**A Project Report On**

# Calculating Semantic Similarity of GO Terms

(Term Project)

**Submitted in Partial Fulfillment of**

## Artificial Intelligence (CSCE 520)

## Fall 2015


**Submitted to:**
**Dr. Rasiah Logantharaj**
**Professor Computer Science**
**Center for Advanced Computer Sciences**
**University of Louisiana at Lafayette**

**Submitted by:**
**Prakash Subedi**
**Yamini Joshi**
**Matinsadat Hosseini**

# TABLE OF CONTENTS

# I. INTRODUCTION

Gene ontology (GO) describes the attributes of genes and gene products (either RNA or protein, resulting from expression of a gene) using a structured and controlled vocabulary. GO consists of three ontologies: biological process (BP), cellular component (CC) and molecular function (MF), each of which is modeled as a directed acyclic graph[1].

In this project, we have implemented a method to measure the semantic similarity of GO terms for human genes using Python. For calculating the semantic similarity between two GO terms, we have used the following formula from[1]:

$$\frac{2 * LnP(c1\,, c2)}{\ln\big(P(c1)\big) + \ln(P(c2))}$$

# II. WORKFLOW

The specifications of Input data were in Table 1:

Table 1: Specifications of Input

| | |
|---|---|
| OBO XML File | go_daily-termdb.obo-xml |
| Size | 55.8 MB |
| Gene annotation File | gene_association.goa_human |
| Size | 84.0 MB |

This project is divided into the following phases:

1. GO Term Identification: For this phase, we parsed the gene annotation file and gathered information like frequency about each GO term.
2. Graph Generation: In this phase, we parsed the obo-xml file and created a parent to child mapping of GO terms. We also mapped GO_IDs to objects in this phase.
3. Update Node Object: In this phase, we traversed the graph to calculate the cumulative frequency and ancestors of each GO term.
4. Serialization: This phase involved writing the graph and node data structures to memory. We used PICKLE library to generate a “cache” for the data structures.
5. Calculate Semantic Similarity: In this phase, we calculated the semantic similarity between all pairs of GO terms in given two genes.

$$\frac{2 * LnP(c1\,, c2)}{\ln\big(P(c1)\big) + \ln(P(c2))}$$

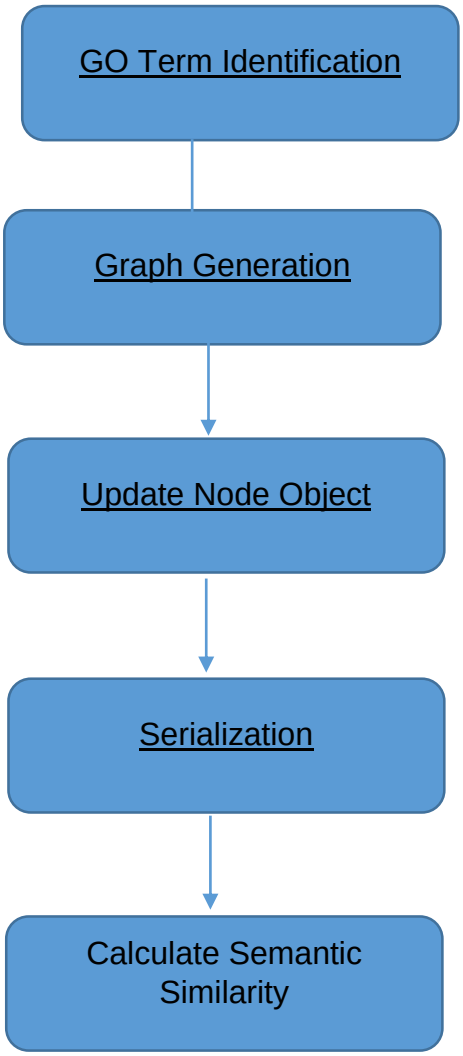


Fig. 1: Project Workflow

# III. Data Structures Used

The following are the main data structures used in the project:

## Class Node:

Node class contains go_Id, list of parents, frequency of node, cumulative frequency and list of ancestors. 3 different function are defined in this class, the definition of the class is in fig 3

- get Id: return Go_Id
- Namespace
- Add parents

```
1.  class Node:                                  #GO term to object mapping
2.          go_id = GO_ID
3.          Parents                              #List of immediate parents
4.          f                                    #Freq of the node
5.          cf                                   #c=Cumulative freq of the node
6.          cfUpdated                             False
7.          ancestors                            #Mapping of ancestors to distance


8.  getID(): return self.go_id
9.  namespace(ns): namespace = ns
10. addParent(node): parents.append(node)
```

*Fig. 3: Node class*

## Class cache:

The class cache objects are defined in Fig. 4:

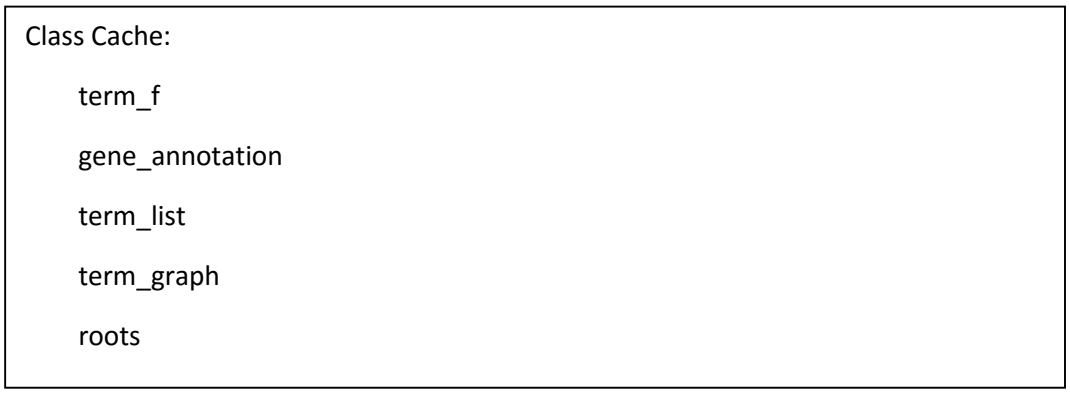


*Fig. 4: Cache Class*

# IV. IMPLEMENTATION

We used Python 2.7 and ran the project on a Vaio Laptop with Intel core i7 and 4G memory. The following is a list of libraries used in this project:

- copy: copy.deepcopy(*x*) Return a deep copy of *x*.
- re: regular expression
- defaultdict: for default valued dictionaries
- untangle: to parse XML file
- time: To calculate total time taken by a procedure
- pickle: to serialize data and write to a file
- math: for log calculation
- itertools: iterator building blocks
- sys: to access system components

## Main function:

In the main function, we have computed the total time taken by our program (line 1 and 3) for gene similarity object instantiation. Based on list of Go_Id mapped to both Gene_Ids, we generated all possible pairsof GO Terms. The pseudo code of main is in the Fig. 2.

Input: Gene Ids as command line argument

Output: Maximum similarity of a pair from given input genes

```
1.  start_time = current time
2.  Gene_Similiarity = GeneSimiliarity()                    # Gene Similarity Object instantiation
3.  total_time = current time - start_time
4.  gene1, gene2 = Command line args[1], Command line args[2]
5.  go_list1 = Gene_Similiarity.gene_annotation[gene1]      #List of all GO terms mapped to gene1
6.  go_list2 = Gene_Similiarity.gene_annotation[gene2]      #List of all GO terms mapped to gene2
7.  go_pairs = all possible GO Terms Pairs from gen1 and gene2
8.  for each pair in go_pairs:
9.          maxsim =Max(semantic similarity of pair)
10. print maxsim as semantic similarity
```

*Fig. 2: main function*

## Gene Similarity:

## Cache:

When cache is not in the directory, gene association file is parsed. The root is identified by <isroot> tag and after that the frequency and ancestors of each Go_term are

calculated. All the required data structures are then saved in cache to reduce further calculation.

```
1.  if cache exists
2.          cache = Open cache using pickle                    #cache=pickled graph file
3.          term_f = load term_f from cache
4.          gene_annotation = cache.gene_annotation
5.          term_list = cache.term_list
6.          term_graph = cache.term_graph
7.          roots = cache.roots
8.  else:
9.          Generate term_freq (GO Term to freq mapping) and GO Term to Object mapping
10.         term_f, gene_annotation = parseGeneAssociation()
11.         Create the graph data structure
12.         term_list = Gene to GO Terms mapping
13.         graph = parent to child GO Term mapping
14.         roots = values in <isroot> tag of XML file
15.         Calculate and store cumulative frequencies and ancestor of each GO Term
16.         Write all the mappings to cache using pickle
```

*Fig.5: Cache*

## Parse Gene Association:

```
1. Load gene associaiton file
2. term_f = a default dictionory
3. gene_annotation = a default dictionary
4. find all GO ID and gene id in the file using regex
5.     Increase freq of GO Term by 1
6.     Add Go Term to gene id mapping
7.     gene_annotation[gene_id].append(go_id)
```

*Fig. 6: Parse gene association*

## Parse Annotations:

After parsing XML file with Untangle Library and specifying the root (in different namespaces such as Cellular component, Biological process, molecular function) we create a graph as parent to children map and create node object for each term. Then the frequency and immediate parents of each node are updated. The output of this

```
1.  Initialize term_list                                    #Str GO_ID-> Node object
2.  Initialize term_graph with defaultdict(list)            #Graph str-> str list
3.  Initialize roots = {}
4.  Parse the XML file using Untangle
5.  for <term> in XML file:
6.          child = term.value
7.          if <is_root> is defined:
8.              roots[namespace] = child
9.          if <child> is defined:
10.             f = term_f[child]
11.             '''specify each node in each name space'''
12.             Initialie node object
13.         node = Node(child, f)
14.         node.namespace = <namespace>
15.         for parent in <is_a>:
16.             add mapping entry from parent to child in graph
17.             Update parent of their node
18.             Map GO ID to Node object
19. return term_list, term_graph, roots                     #go_id -> node, graph, roots
```

*Fig. 7: Parse annotation*

function is the roots, parents of each node and child of each node, so our graph is completed.

## compute_cf_and_ancestors:

Cumulative frequency of the leaf node is equal to its frequency and cumulative frequency of other nodes are the summation of its children cf and its frequency (but this number divided by the number of its parents). So we use parents of each node to compute its ancestors.

```
1.  Compute cf and ancestors(current node):
2.  for each leaf:
3.          cumulative freq = freq
4.          the accumulative frequency of the last child is equal to its frequency
5.          store the list of children
6.          childList = term_graph[current node]
7.  for child in childList:
8.           Load the object correspond to the child
9.           temp = copy of the ancestors of child
10. for key,val in temp.iteritems():
11.              Increment the val by 1
12.              if key is already an acestor
13.              Compare val and store the least val
14. childObj.ancestors[currnt node node] = 1
15. compute_cf_and_ancestors(child)
16. term_list[a].cf += float(childObj.cf) / len(childObj.parents)
17. return True
```

*Fig. 8: compute_cf_and_ancestors*

**Semantic Similarity:**

Both genes should be from the same namespace. We have list of ancestors of each node. Based on the intersection of ancestors list, common ancestors will computed. The closest is the ancestors with minimum distance (may be more than one ancestors). For each node we compute the probability based on the frequency. Compute the IC(c) = -log P(c) for both node. Compute similarity based on $\frac{Max(Ic(c))}{(0.5\ (IC(a) + IC(b)))}$

```
1.  semantic_similiarity(a, b) :
2.  if the name space of a and b is different print error of different namespace
3.  if a and b are the same similarity =1
4.  initialize closest ancestor
5.  aAncestores = term_list[a].ancestors
6.  bAncestores = term_list[b].ancestors
7.  for id in intersection of aAncestores and bAncestores:
8.          p = float(term_list[id].cf) / root_f
9.          ic = -1*math.log(p)
10.         if ic is equal to maxIC:
11.                 Add this Id to closest ancestor
12.         Else if ic change the max:
13.                 Remove the previous closest ancestors and add the new one
14. a_ic = -1*math.log(self.term_list[a].cf/root_f)
15. b_ic = -1*math.log(self.term_list[b].cf/root_f)
16. for ancestor in closestAncestors:
17.         sim_t = (2*maxIC)/(a_ic+b_ic)
18.        if sim_t > sim:
19.                 sim = sim_t
20. return sim
```

*Fig. 9:* ***Semantic Similarity***

# V. CONCLUSION

The existing code takes about 120 seconds to generate the main data structures and write them to a file. Based on this project, the following conclusions can be made:

- For a pair of genes, to calculate GO term similarity, we need to make a cross product of all GO terms in the genes and then calculate the semantic similarity for each pair. We could successfully calculate semantic similarity using this approach.

- To calculate semantic similarity, the GO terms must belong to the same namespace.

# VI. FUTURE WORK

The existing model takes the genes as command line arguments and works only for human genes. The model can be refined to add a better user interface. This interface can provide the option of selecting a species for whom we want to calculate the semantic similarity.

# VII. REFERENCES

[1] Measure the Semantic Similarity of GO Terms Using Aggregate Information Content. Xuebo Song, Lin Li, Pradip K. Srimani, Philip S. Yu, and James Z. Wang