Qrator: A population and Curation tool for the GlycO Ontology

by

Rajesh Narra

(Under the direction of John A. Miller and William S. York)

Abstract

Qrator is a tool for curating and populating glycan structures into the Glycomics Ontology (GlycO). It takes glycan structures from various web based glycan databases and converts them to an interoperable XML format and then to a Java Object model which represents them as a tree structure. An efficient tree-structure alignment algorithm is implemented, similar to a sequence alignment algorithm, but for a branched tree structures. The tree matching results in a list of glycan structures in descending order of similarity to a canonical representation of acceptable structures, as implemented in the GlycO ontology. All the matching and partially matching structures can be viewed graphically along with suggestions for editing structures to make them match exactly with canonical representation. When a perfect match is found, Qrator adds the input structure to GlycO. When the input structure does not match perfectly, it provides a convenient way to edit the input structure and revalidate the edited structure with by comparison to the canonical representation in GlycO.

Index words: Ontology Population, Tree Matching, Glycan Matching

Qrator: A population and Curation tool for the GlycO Ontology

by

Rajesh Narra

B.Tech, Jawaharlal Nehru Technological University, India, 2006

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2008

Qrator: A population and Curation tool for the GlycO Ontology

by

Rajesh Narra

Approved:

Major Professors: John A. Miller
William S. York

Committee: Thiab Taha
Budak Arpinar

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2008

Dedication

To my Parents and Shravya.

Acknowledgments

I owe immeasurable debts of gratitude to so many people who have supported me along the way. I would especially like to thank Dr. William S. York and Dr. John A. Miller. They helped throughout this work, and gave me an opportunity to have many valuable experiences. I am very grateful for having an exceptional committee for my thesis and wish to thank the other committee members, Dr. Budak Arpinar and Dr. Thiab Taha for sparing their precious time and offering valuable suggestions. Special thanks to Shravya S. Nimmagadda, who could not have been more supportive of me throughout this entire work and a reliable friend. I would also like to express my gratitude to Dr. Kochut for his patient support and encouragement.

Finally, I would like to thank my parents who gave me strength when I am mostly in need of it.

Table of Contents

List of Figures

Chapter 1

Introduction

Glycans are complex carbohydrate structures that are synthesized by living cells and play key roles in the development and maintenance of these cells. They are complex tree structures whose nodes are simpler monosaccharide residues and whose edges are chemical bonds between these residues. Research in glycoproteomics studies the interaction of these glycans, genes and proteins and the biological process in which they participate. Despite their complexity, modern experimental techniques such as mass spectrometry and Nuclear Magnetic Resonance (NMR) has provided much knowledge about glycan structures. Due to enormous growth of data stored in web accessible databases, the task of querying, sharing and correlating structurally related data has become exceptionally complex and therefore unwieldy. As a result, biologists require intuitive mechanisms for creating and managing this information. This would be facilitated by semantic annotation of the accumulated knowledge. To capture a formalized description of this complex domain, an ontology for glycans, *GlycO*[1] is utilized. The strength of GlycO is its canonical representation of the building blocks of glycans (monosaccharide residues) and their relationships to each other and to the biochemical processes by which they are formed. The implementation of this canonical representation within GlycO is called *GlycoTree*. These canonical residue representations provide rich semantic context for describing the relationships between between chemical structures and a host of biological interactions.

In order to be useful, the GlycO ontology must be populated with valid structures. A large collection of structures are available via trusted sources such as CarbBank[2], KEGG[3] and SweetDB. However, due to differences in representation standard and human misconceptions,

many errors exist in these structural databases. The canonical representation of structures in GlycO provides a mechanism to specify biologically reasonable structural motifs and thereby identify errors in these structural databases. Furthermore, the structural integrity of GlycO is maintained by populating it exclusively with structures that are consistent with the canonical representation.

Hence each new glycan structure must be validated before using it to populate the ontology. This paper describes a user friendly curation and population tool called *Qrator* that implements an approximate tree matching algorithm. This algorithm identifies subtrees of the canonical *GlycoTree* that best match the input structure. If the given structure does not have a perfect match in the GlycoTree, the tool displays the closest match and suggests corrections to the structure or extensions to GlycoTree that result in an exact match so that it is appropriate add the new structure as an instance in the ontology.

This document is organized as follows. Chapter 2 briefly introduces the required preliminaries. Chapter 3 summarizes related work. Chapter 4 presents our Tree-Matching algorithm. Chapter 5 describes the architectural workflow of the entire process. Chapter 6 presents the data acquisition and format conversion steps. Chapter 7 describes the entire process of curation and population. Chapter 8 presents the Qrator tool and finally Chapter 9 presents conclusions and outlines future work.

Chapter 2

Preliminaries

This section will define preliminary information used in the rest of paper, including relevant concepts in semantics (i.e., ontologies), glycobiology and graph theory.

2.1 Graph Theory

A *tree* is defined as an acyclic connected graph, whose vertices are referred to as *nodes*. A *rooted tree* is a tree having a specific node called the root, from which rest of the tree extends. Consider a node x in a rooted tree T with root r. Nodes that are directly connected to node x (distal to the root node) are called children of x and conversely, x is parent of these children. An *ordered tree* is a rooted tree in which the children of each node are ordered. A *labeled tree* is a tree in which a label is attached to each node. An *unordered tree* is rooted tree which is not an ordered tree. Note all trees we consider in this paper are *labeled unordered trees*.

2.2 Subtree Isomorphism

Given two trees T_1 and T_2, find all possible subtrees of T_2 that are isomorphic to T_1 (i.e., find subtrees of T_2 that are identical to T_1) or decide that there is no such subtree. Our algorithm is variant of subtree isomorphism discussed in Chapter 4.

2.3 Web Ontology Language (OWL)

The Web Ontology Language (OWL) is one of the knowledge representation languages for authoring ontologies and is recommended by World Wide Web Consortium. OWL provides

the ability to specify classes and properties using a formal description logic. There are three variations in OWL with different levels of expressiveness.

- **OWL Lite** provides a classification hierarchy and limited constraints.
- **OWL DL** provides maximum expressiveness while ensuring decidability.
- **OWL Full** provides maximum syntactic freedom, limited only by RDF. However, OWL Full makes no guarantees of decidability.

An ontology expressed in OWL Lite is automatically expressed in OWL DL, similarly ontology expressed in OWL DL is automatically expressed in OWL FULL.

2.4 Glycomics Ontology (GlycO)

The Glycomics Ontology (GlycO) focuses on the glycoproteomics domain to model the structure and functions of glycans and glycoconjugates, the enzymes involved in their biosynthesis and modification, and the metabolic pathways in which they participate. GlycO is intended to provide both a schema and a sufficiently large knowledge base, which will allow classification of concepts commonly encountered in the field of glycobiology in order to facilitate automated reasoning and information analysis in this domain. The GlycO schema exploits the expressiveness of OWL-DL to place restrictions on relationships, thus making it suitable to be used as a means to classify new instance data. These logical restrictions are necessary due to the chemical nature of glycans, which have complex, branched structures that cannot be represented in any simple way. Glycans are thus distinguished from DNA (e.g., genes) and proteins, which can be represented (at least in their most basic forms) as simple character strings. The structural knowledge in GlycO is modularized, in that larger structures are semantically composed from smaller canonical building blocks. In particular, glycan instances are modeled by linking together several instances of canonical monosaccharide residues, which embody knowledge of their chemical structure (e.g., β-D-GlcpNAc) and context (e.g., attached directly to the Asn residue of a protein). This bottom-up semantic

modeling of large molecular structures using smaller building blocks allows structures in GlycO to be placed in a biochemical context by describing the specific interactions of its component parts with proteins, enzymes and other biochemical entities.

2.5 GlycoTree

A GlycoTree (Figure 2.1) is a labeled ordered tree whose nodes are canonical residues, as defined in the GlycO ontology. The current version[4] of GlycO is limited to a single GlycoTree, which focuses on a subclass of glycans called N-glycans. The N-glycan GlycoTree subsumes a large fraction of the N-glycans whose structures have been chemically characterized. That is, many of the known N-glycan structures can be completely specified by choosing a subset of the nodes of GlycoTree, to form a connected subtree rooted at one of the residue nodes of the N-glycan GlycoTree. Observation of valid N-glycan structures (as judged by a qualified curator) that are not subsumed by the current version of the N-glycan GlycoTree provide justification to extend the N-glycan GlycoTree by addition of new canonical residues.

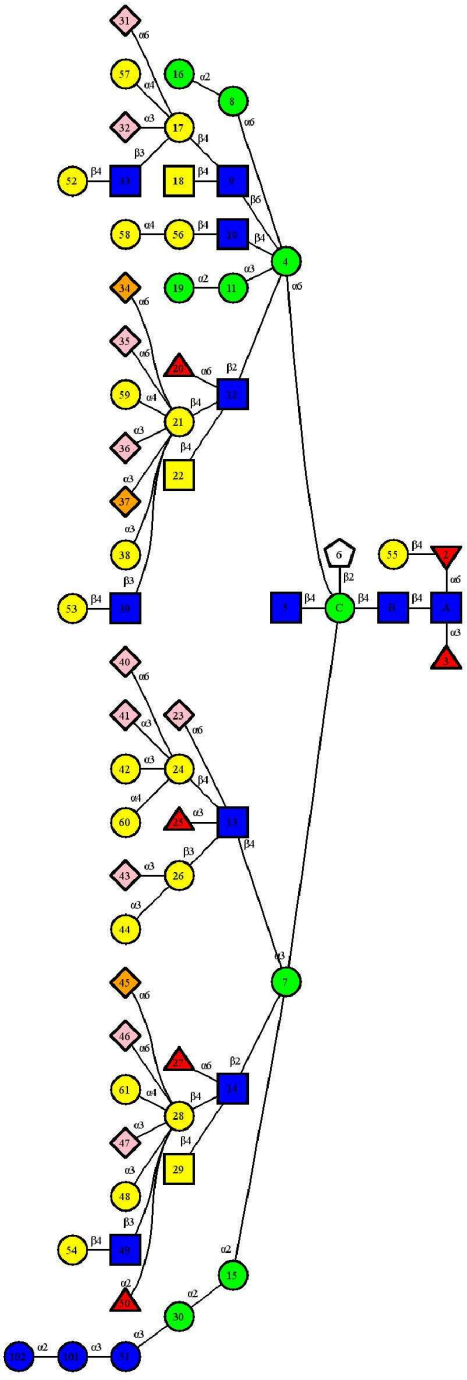


Figure 2.1: GlycoTree

Chapter 3

Related Work

Recently the problem of measuring the similarity of two trees has been a focus of researchers in various scientific fields such as computational biology [5, 6, 7, 8], information extraction from web pages [9, 10]. Tai [11] was the first to generalize the edit model from strings to tree structures, where he used the concept of a "Tree edit distance" to compare the difference between two structures. The tree edit distance between two trees is defined as the minimum cost of a series of elementary edit operations needed to transform the first tree into the second. Since then, tree mapping has been attracting the interest of researchers. The tree mapping between two trees is a set-theoretic description of the transformation from one tree to the other. Intuitively, a tree mapping describes the transformation between two trees by means of a set of node pairs, whereas a tree edit algorithm describes how to transform one tree into the other. Tree mapping allows us to understand and investigate tree edit distance in a qualitative and abstract way. In addition to the tree edit distance proposed by Tai (Tai distance), various tree edit distance measures have been proposed in the past three decades. For example, the algorithms for computing the structure-preserving [12], constrained [13], structure-respecting [14], less-constrained [15], and bottom-up [16] distance measures were proposed according to the definitions using the notion of tree mapping, i.e. these measures have the corresponding tree mapping definitions.

In sequence alignment methods, the "similarity score" is often used. A higher similarity score value indicates better alignment. Based on this similarity score approach various approximate tree matching algorithms have been proposed, including KCaM algorithms [17] and labeled subtree homeomorphism [18].

Here, we consider special cases in which there are at most K differences between two input trees. For these special cases Shasha and Zhang [19] developed an $O(K^2nH)$ time algorithm for the unit cost edit distance problem for ordered trees where H is the minimum height of two input trees and n is the maximum size of input Trees. Daiji and Tatsuya [20] developed an algorithm that is able to find the largest common subtrees allowing at most K differences between two input trees. Their algorithm, [20] implemented as a Tree-edit distance method, is claimed to work in linear time for constant K and two rooted and unordered trees. Aoki, Yamaguchi and Okuno [17] developed different approximate Tree matching algorithms for querying the carbohydrate database. Their approximate matching algorithm is able to find a maximum common subtree between two input trees allowing gaps in their match. For this purpose they included a Global Matching subroutine and local matching subroutine in their algorithm. The global matching algorithm recursively calls the local matching procedure for all subtrees of the two input trees that have not been matched. After the single largest matching tree has been found, the resulting unmatched portions of both trees are then used as the input trees for another round of the local exact matching algorithm.

In populating the GlycO ontology, we require an algorithm that identifies and assigns one or more subtrees (of a GlycoTree) that best match each input glycan. If no perfect match exists, the algorithm should calculate the similarity score of the input structure with all possible subtrees of the GlycoTree and return the top K (where K is specified by the user) similarity scores (in descending order) along with representations of the corresponding subtrees. We developed an algorithm that fulfills this explicit requirement, although it does not execute in linear time. Our algorithm takes two trees (GlycoTree, InputStructure) and returns all perfect matches, if they exist. Otherwise it returns the top K subtrees in descending order of similarity scores. It also provides graphical representations of the input structure and the (partially) matching subtrees of the GlycoTree, allowing the user to identify specific nodes in the subtree that do not precisely match the corresponding nodes of the input structure. This facilitates curation by providing a rational basis upon which the user can decide

whether to edit the input structure, discard the input structure, or extend the GlycoTree to accommodate the input structure.

Chapter 4

Tree-Matching

4.1 Description of Algorithm

In this section, we describe our Tree-Matching algorithm. It is a variant of a maximum common subtree isomorphism algorithm. The main difference is that maximum common subtree isomorphism algorithm identify all possible maximum subtrees between two trees T_1 and T_2, whereas our algorithm finds all possible subtrees of T_2 that are similar to T_1 and displays the list of subtrees in descending order of their similarity scores.

4.1.1 Conventions used

T_1 and T_2 are two ordered labeled rooted trees whose roots are r_1 and r_2 respectively. For a node u, *child(u)* are the child nodes of u. For a node v in tree T, *T(v)* denotes the subtree of T induced from v and its descendents. Finally, *w(u,v)* is the similarity score assigned when a single node u of tree T_1 and a single node v of tree T_2 are compared.

The method *"root(T)"* returns the root node of the Tree T.

The subprocedure *ENQUEUE (Q, s)* adds an element *"s"* to the end of the queue *"Q"*.

Similarly, the sub procedure DEQUEUE(Q) removes the first element currently existing in the queue *"Q"*.

The method *numberOfChildren(v)* returns the number of child nodes of *"v"*. The procedure $MAX\_WEIGHT\_BIPARTITE\_GRAPH(X, Y, edge\_weights[])$ returns the list of edges that have highest sum of weights and maximum cardinality in the matching. Here X and Y are set of nodes in the bipartite graph and the $edge\_weights[]$ is an array containing the weights of the edges that connects the nodes in X and Y.

4.1.2 TREE_MATCHING Algorithm

This algorithm accepts two trees T_1 and T_2 as inputs. The algorithm now traverses the tree T_2 in a Breadth First Search (BFS) manner and sends each visited node in the tree T_2 along with the root node in the tree T_1 to another procedure called the $SIMILARITY\_SCORE$ which returns a similarity score between the tree T_1 and the subtree of the tree T_2 rooted at the visited node. The structure $similarityScore[u]$ is used to store similarity score between given tree T_1 and the subtree of tree T_2 rooted at node u (where "u" is the current visited node). Finally, the $TREE\_MATCHING$ Algorithm returns this $similarityScore$ array.

Algorithm 1 TREE-MATCHING(T_1,T_2)

$r \leftarrow root(T_1)$
$s \leftarrow root(T_2)$
$Q \leftarrow \phi$
$similarityScore[] \leftarrow \phi$ //size of the similarityScore is size of Tree T_2
$ENQUEUE(Q, s)$
while $Q \neq \phi$ **do**
    $u \leftarrow DEQUEUE(Q)$
    for *each* $v \in children(u)$ **do**
        $ENQUEUE(Q, v)$
    end for
    $similarityScore[u] = SIMILARITY\_SCORE(r, u)$//return root node of mapping tree
end while
return similartyScore;

4.1.3 SIMILARITY_SCORE Algorithm

This algorithm accepts two nodes "r" and "u" where "r" is the node from the tree T_1 and "u" is the node from tree T_2. Initially, the algorithm determines the similarity score between these two nodes by using a procedure *getMatchScore(r,u)*. The score returned by this procedure corresponds to the scoring scheme to be followed. In our case, the maximum score assigned for a perfect match between two given residues is 5, as explained in section 7.1. If the current node from T_1 which is "r" is a leaf node, then the algorithm returns the score obtained from the *getMatchScore* procedure and exits. Otherwise, for each of the child nodes in r and u, the following procedure is executed.

Now let $r_1, r_2, ...r_p$ denote the children of node r in tree T_1 and $u_1, u_2, ...u_q$ denote the children of node u of the tree T_2, respectively. The Similarity score *Score*$[r_i, y_j]$ for *i=1..p* and *j=1..q* is calculated recursively. Now a bipartite Graph G is constructed by X and Y where X is the set of children of r and Y is the set of children u and each node in X is connected to each node in Y. An *edge* (r_i, u_j) in G is annotated with weight *Score*$[r_i, y_j]$. The similarity score between subtree of tree T_1 rooted at node r and subtree of tree T_2 rooted at node u is computed using the node-to-node similarity, which is *w(r, u)* plus the sum of the weights of the matched edges returned by the procedure $MAX\_WEIGHT\_BIPARTITE\_GRAPH$.

Note: The *edge.getSource()* method in Line 15 in the $SIMILARITY\_SCORE$ algorithm returns r_i where r_i is a child node of r. And also the *edge.getTarget()* method returns $\psi(r_i)$ where $\psi(r_i)$ is some node u_j that has a match with r_i.

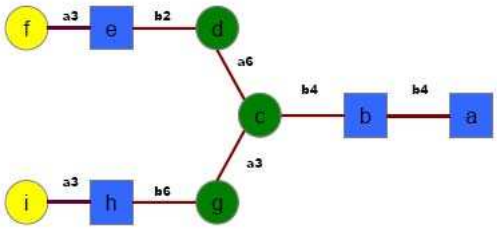
The above algorithm only describes the procedure by which similarity scores between tree T_1 and all possible subtrees of tree T_2 are obtained. We used a mapping tree structure in order to keep track of the nodes corresponding to the scores calculated. For simplicity, this is not included in the above procedures.

Algorithm 2 SIMILARITY_SCORE(r,u)

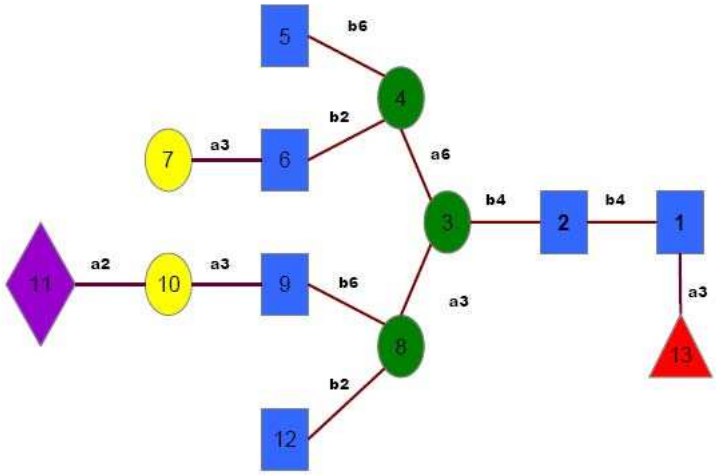
1: $w(r,u) \leftarrow getMatchScore(r,u)$
2: $totalScore = w(r,u)$
3: **if** $child(r) = \phi$ **then**
4: **return** $totalScore$
5: **else**
6: $Score[1...child(r), 1...child(u)] \leftarrow \phi$
7: **if** $numberofChildren(r) \leq numberofChildren(u)$ **then**
8: **for** $r_i \epsilon\ child(r)$ **do**
9: **for** $u_j \epsilon\ child(u)$ **do**
10: $Score[r_i, u_j] = SIMILARITY\_SCORE(r_i, u_j)$
11: **end for**
12: **end for**
13: $childlist\ L = MAX\_WEIGHT\_BIPARTITE\_GRAPH(child(r), child(u), Score)$
14: **for** $edge\ e \epsilon\ L$ **do**
15: $totalScore = totalScore + Score[e.getSource(), e.getTarget()]$
16: **end for**
17: **end if**
18: **return** $totalScore$
19: **end if**

4.2 Example

The algorithms discussed in the section 4.1 are illustrated by using the following example. Consider the trees T_1 and T_2 shown in figure 4.1(a) and 4.1(b). The goal is to find all possible subtrees of Tree T_2 that are similar to Tree T_1. Our Algorithm 1 traverses the Tree T_2 in Breadth First Search (BFS) manner and for every visited node it calls Algorithm 2 with the root node of Tree T_1 and currently visited node of Tree T_2 as arguments. It stores the returned similarity score between Tree and subtree of Tree T_2 rooted at current visited node of Tree T_2. In this section we briefly discuss the algorithm 2 by considering root nodes of two trees T_1 and T_2. The Maximum similarity score between the subtree of Tree T_1 rooted at node v and the subtree of Tree T_2 rooted at node w has score equal to the similarty score between two nodes v and w plus the score of a maximum weight bipartite matching



(a) tree T_1



(b) tree T_2

Figure 4.1: Tree T_1 and Tree T_2

in G, where G is bipartite Graph($\{v_1, v_2, v_3, ...v_p\}$, $\{w_1, w_2, w_3, ...w_p\}$, E) for $v_1, v_2, ...v_p$ and $w_1, w_2, ...w_q$ be the children of nodes v and w respectively.

Let a score of *2* be the maximum similarity score assigned for two individual nodes v of Tree T_1 and w of Tree T_2 in this example, i.e, a score of *1* is assigned if the nodes u and v are equal (in shape) and a score of *1* is assigned if the labels of the edges that connect each node them to their parent are equal (for example w(b,2) = 2 since both b and 2 have same shape and the edge label that connect to its parent is *b4* in both cases).

Let us first start with nodes *a* and *1* of Trees T_1 and T_2 respectively. In order to decide the similarity score between the subtree of tree T_1 rooted at node *a* and the subtree of T_2

rooted at node *1*, the following maximum weight bipartite matching problem has to be solved:

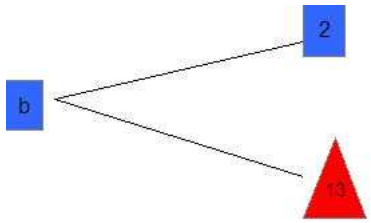


Figure 4.2: bipartite graph between children of nodes *"a"* and *"1"*

However, solving this bipartite matching problem (Figure 4.2) involves (recursively) solving further maximum weight bipartite matching problems. First in order to find the similarity score between subtree of Tree T_1 rooted at node "b" and subtree of Tree T_2 rooted at node "2", the following maximum weight bipartite matching problem also needs to be solved

Figure 4.3: bipartite graph between children of nodes *"b"* and *"2"*

This in turn, requires solving the following maximum weight bipartite matching problem, in oder to find the maximum similarity score between the subtree of Tree T_1 rooted at node "c" and subtree of Tree T_2 rooted at node "3".

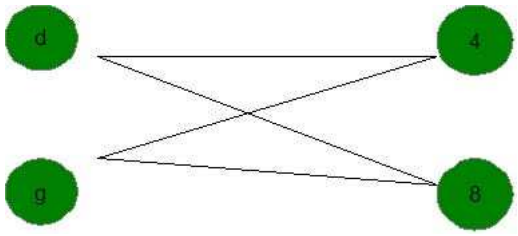


Figure 4.4: bipartite graph between children of nodes *"c"* and *"3"*

In order to solve the above maximum weight bipartite matching problem, the weights of the edges in the graph are to be determined, which involves solving further maximum weight bipartite matching problems. First, in order to find the similarity score between the subtree of TreeT_1 rooted at node "d" and the subtree of Tree T_2 rooted at node "4" the following maximum weight bipartite matching problem has to be addressed:

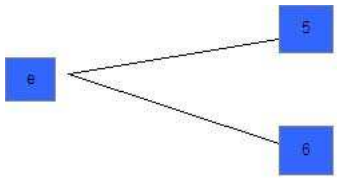


Figure 4.5: bipartite graph between children of nodes *"d"* and *"4"*

In order to solve above maximum weight bipartite matching problem (Figure 4.5) the following maximum weight bipartite matching has to be solved.

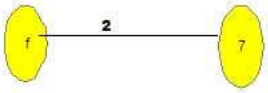


Figure 4.6: Bipartite graph between children of nodes *"e"* and *"6"*

The similarity score between the subtree of Tree T_1 rooted at node "f" and subtree of Tree T_2 rooted at node "7" is the similarity score between these two nodes alone, since these are leaf nodes. Since both these nodes have same shape and edge label, they have

similarity score of "2". Hence, the maximum bipartite matching problem shown in Figure 4.6.

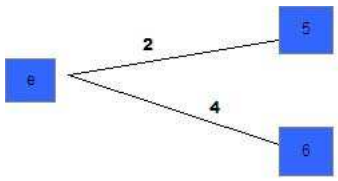


Figure 4.7: Bipartite graph between children of nodes *"d"* and *"4"* with scores

In the maximum weight bipartite graph shown in figure 4.7 the edge weight between the nodes "e" and "6" is calculated as "4". This is because the similiarity score returned from bipartite graph shown in figure 4.6 is "2" plus the similarity score between the two nodes "e" and "6" which is also "2". Since "5" is leaf node of Tree T_2, the similarity score between the subtree of Tree T_1 rooted at node "e" and the subtree of T_2 rooted at "5" is 2.

Similarly, in order to obtain similarity score between the subtree of Tree T_1 rooted at node "d" and "8", following maximum weight bipartite matching problem has to be solved:

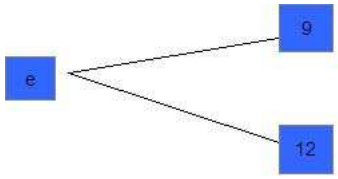


Figure 4.8: Bipartite graph between children of nodes *"d"* and *"8"*

which, in turn, requires solving the following maximum weight bipartite matching problem.

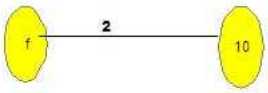


Figure 4.9: Bipartite graph between children of nodes *"e"* and *"9"*

The similarity score between the subtree of Tree T_1 rooted at node "f" and subtree of Tree T_2 rooted at node "10" is the similiraity score only between these two nodes, since these are the leaf nodes. Since both these nodes have same shape and edge label, they have similarity score of "2". Hence, the maximum bipartite matching problem shown in Figure 4.9 is solved.

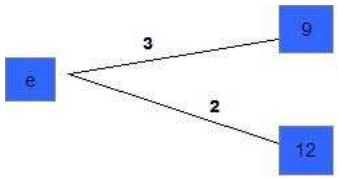


Figure 4.10: Bipartite graph between children of nodes *"d"* and *"8"*

In the maximum weight bipartite graph shown in figure 4.10 the edge weight between the nodes "e" and "9" is calculated as "3", since a similiarity score returned from bipartite graph shown in figure 4.9 is "2" plus similarity score between these two nodes "e" and "9" is "1" (since they have different edge labels). Since node "12" is leaf node of Tree T_2, the similarity score between the subtree of Tree T_1 rooted at node "e" and that of T_2 rooted at "12" is 2.

Similarly, in order to obtain similarity score score between the subtree of Tree T_1 rooted at node "g" and "8", the following maximum weight bipartite matching problem has to be resolved:

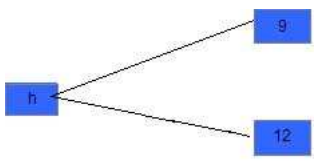


Figure 4.11: Bipartite graph between children of nodes *"g"* and *"8"*

which, in turn, requires solving the following maximum weight bipartite matching problem.

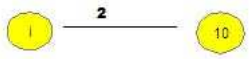


Figure 4.12: Bipartite graph between children of nodes *"h"* and *"9"*

The similarity score between the subtree of Tree T_1 rooted at node "i" and subtree of Tree T_2 rooted at node "10" is just the similarity score between these two nodes, since these are leaf nodes. Since both these nodes have same shape and edge label, they have similarity score of "2". The maximum bipartite matching problem shown in Figure 4.12 is solved.

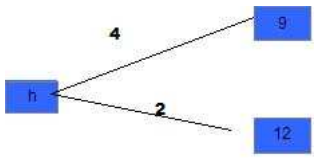


Figure 4.13: Bipartite graph between children of nodes *"g"* and *"8"*

In the maximum weight bipartite graph shown in figure 4.13 the edge weight between the nodes "h" and "9" is calculated as "4". This is the similiarity score returned from bipartite graph shown in figure 4.12 which is 2, plus similarity score between these two nodes "h" and "9" which is also 2. Since node "12" is leaf node of Tree T_2, the similarity score between the subtree of Tree T_1 rooted at node "h" and T_2 rooted at "12" is 2.

Finally, in order to obtain the similarity score score between the subtree of Tree T_1 rooted at node "g" and "4", the following maximum weight bipartite matching problem has to be addressed:

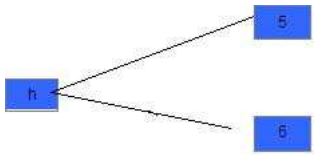


Figure 4.14: Bipartite graph between children of nodes *"g"* and *"4"*

which, in turn, requires solving the following maximum weight bipartite matching problem.

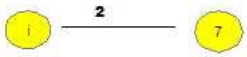


Figure 4.15: Bipartite graph between children of nodes *"h"* and *"6"*

The similarity score between the subtree of Tree T_1 rooted at node "i" and subtree of Tree T_2 rooted at node "7" is the similiraity score between these two nodes, since these are leaf nodes. Since both these nodes have the same shape and edge label, they have similarity score of "2". The maximum bipartite matching problem shown in Figure 4.15 is solved.

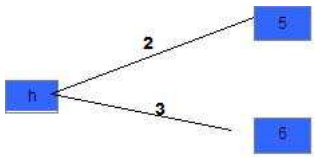


Figure 4.16: Bipartite graph between children of nodes *"g"* and *"4"*

In the maximum weight bipartite graph shown in figure 4.16 the edge weight between the nodes "h" and "6" is calculated as "3", since a similiarity score returned from bipartite graph shown in figure 4.15 is 2 plus similarity score between these two nodes "h" and "6" is 1 (since they have different edge labels). But due to the node "5" being a leaf node of Tree

T_2, the similarity score between the subtree of Tree T_1 rooted at node "h" and the subtree T_2 rooted at "5" is 2.

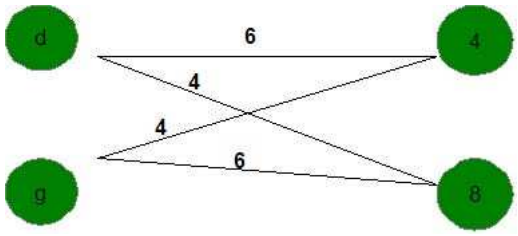


Figure 4.17: Bipartite graph between children of nodes *"c"* and *"3"*

By solving all maximum weight bipartite graph shown in figure 4.8, figure 4.10 , figure 4.13, figure 4.16, we get the weights of edges for the above bipartite graph (figure 4.17). By solving this bipartite graph we get maximum weight "12", which is the sum of the weights of edges that connect nodes "d" and "4", "g" and "8". That is, we have higher similarity score between subtree of Tree T_1 starting at node "d" and subtree of Tree T_2 starting at node "4" than that obtained in the subtree of Tree T_2 starting at node "8". Similarly we have higher similarity score between subtree of Tree T_1 starting at node "d" and subtree of Tree T_2 starting at node "8" than that obtained in the subtree of Tree T_2 strting at node "4".

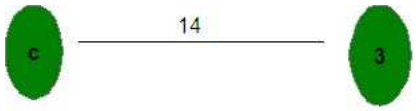


Figure 4.18: Bipartite graph between children of nodes *"b"* and *"2"*

By solving the maximum bipartite matching graph problem of the child nodes of "c" and "3" we obtained a similarity score of 12. This similarity score of child nodes is now added to that of the similarity score between "c" and "3", which is 2. The total similarity score thus calculated between the subtree of Tree T_1 rooted at node "c" and the subtree of Tree

T_2 rooted at node “3” is “14” (figure 4.18).

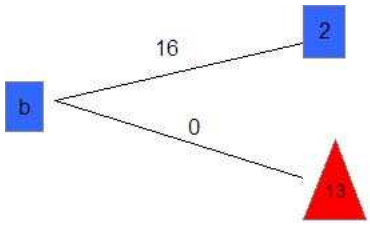


Figure 4.19: Bipartite graph between children of nodes *“a”* and *“1”*

Finally, considering between the child nodes of node “a ” and “1”, a maximum score of 16 is obtained between the node “b” and “2”, which is the highest score calculated among the child nodes of “a” and “1”. Therefore, the total similarity score between the subtree of Tree T_1 rooted at node “a” and the subtree of Tree T_2 rooted at node “1” is 18 as shown in figure 4.20.

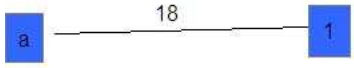


Figure 4.20: Bipartite graph between nodes *“a”* and *“1”*

In our case, we are not only interested in finding the maximum similarity scores among the subtrees, but we also need to store the path through the tree by which this maximum similarity is obtained. For this pupose a Tree structure is created internally in which each node consists of information above the nodes matched in each tree along with their similarity scores. Figure 4.21 presents the tree consisting of the matching information for the example illustrated in this section.

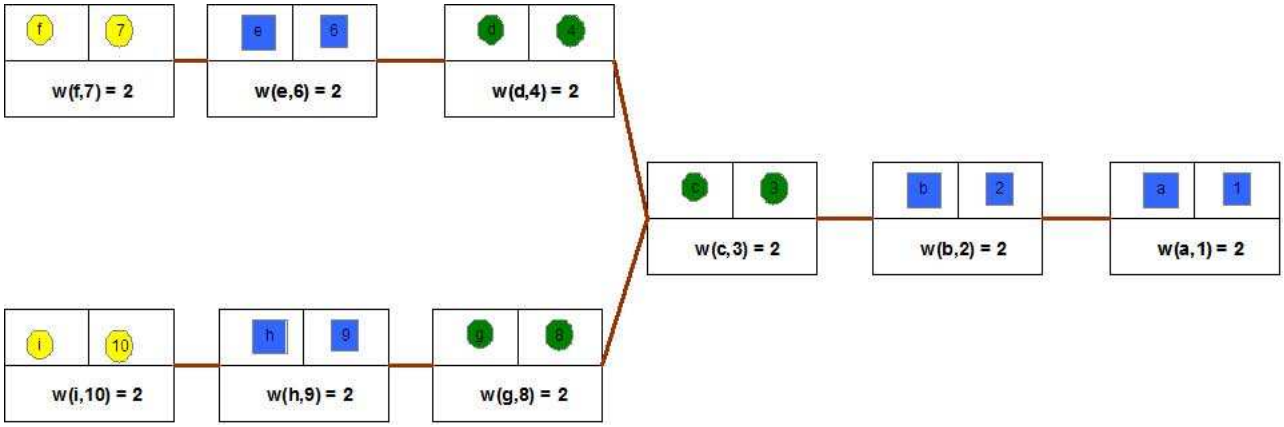


Figure 4.21: Tree structure that is used to store the matches and corrections in given Tree

Chapter 5

Architecture

Figure 5.1 illustrates the work flow of the Qrator. Initially input structures are taken from various web based sources such as CarbBank, KEGG and SweetDB. These structures are in various formats which are then converted to a standard Glyde-II format in the format converter process. The Glyde-II XML document is parsed to create a Glycomics Object Model (GOM) in Java in which the molecule and all its residues are represented as Java objects that are linked together. Similarly GlycoTree is also represented in GOM in order to improve the performance and also ease the programmatic access to various objects. The approximate tree matching algorithm now considers the GOM representation of both GlycoTree and the input structure and finds the closest match. If the closest match found is a perfect match then this structure is instantiated in GlycO. Otherwise, the user can request the top K suggested structures to be displayed by the Qrator. Now the user can select one of the suggested structures and instantiate it in GlycO. On the other hand, the user can edit the input structure manually and then repeat the process of finding the closest match. An additional feature offered to trusted users is the option to edit the GlycoTree. If the user's input structure is certified to be correct then GlycoTree should be updated, so the user could opt to edit the GlycoTree itself.

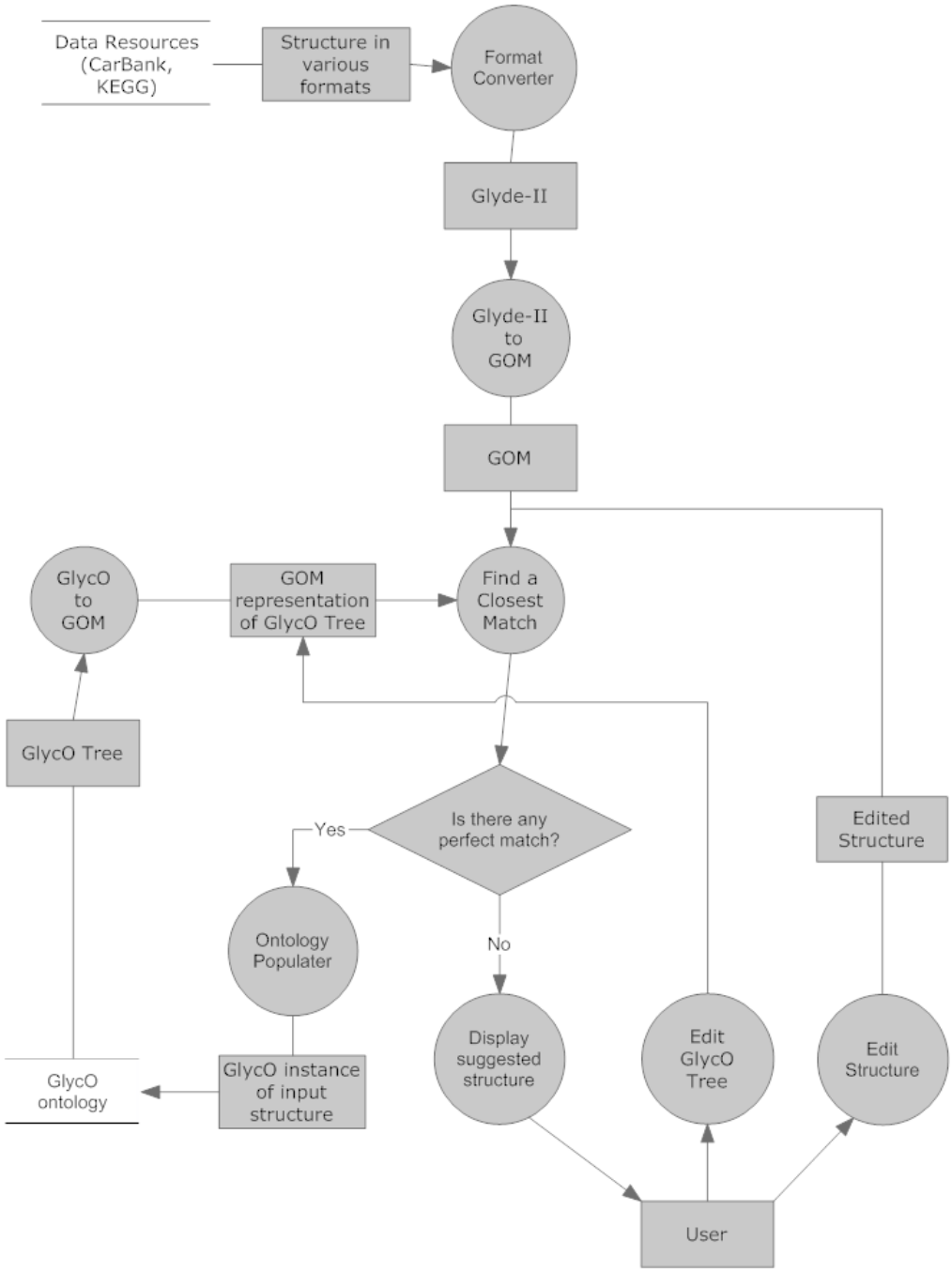


Figure 5.1: Work flow for GlycO Population

Chapter 6

Data Acquistion and Format Conversion

Creation of an Ontology does not end with schema design. It is more useful when actual domain knowledge in the form of instances are added. In this section, pre-processing steps that are performed before population are described.

6.1 Data Sources

Several data sources such as CarbBank, KEGG and SweetDB offer a good place to obtain glycan data. Though CarbBank is discontinued, it still contains many useful structures which are used as references in other structures. Hence CarbBank is chosen as a potential resource for the initial population of the Glycomics Ontology (GlycO).

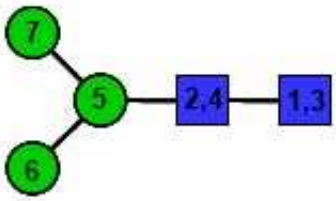
6.2 Format Conversion

In order to disambiguate the potential instances, the textual description of the structure was converted into the internal GOM representation. Each database has its own representation for glycans. Structures from CarbBank are represented in IUPAC format, which is non-unique two-dimensional textual representation. Using the web service provided by our colleagues at the German Cancer Research Center who are maintaining GlycomeDB (www.glycomedb.org/About.action), these structure are converted from IUPAC to structurally unambiguous LINUCS format. Unfortunately LINUCS is not able to disambiguate different naming conventions. For this purpose, we are converting strucutres into the XML-based Glycan Data Exchange(GLYDE-II) format, which semantically disambiguates the different naming conventions of monosaccharide residues, using web service

(www.glycomedb.org/About.action). Moreover, the structurally rich XML format offers more flexible programmatic input to the algorithm. This Glyde-II format is explained with an example in section 6.3.

6.3 Glyde-II

GLYDE-II is a standard for representation of the chemical structures of complex glycans that is based on a connection table formalism using XML syntax. The GLYDE-II standards contains two parts, syntax and implementation which are conceptually distinct to one another. The syntax document is completely specified by a framework called PARCHMENT (PARtonomy of CHeMical ENTities) which allows complete structure of biological molecules. The GLYDE-II implementation also includes a set of rules, naming conventions for the parts that are absolutely required for representational consistency and disambiguation. In this format the largest structure are defined by their parts. For example, a complex entity such as a molecule consists of parts that are connected to each other. In this case a part can be a moiety, a residue or a bound atom. Two parts of an entity are connected by a link. Here in our example, a molecule is an independent entity which is not be connected to another molecule. Conversely, the parts of this entity are linked together. The other two independent entities are, free atom which is not bound to any other atom and an aggregate which is composed of other independent entities. In a GLYDE-II XML document the root node is always “GLYDE-II”. The following figure shows an example structure followed by the GLYDE-II specification for that structure.




```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE GlydeII SYSTEM "http://glycomics.ccrc.uga.edu/GLYDE-II/GLYDE-II_v0.7.DTD"[
  <!ENTITY mDBget "http://www.monosaccharideDB.org/GLYDE-II.jsp?G">
]>
<GlydeII>

   <molecule subtype="glycan" id="M3N2" name="pentaglycoside">
      <residue subtype="substituent" partid="1" ref="&mDBget;=n-acetyl"/>
      <residue subtype="substituent" partid="2" ref="&mDBget;=n-acetyl"/>
      <residue subtype="base_type" partid="3" ref="&mDBget;=b-dglc-hex-1:5"/>
      <residue subtype="base_type" partid="4" ref="&mDBget;=b-dglc-hex-1:5"/>
      <residue subtype="base_type" partid="5" ref="&mDBget;=b-dman-hex-1:5"/>
      <residue subtype="base_type" partid="6" ref="&mDBget;=a-dman-hex-1:5"/>
      <residue subtype="base_type" partid="7" ref="&mDBget;=a-dman-hex-1:5"/>

      <residue_link from="1" to="3">
         <atom_link from="N1" to="C2" from_replaces="O2" bond_order="1"/>
      </residue_link>
      <residue_link from="2" to="4">
         <atom_link from="N1" to="C2" from_replaces="O2" bond_order="1"/>
      </residue_link>
      <residue_link from="4" to="3">
         <atom_link from="C1" to="O4" to_replaces="O1" bond_order="1"/>
      </residue_link>
      <residue_link from="5" to="4">
         <atom_link from="C1" to="O4" to_replaces="O1" bond_order="1"/>
      </residue_link>
      <residue_link from="6" to="5">
         <atom_link from="C1" to="O3" to_replaces="O1" bond_order="1"/>
      </residue_link>
      <residue_link from="7" to="6">
         <atom_link from="C1" to="O6" to_replaces="O1" bond_order="1"/>
      </residue_link>
   </molecule>
</GlydeII>
```

Figure 6.1: GLYDE-II representation for pentaglycoside molecule

6.4 Glycomics Object Model(GOM)

The Glycomics Object Model (GOM) provides a convenient way to translate GLYDE-II XML documents into other forms such as GlycO OWL instances. The object model was designed using UML and the classes were coded in Java. The correspondence with GLYDE-II is nearly one-to-one, as shown in the UML Class Diagram for GOM in Figure 6.2.

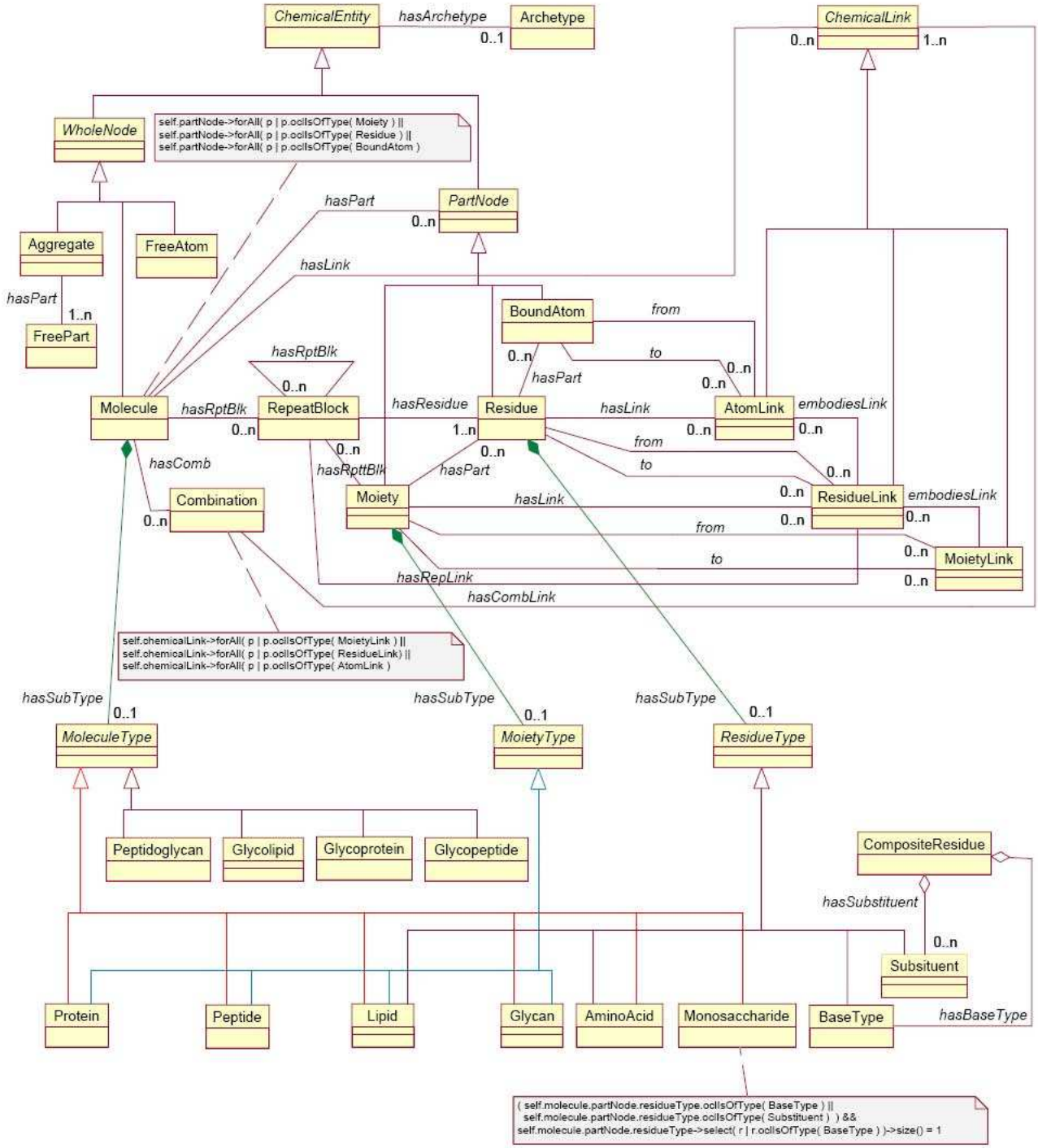


Figure 6.2: UML diagram for GOM

The above displayed GOM is a composite model in which the molecule and all of its residues are represented as Java objects that are linked together. A GLYDE-II structure document described in section 6.3, is parsed to create a DOM tree using an XML parser. This DOM tree is traversed to create a Glycomics Object Model (GOM) in Java. Before we validate our input structure, GlycoTree from GlycO is extracted and is represented in the GOM. Using Java Serialization, the above mentioned conversion process is done only once after each change in GlycoTree, which is rare. By this we improved the speed of our procedure compared to that specified in [21], the GlycO is queried constantly.

Chapter 7

Curation and Population

The glycan structures that are extracted from the data sources contain many errors. Each input structure's linkage pattern must be compared with the pre-existing GlycoTree. For this purpose an additional processing step is added to the workflow called Curation. This is done just before the Population of input glycan structure to GlycO. This next section describes these steps of curation and population.

7.1 Curation

As we described in section 2.4 glycans in GlycO are defined as a tree structure of canonical residues, which function as a building blocks for more complex carbohydrates. Each instance that is added to GlycO should comply with this tree structure. Hence we need a curation step that identifies and assigns the subtree that corresponds to a particular glycan that is to be instantiated in the ontology. This is achieved by sub-tree isomorphism. For curation there is a need for an algorithm that not only finds perfect matches, but also allow some gaps in alignment so that a user can identify sites where the input structure fails and thereby correct the input structure. Nodes of the trees represent monosaccharides and edges represent the linkage between them. For determining similarity between two monosaccharides a simple matching of their names is not sufficient. There are several other factors that need to be taken into account for matching, for example, the residue's anomeric configuration, absolute configuration, and linkage number to its parent. Hence for this purpose, the similarity score $w(u,v)$ between two monosaccharides, u and v is calculated using the following procedure:

A score of 1 is assigned when the residue meets the following criteria:

1. Both have same basetypes
2. The Anomeric configuration of both residues is equal
3. The Absolute configuration of both the residues is equal
4. Both the residues have the same substituents
5. The edge link number to the parent is the same for both the residues

The following figure illustrates in detail the scoring assignments based upon the above scoring schemes.

(a) β-D-GlcpNAc (b) α-D-Manp

In the above figure, the square in part(a) indicates a β-D-GlcpNAc residue and the circle in part (b) indicates α-D-Manp. Both these residues are similar in their absolute configuration which is indicated by a "D". However, they differ in their anomeric configuration which is "β" for residue in (a) and "α" in residue in (b). Moreover the residue in (a) consists of a basetype and substituent pair where as that in (b) has only a basetype. Finally, they also differ in linkages to their parents.

Taking into account the similarities and dissimilarities among the above residues the similarity score assigned for these residues is "1".

As per our scoring scheme we can say that two monosaccharide residues are identical if their similarity score is 5. Using the Tree-Matching algorithm along with this scoring scheme,

we are not only able to identify the subtree that corresponds to a particular glycan, but we are also able to find the nodes at which input glycan structures are failing to match with GlycoTree. Moreover, this enables us to give suggested structures for making corrections to the input structure, so that it can be identified with a subtree of GlycoTree.

7.2 Population

Once a perfectly matched subtree of GlycoTree is found for the given glycan structure linkage pattern, this input structure is used to populate GlycO. For populating and accessing the ontology we are using the open source Jena API [22] which is developed by *HP labs* as part of their *Semantic Web Programme*. Along with the structure we are also adding the information of its identifiers in other databases like KEGGID, CCSD, etc.

Chapter 8

Qrator

Qrator is a tool developed for the curation and population procedure described in Chapter 5. Though gathering structures and converting formats is done with out any human intervention, curation requires a domain expert for reviewing the structures that do not have any perfect matches in the GlycoTree. Most of the experts in this domain are biologists who may not have computer science background. Qrator makes this task easy by showing the pictorial representation of input glycan structures and the subtree of GlycoTree it matches. If the given input glycan structure has no perfect matches, it displays the top K matches with suggested corrections for making it a perfect match in GlycoTree. Qrator provides three options for each structure using the buttons “create”, “change”, “discard”. Using “Create Button”, a user can create instance of this glycan structure for GlycO in cases of perfect match. For imperfect matches, the user has the option to choose one of the suggested structures and create an instance of it in the ontology using the ”Create” button. The “discard” button discards the given glycan structure. Figure 8.2 is a screen shot of the Qrator. The topmpost panel is the pictorial representation of the given glycan structure (the CarbBankID of the structure shown in figure is CCSD:4419). The image in the bottom left is a replica of the input structure in which nodes that failed to match with GlycoTree are displayed in grey color. The image on the right in the bottom panel shows the suggested structure for the given input structure to make it a perfect Match in a GlycoTree. In this case, two nodes failed due to mismatches of edge link numbers (both have 3 where suggested structure shows 4). CFG Nomenclature (http://glycomics.scripps.edu/CFGnomenclature.pdf) is followed for pictorial

representation of glycans. Figure 8.1 represents common monosaccharide names and their symbols.

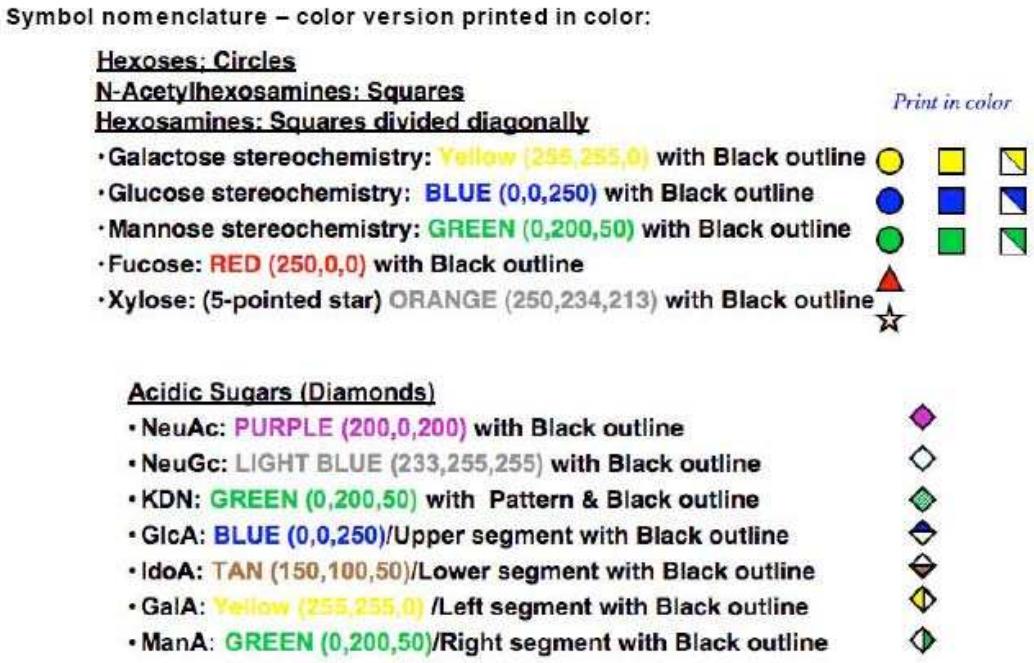


Figure 8.1: Common monosaccharide names and their symbols

Another type of mismatch is shown in Figure 8.3 in which the error nodes are represented in black, since these nodes are not present in the GlycoTree and therefore no comparison is made. When error nodes are in black, Qrator suggests that these nodes should be removed for a perfect match. Error nodes in grey color suggest that they have to be changed to match perfectly but not necessarily removed.

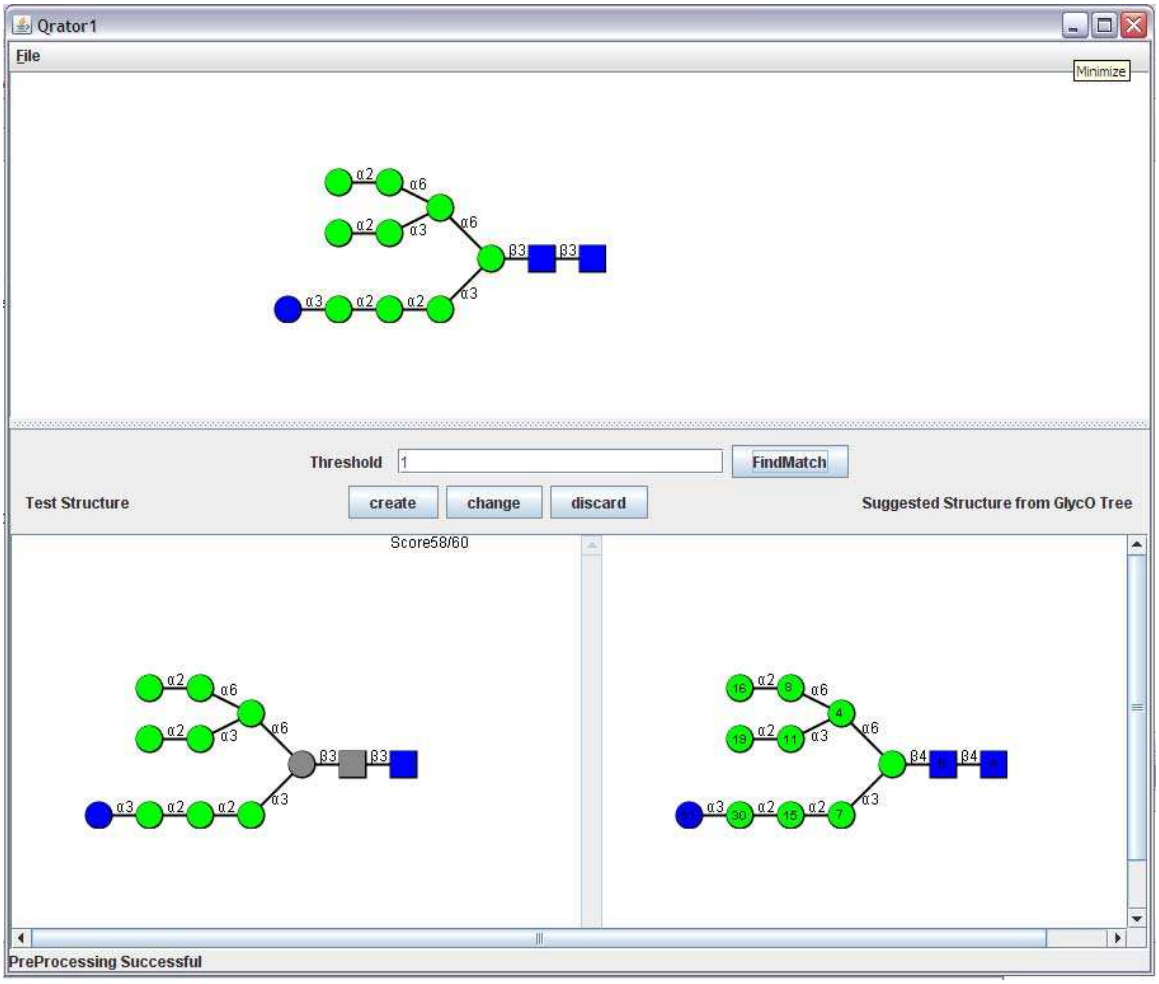


Figure 8.2: Screen shot of Qrator. Image in upper panel is the input structure, left side image in bottom panel shows the nodes where error occurs in gray color and right side image in bottom panel is the suggested structure

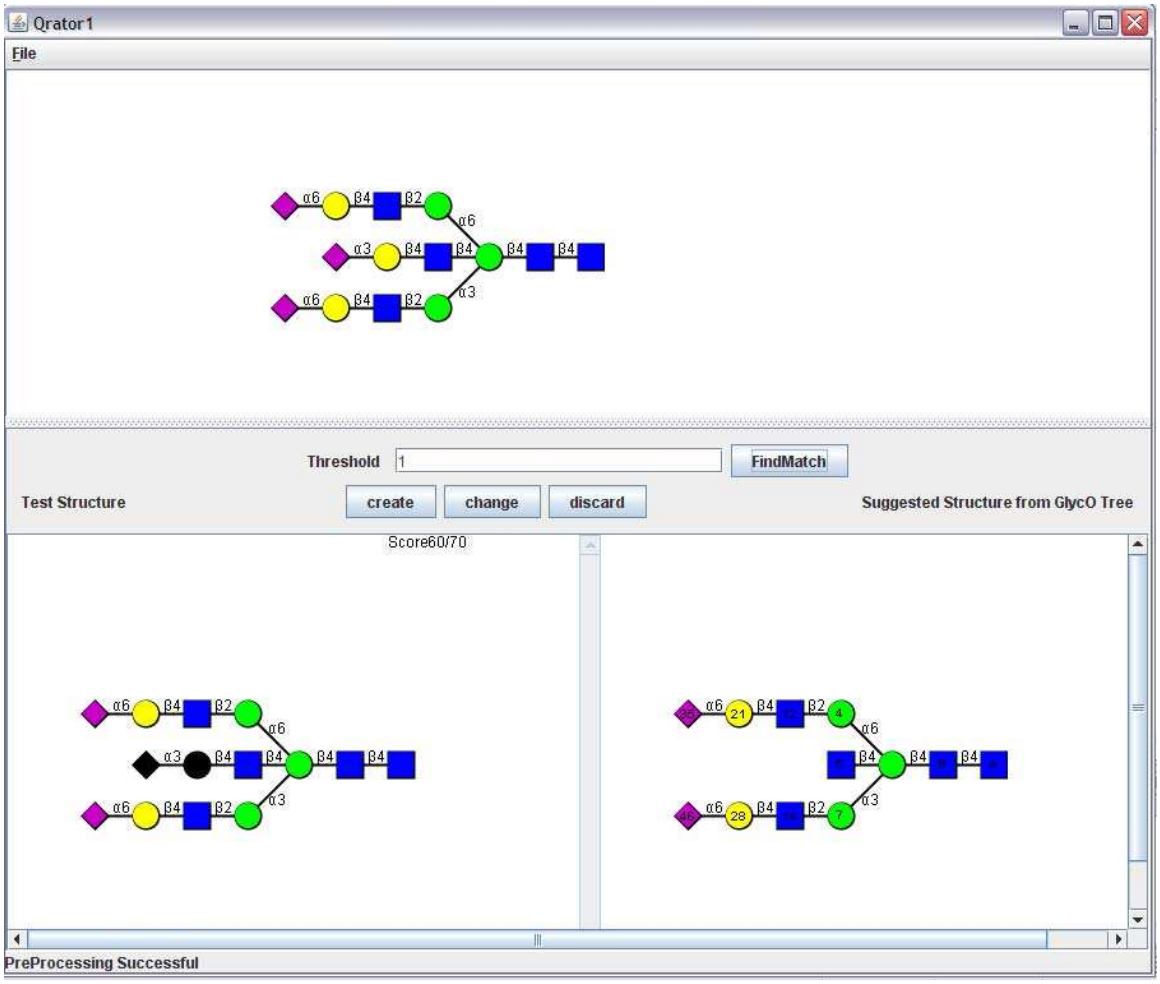


Figure 8.3: Screen shot of Qrator. Image in upper panel is the input structure, left side image in bottom panel shows the nodes where error occurs in black color and right side image in bottom panel is the suggested structure

Chapter 9

Conclusions and Future Work

This work describs a Tree-Matching algorithm that facilitates the curation of glycan structures from various trusted glycan databases. This process of curation helps users to make appropriate changes to the input structures before populating the ontology. This process helps to maintain the canonical richness of the existing ontology.

By providing a good user interface, we were able to encapsulate all the working details such as tree matching and structure alignment. Hence this tools appeals to a variety of end users such as biologists. The input from the users to select the number of suggested structures one would like to be displayed, which offers more control upon the number of structure suggestions. More trusted users also have the permissions to change the GlycoTree itself.

At this point most of the software for Qrator is complete. Work is being done to make this tool as a web service so as to make it available to more number of users. Efforts are being made to develop a more general and user friendly interface that would help users to curate the structures and populate the ontology. In the next few months, we plan to make Qrator available on the web as a generally accessible curation tool for glycan structures.

Bibliography

[1] Thomas,C.,Sheth,A.,York,W.S.: Modular ontology design using canonical building blocks in the biochemistry domain. In: Proc of the 4th Int. Conference on Formal Ontologies in Information Systems (FOIS), Baltimore, 2006.

[2] CarbBank, http://ncbi.nlm.nih.gov subdirectory /repository/carbbank.

[3] KEGG: Kyoto Encyclopedia of Genes and Genomes, http://www.genome.jp/kegg/.

[4] Glycomics Ontology, http://lsdis.cs.uga.edu/projects/glycomics/2006/GlycO_v0_95.owl.

[5] T. Akutsu, Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots, Discrete Applied Mathematics 104, 2000, pp. 45-62.

[6] Y. Sakakibara, Pair hidden markov models on tree structures, Bioinformatics 19 2003, pp. 232-240.

[7] M. Hochsmann, T. Toller, R. Giegerich, and S. Kurtz, Local similarity in RNA secondary structures, Proc. of the Computational Systems Bioinformatics (CSB), IEEE, 2003, pp. 159-168.

[8] J. Allali and M.-F. Sagot, Novel tree edit operations for RNA secondary structure comparison, WABI 2004, LNBI 3240, 2004, pp. 412-425.

[9] M. Collins and N. Duffy, Convolution kernels for natural language, Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001], MIT Press, 2001, pp. 625-632.

[10] A. Hogue and D. Karger, Thresher: Automating the unwrapping of semantic content from the world wide web, Proc. of 14th International World Wide Web Conference (WWW), 2005, pp. 86-95.

[11] Tai, K., The tree-to-tree correction problem, Journal of the ACM,26, 1979, pp. 422-433.

[12] E. Tanaka, A note on a tree-to-tree editing problem, International Journal of Pattern Recognition and Artificial Intelligence 9, 1995, no. 1, pp. 167-172.

[13] K. Zhang, A constrained edit distance between unordered labeled trees, Algorithmica 15, 1996, pp. 205-222.

[14] T. Richter, A new measure of the distance between ordered trees and its applications, Tech. Report 85166-CS, Dept. of Computer Science, Univ. of Bonn, 1997.

[15] C. L. Lu, Z.-Y. Su, and C. Y. Tang, A new measure of edit distance between labeled trees, COCOON, Lecture Notes in Computer Science, vol. 2108, 2001, pp. 338-348.

[16] G. Valiente, An efficient bottom-up distance between trees, Proc. of 8th International Symposium on String Processing and Information Retrieval (SPIRE), IEEE Computer Science Press, 2001, pp. 212-219.

[17] K. F. Aoki, A. Yamaguchi, Y. Okuno, T. Akutsu, N. Ueda, M. Kanehisa, and H. Mamitsuka, Efficient tree-matching methods for accurate carbohydrate database queries, Genome Informatics 14 , 2003, pp. 134-143.

[18] R.Y. Pinter, O. Rokhlenko, D. Tsur, M. Ziv-Ukelson, Approximate labelled subtree homeomorphism, in: Proc. 15th Annu. Symp. Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 3109, Springer, Berlin, 2004, pp. 55-69.

[19] Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems, SIAM J. Computing 18, 1989, pp. 1245-1262.

[20] Daiji, F., Tatsuya, A.: Fast Algorithms for Comparision of Similar Unordered Trees, in: ISAC 2004, Lecture Notes in Computer Science, Vol. 3341, Springer, Berlin, 2004, pp. 452-463.

[21] Sahoo, S.S., Thomas,C.,Sheth,A.,York,W.S., Tartir, S.: Knowledge Modeling and its Application in Life Sciences: A Tale of two Ontologies. In: World Wide Web Conference[WWW 2006],Edinburgh,Scotland, 2006.

[22] Jena Semantic Web Framework, http://jena.sourceforge.net/.