this reason, the similarity between two edges can be defined via the similarity between the nodes relating to them and the descriptions of their inputs and outputs. For an edge e, let $s(e)$ and $d(e)$ denote the source and destination of e, respectively; Furthermore, let $lab(e) = (o_{s(e)}, i_{d(e)})$, where $o_{s(e)}$ is the output of $s(e)$ that is used as the input $i_{d(e)}$ of $d(e)$. We define:
  - `sim_ed_nod(`e_1, e_2`)` : measures the similarity of two edges by considering the similarity of the nodes related to the edges and is defined by $sim\_ed\_nod(e_1,e_2) = \frac{1}{2} * (sim\_nodes(s(e_1), s(e_2)) + sim\_nodes(d(e_1), d(e_2)))$.
  - `sim_ed_re(`e_1, e_2`)` : measures the similarity of two edges by considering distance between their labels and is defined by $sim\_ed\_re(e_1,e_2) = \frac{1}{1 + d\_ed\_ont(lab(e_1), lab(e_2))}$, where $d\_ed\_ont(lab(e_1), lab(e_2)) = \frac{1}{2} * (d\_nodes\_onto(o_{s(e_1)}, o_{s(e_2)}) + d\_nodes\_onto(i_{d(e_1)}, i_{d(e_2)}))$.

  The similarity between two edges, denoted by **sim_edges(e_1, e_2)**, is then defined as the weighted sum between their two similarity measures. $sim\_edges(e_1,e_2) = w_{node} * sim\_ed\_nod(e_1,e_2) + w_{label} * sim\_ed\_re(e_1,e_2)$. where $0 \leq w_{node}, w_{label} \leq 1$ are weight values of corresponding attributes contributions such that $w_{node} + w_{label} = 1$. In our implementation, we assign the values of w_{node}, w_{label} are 0.5 and 0.5 respectively. We define

$$sim\_edges\_workflows(G_1,G_2) = 2 * \frac{\sum_{e_1 \in E_1} \sum_{e_2 \in E_2} sim\_edges(e_1,e_2)}{|E_1| + |E_2|}.$$

- **Topological Similarity.** By considering workflows as graphs, we can also consider their similarity based on the amount of changes needed to convert one to the other. The notion of an *Edit-Distance*, denoted by $dist\_topo(G_1, G_2)$, between two graphs—the smallest number of changes (insertions, deletions, substitutions, etc.) required to transform one structure to another—has been introduced and algorithms for computing it have been developed

by Zhang and Shasha (1989). The topological similarity between two graphs is defined by $sim\_topo(G_1,G_2) = \frac{1}{1+dist\_topo(G_1,G_2)}$.

Having defined various types of similarities between elements of the workflows, we can now define the similarity between two workflows as a weighted sum of these similarities:

$$\begin{aligned} sim\_workflows(G_1,G_2) = \quad & w_{no} * sim\_nodes\_workflows(G_1,G_2) + \\ & w_{ed} * sim\_edges\_wf(G_1,G_2) + w_{to} * sim\_topo(G_1,G_2) \end{aligned}$$

where $w_{no}, w_{ed}, w_{to} \in [0,1]$ and $w_{no} + w_{ed} + w_{to} = 1$. In our implementation, we use $w_{no} = 0.45, w_{ed} = 0.35$ and $w_{to} = 0.2$. Here, the emphasis is still the similarity between nodes. This is because the nodes are the main components of a workflow. For this reason, we place greater emphasis on the edges than the topology of a workflow, because the labels of the edges are also critical to the workflow. Due to the fact that the computation of the similarity between two workflows is deterministic, deals mostly with real numbers, and the fact that new answer set solvers allow for the integration of external atoms as described by (Kaminski et al. 2017), we implemented a package for computing the similarity between two workflows as a Python library and used this as external predicates in ASP.

4.2 ASP-Code for Replanning

Given a workflow and some modifications requested by a user, the similarity measure introduced in the previous subsection can be used to select the workflow that satisfies the user's requests and is most similar to the original one. We next present the ASP implementation addressing each of the change categories discussed at the beginning of this section and then selecting the *most similar workflow*. We will use a simple original workflow for generating a *gene-species reconciliation tree* from *a set of gene names* (Figure 3) as a running example. In this workflow, each node (circle) represents a service (e.g., *a* is a service) and each connection between two nodes represent an edge with its label (e.g., the edge from *a* to *b* has *oa* as an output of *a* which is used as input *ib* of *b*). Here, *a*, *b*, *c*, *d*, *e* and *f* are the short name for the services *Get_GeneTree_from_Genes, Ext_Species_from_GeneTree, Resolved_Names_OT, Get_PhyloTree_OT_V1, GeneTree_Scaling_V1* and *Get_ReconciliationTree* respectively.

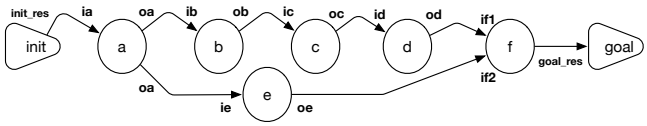

Fig. 3: Original workflow

Intuitively, the workflow in Figure 3 represents a workflow generated by the planning engine, encoded by the set of atoms:

$$\begin{aligned} \{&initially(init\_res, dfi), occ(a,1), occ(b,2), occ(c,3), occ(d,4), occ(e,5), occ(f,6), \\ &finally(goal\_res, dfg), map(a,ia,1,init\_res,0), map(b,ib,2,oa,2), map(c,ic,3,ob,3), \\ &map(d,id,4,oc,4), map(e,ie,5,oa,2), map(f,if1,6,od,5), map(f,if2,6,oe,6).\} \end{aligned}$$

In this encoding, $initially/2$ ($finally/2$) states that the input (goal) with its data type; $occ(x,i)$ states that the service x must occur at the step i; $map(s,i,t1,o,t2)$ says that an output o of step $t2$ is an input i of service s at step $t1$.

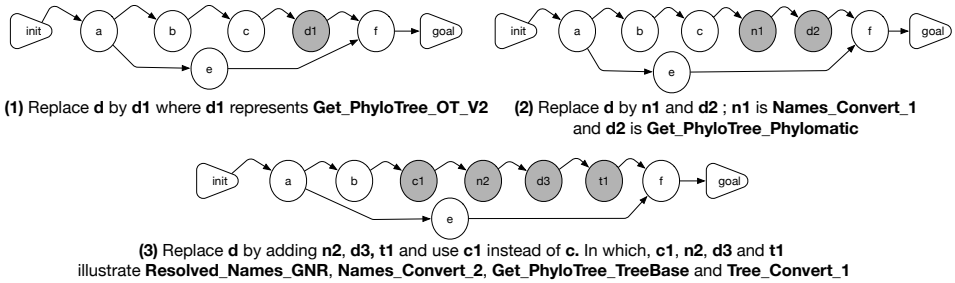

Fig. 4: Some updated workflows with avoidance the service d

4.2.1 IO Request

For an IO request, all that needs to be changed is the ASP encoding sent to the ASP-planner, specifically atoms of the form `initially` and/or `finally` will be updated to reflect the request. The planning engine will be executed and returned the most similar workflow to the current one.

4.2.2 Avoidance Request

A request to avoid using a service s (or a class of services c) will be translated to the atom $do\_not\_use(s)$ ($do\_not\_use(c)$) and supplied to the planning engine. We use the following ASP-rules to enforce this request:

$$is\_used(C) :- \ member(X,C), occ(X,\_). \tag{2}$$

$$is\_used(X) :- \ occ(X,\_). \tag{3}$$

$$:- \ is\_used(X), do\_not\_use(X). \tag{4}$$

The first two rules determine the class of service `C` or the service X is used in the workflow. The third enforces the request of the user to avoid the use of the service or a class of services. Some possible resulting workflows satisfying the request $do\_not\_use(d)$ are shown in Figure 4.

4.2.3 Inclusion Request

A request to include a service s (or a class of services c) will be translated to the atom $must\_used(s)$ ($used(c)$) and supplied to the planning engine. The rules (2)-(4) are used with the rule:

$$:- \ must\_used(X), not \ is\_used(X). \tag{5}$$

to make sure that the request to include some service is satisfied. Figure 5 displays some possible updated workflows with the request of using **(1)** `e2` (*GeneTree_Scaling_V2*) or **(2)** `e3` (*GeneTree_Scaling_V3*)

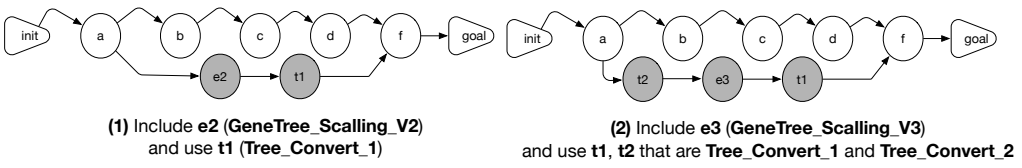

Fig. 5: Updated workflows with inclusion request of `e2` or `e3`

4.2.4 Insertion Request

This is a special case of an inclusion request. Specifically, it specifies where the service should be included. This request is translated into the ASP atoms of the form $before(x,y)$ (service x must be executed before service y) and $must\_used(x)$. To implement this, we add to the before but also some after statements

$$is\_before(C,D) :- \ occ(C,T), occ(D,T_1), T < T_1. \tag{6}$$

$$:- \ before(C,D), not \ is\_before(C,D). \tag{7}$$

Figure 6 shows some workflows accommodating the request "*use service e after a and before b*", encoded by $\{must\_used(e), before(a,e), before(e,c).\}$.

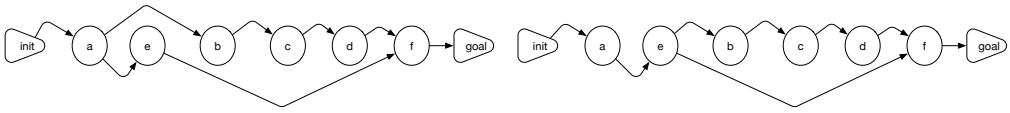

Fig. 6: Service e is executed after a and before c

4.2.5 Selecting a Most Similar Workflow

Let Π_R be the set of rules (2)-(7) for enforcing the requests of the users and C denote the set of atoms encoding the requests of an user. Our goal is to compute a new workflow that satisfies C and is as similar as possible to the original workflow. It is easy to see that $\Pi_L \cup \Pi_R \cup C$ will return workflows satisfying C. As such, we only need to identify, among all solutions provided by $\Pi_L \cup \Pi_R \cup C$, the workflow that is most similar to the original workflow. This can be achieved by encoding the original workflow, providing it as input, and exploiting the Python package for computing the similarity of two workflows (subsection 4.1). To do so, let us assume that the original workflow is encoded as a set of atoms O of the form $old\_occ(s,t)$ and $old\_map(s,i,t_1,o,t_2)$. There are different possibilities here:

- **Computing exact value of similarity using ASP**: theoretically, the exact value of similarity between the constructed workflow and the original one can be computed in the ASP using an external call to $sim\_workflows$ and the most similar workflow can be computed using the `maximize` statement by

$$sim(V) :- \ V = @sim\_workflows(wf_1, wf_2). \tag{8}$$

$$\texttt{\#maximize}\{V : sim(V)\}. \tag{9}$$

  where wf_1 and wf_2 are two sets of atoms encoding the two workflows (the original and the computed one), this is essentially the sets of atoms of the forms $old\_occ(.)$, $old\_map(.)$, $occ()$, and $map()$ that is generated by the planning engine and supplied in O. This means that we need a set data structure to implement this approach. We did not implement this due to the fact that **clingo** does not yet provide a construct for set. We hope to work with the **clingo** group to introduce set as a basic data type for use besides the aggregate functions.
- **Approximating the value of similarity using ASP**: Since the set of facts of the form $occ(s,t)$ correspond to the nodes in the workflow, we can approximate the similarity of two workflows by considering only the similarity between nodes of the two workflows (old and new). This

can be realized by the following rules:

$$\begin{aligned} single\_sim\_nodes(X,Y,Z) :- \;& old\_occ(X,I_1), occ(Y,I_2), Z = @sim\_nodes(X,I_1,Y,I_2). \\ sum\_sim\_nodes(S) :- \;& S = \#sum\{Z,X,Y : single\_sim\_nodes(X,Y,Z)\}. \\ sim\_nodes\_workflows(R) :- \;& sum\_sim\_nodes(S), NO = \#count\{X,I_1 : old\_occ(X,I_1)\}, \\ & NN = \#count\{Y,I_2 : occ(Y,I_2)\}, R = 2 * S/(NO+NN). \\ & \#\texttt{maximize}\{R : sim\_nodes\_workflows(R)\}. \end{aligned}$$

It is easy to see that the above rules implement formula (1).

- **Computing exact value of similarity using multi-shot ASP**: This approach has been implemented using the multi-shot ASP (Kaminski et al. 2017). Basically, a Python wrapper is used to control the search for the most similar workflow to the original one. It implements the following loop:

```
for each answer set of Π_L ∪ Π_R ∪ C
  compute the similarity ν of the solution and the original workflow
  if it is greater than the current value (initiated with -1)
  then keep the solution
```

4.3 Refining a Workflow: Case-Studies

We illustrate the new features of our system using three use cases developed in the Phylotastic project.

4.3.1 From a Set of Gene Names to a Reconciliation Tree in `newick` Format

In this use case, the user wants to generate a phylogenetic reconciliation tree in the `newick` format from a set of gene names, whose format is `set_of_strings`. The user can use the *Workflow Configuration Module* to design an initial workflow with two nodes (initial node and goal node) with this information as described by Nguyen et al. (2018). The module will then convert this information into ASP-atoms `initially(setOfGeneNames,set_of_strings)` and `finally(reconciliationTree,newickTree)`, which will be sent the planning engine. The result is the workflow shown in Figure 7, where the triangles identify the input and output and the name of the service that should be executed at each step. For example, a `gene_based_extraction` service should be executed at step 1 and a `names_extraction_tree` service needs to be executed at step 2. The workflow shown in the figure is also the one with highest QoS (1.4224).

Fig. 7: Original Workflow

The user, after examining the workflow, determines that the input `GeneTree` of `tree_reconciliation` service needs to be scaled before being processed by `tree_reconciliation`; and, this requires that the service `gene_tree_scaling` should

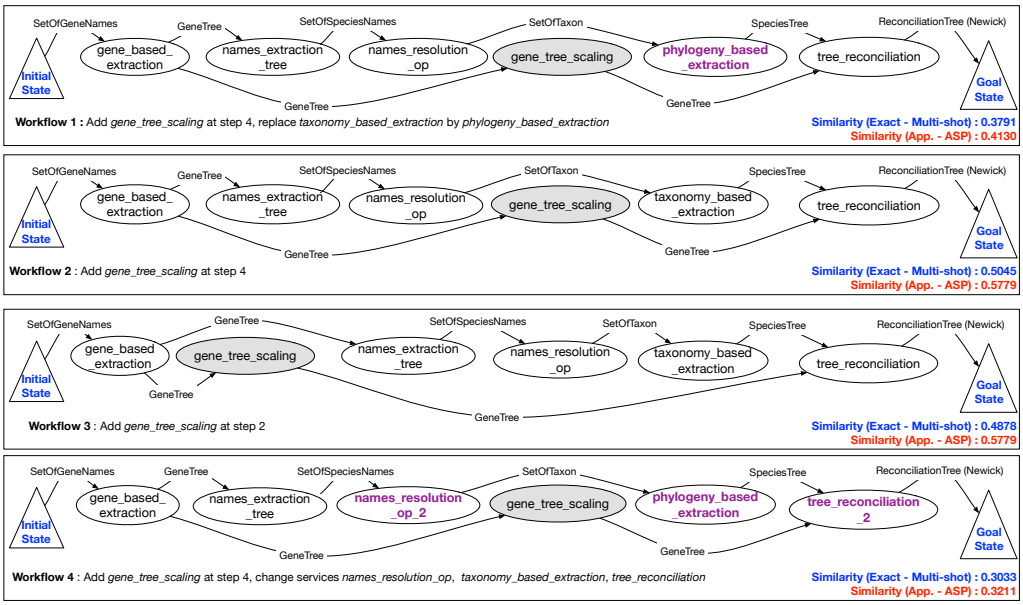

Fig. 8: Possible updated workflows by 2 approaches

be inserted after `gene_based_extraction` and before `tree_reconciliation` service. The experimentation has been performed on a machine running MacOS 10.13.3 with 8GB DDRam 3 and a 2.5GHZ Intel-Core i5 (3rd Generation) with 54 classes of services and 125 concrete specific services in Ontology domain.

- **Approximating the value of similarity using ASP**: Using this approach, there are different updates to the original workflow. Four of them with the highest similarity values are displayed in Figure 8. The highest similarity value of 0.5779 comes from WF_2 and WF_3. Observe that these workflows have only one modification (`gene_tree_scaling` occurs at step 4 in WF_2 and step 1 in WF_3). Furthermore, WF_1 has two changes (adding `gene_tree_scaling` and replacing `taxonomy_based_extraction` by `phylogeny_based_extraction`); and WF_4 has three changes. The total processing time of this approach is 35.183 seconds.
- **Computing exact value of similarity using multi-shot ASP**: The previous approach only approximates the similarity values of the new solutions and the original workflow. Using the multi-shot ASP, we can calculate the exact value of the similarity. Figure 8 shows the same 4 workflows considered in the previous approach with their exact value of similarity. The clear winner is WF_2. The system takes 37.937 seconds to compute these updates.

4.3.2 Generating a Species Tree in `newick` Format from Free Plain-Text

The second use case is concerned with the generation of a species tree in `newick` format from a text document (`plain_text` format). Figure 9 (Workflow 0) displays the workflow with the highest QoS value. The first revision that an user asked is the system is to avoid using the `Get_PhyloTree_Phylomatic_V2` service. The resulting most similar workflow is displayed in Figure 9 (Workflow 1) in which the service is replaced by the service `Get_PhyloTree_OT_V2`. However, for this service to be used, the service `convert_taxa_GNR_to_Phylomatic` must be replaced with `convert_taxa_GNR_to_OT_format`. The second revision was to force the use of

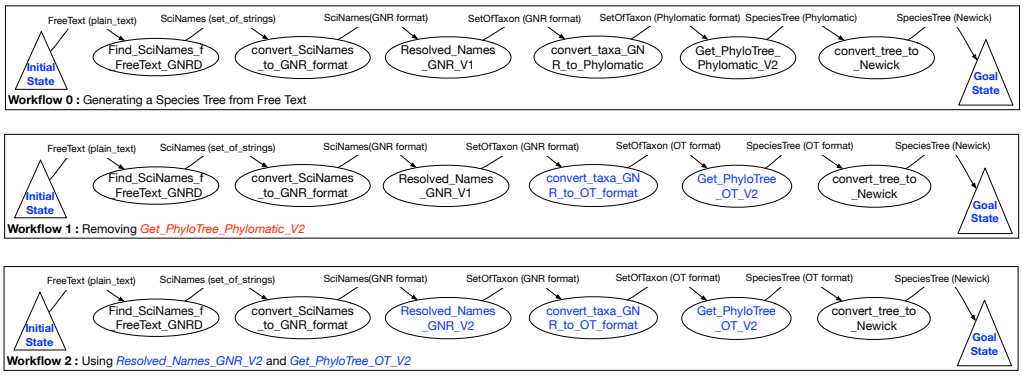

Fig. 9: Generating a Species Tree from Free Text

the service `Resolved_Names_GNR_V2`, the third workflow Figure 9 (Workflow 2) satisfies this request.

4.3.3 Generating a Chronogram from a Web-site Content

A `chronogram` is a scaled species tree with branch lengths. In this use case,

- An user has an initial request to create a chronogram in the `newick` format and meta-data of this tree in the `set_of_strings` format from a web-site content, which is specified by a URL (`http_url` format), and a name of scaling method (`string` format). The workflow generated by the planning engine is shown in Figure 10 (Workflow 0). In this workflow, the output components `chronogram` and `meta-data` are produced by services `Get_Chronogram_ScaledTree_DL_V2` and `Get_MetaData_Chronogram_DL_V2` respectively. Both of them use `species_tree` as an input and this resource is generated by service `Get_PhyloTree_PhyloT_V2` in previous step.
- The user then requests that the services `Get_PhyloTree_OT_V2` and `Resolved_Names_OT_V2` have to be used. The resulting most similar workflow is presented in Figure 10 (Workflow 1) in which services `Resolved_Names_GNR_V1`, `convert_taxa_GNR_to_phyloT` and `Get_PhyloTree_PhyloT_V2` are replaced by `Resolved_Names_OT_V2`, `Get_PhyloTree_OT_V2` and `convert_Tree_to_Newick` respectively.
- Instead of the above request, the user requests that `Get_MetaData_Chronogram_DL_V2` is executed **after** `Get_Chronogram_ScaledTree_DL_V2`. Figure 10 (Workflow 2) shows the most similar workflow to Workflow 0 that satisfies this request. In this workflow, `Get_MetaData_Chronogram_DL_V2` will use the output of `Get_Chronogram_ScaledTree_DL_V2` as its input instead of the output from `Get_PhyloTree_PhyloT_V2`.

5 Conclusion and Future Works

In this paper, we described two enhancements to the Phylotastic project that allow users to select their most preferred workflow and modify a workflow and obtain the most similar workflow to the original one. In the process, we defined the notion of quality of service of a workflow and the notion of similarity between two workflows. We discuss their implementation and their use

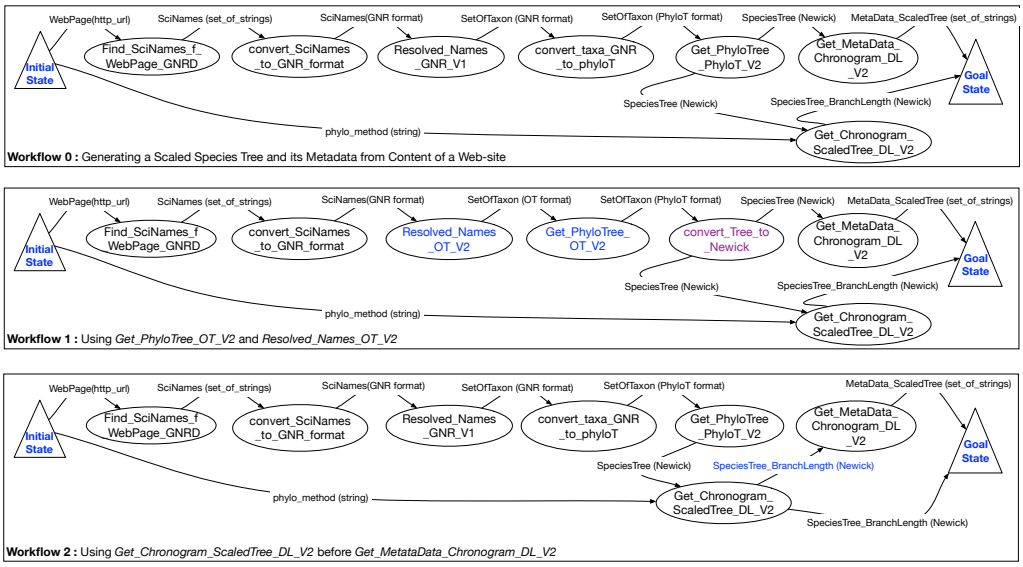

Fig. 10: Generating a Scaled Species Tree and its Metadata from Content of a Web-site

in enhancing the capabilities of the Phylotastic system. The proposed system is currently begin evaluated by biology researchers participating to the Phylotastic project. Our immediate future considerations are: *(i)* investigate the use of node QoS or similarity in the ASP-planning engine to assist the computation of most preferred (or similar) workflow; *(ii)* study other extensions of **clingo** (e.g., clingcon) that allow a tighter integration of CSP and ASP in computing most preferred (or similar) workflows; *(iii)* evaluate the scalability and efficiency of the system when more web services are registered to Phylotastic.

References

A. Stoltzfus et al. 2013. Phylotastic! Making tree-of-life knowledge accessible, reusable and convenient. *BMC Bioinformatics 14*.

Antunes, G., Bakhshandeh, M., Borbinha, J. L., Cardoso, J., Dadashnia, S., Francescomarino, C. D., Dragoni, M., Fettke, P., Gal, A., Ghidini, C., Hake, P., Khiat, A., Klinkmüller, C., Kuss, E., Leopold, H., Loos, P., Meilicke, C., Niesen, T., Pesquita, C., Péus, T., Schoknecht, A., Sheetrit, E., Sonntag, A., Stuckenschmidt, H., Thaler, T., Weber, I., and Weidlich, M. 2015. The process model matching contest 2015. In *Enterprise Modelling and Information Systems Architectures, Proceedings of the 6th Int. Workshop on Enterprise Modelling and Information Systems Architectures, EMISA 2015, Innsbruck, Austria, September 3-4, 2015.* 127–155.

Becker, M. and Laue, R. 2012. A comparative survey of business process similarity measures. *Computers in Industry 63,* 2, 148–167.

Bininda-Emonds, O., Cardillo, M., Jones, K., MacPhee, R., Beck, R., Grenyer, R., Price, S., Vos, R., Gittleman, J., and Purvis, A. 2007. The delayed rise of present-day mammals. *Nature 446,* 7135.

Carman, M., Serafini, L., and Traverso, P. 2004. Web Service Composition as Planning. In *Proceedings of ICAPS 2003 Workshop on Planning for Web Services*.

Cracraft, J., Donoghue, M., Dragoo, J., Hillis, D., and Yates, T. 2002. Assembling the tree of life: harnessing life's history to benefit science and society. Tech. Rep. `http://ucjeps.berkeley.edu/tol.pdf`., U.C. Berkeley.

Jetz, W., Thomas, G., Joy, J., Hartmann, K., and Mooers, A. 2012. The global diversity of birds in space and time. *Nature 491,* 7424.

KAMINSKI, R., SCHAUB, T., AND WANKO, P. 2017. A tutorial on hybrid answer set solving with clingo. In *Reasoning Web. Semantic Interoperability on the Web - 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures*. 167–203.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1–2, 39–54.

MCILRAITH, S., SON, T., AND ZENG, H. 2001. Semantic Web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web) 16,* 2 (March/April), 46–53.

MORA, C., TITTENSOR, D., ADL, S., SIMPSON, A., AND WORM, B. 2011. How many species are there on Earth and in the ocean? *PLoS biology 9,* 8.

NGUYEN, T. H., SON, T. C., AND PONTELLI, E. 2018. Automatic web services composition for phylotastic. In *Practical Aspects of Declarative Languages - 20th International Symposium*. 186–202.

PEER, J. 2005. Web Service Composition as AI Planning - a Survey. Tech. rep., University of St. Gallen.

PENNY, D. 2004. Inferring phylogenies.joseph felsenstein. 2003. sinauer associates, sunderland, massachusetts. *Systematic Biology 53,* 4, 669–670.

PROSDOCIMI, F., CHISHAM, B., THOMPSON, J., PONTELLI, E., AND STOLTZFUS, A. 2009. Initial implementation of a comparative data analysis ontology. *Evolutionary Bioinformatics 5*, 47–66.

RAJENDRAN, T. AND BALASUBRAMANIE, D. P. 2009. Analysis on the study of qos-aware web services discovery. *Journal of Computing*, 119–130.

SMITH, S., BEAULIEU, J., STAMATAKIS, A., AND DONOGHUE, M. 2011. Understanding angiosperm diversification using small and large phylogenetic trees. *American Journal of Botology 98,* 3, 404–414.

STOLTZFUS, A., O'MEARA, B., WHITACRE, J., MOUNCE, R., GILLESPIE, E., KUMAR, S., ROSAUER, D., AND VOS, R. 2012. Sharing and Re-use of Phylogenetic Trees (and associated data) to Facilitate Synthesis. *BMC Research Notes 5*, 574.

ZHANG, K. AND SHASHA, D. E. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput. 18,* 6, 1245–1262.