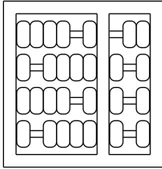

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Linked biology technical aspects – linking phenotypes and phylogenetic trees

E. Miranda *A. Santanchè*

Technical Report - IC-14-06 - Relatório Técnico

February - 2014 - Fevereiro

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Linked biology technical aspects – linking phenotypes and phylogenetic trees

Eduardo Miranda * André Santanchè †

Abstract

A large number of studies in biology, including those involving phylogenetic trees reconstruction, result in the production of a huge amount of data – e.g., phenotype descriptions, morphological data matrices, etc. Biologists increasingly face a challenge and opportunity of effectively discovering useful knowledge crossing and comparing several pieces of information, not always linked and integrated. Ontologies are one of the promising choices to address this challenge. However, the existing digital phenotypic descriptions are stored in semi-structured formats, making extensive use of natural language. This technical report is related to a research developed by us [1] to addresses this problem, adding an intermediate step between semi-structured phenotypic descriptions and ontologies. It remodels semi-structured descriptions to a graph abstraction in which the data are linked. Graph transformations subsidize the transition from semi-structured data representation to a more formalized representation with ontologies. The present technical report drills down implementation details of our system. It provides a module to ingest phylogenetic trees and phenotype descriptions – represented in semi-structured formats – into a graph database. Additionally, two approaches to combine distinct data sources are presented and an algorithm to trace changes in phylogenetic traits of trees.

*Institute of Computing – State University of Campinas, 13081-970, Campinas, Brazil. Work partially financed by (CNPq 138197/2011-3), the Microsoft Research FAPESP Virtual Institute (NavScales project), CNPq (MuZOO Project and PRONEX-FAPESP), INCT in Web Science(CNPq 557.128/2009-9) and CAPES, as well as individual grants from CNPq.

†Institute of Computing – State University of Campinas, 13081-970, Campinas, Brazil. Work partially financed by (CNPq 138197/2011-3), the Microsoft Research FAPESP Virtual Institute (NavScales project), CNPq (MuZOO Project and PRONEX-FAPESP), INCT in Web Science(CNPq 557.128/2009-9) and CAPES, as well as individual grants from CNPq.

Contents

1 Introduction **3**

2 Basic concepts **3**
2.1 Standards for Phenotype Description . . . 3
2.2 Life Science Identifiers (LSIDs) . . . 5
2.3 The proposed graph data model . . . 5

3 System Architecture and Implementation Details **6**
3.1 SDD Parser . . . 6
3.2 Tree Output . . . 8
3.3 Global Names Resolver (GNR) . . . 9
3.4 Graph Importer . . . 9
3.5 Graph Database . . . 10
3.6 Similarity Index . . . 12
3.6.1 Practical Implementation of the Similarity Measure . . . 13
3.7 Tracing the Evolutionary History . . . 14

4 Conclusion **17**

A Demonstration **21**
A.1 SDDParser.py . . . 21
A.2 TeeOutput.py . . . 26
A.3 GlobalNamesResolver.py . . . 30
A.4 GNRResultObject.py . . . 33
A.5 ITISServices.py . . . 35
A.6 CoLServices.py . . . 37
A.7 GraphImporter.py . . . 40
A.8 SimilarityIndex.py . . . 48
A.9 TraceEvolutionaryHistory.py . . . 51

1 Introduction

In 1859 Charles Darwin published *On the Origin of Species* which is considered the foundation of evolutionary biology. In his book, Darwin set forth the theory of evolution and natural selection. It argues that all life is related and has descended from a common ancestor. The Tree of Life is a metaphor to describe the relationships between living and extinct organisms through their common ancestors. More precisely, it is an abstract form to represent hypotheses about evolutionary relationships, in which all species that have ever existed are taken together with relationships among them, describing their evolutionary lineages. In this abstract representation, the taxa are the leaves of the tree and the internal nodes are common ancestors, or hypothetical taxa.

This huge and complex tree is split into smaller branches, which are investigated separately and then incorporated into the tree. Evolutionary biologists normally work in relatively small chunks of the tree, analyzing a very specific subset of species. A fundamental challenge in this scenario is the creation of a complete evolutionary Tree of Life [2], assembling genomic and morphological data so as to congregate the phylogenetic relationships among all known living or extinct organisms [3, 4, 5]. The integration of these data may contribute to better understand how a morphological trait became organized and evolved over time [6], how organisms interact and how life on Earth came to be.

The main goal of this research is to design and implement a linked biology approach to automatically connect and combine data from independent semi-structured resources of phenotype descriptions and/or phylogenetic trees, exploiting their latent semantics. We propose a graph data model that plays a crucial role, since it is the basis of our linking discovery and combination process. It contributes assisting biologists in the exploration of existing biology assets related to phenotype descriptions and their latent semantics. The present work details algorithms, implementation aspects and the database model related to our research.

The text is organized as follows. Section 2 synthesizes basic concepts necessary for understanding the text. Section 3 discusses implementation details of our system and presents some results. Section 4 presents concluding remarks. In the Appendix the source code is provided with comments explaining its functionalities.

2 Basic concepts

In this section, we highlight basic concepts adopted in this text. Subsection 2.1 introduces some key elements of XML formats for phenotype description. Subsection 2.2 we details the Life Science Identifier which is one of the solutions for data interconnection. Subsection 2.3 presents an overview of our proposed graph model.

2.1 Standards for Phenotype Description

There is a wide variety of representation formats for phenotype descriptions adopted by information systems and open standards, which represent differently the same information. In [1] we analyze four of them – Xper^2, SDD, Nexus and NeXML – looking for a common denominator which is the foundation for our graph-based model. SDD, Nexus and NeXML are widely adopted open standards. Xper^2 (*http://lis-upmc.snv.jussieu.fr/lis/*) is a management system adopted by the systematist community, for storing, editing and analyzing phenotype descriptive data. It focuses mainly on taxonomic descriptions, allowing creation, sharing and comparison of identification keys [7, 8]. Xper^2 was developed in the Laboratoire Informatique & Systématique of the University Pierre et Marie Curie and this work is part of a bigger project in collaboration with this lab. Therefore, Xper^2 was adopted for our practical implementation.

In order to transform phenotype observations into digital records and generalize them – e.g., devising general characters and states observed in a genus of monitor lizards – the biologist may use a tool as Xper[2]. Phenotype descriptions can be stored in the Xper[2] native format or can be exported to the SDD open format. The Structured Descriptive Data (SDD) (*http://wiki.tdwg.org/SDD*) is a platform and application-independent XML-based standard developed by the Biodiversity Information Standards (historic acronym: TDWG) for recording and exchanging descriptions of biological and biodiversity data of any type [9]. SDD is adopted by several other phenotype description tools – e.g., Lucid Central (*http://www.lucidcentral.org*) and Linnaeus II (*http://www.eti.uva.nl/*).

We further introduce some key elements of the SDD format, which are recurrent in the formats confronted in [1]. A SDD description comprises, in a single file, a domain schema and its instances. Figure 1 shows a diagram with a fragment of a SDD file containing the description of a varanus lizard. A (C,CS) description in SDD has two main blocks: (i) defines the characters involved and their possible states – Figure 1 top; (ii) describes an Operational Taxonomic Unit (OTU) using the characters defined in (i) – Figure 1 bottom. OTU is a biology term which refers to a given taxon at the rank adopted to the study – e.g., a specimen, a species, a genus etc.

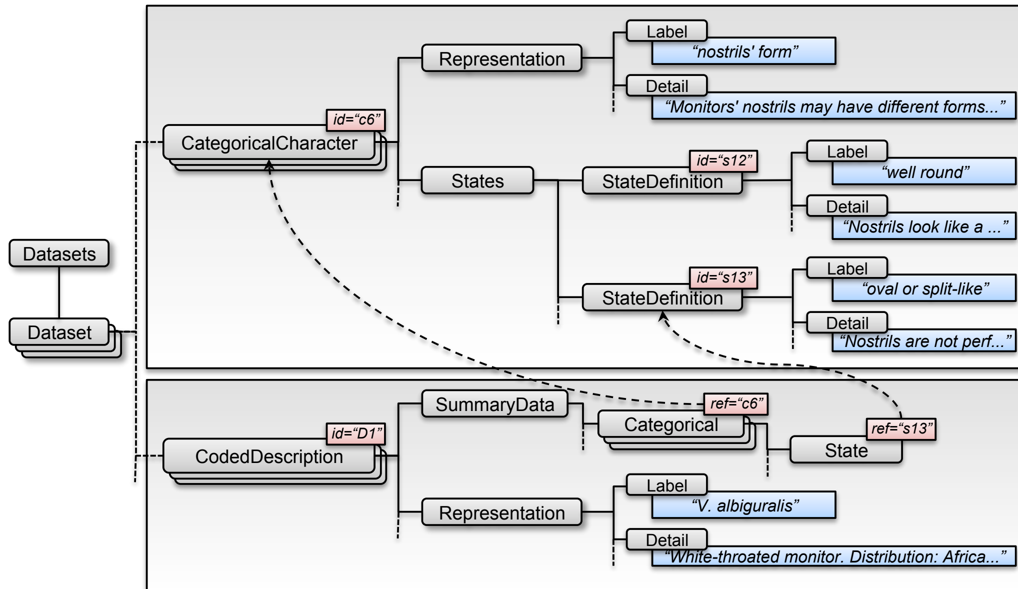

Figure 1: Fragment of SDD Schema with Instances

<*CategoricalCharacter*>s and their <*States*> (shown in Figure 1 top) are primitives to describe an OTU [9]. Each <*CategoricalCharacter*> has its <*Representation*> – comprising a label and a description as plain texts – and a set of <*StateDefinition*> elements with their possible states. <*CategoricalCharacter*> and <*StateDefinition*> elements defined here will be referred throughout the XML document by their ids. The <*CodedDescription*> (Figure 1 bottom) links the described OTU to *States* of each <*CategoricalCharacter*>. It has two essential items: (i) the described OTU, where its name and description are listed in natural language under <*Representation*>; (ii) a set of character and values (<*Categorical*> and <*State*>), which address the characters defined in the previous section through the *ref* attribute. It is possible and usual to assign multiple character-states for a given OTU (i.e. in case of polymorphism). A first integration, problem observed here is that each character or OTU described does not have a global unique identification among documents. Therefore, the description can only be used by the document where it was declared and it is not

possible to guarantee the equivalence of two or more <*CategoricalCharacters*>.

2.2 Life Science Identifiers (LSIDs)

One of the problems faced in life science is related to the identification of objects within and across repositories [10]. More precisely, an object may refer to a taxon, gene, anatomical feature, phenotypic description, geographical location etc. Integrating data from different sources is not straightforward and uniquely identifying these objects is undoubtedly a key point for the success of our proposed solution.

During the 18th century, Carolus Linnaeus introduced the binomial nomenclature for naming species that is the basis of modern classification [11]. This system basically concatenates 2 Latim words, where the first part identifies the species genera and the second one the species itself. The binomial nomenclature has been used for the last 250 years [11] and the biological information related to organisms is historically annotated by species names. Hence, the binomial name would appear to be a logical candidate to index information available about species. However, misspelling problems are often encountered [12, 13], moreover, taxonomic names are not unique identifiers [14, 15] because scientists may use (i) similar names to different species (homonyms) or (ii) multiple names for the same specie (synonyms) [10, 16].

Furthermore, each organization has its own means of defining a key, which makes the problem even harder to solve. For example, the species *Aotus ericoides* has the id 11479744 on the Catalogue of Life (CoL), id 42472 on the Australian Plant Name Index (APN), id 643314 on the Encyclopedia of Life (EoL), id 129761-3 on the The International Plant Names Index (IPNI), id 700844 on the Universal Biological Indexer and Organizer (uBio) etc.

In order to address this issue, some organizations – e.g., Universal Biological Indexer and Organizer (uBio), Integrated Taxonomic Information System (ITIS), Catalogue of Life (CoL), The International Plant Names Index (IPNI), National Center for Biotechnology Information (NCBI) etc. – incorporated into their projects the concept of Life Science Identifiers (LSIDs), proposed by the Object Management Group (OMG) (*http://www.omg.org/*). LSID is a persistent, location-independent resource identifier, whose purpose is to uniquely identify biological resources [17]. The persistent property refers to the fact that LSID identifiers are unique, can be assigned to only one object forever and they never expire. The location-independent property specifies that each authority locally creates LSIDs and they are the responsible to guaranteeing the uniqueness of LSIDs.

2.3 The proposed graph data model

In this section we will present an overview of our proposed graph model. From the numerous graph data models proposed – see [18, 19, 20] for more details – the *property graph* model was adopted in the present work. In a *property graph*, nodes and relationships can maintain extra metadata as a set of key/value pairs. Moreover, relationships are typed, enabling to create multi-relational networks with heterogeneous sets of edges. Different from single-relational networks, in which edges are of the same type, multi-relational networks are more appropriate to represent complex domain models, due to the variety of relationship types in the same graph [21]. For example: relationships may either represent membership in a social group (family membership) or professional relationships (employer-worker relationship) simultaneously in the same network.

Figure 2 shows our graph data model. The tables below the nodes/edges represent their types and metadata. We mapped the SDD format to the graph model as follows: OTUs are entities (e.g., "Varanus prasinus") and, therefore, were mapped to nodes. A future target of this project is to enrich our model by associating identifiable entities to ontology concepts. One may consider to map Characters and Characters States to key/value pairs, to be related to OTU nodes. However, we decided to map Characters to nodes, in order to unify in the same node equivalent characters observed

in several OTUs and, in a future work, to relate the unified characters with ontologies. Finally, the Character-state makes a semantic bridge (relationship) between OTUs and Characters. Thus, a statement like *"Varanus gouldi ventral pattern is randomly scattered dark spots"* is represented in our model as *Varanus gouldi* (node) → *randomly scattered dark spots* (edge) → *ventral pattern* (node).

Our model comprises, in a single place, phenotype descriptions and phylogenetic trees. For this reason a new node called HTU (Hypothetical Taxonomic Unit) is present in this model. HTUs are internal nodes in phylogenetic trees that represent an inferred ancestral organism. HTUs are hypothetical common ancestors of OTUs nodes and, therefore, can only be connected to themselves (HTU → HTU) or to OTUs (HTU → OTU). For the sake of modeling simplicity, only the *TreeEdge* relationship is allowed between HTU → HTU and HTU → OTU. Finally, there is also a character-state relationship between HTU nodes and character nodes that are strictly created by some algorithms.

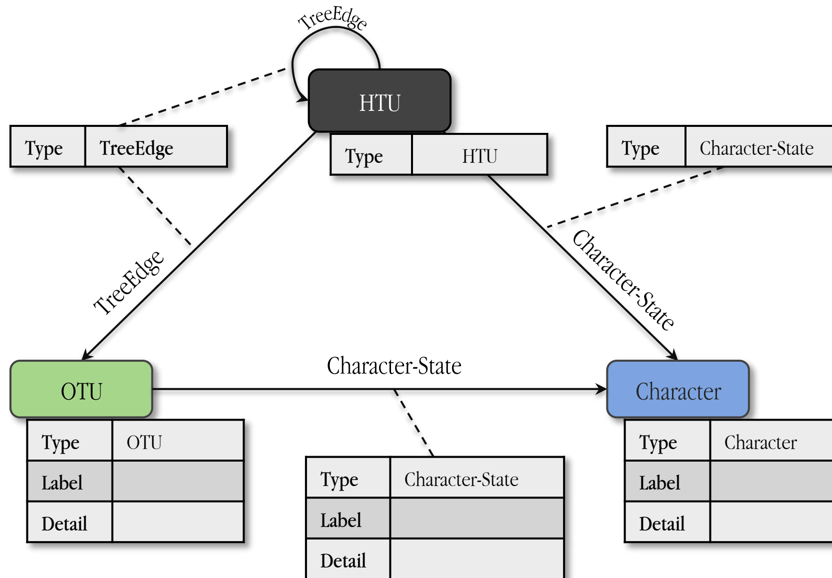

Figure 2: Property Graph Model

3 System Architecture and Implementation Details

In this section, we analyze the system architecture and its implementation details, in order to present its main functionalities and operational features. The text is presented progressively. The core functionalities are shown in the first subsections and the algorithms are presented later.

We have developed our platform on top of the Neo4j graph database (*http://www. neo4j.org/*), mainly due to its widespread adoption. Our implementation uses the Python programming language and Py2neo (*http://book.py2neo.org/*), which is an interface connecting Python and Neo4j via REST API. The adopted query language was *Cypher*, which is a declarative graph query language.

3.1 SDD Parser

Our SDD Parser has all functionalities to parse an SDD file (for implementation details see Appendix A.1) using the Python *xml.dom.minidom*, which is a minimal implementation of the Document Ob-

ject Model interface. Listing 1 shows an SDD fragment of a Varanus knowledge base [1], of which Figure 1 is a simplified abstraction. In addition, all main SDD structures presented in Figure 1 and Listing 1 – Representation, StateDefinition, CategoricalCharacter, Categorical and CodedDescription – were processed to produce our graph.

Listing 1: Varanus.sdd.xml

```
<Characters>
 ...
 <CategoricalCharacter id="c6">
  <Representation>
   <Label>nostrils' form</Label>
   <Detail>Monitors' nostrils mayhave different forms.&lt;br&gt;Look
       at the head in side view or dorsal view in order to
      appreciate this characteristic.</Detail>
   <MediaObject ref="m40"/>
  </Representation>
  <States>
   <StateDefinition id="s12">
    <Representation>
     <Label>well round</Label>
     <Detail>Nostrils look like a quite perfect circle.</Detail>
    </Representation>
   </StateDefinition>
   <StateDefinition id="s13">
    <Representation>
     <Label>oval or split-like</Label>
     <Detail>Nostrils are not perfectly round: they are oval or they
         present a split-like form.</Detail>
    </Representation>
   </StateDefinition>
  </States>
 </CategoricalCharacter>
 ...
</Characters>
...
<CodedDescriptions>
 <CodedDescription id="D1">
  <Representation>
   <Label>V. albiguralis</Label>
   <Detail>White-throated monitor&lt;br&gt;&lt;br&gt;Distribution:
      Africa (West and South).&lt;br&gt;&lt;br&gt;CITES: appendix II
      .</Detail>
   <MediaObject ref="m1"/>
  </Representation>
   <SummaryData>
    ...
    <Categorical ref="c6">
```

[1]Knowledge base of the genus Varanus
(*http://lis-upmc.snv.jussieu.fr/xper2/infosXper2Bases/details_base.php?id_base=86*)

```
37      <State ref="s13"/>
38     </Categorical>
39     ...
40    </SummaryData>
41  </CodedDescription>
42 </CodedDescriptions>
```

3.2 Tree Output

The present work also draws upon phylogenetic trees generated from LisBeth (*http://lis-upmc.snv.jussieu.fr/lis/*). LisBeth is a cladistics software for phylogenetics and biogeography [22] that implements the three-item analysis (3ia) method of phylogenetic inference [23]. It minimizes the conflictual relationships within a set of characters, or maximizes the compatible relationships so as to reconstruct one or several optimal tree(s). We implemented a *TreeOutput* class, which abstracts the functions of interacting with LisBeth output files (for implementation details see Appendix A.2). Listing 2 displays two fragments of a LisBeth output file, focusing in the elements processed in this work, i.e. taxons with their ids and the retained tree – newick tree which is a way to represent a tree in computer-readable form, using parentheses and commas. The *TreeOutput* main function combines the retained tree with the taxon names, retrieved in previous steps, and returns a root node to a tree that represents the retained tree. In this new tree, the internal nodes are renamed to *HTU* and the leaf nodes to its respective taxon names (see Figure 3).

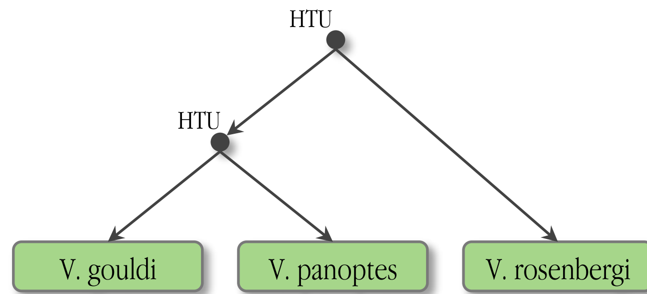

Figure 3: Retained Tree Example

Listing 2: LisBethOutput.3iz

```
1  ...
2  —<D02>—
3  ...
4  Taxa (3) :
5  .                    3  V. gouldii
6  .                    7  V. panoptes
7  .                   12  V. rosenbergi
8  —<F02>—
9  ...
10 —<D06>—
11 ...
12
```

```
13 Retained trees : 1
14 .     1: ((3 7) 12)
15 —<F06>—
16 ...
```

3.3 Global Names Resolver (GNR)

In order to find a valid LSID, we adopted the Global Names Resolver (GNR) web service (*http://resolver.globalnames.org/*) that executes exact or fuzzy matching against canonical forms of scientific names in 170 distinct data sources. The Canonical form (cf) is the simplest, most complete and unambiguous form of a name. The Canonical form of scientific names consists of the genus and species – when applied – with no authorship, rank, nomenclatural annotation or subgenus.

Our system used three of the six types of matching offered by the GNR resolver: (i) exact matching; (ii) exact matching of canonical forms – this process reduces a given name to its canonical form and checks it for an exact match; (iii) fuzzy matching of canonical forms – uses a modified version of the TaxaMatch algorithm [13] and intends to work around misspellings errors. It does a fuzzy match of the canonical form of a given name – even with mistakes – against spellings considered correct. The GNR resolver reports the matching quality (*"confidence score"*) for each match. The other three remaining matching types are: (iv) exact matching of specific parts of names, (v) fuzzy matching of specific parts of names and (vi) exact matching of genus part of names. They were not adopted because we focused in complete names in their canonical form.

Our algorithm extracts all plain text taxon entities present in the SDD file and, for each one, it uses the GNR to transform the taxon name to its canonical form. Only those taxons with confidence score above of 0.988 are considered. After that, the algorithm makes use of the GNR resolver to search for its LSID (for implementation details see Appendix A.3) – only exact matches are considered. The GNR results have the output field *"local id"* which, in the case of uBio, is the LSID. Moreover, we prioritized the uBio LSID, since it indexes and organizes until now more than 11 million names. But there are cases in which the GNR resolver does not retrieve any result from the uBio. In these cases, the algorithm makes use of the Integrated Taxonomic Information System (ITIS) web services (*http://www.usgovxml.com/DataService.aspx?ds=ITIS*), in order to obtain the LSID (for implementation details see Appendix A.5). ITIS is a reliable taxonomic base for species, with more than 740 thousand common names and scientific names indexed. If none of the services return a valid LSID, we also implemented a class to interact with the CoL web service (*http://www.catalogueoflife.org/col/webservice*), attempting to obtain a valid LSID (for implementation details see Appendix A.6).

3.4 Graph Importer

Graph Importer is an object class written in Python that is responsible for coupling the phylogenetic trees and phenotype descriptions into the graph database. The insertion process follows the sequence: (1) Starts parsing the SDD XML file and the LisBeth output file – see Listing 1 and 2 respectively. (2) Creates a taxon node for each taxon present in the SDD file – see Figure 1 bottom, tag *<Representation>*. In this process, it searches for a valid LSID for each taxon node, using the GNR web service, ITIS web service or CoL web service. If the LSID is not found, it creates a taxon node without LSID. (3) Joins the taxon nodes to the tree structure, extracted from the LisBeth output file. (4) A node is created for each character in the SDD file – see Figure 1 top, tag *<Representation>*. (5) The taxon nodes are linked to the character nodes by their character-states – see Figure 1 top, tag *<States>/<StateDefinition>*. It will exist character-state relationships where exists a pair *<SummaryData>/<Categorical>* and *<SummaryData>/<Categorical>/<State>* – see

Figure 1 bottom, tag <*SummaryData*>. For implementation details see Appendix A.7. Figure 4 shows a visual representation of the retained tree combined with the taxon nodes provided in Listing 2. The figure shows that the edges depart from taxon nodes toward character nodes.

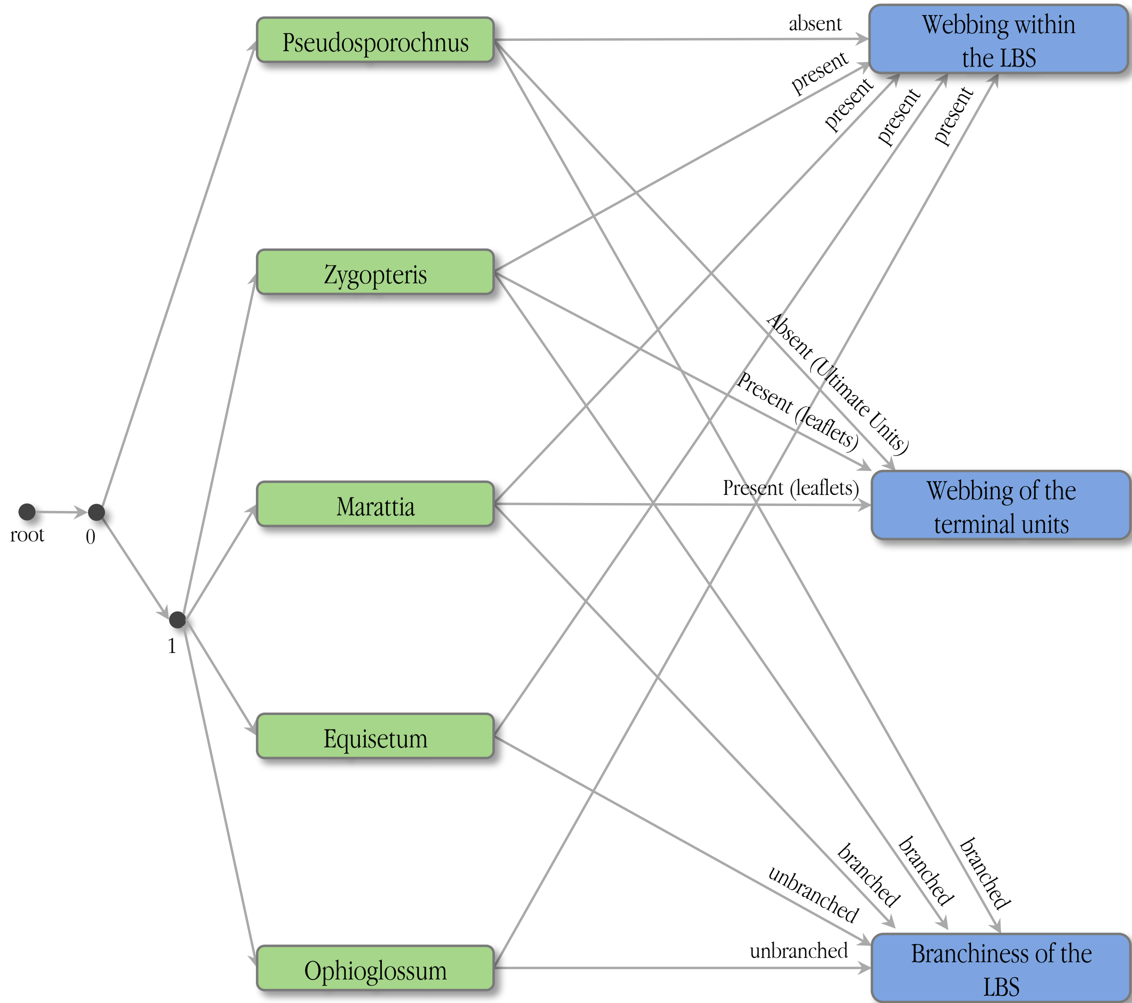

Figure 4: Real Example

3.5 Graph Database

We implemented a GraphDB class, which abstracts and centralizes all database operations. We describe each function header, followed by a short description of the main *Cypher* queries used in the system.

```
getNodeByLSID( LSID ):
// Returns a node for the supplied LSID.
START n=node( * )
WHERE n.lsid = 'LSID'
```

```
5  RETURN n
6
7  getOutgoingAdjacentNodes( GivenNode ):
8  // Returns all nodes to which the given node points to.
9  START n=node( GivenNode.id )
10 MATCH (n)-->(c)
11 RETURN DISTINCT c
12
13 getIncomingAdjacentNodes( GivenNode ):
14 // Returns all nodes that points to the given node.
15 START n=node( GivenNode.id )
16 MATCH (c)-->(n)
17 RETURN DISTINCT c
18
19 getIncomingAdjacentRelationships( GivenNode ):
20  // Returns all relationships incoming to a given node.
21 START n=node( GivenNode.id )
22 MATCH ()-[r]->(n)
23 RETURN r
24
25 getIncomingAdjacentNodesWithRelationshipInBetween( GivenNode,
      GivenRelationship ):
26 // Returns all nodes, ordered by their label, that points to a given
       node with a given relationship in between.
27 START n=node( GivenNode.id )
28 MATCH (c)-[:GivenRelationship.label]->(n)
29 RETURN c
30 ORDER BY c.label
31
32 getOutgoingRelationships( GivenNode ):
33 // Returns all relationships outgoing from a given node.
34 START n=node( GivenNode.id )
35 MATCH (n)-[r]->()
36 RETURN r
37
38 getDistinctRelationshipsInBetween( GivenNodeA, GivenNodeB ):
39 // Returns all distinct relationships that exists between nodes A
      and B.
40 START a=node( GivenNodeA.id ), b=node( GivenNodeB.id )
41 MATCH (a)-[r]-(b)
42 WITH COLLECT( DISTINCT TYPE( r ) ) as rels
43 RETURN rels
44
45 getDescriptionNodesOfATree( TreeRoot )
46 // Returns all distinct description nodes id, character or character
      -states depending on the schema, that are conected to a given
      tree.
47 START root=node( TreeRoot.id )
48 MATCH (root)-[*..]->(d)
```

```
49 WHERE d.type = 'description'
50 RETURN DISTINCT ID(d)
51
52 deleteNodeRelationshipsExceptLabel( GivenNode, RelationshipLabel ):
53 // Deletes all node relationships except for a given relationship
     label.
54 START n=node( GivenNode.id )
55 MATCH n-[r]->()
56 WHERE NOT( r.label = 'RelationshipLabel' ) AND NOT( r.type = '
     TreeEdge' )
57 DELETE r
58
59 deleteRelationshipsTypeFromNode( GivenNode, RelationshipType ):
60 // Deletes all node relationships of a given type.
61 START n=node( GivenNode.id )
62 MATCH n-[r]->()
63 WHERE r.type = 'RelationshipType'
64 DELETE r
```

3.6 Similarity Index

We are proposing a heuristic similarity measure that computes the similarity degree between two morphological character descriptions. This measure will represent how closely related they are. The similarity index ($\boldsymbol{S}_i$) is based on 2 weighted aspects. 25% of the index is calculated based on the taxa being described, i.e. it analyzes if two given characters (C_1 and C_2) describe the same taxa. The other 75% are based on the meaning of the character-states. It checks if the state labels being used are the same. This heuristic is still a work in progress. The weights assigned to parts of the index are configurable and their values were calibrated based on observations.

Let $G = (V(G), E(G))$ be a directed graph with vertex-set $V(G) = \{v_1, ..., v_n\}$ and edge-set $E(G) = \{e_1, ..., e_m\} \subset \{(v_i, v_j)|v_i, v_j \in V(G)\}$. Let C_1, $C_2 \in V(G)$ be two distinct vertices of G. We define the following sets:

$$N_{C_1} = \{v_i \in V(G) \mid (v_i, C_1) \in E(G)\} \quad (1)$$

$$N_{C_2} = \{v_i \in V(G) \mid (v_i, C_2) \in E(G)\} \quad (2)$$

$$S_1 = \frac{|N_{C_1} \cap N_{C_2}|}{\max\{|N_{C_1}|, |N_{C_2}|\}} \quad (3)$$

Let $f : E(G) \to \Upsilon$ be a labeling function, where Υ is a set of labels, and $f(e) \in \Upsilon$ is the label of edge $e \in E(G)$. We define the following sets:

$$L_{C_1} = \{e \mid e = f((v_i, C_1)) \in \Upsilon \text{ and } (v_i, C_1) \in E(G) \text{ and } v_i \in V(G)\} \quad (4)$$

$$L_{C_2} = \{e \mid e = f((v_i, C_2)) \in \Upsilon \text{ and } (v_i, C_2) \in E(G) \text{ and } v_i \in V(G)\} \quad (5)$$

$$S_2 = \frac{|L_{C_1} \cap L_{C_2}|}{\max\{|L_{C_1}|, |L_{C_2}|\}} \quad (6)$$

$$Similarity\ Index(S_i) = 0.25 * S_1 + 0.75 * S_2 \quad (7)$$

S_1 defines a rate of common OTU vertices with edges for two given characters C_1 and C_2. The S_1 result lies between 0 (no common OTUs) and 1 (all OTUs are common). N_{C_1} is the subset

of incoming adjacent vertexes of C_1 and N_{C_2} is the subset of incoming adjacent vertexes of C_2. Incoming adjacent vertexes of both C_1 and C_2 are always OTU vertexes, as shown in Figure 2. S_2 defines a rate of common labels of the incoming edges (character-states) for the characters C_1 and C_2. The S_2 result also lies between 0 (no common character-states) and 1 (all character states are common). L_{C_1} and L_{C_2} are the subset of incoming adjacent edge labels (character-states) of C_1 and C_2 respectively.

It is important to note that the character labels of C_1 and C_2 are not being taken into account in the S_i formula. This intends to avoid weighting in favor of two identical textual characters that do not have the same meaning, and to avoid weighting against two textual characters that are identical but do not have the same meaning. In practice, this will make the solution independent of the label and applicable for both presented scenarios (same label but different meanings and different labels and same meaning). Additionally, the symmetric property of equality is satisfied.

3.6.1 Practical Implementation of the Similarity Measure

Our system is able to draw a chart as illustrated in Figure 5, whose algorithm is inspired by the hierarchical edge bundling example (*http://mbostock.github.io/d3 /talk/20111116/ bundle.html*) of D3.js (*http://d3js.org/*) library. D3.js is a JavaScript library for manipulating documents and it has a wide variety of powerful visualization components. In the case of the hierarchical edge bundling example, it is necessary to provide only a "name" for each node and, inside a related "imports" sentence, the node name to where an edge must be created to. Listing 3 shows the JSON file that encodes the data used to generate Figure 5 (for implementation details see Appendix A.8).

Listing 3: RealExample.json

```
1  [
2  {"name": "root.Cauline cladotaxy" , "imports": ["root.Cauline
      cladotaxy", "root.Phyllotaxy"]},
3  {"name": "root.Protoxylem position within the cauline stele" , "
      imports": ["root.Protoxylem position within the cauline stele"]},
4  {"name": "root.Organotaxy of the LBS" , "imports": ["root.Cauline
      cladotaxy", "root.Phyllotaxy"]},
5  {"name": "root.Xylem configuration in the leaflets" , "imports": []}
      ,
6  {"name": "root.Planation" , "imports": []},
7  {"name": "root.Development of the foliar organ" , "imports": []},
8  {"name": "root.Phyllotaxy" , "imports": []},
9  {"name": "root.Xylem configuration in the rachis" , "imports": []},
10 {"name": "root.Cauline cladotaxy" , "imports": []},
11 {"name": "root.Protoxylem position within the cauline stele" , "
      imports": []},
12 {"name": "root.Xylem configuration in the rachis" , "imports": []},
13 {"name": "root.Extent of the planation" , "imports": []},
14 {"name": "root.Presence of planated parts within the LBS" , "imports
      ": []},
15 {"name": "root.Xylem configuration in the leaflets" , "imports": []}
      ,
16 {"name": "root.Development of the LBS" , "imports": ["root.
      Development of the foliar organ"]}
17 ]
```

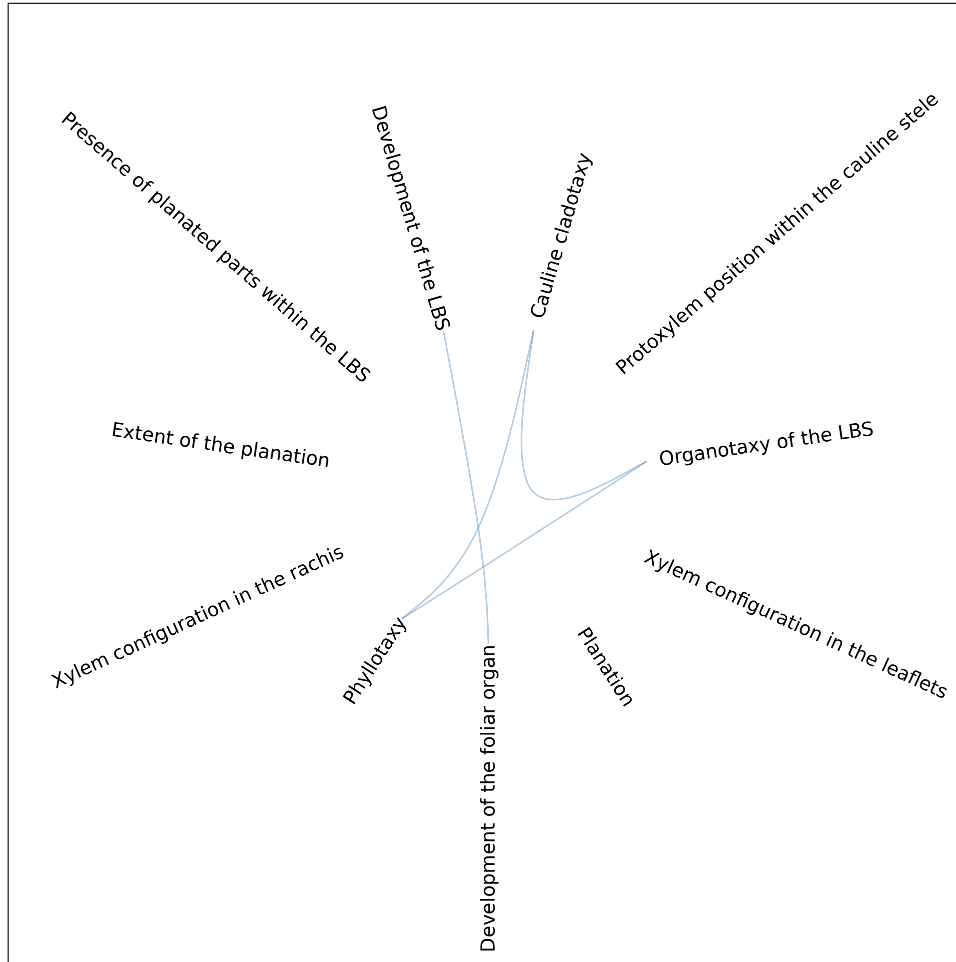

Figure 5: Practical Implementation

3.7 Tracing the Evolutionary History

The *TraceEvolutionaryHistory* class abstracts an important algorithm that traces a phylogenetic history of traits changes (for implementation details see Appendix A.9). This algorithm was built on top of our graph data model. It searches in a given tree for traits (characters) that might be the "responsible" for a tree branching, in which branching is considered as any division from a particular ancestor. For example, Figure 4 has two Hypothetical Taxonomic Units (HTU), in which the least nested one after the root has the *Pseudosporochnus* node and another HTU node as children. A typical question that motivated us to create such an algorithm was: What differentiates *Pseudosporochnus* from the other nodes?

The algorithm is divided into two recursive methods that are invoked in sequence. The first one *BottomUpAggregation* starts from a given point in the tree and goes down until it reaches Operational Taxonomic Unit (OTU) nodes. At this point, the method retrieves all outgoing relationships from the OTU node and starts going back towards the root. While the method is traversing internal HTU nodes ($current_{HTU}$) from the leaves back towards the root, it performs an union operation with the outgoing relationships of all children nodes – one occurrence for each type of relationship –

and then, for each type of relationship of the resulting union, the method creates an edge departing from the current HTU ($current_{HTU}$) towards the original ending point of the relationship. In the end, the method returns all relationships outgoing from all nodes, including the intermediary HTU nodes ($current_{HTU}$). Figure 6 shows the result of *BottomUpAggregation* method being applied on the graph of Figure 4.

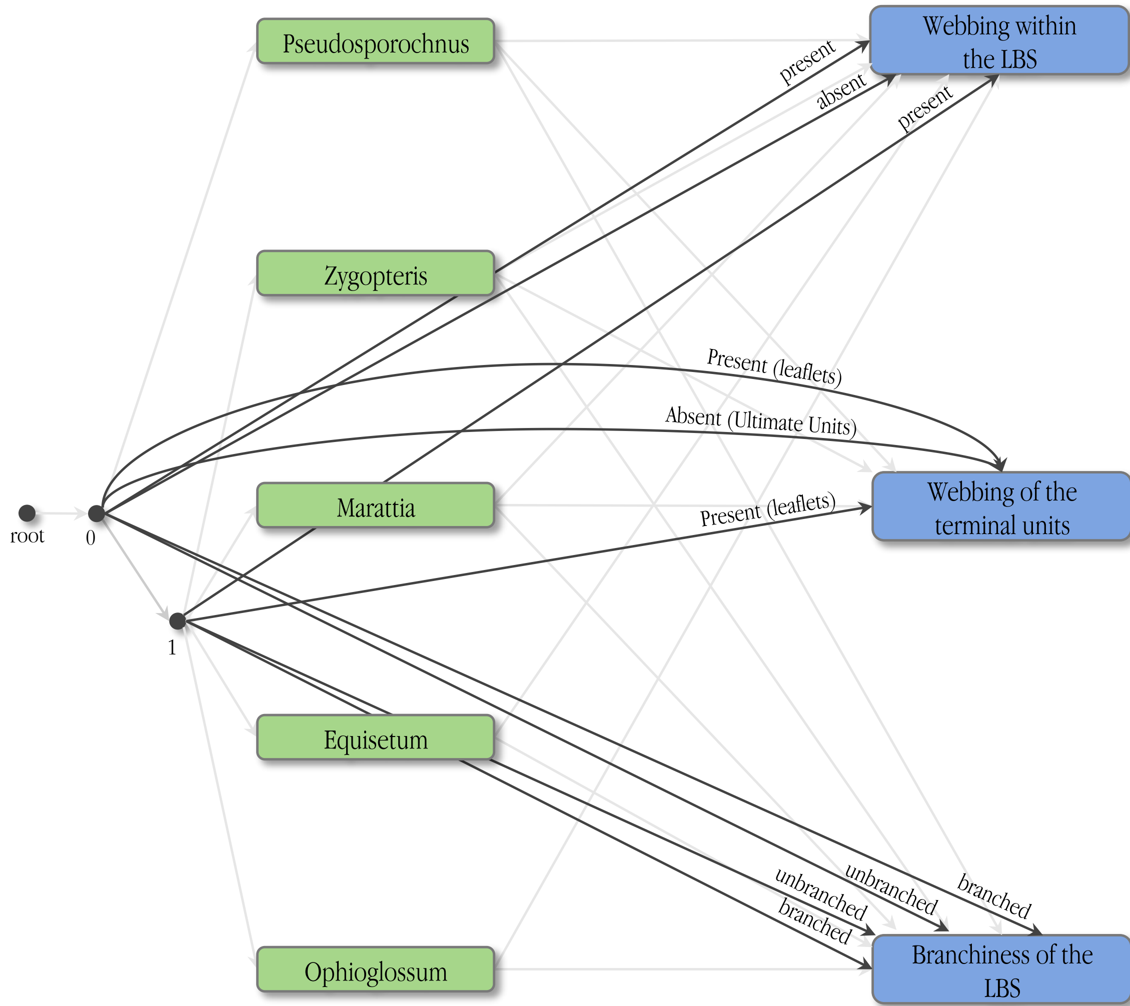

Figure 6: Bottom Up Aggregation

The second part of the algorithm is called *TopDownRefining*. This method is triggered after the *BottomUpAggregation* method, going to the same starting node provided in the *BottomUpAggregation* method. It starts from a given node (node_n) traversing down the three and, in every HTU it reaches, it subtracts the set of character-states that starts in its children nodes ($\text{node}_{children}$) and points to a given character ($\text{node}_{character}$), from the set of character-states starting from itself (node_n) pointing to the same character node ($\text{node}_{character}$).

For example, in Figure 6, consider the least nested node ($node_0$), just after the root and linked to the *Webbing within the LBS* character node ($node_{webbingLBS}$). There are two edges connecting the

$node_0$ and the $node_{webbingLBS}$ with values *present* and *absent*. The *present* edge comes from the most nested part of the tree, composed of the nodes *Zygopteris*, *Marattia*, *Equisetum* and *Ophioglossum*, nested by node 1 ($node_1$) – see Figure 4. The *absent* comes from *Pseudosporochnus* node – see Figure 4.

When the algorithm reaches $node_0$ it will subtracts the set of character-states (edges) outgoing from *Pseudosporochnus* toward $node_{webbingLBS}$ from the set of outgoing character-states (edges) outgoing from $node_0$ toward $node_{webbingLBS}$. This set subtraction will be {*present*, *absent*} – {*absent*} = {*present*}. If the set subtraction result is not empty, it creates an edge called *"EvolvedTrait"* from itself ($node_0$) toward the character ($node_{webbingLBS}$) as shown in Figure 7.

Also, the algorithm will subtracts the set of character-states (edges) outgoing from $node_1$ toward $node_{webbingLBS}$ from the set of character-states (edges) outgoing from $node_0$ toward $node_{webbingLBS}$. This set subtraction will also not be empty ({*present*, *absent*} – {*present*} = {*absent*}) but the *"EvolvedTrait"* edge is created only once between $node_0$ and $node_{webbingLBS}$.

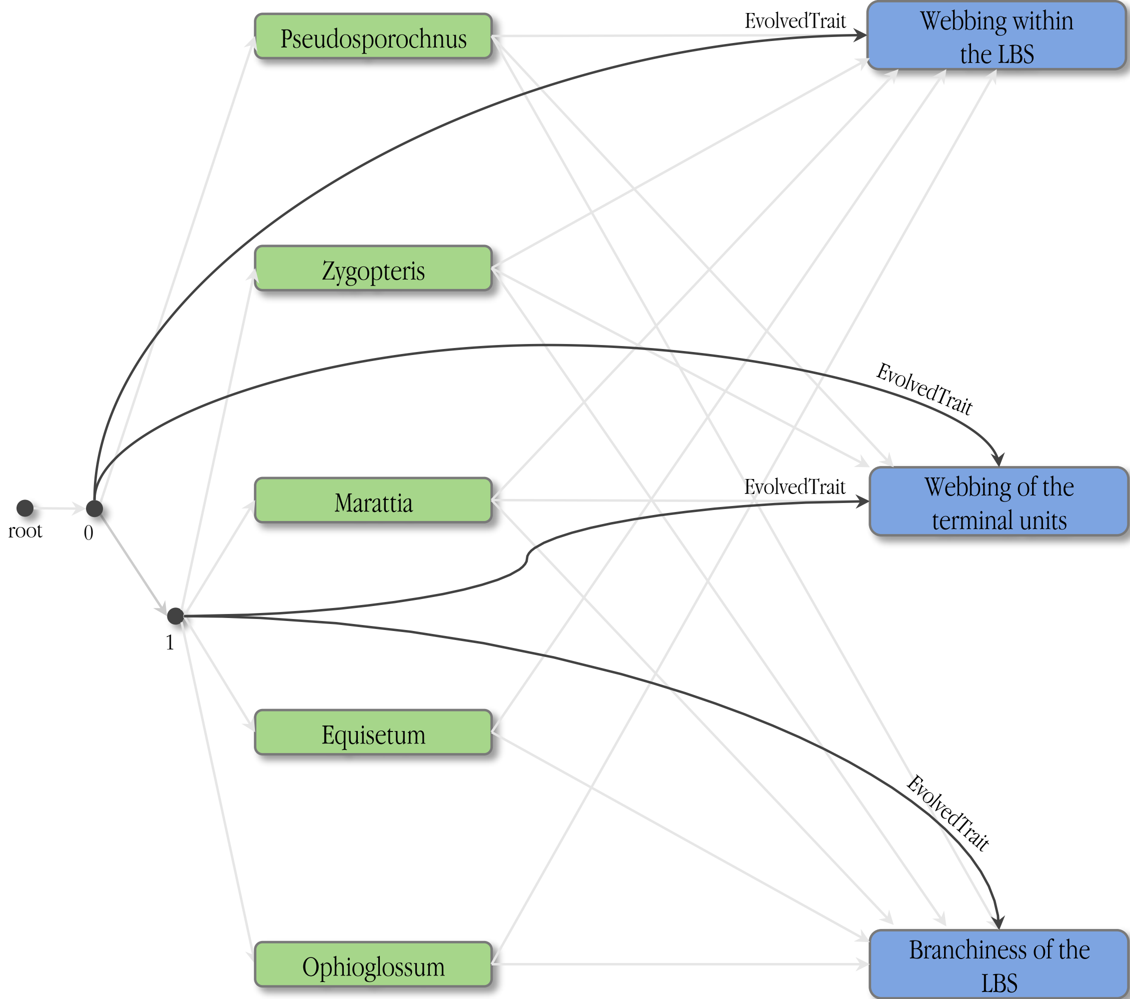

Figure 7: Top Down Refining

In a second iteration, the algorithm will reach $node_1$ and it will individually subtracts $node_1$ children nodes (*Zygopteris*, *Marattia*, *Equisetum* and *Ophioglossum*) outgoing character-states toward $node_{webbingLBS}$ from the set of character-states outgoing from $node_1$ toward $node_{webbingLBS}$. All those set subtractions will be $\{present\} - \{present\} = \emptyset$. In such a case (empty set, $\emptyset$), no *"EvolvedTrait"* edge is created, as can be seen in Figure 7.

Finally there is a visual tool that presents to the user the tree structure with all characters flagged with the "EvolvedTrait" edge, i.e. the characters that the algorithm "suspect" of being responsible for the branching. Figure 8 is a screenshot of our visual tool.

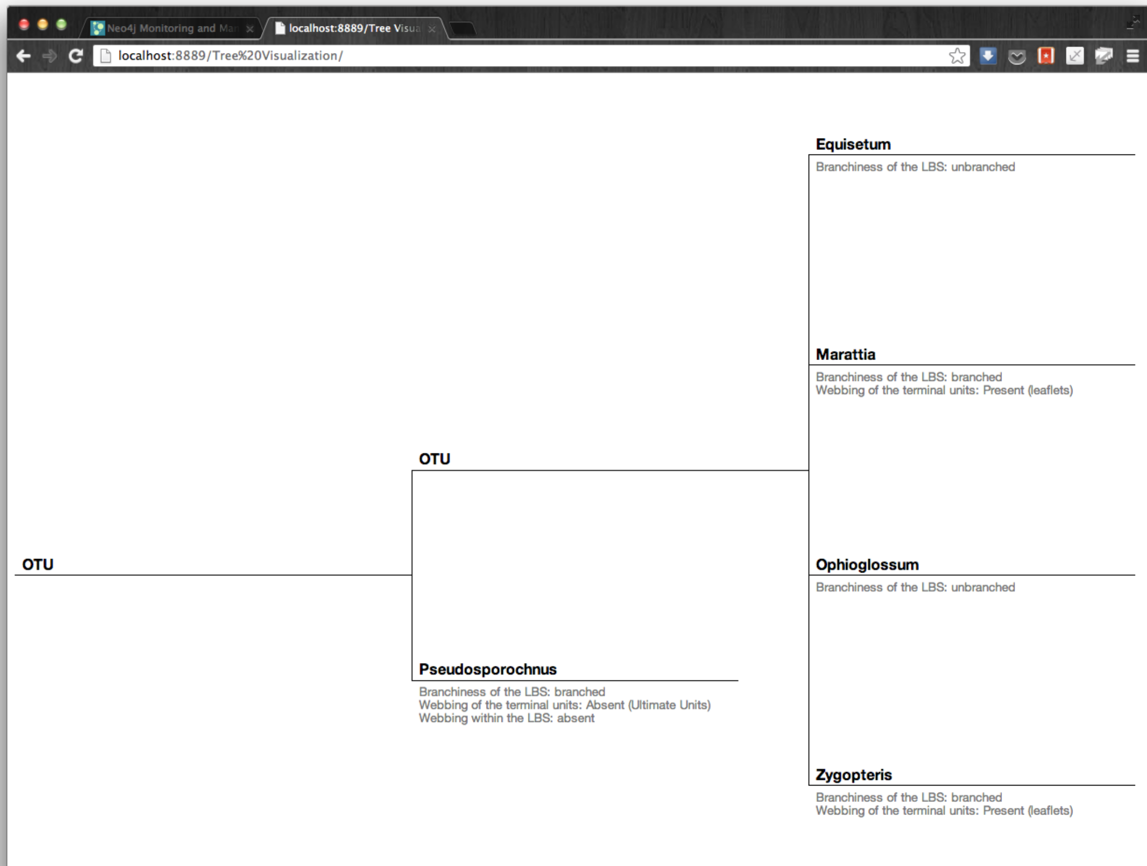

Figure 8: Evolved Traits Visualization

4 Conclusion

In this technical report we showed the main functionalities and operational features of the system. We mapped the SDD format to the graph model, remodeling semi-structured descriptions to a graph abstraction, in which the data are linked enabling coupling phylogenetic trees and phenotype descriptions. We drilled down the interconnection process through LSID unification, showing the

required steps to obtain a valid LSID and implementation details of the services used in this process. We presented details regarding a visualization tool implemented on top of the D3.js, to visualize our proposed similarity measure. Such a solution will not only help discovering characters similarity, but will be very important in the next stage of this project, which is the mapping from the graph towards ontologies. Furthermore, an algorithm to trace the phylogenetic history of traits changes has been shown. Finally, *Cypher* database queries and the main classes and methods of the system were provided with detailed comments for each method.

References

[1] Miranda, E., Santanchè, A.: Unifying phenotypes to support semantic descriptions. VI Brazilian Conference on Ontological Research (Ontobras) (09 2013)

[2] Parr, C.S., Guralnick, R., Cellinese, N., Page, R.D.: Evolutionary informatics: unifying knowledge about the diversity of life. Trends in ecology & evolution **27**(2) (2012) 94–103

[3] Ciccarelli, F.D., Doerks, T., Von Mering, C., Creevey, C.J., Snel, B., Bork, P.: Toward automatic reconstruction of a highly resolved tree of life. Science **311**(5765) (2006) 1283–1287

[4] Delsuc, F., Brinkmann, H., Philippe, H.: Phylogenomics and the reconstruction of the tree of life. Nature Reviews Genetics **6**(5) (2005) 361–375

[5] Miller, M.A., Pfeiffer, W., Schwartz, T.: Creating the cipres science gateway for inference of large phylogenetic trees. In: Gateway Computing Environments Workshop (GCE), 2010, IEEE (2010) 1–8

[6] Mabee, P.M.: Integrating evolution and development: the need for bioinformatics in evo-devo. BioScience **56**(4) (2006) 301–309

[7] Ung, V., Causse, F., Vignes Lebbe, R.: Xper^2: managing descriptive data from their collection to e-monographs. (2010)

[8] Ung, V., Dubus, G., Zaragüeta-Bagils, R., Vignes-Lebbe, R.: Xper2: introducing e-taxonomy. Bioinformatics **26**(5) (2010) 703–704

[9] Hagedorn, G.: Structuring Descriptive Data of Organisms – Requirement Analysis and Information Models. PhD thesis, Universität Bayreuth,Fakultät für Biologie, Chemie und Geowissenschaften (11 2007)

[10] Page, R.: Biodiversity informatics: the challenge of linking data and the role of shared identifiers. Briefings in Bioinformatics **9**(5) (2008) 345–354

[11] Godfray, H., et al.: Challenges for taxonomy. Nature **417**(6884) (2002) 17–19

[12] Adler, P.H., Crosskey, R.W.: World blackflies (diptera: Simuliidae): a comprehensive revision of the taxonomic and geographical inventory [2013] (2013) Accessed on July 08 2013.

[13] Rees, T.: Taxamatch, a "fuzzy" matching algorithm for taxon names, and potential applications in taxonomic databases. In Weitzman, A., Belbin, L., eds.: Provisional Abstracts of the 2008 Annual Conference of the Taxonomic Databases Working Group, Fremantle, Australia, Biodiversity Information Standards (TDWG) and the Missouri Botanical Garden (2008)

[14] Kennedy, J., Kukla, R., Paterson, T.: Scientific names are ambiguous as identifiers for biological taxa: Their context and definition are required for accurate data integration. In: 2nd Intl. Workshop on Data Integration in the Life Sciences (DILS). LNCS 3615 (July 2005) 80–95

[15] Patterson, D., Cooper, J., Kirk, P., Pyle, R., Remsen, D.: Names are key to the big new biology. Trends in ecology & evolution **25**(12) (2010) 686–691

[16] Bisby, F.: The quiet revolution: biodiversity informatics and the internet. Science **289**(5488) (2000) 2309–2312

[17] Clark, T., Martin, S., Liefeld, T.: Globally distributed object identification for biological knowledgebases. Briefings in bioinformatics **5**(1) (2004) 59–70

[18] Angles, R.: A comparison of current graph database models. In: Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on. (2012) 171–177

[19] Angles, R., Gutierrez, C.: Survey of graph database models. ACM Computing Surveys (CSUR) **40**(1) (2008) 1

[20] Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly Media, Inc. (2013)

[21] Rodriguez, M.A., Shinavier, J.: Exposing multi-relational networks to single-relational network analysis algorithms. Journal of Informetrics **4**(1) (2010) 29 – 41

[22] Bagils, R.Z., Ung, V., Grand, A., Vignes-Lebbe, R., Cao, N., Ducasse, J.: Lisbeth: New cladistics for phylogenetics and biogeography. Comptes Rendus Palevol **11**(8) (2012) 563 – 566

[23] Nelson, G., Platnick, N.I.: Three-taxon statements: A more precise use of parsimony? Cladistics **7**(4) (1991) 351–366

A Demonstration

In this section we present the source code of the system, according to the graph data model presented in previous sections. The code is modularized in files and each file has a class with methods, all with comments explaining their functionality.

A.1 SDDParser.py

```
1  import os,sys
2
3  from xml.dom import minidom
4  from collections import OrderedDict
5
6  from Representation import *
7  from StateDefinition import *
8  from CategoricalCharacter import *
9  from Categorical import *
10 from CodedDescription import *
11
12 class SDDParser:
13
14  def __init__(self, SDDFile):
15
16   self.CategoricalCharacters = self.__parseCategoricalCharacter(
       SDDFile )
17   self.CodedDescriptions  = self.__parseCodedDescription( SDDFile )
18
19  def __parseRepresentation(self, Repr) :
20   """
21   Representation is a plain text label and description block found
       inside CategoricalCharacter, StateDefinition and
       CodedDescription blocks.
22   Args: A XML Representation block and its content.
23   Returns: A SDD Representation object.
24   """
25
26   label = ''
27   detail = ''
28
29   if Repr :
30
31    if 0 < Repr.getElementsByTagName('Label').length :
32     label = Repr.getElementsByTagName('Label')[0].childNodes[0].
       nodeValue.strip()
33
34    if 0 < Repr.getElementsByTagName('Detail').length :
35     detail = Repr.getElementsByTagName('Detail')[0].childNodes[0].
       nodeValue.strip()
```

```
36
37   return Representation( label, detail )
38
39
40  def __parseStateDefinitions(self, StateDefinitions):
41   """
42   StateDefinition has its own id and a Representation block. It is
        define inside the CategoricalCharacter/States block in which
        the States groups together all possible states (StateDefinition
        ) observed at a given Categorical Character.
43   Args: All XML StateDefinition blocks of a particular
        CategoricalCharacter/States block.
44   Returns: A dictionary of StateDefinition object.
45   """
46
47   # Dictionary with all state definition nodes
48   SStateDefinitionsDictionary = {}
49
50   for State in StateDefinitions :
51
52    Id = State.getAttributeNode('id').nodeValue
53
54    Repr = State.getElementsByTagName('Representation')[0]
55
56    Representation = self.__parseRepresentation( Repr )
57
58    # Add node to Dictionary
59    SStateDefinitionsDictionary[Id] = StateDefinition( Id ,
        Representation )
60
61   return SStateDefinitionsDictionary
62
63
64  def __parseStates(self, States):
65   """
66   State is define inside CodedDescription/SummaryData/Categorical
        and it links a taxon CategoricalCharacter to its possible
        States through the ref parameters.
67   Args: All XML State blocks of a particular CodedDescription/
        SummaryData/Categorical block.
68   Returns: An array with StateDefinitions references.
69   """
70
71   # Array with references to StateDefinitions
72   StatesDictionary = []
73
74   for state in States :
75
76    ref = state.getAttributeNode('ref').nodeValue
```

```
77        StatesDictionary.append( ref )
78
79      return StatesDictionary
80
81
82    def __parseSummaryData(self, Categoricals):
83      """
84      Categorical is a reference to a CategoricalCharacter object and is
          composed by a list of references to possible states that a
          given taxon can take.
85      Args: All XML Categorical blocks of a particular CodedDescription/
          SummaryData block.
86      Returns: A dictionary of Categorical objects.
87      """
88
89      # Dictionary of Categorical objects
90      SummaryDataDictionary = {}
91
92      for c in Categoricals :
93
94        ref = c.getAttributeNode('ref').nodeValue
95
96        s = c.getElementsByTagName('State')
97
98        States = self.__parseStates( s )
99
100       SummaryDataDictionary[ref] = Categorical( ref , States )
101
102     return SummaryDataDictionary
103
104
105   def __parseCategoricalCharacter(self, SDDFile):
106     """
107     CategoricalCharacter has its own id, a Representation block and a
          States block.
108     Args: A SDD file name.
109     Returns: A dictionary with all CategoricalCharacters objects in
          the given file.
110     """
111
112     CC = SDDFile.getElementsByTagName('CategoricalCharacter')
113
114     # Dictionary with all CategoricalCharacters objects
115     CategoricalCharacters = {}
116
117     for Character in CC:
118
119       Id = Character.getAttributeNode('id').nodeValue
120
```

```
121     States = Character.getElementsByTagName('StateDefinition')
122     Repr   = Character.getElementsByTagName('Representation')[0]
123 
124     Representation    = self.__parseRepresentation( Repr )
125     SStateDefinitionsDictionary = self.__parseStateDefinitions(
           States )
126 
127     CategoricalCharacters[ Id ] = CategoricalCharacter( Id ,
           SStateDefinitionsDictionary, Representation )
128 
129    return CategoricalCharacters
130 
131 
132   def __parseCodedDescription(self, SDDFile):
133    """
134    CodedDescription has its own id, a Representation block and a
          SummaryData block.
135    Args: A SDD file name.
136    Returns: A dictionary with all CodedDescription objects in the
          given file.
137    """
138 
139    CD = SDDFile.getElementsByTagName('CodedDescription')
140 
141    # Dictionary with all CodedDescriptions objects
142    CodedDescriptions = {}
143 
144    for Description in CD:
145 
146     Id = Description.getAttributeNode('id').nodeValue
147 
148     SD   = Description.getElementsByTagName('Categorical')
149     Repr = Description.getElementsByTagName('Representation')[0]
150 
151     Representation  = self.__parseRepresentation( Repr )
152     SummaryDataDictionary = self.__parseSummaryData( SD )
153 
154     CodedDescriptions[ Id ] = CodedDescription( Id ,
           SummaryDataDictionary, Representation )
155 
156    return CodedDescriptions
157 
158 
159   def getAllSates(self):
160    """
161    Returns a dictionary of all 'StateDefinitions' elements.
162    """
163 
164    States = {}
```

```
165
166     for key, CategoricalCharacter in self.CategoricalCharacters.
            iteritems():
167
168       States.update( CategoricalCharacter.States )
169
170     OrderedStates = OrderedDict( sorted( States.items() ) )
171
172     return OrderedStates
173
174
175   def getAllTaxons(self):
176     """
177     Returns a list of all taxons elements.
178     """
179
180     Taxons = []
181
182     for key, CodedDescription in self.CodedDescriptions.iteritems():
183
184       Taxons.append( CodedDescription.Representation )
185
186     return Taxons
187
188
189   def getAllCharacters(self):
190     """
191     Returns a dictionary of all 'CategoricalCharacter' elements.
192     """
193
194     Characters = {}
195
196     for key, CategoricalCharacter in self.CategoricalCharacters.
            iteritems():
197
198       Characters[ CategoricalCharacter.id ] = ( CategoricalCharacter.
              Representation )
199
200     OrderedCharacters = OrderedDict( sorted( Characters.items() ) )
201
202     return OrderedCharacters
```

A.2 TeeOutput.py

```
import re
import shlex
import mmap
import sys

from TreeNode import *
from NodeTypes import *

class TreeOutput:

 def __init__(self, _TreeOutputFile ):

  self.TreeOutputFile = _TreeOutputFile


 def __parseNewickTree( self, NewickTree , parentNode ) :
  """
  Newick tree format (New Hampshire tree format) is a way of
     representing trees in computer-readable form using parentheses
     and commas.
  Args:
   NewickTree: A NewickTree string. For example: (((((((12 18) 22)
     13) 3) 7) 30) (23 25))
   parentNode: A node to where NewickTree tree will be attached to.
  """

  opened = False
  substring = NewickTree

  i = j = begin = end = 0

  for c in NewickTree :

   if c == '(' :
    i += 1

    if not opened :
     begin = j

    opened = True

   elif c == ')' :
    i -= 1

   if opened and i == 0 :
    # (opened and i == 0) means that opening round bracket '(' and
       the corresponding closing round bracket ')' was found.
```

```
44      # It will recursively call __parseNewickTree with brackets
            content. Also, it will remove parentheses block and content
            from NewickTree.
45
46      opened = False
47
48      childrenWithBrackets  = NewickTree[ begin : j + 1 ]
49      childrenWithNoBrackets = NewickTree[ begin + 1 : j ]
50
51      child = TreeNode(None)
52      parentNode.appendChild( child )
53
54      self.__parseNewickTree( childrenWithNoBrackets , child )
55
56      substring = substring.replace( childrenWithBrackets, "" )
57
58     j += 1
59
60   if "(" not in substring:
61    # When this condition is satisfied, it means that substring will
        only have leaves nodes or it is empty.
62
63     my_splitter = shlex.shlex(substring, posix = True )
64     my_splitter.whitespace += ','
65     my_splitter.whitespace_split = True
66
67     for n in my_splitter :
68      parentNode.appendChild( TreeNode(n) )
69
70
71  def getNewickTree( self ) :
72   """
73   This method looks into the file in search of the Newick Tree and
       returns it.
74   The process is pretty straightforward:
75    1. Set the file's current position to the o occurence of '
        Retained trees'
76    2. Reads this line and discards it
77    3. Reads the next line, which supposedly should contain the
        Newick Tree
78    4. Get the Newick Tree
79   """
80
81   _file = open( self.TreeOutputFile )
82   memorymap = mmap.mmap( _file.fileno(), 0, access = mmap.
       ACCESS_READ )
83
84   RetainedTreesPosition = memorymap.find("Retained trees")
85   memorymap.seek( RetainedTreesPosition )
```

```
86     memorymap.readline()
87     FirstRetainedTreeLine = memorymap.readline()
88     memorymap.close()
89 
90     # First occurence of ')'
91     begin = FirstRetainedTreeLine.find('(')
92 
93     # Last occurence of '('
94     end  = FirstRetainedTreeLine.rfind(')')
95 
96     NewickTree = FirstRetainedTreeLine[ begin : end + 1 ]
97 
98     return NewickTree
99 
100
101   def getTaxons( self ) :
102     """
103     Get all taxons listed right bellow 'Taxa (# taxons)' inside —<D02
          >— block and return all those taxons.
104     """
105
106     _file = open( self.TreeOutputFile )
107     memorymap = mmap.mmap( _file.fileno(), 0, access = mmap.
          ACCESS_READ )
108
109     BlockBegin = memorymap.find("<D02>")
110     BlockEnd  = memorymap.find("<F02>")
111
112     TaxaPosition = memorymap.find( "Taxa" , BlockBegin , BlockEnd )
113
114     memorymap.seek( TaxaPosition )
115     TaxaLine = memorymap.readline()
116     TotalTaxa = int( re.search( re.escape( '(' ) + "(.*?)" + re.escape
          ( ')' ), TaxaLine ).group(1) )
117
118     TaxonsDictionary = {}
119
120     for i in range( TotalTaxa ):
121
122      line = memorymap.readline()
123
124      index = line[ 1 : 21 ].strip()
125      taxon = line[ 22 : ].strip()
126
127      TaxonsDictionary[ index ] = taxon
128
129     memorymap.close()
130
131     return TaxonsDictionary
```

```
132
133
134 def __RenameTreeNodes(self, subTree , TaxonsDictionary ):
135
136     """
137     In a rooted phylogenetic tree, each node is called a taxonomic
        unit. Internal nodes are generally called hypothetical
        taxonomic units (HTUs) as they cannot be directly observed.
138     Args:
139      subTree: Is a branch of the tree.
140      TaxonsDictionary: A list of taxons present in the 3iz file.
141     """
142
143     if subTree.nodes:
144
145      subTree.value = str( NodeTypes.HTU )
146
147      for n in subTree.nodes:
148       self.__RenameTreeNodes( n, TaxonsDictionary )
149
150     else:
151      subTree.value = TaxonsDictionary[ subTree.value ]
152
153
154 def getTaxonsTreeStructure( self ):
155     """
156     It parse the NewickTree string into a tree structure with
        Hypothetical Taxonomic Units as internal nodes and the correct
        Taxon name as the leaves.
157     """
158
159     NewickTree = self.getNewickTree()
160     TaxonsDictionary = self.getTaxons()
161
162     root = TreeNode(None)
163     self.__parseNewickTree( NewickTree , root )
164
165     self.__RenameTreeNodes( root , TaxonsDictionary )
166
167     return root
```

A.3 GlobalNamesResolver.py

```
from bs4 import BeautifulSoup
from GNRResultObject import *
import urllib2
from enumerator import *

class GlobalNamesResolver:

 def __init__(self) :
  self.url = 'http://resolver.globalnames.org/name_resolvers.xml?
     names='

  # Names Data Sources <http://resolver.globalnames.org/data_sources
     >
  # ID   Source
  # 169  uBio NameBank
  #   1  Catalogue of Life
  #   3  ITIS
  self.DataSources = enum( CatalogueOfLife = 1, ITIS = 3,
     uBioNameBank = 169 )

  self.DataSourceIds = [self.DataSources.CatalogueOfLife, self.
     DataSources.ITIS, self.DataSources.uBioNameBank ]


 def getResultsObjects( self, ScientificName ):

  ScientificName = ScientificName.replace(' ' , '%20')

  url = self.url + ScientificName

  if len( self.DataSourceIds ) > 0 :
   url = url + '&data_source_ids='

   for _id in self.DataSourceIds :
    url = url + str( _id ) + '|'

  try:
   GNRServiceUrlResponse = urllib2.urlopen( url ).read()

  except urllib2.HTTPError, e:
   print "HTTP error: %d" % e.code
  except urllib2.URLError, e:
   print "Network error: %s" % e.reason.args[1]

  SoupGNRResponse = BeautifulSoup( GNRServiceUrlResponse )

  results = SoupGNRResponse.findAll('result')
```

```
44
45    GNRResultObjects = []
46
47    for result in results :
48
49     DataSourceId    = result.find('data-source-id', {'type': 'integer
         '} )
50     DataSourceTitle = result.find('data-source-title')
51     gniUUID         = result.find('gni-uuid')
52     NameString      = result.find('name-string')
53     CanonicalForm   = result.find('canonical-form')
54     TaxonId         = result.find('taxon-id')
55     LocalId         = result.find('local-id')
56     MatchType       = result.find('match-type', {'type': 'integer'} )
57     Prescore        = result.find('prescore')
58     Score           = result.find('score', {'type': 'float'} )
59
60     DataSourceId    = DataSourceId.contents[0]    if DataSourceId
         else ""
61     DataSourceTitle = DataSourceTitle.contents[0] if DataSourceTitle
         else ""
62     gniUUID         = gniUUID.contents[0]         if gniUUID
         else ""
63     NameString      = NameString.contents[0]      if NameString
         else ""
64     CanonicalForm   = CanonicalForm.contents[0]   if CanonicalForm
         else ""
65     TaxonId         = TaxonId.contents[0]         if TaxonId
         else ""
66     LocalId         = LocalId.contents[0]         if LocalId
         else ""
67     MatchType       = MatchType.contents[0]       if MatchType
         else ""
68     Prescore        = Prescore.contents[0]        if Prescore
         else ""
69     Score           = Score.contents[0]           if Score
         else ""
70
71     obj = GNRResultObject( DataSourceId, DataSourceTitle, gniUUID,
         NameString, CanonicalForm, TaxonId, LocalId, MatchType ,
         Prescore , Score )
72
73     GNRResultObjects.append( obj )
74
75    return GNRResultObjects
76
77
78  def getCanonicalForm( self, ScientificName ) :
79    """
```

```
80   Returns the canonical forms of a given scientific name.
81   """
82
83   objects = self.getResultsObjects( ScientificName )
84
85   CanonicalForms = set( [ ] )
86
87   for obj in objects :
88
89    match = int(obj.MatchType)
90
91    # 1 - Exact match
92    # 2 - Exact match by canonical form
93    # 3 - Fuzzy match by canonical form
94    if match == 1 or match == 2 or match == 3 :
95
96     if 0.988 <= float(obj.MatchType) :
97
98      # Add canonical form to the set
99      CanonicalForms = CanonicalForms | set( [ obj.CanonicalForm ] )
100
101  if 1 == len( CanonicalForms ):
102
103   return sorted(CanonicalForms)[0]
104
105  return None
106
107
108 def getLSIDFromCanonicalForm( self, CanonicalForm ) :
109  """
110  Returns the LSID of a given Canonical Form. Only uBio NameBank
        LSID are retrieved and still only if a exact match occur.
111   """
112
113  ResultsObjects = self.getResultsObjects( CanonicalForm )
114
115  for obj in ResultsObjects :
116
117  if int(obj.MatchType) == 1:
118
119    if int(obj.DataSourceId) == self.DataSources.uBioNameBank:
120
121     return obj.LocalId
122
123  return None
```

A.4 GNRResultObject.py

```
class GNRResultObject:

 def __init__(self, _DataSourceId, _DataSourceTitle, _gniUUID,
    _NameString, _CanonicalForm, _TaxonId, _LocalId, _MatchType,
    _Prescore, _Score ):

  # The id of the data source where a name was found.
  self.DataSourceId = _DataSourceId

  # The data source title where a name was found.
  self.DataSourceTitle = _DataSourceTitle

  # An identifier for the found name string used in Global Names.
  self.gniUUID = _gniUUID

  # The name string found in this data source.
  self.NameString = _NameString

  # A "canonical" version of the name generated by the Global Names
     parser
  self.CanonicalForm = _CanonicalForm

  # Tree path to the root if a name string was found within a data
     source classification.
  # self.ClassificationPath

  # self.ClassificationPathRanks

  # Same tree path using taxon_ids
  # self.ClassificationPathIds

  # An identifier supplied in the source Darwin Core Archive for the
      name string record
  self.TaxonId = _TaxonId

  # Shows id local to the data source (if provided by the data
     source manager)
  self.LocalId = _LocalId

  # Explains how resolver found the name. If the resolver cannot
     find names corresponding to the entire queried name string, it
     sequentially removes terminal portions of the name string until
      a match is found.
  # 1 - Exact match
  # 2 - Exact match by canonical form of a name
  # 3 - Fuzzy match by canonical form
  # 4 - Partial exact match by species part of canonical form
```

```
39 # 5 - Partial fuzzy match by species part of canonical form
40 # 6 - Exact match by genus part of a canonical form
41 self.MatchType = _MatchType
42
43 # Displays points used to calculate the score delimited by '|' --
     "Match points|Author match points|Context points". Negative
     points decrease the final result.
44 self.Prescore = _Prescore
45
46 # A confidence score calculated for the match.
47 # 0.5 means an uncertain result that will require investigation.
48 # Results higher than 0.9 correspond to 'good' matches.
49 # Results between 0.5 and 0.9 should be taken with caution.
50 # Results less than 0.5 are likely poor matches.
51 # The scoring is described in more details on http://resolver.
     globalnames.org/about
52 self.Score = _Score
```

A.5 ITISServices.py

```
import suds

class ITISServices:

 url = "http://www.itis.gov/ITISWebService.xml"
 client = None

 def __init__(self):
  self.client = suds.client.Client( self.url )


 def getTSNfromScientificName(self, ScientificName ):
  """
  Taxonomic Serial Number (TSN) which is the primary key for the
      scientific name. This method returns a TSN if the provided
      ScientificName is found and None otherwise.
  """

  self.client.service.searchByScientificName( ScientificName )

  ScientificNamesResponse = self.client.last_received().getChild("
      soapenv:Envelope").getChild("soapenv:Body").getChild("ns:
      searchByScientificNameResponse").getChild("ns:return").
      getChildren("ax21:scientificNames")

  for sn in ScientificNamesResponse:

   tsn = sn.getChild("ax21:tsn")

   if tsn != None :
    return tsn.getText()

  return None


 def getLSIDfromTSN(self, tsn ):
 """
 Given a TSN this method returns a LSID if found and None otherwise.
 """

  self.client.service.getLSIDFromTSN( tsn )

  LSID = self.client.last_received().getChild("soapenv:Envelope").
      getChild("soapenv:Body").getChild("ns:getLSIDFromTSNResponse").
      getChild("ns:return").getText()

  if LSID:
```

```
41      return LSID
42
43    return None
```

A.6 CoLServices.py

```
from BeautifulSoup import BeautifulSoup
import urllib2

class CoLServices:
 """
 This class contains the main methods to interact with the CoL web
    service.
 """

 def getCoLUrl( self, ScientificName ):
  """
  This method uses a XML scraping technique to get the URL of the
     given Scientific Name from the webservice response.
  """

  url = 'http://www.catalogueoflife.org/col/webservice?name='

  ScientificName = ScientificName.replace(' ' , '%20')

  try:
   CoLWebServiceUrlResponse = urllib2.urlopen(url + ScientificName).
      read()
  except urllib2.HTTPError, e:
   print "HTTP error: %d" % e.code
  except urllib2.URLError, e:
   print "Network error: %s" % e.reason.args[1]

  SoupCoLWebServiceResponse = BeautifulSoup(CoLWebServiceUrlResponse
     )

  tagresult = SoupCoLWebServiceResponse.findAll('result')

  CoLUrl = tagresult[0].find('url').contents[0]

  if CoLUrl:
   return CoLUrl


 def getCoLSpecieID( self, ScientificName ):
  """
  This method uses a XML scraping technique to get the ID of the
     given Scientific Name from the webservice response.
  """

  url = 'http://www.catalogueoflife.org/testcol/webservice?name='

  ScientificName = ScientificName.replace(' ' , '%20')
```

```
43
44    try:
45     CoLWebServiceUrlResponse = urllib2.urlopen(url + ScientificName).
          read()
46    except urllib2.HTTPError, e:
47     print "HTTP error: %d" % e.code
48    except urllib2.URLError, e:
49     print "Network error: %s" % e.reason.args[1]
50
51    SoupCoLWebServiceResponse = BeautifulSoup(CoLWebServiceUrlResponse
        )
52
53    result = SoupCoLWebServiceResponse.find('result')
54
55    if result:
56     findID = result.find('id')
57
58     if findID:
59      SpecieID = findID.contents[0]
60
61      if SpecieID:
62       return SpecieID
63
64    return None
65
66
67   def getLSIDfromSpecieID( self, SpecieID ):
68    """
69    This method uses a HTML screen-scraping technique to get the LSID
        of the given SpecieID.
70    """
71
72    url = 'http://www.catalogueoflife.org/testcol/details/species/id/'
73
74    try:
75     SpecieDetailsCoLUrlResponse = urllib2.urlopen(url + SpecieID).
          read()
76    except urllib2.HTTPError, e:
77     print "HTTP error: %d" % e.code
78    except urllib2.URLError, e:
79     print "Network error: %s" % e.reason.args[1]
80
81    SoupSpecieDetailsCoLUrlResponse = BeautifulSoup(
        SpecieDetailsCoLUrlResponse)
82
83    LSID = SoupSpecieDetailsCoLUrlResponse.find('span', {'class': '
        lsid'}).contents[0]
84
85    return LSID
```

```
86
87
88  def getLSIDfromSpecieUrl( self, SpecieUrl ):
89   """
90   This method uses a HTML screen-scraping technique to get the LSID
        of the given SpecieUrl.
91   """
92
93   try:
94    SpecieDetailsCoLUrlResponse = urllib2.urlopen(SpecieUrl).read()
95   except urllib2.HTTPError, e:
96    print "HTTP error: %d" % e.code
97   except urllib2.URLError, e:
98    print "Network error: %s" % e.reason.args[1]
99
100  SoupSpecieDetailsCoLUrlResponse = BeautifulSoup(
        SpecieDetailsCoLUrlResponse)
101
102  LSID = SoupSpecieDetailsCoLUrlResponse.find('span', {'class': '
        lsid'}).contents[0]
103
104  return LSID
```

A.7 GraphImporter.py

```
from py2neo import rest, neo4j, cypher

from SDDParser import *
from TreeOutput import *
from GlobalNamesResolver import *
from GraphDB import *
from NodeTypes import *
from RelationshipTypes import *
from ITISServices import *
from CoLServices import *

class GraphImporter:

 SDDFilename = None
 TreeFilename = None

 def __init__(self, _SDDFilename, _TreeFilename, _IgnoreTreeFilename
     ):

  self.SDDFilename = _SDDFilename
  self.TreeFilename = _TreeFilename
  self.IgnoreTreeFilename = _IgnoreTreeFilename


 def __CreateTaxonsNodes(self, CodedDescriptions ) :
  """
  Add to the Graph DB all taxons elements as nodes. In case the
     taxon node already exists, it uses the node in GraphDB rather
     than create a new one.
  Args: CodedDescriptions: A list of all Coded Descriptions elements
     .
  Returns: A dict mapping keys to the corresponding added nodes.
     Each tuple is represented as (Taxon Name , node) where the
     first element of the tuple is the taxon name and the last one
     is the node itself.
   Example:
   {u'Equisetum'  : [Node('http://localhost:7474/db/data/node/142')
      ],
    u'Marattia' : [Node('http://localhost:7474/db/data/node/131')],
    u'Botryopteris': [Node('http://localhost:7474/db/data/node/222')
      ]}
  """

  gdb = GraphDB()
  GDBConn, msg = gdb.getPy2neoGraphDatabaseService()

  if GDBConn is not None:
```

```
39
40    # Dictionary for all taxons nodes
41    TaxonsNodes = {}
42
43    GNR  = GlobalNamesResolver()
44    ITIS = ITISServices()
45    CoL  = CoLServices()
46
47    for key, CodedDescription in CodedDescriptions.iteritems():
48
49     node = None
50
51     taxonName = CodedDescription.Representation.label
52     taxonNameCF = GNR.getCanonicalForm( taxonName )
53
54     lsid = GNR.getLSIDFromCanonicalForm( taxonNameCF )
55
56     if lsid == None :
57       tsn = ITIS.getTSNfromScientificName( taxonNameCF )
58       lsid = ITIS.getLSIDfromTSN( tsn )
59
60       if lsid == None :
61         SpecieID = CoL.getCoLSpecieID( taxonNameCF )
62         lsid = CoL.getLSIDfromSpecieID( SpecieID )
63
64     n = gdb.getNodeByLSID( lsid )
65
66     if n is None:
67
68      # Create taxon node
69      node = GDBConn.create({ 'label' : taxonNameCF ,
70            'detail'   : CodedDescription.Representation.detail ,
71            'sourceId' : CodedDescription.id ,
72            'type'   : str( NodeTypes.OTU ) ,
73            'LSID'   : lsid })
74     else :
75      node = n
76
77     # Add node to Dictionary
78     TaxonsNodes[ CodedDescription.Representation.label ] = node
79
80    return TaxonsNodes
81
82   else:
83    print msg
84    return None
85
86
87  def __CreateStateDefinitionNodes(self, StateDefinitions ) :
```

```
88  """
89  Add to the Graph DB all state definition elements as nodes.
90  Args:
91   StateDefinitions: A dictionary of all 'StateDefinitions' elements
       .
92  Returns:A dict mapping keys to the corresponding added nodes. Each
        tuple is represented as (Id , node) where the first element of
        the tuple, Id (For example: s54) is the SDD.XML
       StateDefinition ID and the last one is the node itself.
93   Example:
94   {u's54': [Node('http://localhost:7474/db/data/node/142')],
95    u's43': [Node('http://localhost:7474/db/data/node/131')],
96    u's46': [Node('http://localhost:7474/db/data/node/222')]}
97  """
98
99  gdb = GraphDB()
100 GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
101
102 if GDBConn is not None:
103
104  # Dictionary for all state definition nodes
105  StateDefinitionsNodes = {}
106
107  for key, State in StateDefinitions.iteritems():
108
109   # Create state definition node
110   node = GDBConn.create({ 'label' : State.Representation.label ,
111         'detail'  : State.Representation.detail ,
112         'sourceId' : State.id ,
113         'type'  : str( NodeTypes.description ) })
114
115   # Add node to Dictionary
116   StateDefinitionsNodes[ State.id ] = node
117
118  return StateDefinitionsNodes
119
120 else:
121  print msg
122  return None
123
124
125 def __CreateCharacterNodes( self, Characters ) :
126 """
127 Add to the Graph DB all characters elements as nodes.
128 Args:
129  Characters: A dictionary of all 'Characters' elements.
130 Returns: A dict mapping keys to the corresponding added nodes.
       Each tuple is represented as (Id , node) where the first
       element of the tuple, Id (For example: c19) is the SDD.XML
```

```
       CategoricalCharacter ID and the last one is the node itself.
131     Example:
132     {u'c19': [Node('http://localhost:7474/db/data/node/396')],
133      u'c18': [Node('http://localhost:7474/db/data/node/395')],
134      u'c5' : [Node('http://localhost:7474/db/data/node/400')]}
135    """
136
137    gdb = GraphDB()
138    GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
139
140    if GDBConn is not None:
141
142     # Dictionary for all characters nodes
143     CharactersNodes = {}
144
145     for ID, Character in Characters.iteritems():
146
147      # Create state definition node
148      node = GDBConn.create({ 'label' : Character.label ,
149            'detail'  : Character.detail ,
150            'sourceId' : ID ,
151            'type'   : str( NodeTypes.description ) })
152
153      # Add node to Dictionary
154      CharactersNodes[ ID ] = node
155
156     return CharactersNodes
157
158    else:
159     print msg
160     return None
161
162
163   def __JoinTaxonsNodesTreeStructureRecursion(self, TaxonsNodes ,
        subTree, parentNode ):
164
165    gdb = GraphDB()
166    GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
167
168    if GDBConn is not None:
169
170     if subTree.nodes:
171
172      # Create Hypothetical Taxonomic Unit node
173      htuNode, = GDBConn.create({ 'label' : str( NodeTypes.HTU ) ,
174             'type' : str( NodeTypes.HTU ) })
175
176      # Join Hypothetical Taxonomic Unit node to its parent node
```

```
177        parentNode.create_relationship_to( htuNode , str(
              RelationshipTypes.TreeEdge ) , { "type" : str(
              RelationshipTypes.TreeEdge ) } )
178
179        for n in subTree.nodes:
180         self.__JoinTaxonsNodesTreeStructureRecursion( TaxonsNodes , n,
               htuNode )
181
182       else:
183        # Get Taxonomic Unit (taxon name) already created, passed
              through TaxonsNodes dictionary
184        tuNode = TaxonsNodes[ subTree.value ][0]
185
186        # Join Taxonomic Unit node to its parent node
187        parentNode.create_relationship_to( tuNode , str(
              RelationshipTypes.TreeEdge ) , { "type" : str(
              RelationshipTypes.TreeEdge ) } )
188
189      else:
190       print msg
191       return None
192
193
194    def __JoinTaxonsNodesTreeStructure(self, TaxonsNodes , Tree ):
195     """
196     Join taxons nodes with the Newick tree structure.
197     """
198
199     gdb = GraphDB()
200     GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
201
202     if GDBConn is not None:
203
204      self.__JoinTaxonsNodesTreeStructureRecursion( TaxonsNodes , Tree,
              gdb.getRootNode() )
205
206     else:
207      print msg
208      return None
209
210
211    def ImportUsingTaxonCharacterStateSchema( self ) :
212     """
213     Schema : Taxon(Node) -> CategoricalCharacter (Edge) ->
           StateDefinition (Node)
214     """
215
216     # Parse the SDD-XML file
217     SDDFile = minidom.parse( self.SDDFilename )
```

```
218
219 SDD = SDDParser( SDDFile )
220
221 CategoricalCharacters = SDD.CategoricalCharacters
222 CodedDescriptions  = SDD.CodedDescriptions
223
224 # Create Taxons nodes in the Graph DB
225 TaxonsNodes = self.__CreateTaxonsNodes( SDD.CodedDescriptions )
226
227 # Join Taxons nodes in a tree structure
228 treeOutput = TreeOutput( self.TreeFilename )
229 tree = treeOutput.getTaxonsTreeStructure()
230 self.__JoinTaxonsNodesTreeStructure( TaxonsNodes, tree )
231
232 # Create State Definition nodes in the Graph DB
233 StateDefinitionsNodes = self.__CreateStateDefinitionNodes( SDD.
        getAllSates() )
234
235 for key, CodedDescription in CodedDescriptions.iteritems():
236
237  # Check if the given key exists in the dictionary. Otherwise does
        not proceed by creating the relationship
238  if CodedDescription.Representation.label in TaxonsNodes:
239
240   for key, SummaryData in CodedDescription.SummaryData.iteritems()
        :
241
242    States = CategoricalCharacters[SummaryData.ref].States
243
244    for StateRef in SummaryData.States :
245
246     # Check if the given key exists in the dictionary. Otherwise
        does not proceed by creating the relationship
247     if StateRef in StateDefinitionsNodes:
248
249      taxonNode   = TaxonsNodes[ CodedDescription.Representation.
        label ][0]
250      StateDefinitionsNode = StateDefinitionsNodes[ StateRef ][0]
251
252      CategoricalCharacter    = CategoricalCharacters[ SummaryData.
        ref ].Representation
253      CategoricalCharacterDetail = CategoricalCharacter.detail if
        CategoricalCharacter.detail else ""
254      relationshipType     = CategoricalCharacter.label.replace(' '
        , '_')
255
256
257      # Join Taxon nodes to State Definition node using
        CategoricalCharacter.label as relationship
```

```
258        taxonNode.create_relationship_to( StateDefinitionsNode ,
              relationshipType , { "label"  : relationshipType ,
259                                "type"   : str( RelationshipTypes.
                                      descriptor ) ,
260                                "Detail" : CategoricalCharacterDetail }
                                      )
261
262
263  def ImportUsingTaxonStateCharacterSchema( self ) :
264   """
265   Schema : Taxon (Node) -> StateDefinition (Edge) ->
         CategoricalCharacter (Node)
266   """
267
268   # Parse the SDD-XML file
269   SDDFile = minidom.parse( self.SDDFilename )
270
271   SDD = SDDParser( SDDFile )
272
273   CategoricalCharacters = SDD.CategoricalCharacters
274   CodedDescriptions  = SDD.CodedDescriptions
275
276   # Create Taxons nodes in the Graph DB
277   TaxonsNodes = self.__CreateTaxonsNodes( SDD.CodedDescriptions )
278
279   # Join Taxons nodes in a tree structure
280   treeOutput = TreeOutput( self.TreeFilename )
281   tree = treeOutput.getTaxonsTreeStructure()
282   self.__JoinTaxonsNodesTreeStructure( TaxonsNodes, tree )
283
284   # Create Characters nodes in the Graph DB
285   CharactersNodes = self.__CreateCharacterNodes( SDD.
         getAllCharacters() )
286
287   for key, CodedDescription in CodedDescriptions.iteritems():
288
289    # Check if the given key exists in the dictionary. Otherwise does
         not proceed by creating the relationship.
290    if CodedDescription.Representation.label in TaxonsNodes:
291
292     for key, SummaryData in CodedDescription.SummaryData.iteritems()
         :
293
294      # Check if the given key exists in the dictionary. Otherwise
          does not proceed by creating the relationship.
295      if SummaryData.ref in CharactersNodes:
296
297       States = CategoricalCharacters[ SummaryData.ref ].States
298
```

```
299         for StateRef in SummaryData.States :
300 
301             taxonNode   = TaxonsNodes[ CodedDescription.Representation.
                    label ][0]
302             CharacterNode = CharactersNodes[ SummaryData.ref ][0]
303 
304             StateDefinition    = States[ StateRef ].Representation
305             StateDefinitionDetail = StateDefinition.detail if
                    StateDefinition.detail else ""
306             relationshipType   = StateDefinition.label.replace(' ', '_')
307 
308             # Join Taxon nodes to Categorical Character node using
                    StateDefinition.label as relationship
309             taxonNode.create_relationship_to( CharacterNode ,
                    relationshipType , { "label" : relationshipType ,
310                                    "type"  : str( RelationshipTypes.
                                           descriptor ) ,
311                                    "Detail" : StateDefinitionDetail } )
```

A.8 SimilarityIndex.py

```
from __future__ import division
import codecs
from py2neo  import rest, neo4j, cypher
from GraphDB import *
from NodeAndRelationshipTypes import *

class SimilarityIndex :

 def CalculateIndex(self, gdb, n1, n2 ) :

  TAaux = gdb.getIncomingAdjacentNodes( n1 )
  TBaux = gdb.getIncomingAdjacentNodes( n2 )

  TA = []
  for n in TAaux : TA.append( n[0] )

  TB = []
  for n in TBaux : TB.append( n[0] )

  setTA = set( TA )
  setTB = set( TB )

  S1 = len( setTA & setTB ) / max( len( setTA ) , len( setTB ) )

  TE1aux = gdb.getIncomingAdjacentRelationships( n1 )
  TE2aux = gdb.getIncomingAdjacentRelationships( n2 )

  TE1 = []
  for r in TE1aux: TE1.append( r[0]["label"] )

  TE2 = []
  for r in TE2aux: TE2.append( r[0]["label"] )

  setTE1 = set( TE1 )
  setTE2 = set( TE2 )

  S2 = len( setTE1 & setTE2 ) / max( len( setTE1 ) , len( setTE2 ) )

  SI = ( 0.25 * S1 + 0.75 * S2 )

  return SI


 def CompareStudies(self, TreeRootStudyA, TreeRootStudyB,
     LowerBoundary, JSONFilename ):
 """
```

```
46   It calculates the Similarity Index for all characters between two
         studies taking them two by two. Only SI greater or equal to
         LowerBoundary are exported into the given Json file.
47   Args:
48    TreeRootStudyA: Study A tree root.
49    TreeRootStudyB: Study B tree root.
50    LowerBoundary: Lower Boundary condition.
51    JSONFilename: Filename where the JSON data should be saved.
52   """
53
54   gdb = GraphDB()
55
56   rangeA = gdb.getDescriptionNodesOfATree( TreeRootStudyA )
57   rangeB = gdb.getDescriptionNodesOfATree( TreeRootStudyB )
58
59   Similarity = SimilarityIndex()
60
61   JSON = "["
62
63   for i in rangeA :
64
65    ni = gdb.getNode( i )
66
67    JSON = JSON + "\n" + '{' + "\"name\": \"{0}\" , \"imports\": [".
         format( "root." + ni["label"] )
68
69    imports = False
70
71    for j in rangeB :
72
73     nj = gdb.getNode( j )
74
75     SI = Similarity.CalculateIndex( gdb, ni, nj )
76
77     if LowerBoundary <= SI :
78      JSON = JSON + "\"{0}\", ".format( "root." + nj["label"] )
79      imports = True
80
81    if imports :
82     # Remove the last comma
83     JSON = JSON[:-2]
84
85    JSON = JSON + "]},"
86
87   for j in rangeB :
88    nj = gdb.getNode( j )
89    JSON = JSON + "\n" + '{' + "\"name\": \"{0}\" , \"imports\": []".
         format( "root." + nj["label"] ) + '},'
90
```

```
91 # Remove the last comma
92 JSON = JSON[:-1]
93
94 JSON = JSON + "\n]"
95
96 text_file = open( JSONFilename, "w")
97 text_file.write( JSON )
98 text_file.close()
```

A.9 TraceEvolutionaryHistory.py

```
import codecs

from py2neo  import rest, neo4j, cypher
from GraphDB import *
from NodeAndRelationshipTypes import *

class TraceEvolutionaryHistory:

 def BottomUpAggregation( self, gdb, node ):
  """
  This method starts from anywhere in the tree and goes down until
      reach Operational Taxonomic Unit (OTU) nodes. When it happens,
      the method basically retrieves all outgoing relationships from
      the reached OTU node and start going back toward the root. When
       the method is traversing internal nodes (Hypothetical
      Taxonomic Units) from the leaves back toward the root it
      performs an union operation with all children nodes outgoing
      relationships - i.e., relationships of the same type are
      ignored - and then for each relationship in the union the
      method creates a relationship of the same type changing the
      starting node to itself and the end node remains the same. In
      the end, the method returns all relationships outgoing from the
       given node.
  Returns: Outgoing relationships of the given node. In case the
      given node is an OTU, it returns only the character-states
      relationships from the given node to character nodes.
  In case the given node is an HTU, the method returns all outgoing
      relationships resulted from the union of its children nodes
      outgoing relationships.
  """

  if node["type"] != NodeTypes.OTU and node["type"] != NodeTypes.
      description :

   NeighborsNodes = gdb.getOutgoingAdjacentNodes( node )

   relationships = []

   for neighbor in NeighborsNodes:

    rels = self.BottomUpAggregation( gdb, neighbor[0] )

    relationships.append( rels )

   # At this point we have all children nodes relationships. In such
       a case, we can implement the first part of the algorithm
       which is duplicate all relationships (union of children nodes
```

```
        relationships) in the given node.
29
30      for rels in relationships:
31
32        if rels is not None:
33
34          for rel in rels :
35
36            if rel[0]["type"] == str( RelationshipTypes.descriptor ) :
37
38              relType = rel[0].type.encode('ascii', 'ignore')
39
40              startNode = node
41              endNode   = rel[0].end_node
42
43              # creating new relationships only where necessary
44              gdb.getPy2neoGraphDatabaseService()[0].
                    get_or_create_relationships( ( startNode, relType,
                    endNode, { "type" : str( RelationshipTypes.descriptor ) }
                     ) )
45
46
47    return gdb.getOutgoingRelationships( node )
48
49
50  def TopDownRefining( self, gdb, node ):
51    """
52    This method essentially should be called just after the
        BottomUpAggregation method passing the same starting node
        provided in BottomUpAggregation method. It starts from the
        given node (gn) back down the tree and in every HTU it
        traverses it compare the character-states starting from itself
        (gn) and pointing to a given character (chaN) with every
        character-states that starts in its children nodes (chiN) and
        points to the same character node (chaN) for all character
        nodes it (gn) points to. In case the comparation result is not
        empty - i.e. the set difference between the chatacter-states
        starting from the given node (gn) and the set of character-
        states starting from the children node (chiN) is not empty - it
         creates a edge called 'EvolvedTrait' from itself (gn) to the
        given character (chaN).
53    """
54
55    if node["type"] != NodeTypes.OTU  and node["type"] != NodeTypes.
        description :
56
57      NeighborNodes = gdb.getOutgoingAdjacentNodes( node )
58
59      tuNeighborNodes = []
```

```
60    descriptionNeighborNodes = []
61
62    for n in NeighborNodes:
63
64     if n[0]["type"] == NodeTypes.HTU or n[0]["type"] == NodeTypes.
         OTU :
65
66      tuNeighborNodes.append( n )
67
68     elif n[0]["type"] == NodeTypes.description :
69
70      descriptionNeighborNodes.append( n )
71
72    for tu in tuNeighborNodes:
73
74     for desc in descriptionNeighborNodes:
75
76      # Set Semantics
77      # http://www.itmaybeahack.com/book/python-2.6/html/p02/
          p02c06_sets.html
78
79      nodeOutgoingRelationshipTypes = set( gdb.
          getDistinctRelationshipsInBetween( node  , desc[0] ) )
80      descOutgoingRelationshipTypes = set( gdb.
          getDistinctRelationshipsInBetween( tu[0] , desc[0] ) )
81
82      # diff will have elements that exist in
          nodeOutgoingRelationshipTypes and does not exists in
          descOutgoingRelationshipTypes
83      diff = nodeOutgoingRelationshipTypes -
          descOutgoingRelationshipTypes
84
85      # Removes EvolvedTrait relationship
86      Difference = diff - set( [ str( RelationshipTypes.EvolvedTrait
          ) ] )
87
88      if 0 < len( Difference ) :
89
90       # Creates a new type of relationship (EvolvedTrait) which are
           the traits that changed from node to htu
91       startNode = node
92       endNode   = desc[0]
93       relType   = str( RelationshipTypes.EvolvedTrait )
94
95       gdb.getPy2neoGraphDatabaseService()[0].
           get_or_create_relationships( ( startNode, relType, endNode,
             { "type" : relType } ) )
96
97
```

```
98      gdb.deleteRelationshipsTypeFromNode( node, str( RelationshipTypes
            .descriptor ) )
99
100
101     for tu in tuNeighborNodes:
102      self.TopDownRefining( gdb, tu[0] )
103
104
105   def __JSONencodingRecursion( self, gdb, node, TraitNodes, nesting )
        :
     """
106
107    It is part of the JSONencoding method.
108    Args:
109     node: Given node.
110     TraitNodes: Is the list of character nodes that node's parent has
            a 'EvolvedTrait' edge pointing to.
111     nesting: Is the space (padding) on the left.
112    Returns: JSON string.
113    """
114
115    if node["type"] != NodeTypes.OTU  and node["type"] != NodeTypes.
         description :
116
117     EvolvedTraitNodes = gdb.
          getIncomingAdjacentNodesWithRelationshipInBetween( node,  str(
           RelationshipTypes.EvolvedTrait ) )
118
119     NeighborNodes = gdb.
          getIncomingAdjacentNodesWithRelationshipInBetween( node,  str(
           RelationshipTypes.TreeEdge ) )
120
121     json = ''
122     json = json + "\n" + ' '.ljust( nesting ) + "{"
123
124     json = json + "\n" + ' '.ljust( nesting + 2 ) + "\"{0}\" :
          \"{1}\",".format( "otu" , NodeTypes.OTU )
125     json = json + "\n" + ' '.ljust( nesting ) + "\"parents\" : ["
126
127     for nn in NeighborNodes:
128
129      result = self.__JSONencodingRecursion( gdb, nn[0],
           EvolvedTraitNodes , nesting + 2 )
130
131      json = json + result + ","
132
133     # Remove the last comma
134     json = json[:-1]
135
136     json = json + "\n" + ' '.ljust( nesting ) + "]"
```

```
137      json = json + "\n" + ' '.ljust( nesting ) + "}"
138
139      return json
140
141    elif node["type"] == NodeTypes.OTU:
142
143      json = ''
144      json = json + "\n" + ' '.ljust( nesting ) + "{"
145      json = json + "\n" + ' '.ljust( nesting + 2 ) + "\"{0}\" :
           \"{1}\",".format( "otu" , node["label"] )
146
147      i = 0
148      for trait in TraitNodes:
149
150        descriptions = gdb.getDistinctRelationshipsInBetween ( node  ,
             trait[0] )
151
152        for desc in descriptions:
153
154          json = json + "\n" + ' '.ljust( nesting + 2 ) + "\"{0}{1}\" :
               \"{2}\",".format( RelationshipTypes.descriptor , str(i) ,
               trait[0]["label"].encode('ascii', 'ignore') )
155          json = json + "\n" + ' '.ljust( nesting + 2 ) + "\"{0}{1}\" :
               \"{2}\",".format( NodeTypes.description , str(i), desc.
               encode('ascii', 'ignore').replace("_"," ") )
156          i = i + 1
157
158      # Remove the last comma
159      json = json[:-1]
160
161      json = json + "\n" + ' '.ljust( nesting ) + "}"
162
163      return json
164
165
166  def JSONencoding( self, JSONFilename, startNode ):
167    """
168    It exports to a JSON format the tree structure with all characters
         the algorithm flagged with 'EvolvedTrait' edge.
169    Args:
170      JSONFilename: Filename where the JSON data should be saved.
171      startNode: Node from where the data start being collected.
172
173    """
174
175    gdb = GraphDB()
176
177    json = self.__JSONencodingRecursion( gdb, startNode, [] , 0 )
178
```

```
179    text_file = open( JSONFilename, "w")
180    text_file.write( json )
181    text_file.close()
```