Methods of Gene Ontology Term Similarity Analysis in Graph Database Environment

Łukasz Stypka[1,2] and Michał Kozielski[1]

[1] Institute of Informatics, Silesian University of Technology
Akademicka 16, 44-100 Gliwice, Poland
{lukasz.stypka,michal.kozielski}@polsl.pl
http://adaa.polsl.pl
[2] Future Processing, Gliwice, Poland
lstypka@future-processing.com

Abstract. The article presents and analyses three graph processing issues that can be identified in three methods of GO term similarity evaluation. The solutions of these problems are implemented in Neo4j graph database environment. Each of the issues can be solved directly by a single Cypher query or can be divided into several queries which results have to be merged. The comparison of the introduced solutions is presented in terms of time and memory effectivness. The results show how to implement the effective solutions of this class of issues.

Keywords: graph database, Neo4j, Gene Ontology, GO term similarity.

1 Introduction

Gene Ontology (GO) [2] is a widely used knowledge base that is continuously developed and corrected. GO enables annotation of gene products to ontology terms representing biological process, molecular function or biological component. Gene Ontology terms are connected by means of relations of different types, such as e.g., *is a*, *part of* or *regulates*.

Gene Ontology is an important source of knowledge utilized in several research projects and analysis [2]. It is modeled as a directed acyclic graph where ontology terms are graph nodes and the edges are defined by the relations between terms. One of the important issues that can be approached in Gene Ontology analysis is evaluation of GO terms similarity. Several methods refering to this problem were introduced [8] and many of these methods require different Gene Ontology graph processing.

Recently, several new types of database management systems have been introduced. One of the interesting and well received by the market is a concept of graph database. A very popular representative of this type of systems is Neo4j [7]. It offers, among others, graph data model, graph oriented query language *Cypher* and Java based API. Recently, a version 2.0 of this system has been released, what has solved several previously existing issues (e.g., efficient memory

© Springer International Publishing Switzerland 2014

use). Neo4j has been already reported as an interesting environment that can support Gene Ontology graph analysis [5].

The goal of this article is to present and analyse three graph processing issues that can be identified in three methods of GO term similarity evaluation. The solutions of these problems are implemented in Neo4j graph database environment. Each of the issues can be solved directly by a single Cypher query or can be divided to several queries which results have to be merged. The comparison of the introduced solutions is presented in the article and also their time and memory effectivness are evaluated. The results show how to implement the effective solutions of this class of issues.

The structure of this work is as follows. Section 2 presents the similarity measures which require different graph processing approaches. The solutions that were introduced and implemented, and the results of the effectiveness analysis are presented in section 3. Conclusions of the work are presented in section 4.

2 Similarity Measures

Three GO term similarity measures are analysed in this study. They have different characteristics and they require different approaches to GO graph processing.

The first two approaches are classified as semantic similarity measures and utilize the concept of *Information Content* $\tau(a)$ of an ontology term a given by the following formula:

$$\tau(a) = -log(P(a)), \tag{1}$$

where $P(a)$ is a ratio of a number of annotations to a term a, to a number of analysed genes.

The basic semantic similarity measure was proposed by Resnik [9] and it takes under consideration only the *Information Content* of the common ancestor $\tau_{ca}(a_i, a_j)$ of the compared terms a_i and a_j:

$$s_A^{(R)}(a_i, a_j) = \tau_{ca}(a_i, a_j). \tag{2}$$

Fig. 1 A presents the common ancestors (terms 1, 2, 5 and 6) of terms 4 and 8.

Other popular approaches requireing identification of the common ancestor term were introduced by Jiang and Conrath [4] and by Lin [6]. Each of these methods was introduced with a purpose other then Gene Ontology term analysis.

The approach based on semantic similarity and taking into account the specificity of GO was presented in [3]. GraSM introduced by Couto et al. [3] extends the above similarity measures by taking into consideration different paths leading to a common ancestor what can also result in different terms regarded as common ancestor. GraSM takes into consideration all such common disjunctive ancestors and instead of the information content of a single common ancestor used in semantic similarity measures it calculates the average of the information content of a disjunctive common ancestors [3]. In that way all the "classical" semantic similarity measures [9,4,6] can be extended to GraSM versions. Fig. 1 B presents the disjunctive common ancestors (terms 1 and 6) of terms 4 and 8.

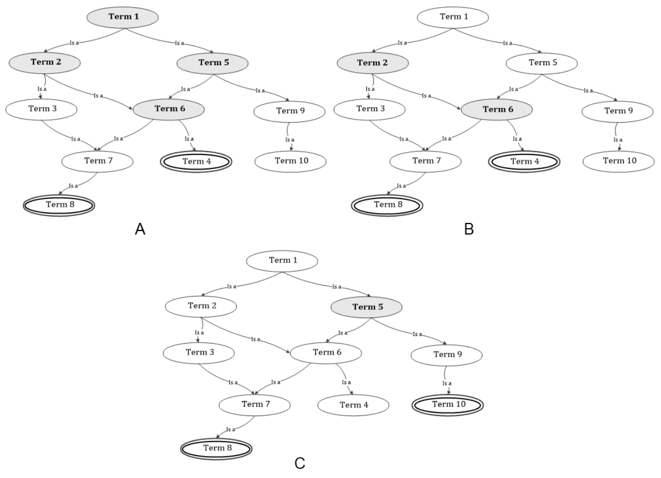


Fig. 1. Illustration of common ancestors (A), disjunctive common ancestors (B), lowest common ancestor (C) of a given pair of terms

The last approach that is considered here defines the distance between two terms a_i and a_j on the basis of a length $l(a_i, a_j)$ of the shortest path between them. Calculating shortest paths in Gene Ontology it has to be taken into consideration that the ontology graph is a directed one. Therefore, the length of a path between two ontology terms that are not connected by a parent-child relation can be set as infinity or it can be calculated as a sum of path lengths leading to the lowest common ancestor. The latter approach was chosen in the work presented. When the shortest paths are calculated then the similarity of the two GO terms can be defined as exponential dependency on a path length $l(a_i, a_j)$ [1]:

$$s_A^{(P)}(a_i, a_j) = e^{-\lambda l(a_i, a_j)}, \tag{3}$$

where λ is a parameter setting the strength of the path length influence on the similarity value. Fig. 1 C presents the lowest common ancestor (term 5) of terms 8 and 10.

3 Experiments and Results

The main goal of the experiments was to compare average query execution time and memory usage level for three types of problems: finding common ancestors, finding the lowest common ancestor and finding disjunctive common ancestors. Each of the issues listed above can be associated with one of the similarity measures presented in section 2 - Resnik, shortest path and GraSM similarity measure respectively.

Twenty terms located deeply in the graph of the gene ontology were selected for the experiments. These were the terms with the significant number of ancestors what enables a proper evaluation of the compared methods.

Table 1. No. of ancestors of the twenty selected terms

Term Id	No. of ancestors	Term Id	No. of ancestors	Term Id	No. of ancestors
GO:0039542	126	GO:0039551	121	GO:0039560	99
GO:0039559	121	GO:0039544	117	GO:0039550	104
GO:0039558	113	GO:0039575	113	GO:0039545	103
GO:0039546	112	GO:0039583	111	GO:0039549	102
GO:0039543	109	GO:0039555	109	GO:0039548	99
GO:0039541	109	GO:0039540	108	GO:0039557	99
GO:0039554	108	GO:0039561	107		

Database Neo4j in version 2.0.6, embedded, was used during the test. In accordance with the recommendations of the creators of the database the following database and the Java virtual machine settings have been used:

Listing 1.1. Database system settings

```
-server -XX:+UseConcMarkSweepGC -Xms200m -Xmx512m -XX:MaxPermSize=512m -XX:+
      UseGCOverheadLimit
```

3.1 Finding Common Ancestors

Each semantic similarity of GO terms (e.g., Resnik similarity) requires identification of a list of common ancestors. The solution of this problem in Neo4j database environment can be approached in two ways. The first one is a comprehensive query of Cypher language referred further as *single*. The example of this query is presented below on listing 1.2.

Listing 1.2. *Single* query identifying a list of common ancestors

```
START firstTerm=node({0}), secondTerm=node({1}) MATCH firstTerm -[*]->ancestor
      <-[*]- secondTerm RETURN distinct ancestor
```

The second method, presented below on listing 1.3, is based on separate ancestors finding for each term, and then calculating the intersection of the sets. Elements of the result set form a set of common ancestors. This approach is refered further as *divide*.

Listing 1.3. *Divide* query identifying a list of common ancestors

```
Set<Term> findCommonAncestors(Long firstTermId, Long secondTermId) {
 return intersection(ancestors(firstTermId), ancestors(secondTermId));
}

@Query("START term=node({0}) MATCH a-[*]->ancestor RETURN distinct ancestor")
Set<Term> ancestors(Long term);
```

Both approaches can be compared according to their execution time and memory usage.

Tab. 2 shows the execution time expressed in milliseconds for both queries presented on lisitings 1.2 and 1.3. Tab. 2 and the rest of the tables presenting

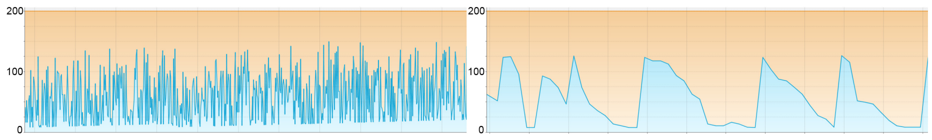
Table 2. Execution time [ms] of *single* (listing 1.2) and *divide* (listing 1.3) queries

Single / Divide	GO:0039542	GO:0039559	GO:0039551	GO:0039544	GO:0039558	GO:0039575	GO:0039546	GO:0039583	GO:0039555	GO:0039543
GO:0039542		8187	8877	8425	7642	8714	7700	8550	8151	7990
GO:0039559	155		7797	7708	7032	8038	7189	8164	7660	7658
GO:0039551	147	136		7704	7037	8009	7185	8155	7652	7662
GO:0039544	177	149	141		6879	7859	7076	7975	7527	7536
GO:0039558	173	126	127	131		7245	6544	7516	7001	7114
GO:0039575	186	136	142	138	122		7413	8402	7888	7896
GO:0039546	164	137	137	137	115	127		7477	7066	7041
GO:0039583	163	138	138	139	117	131	128		8042	8034
GO:0039555	178	148	136	150	129	139	144	137		7475
GO:0039543	174	142	135	150	127	140	133	135	149	

results contain the comparison of only 10 out of 20 terms analysed, what is a consequence of a lack of space. However, this number of measurements presents clearly the existing dependencies.

In the case of a comprehensive query (*single*, 1.2) the average execution time was 7686.5 ms, while in case of the latter approach (*divide*, 1.3) only 142.2 ms. The execution time of the *divide* query is thus on average more than 54 times shorter than it is in the case of a comprehensive query.

The Fig. 2 shows the level of memory usage during the performed experiments. It is worth noting that in the case of the first query (1.2) the value of the maximum memory usage is 150 MB, while in the case of the second one (1.3) it equals 125 MB, what is almost 17% less.



(a) *Single* query (1.2) (b) *Divide* query (1.3)

Fig. 2. Level of memory usage [MB] for method of finding common ancestors

3.2 Finding the Lowest Common Ancestor

The second issue analysed in this work is the problem of a search for the lowest common ancestor in GO graph. This problem can be encountered in the algorithms based on paths and it is based on searching for such ancestor, for which the length of the path between it and the two terms is the shortest. Again, two approaches are presented below. The first of them is based on a *single* query presented on listing 1.4.

Listing 1.4. *Single* query identifying the lowest common ancestor

```
START firstTerm=node({0}), secondTerm=node({1}) MATCH path=firstTerm-[*]->
    commonAncestor<-[*]-secondTerm RETURN commonAncestor  order by length(path)
    limit 1
```

In this query, the mechanism adjusting the pattern in graph with the simultaneous memorizing of its structure was used. Next, common ancestors are sorted by the length of the path, and the resulting set is narrowed down to only one term with the shortest path.

The second method (listing 1.5) relies on finding separate list of ancestors of two terms, the calculation of the intersection of the sets, and then calculating path lengths between terms and common ancestor.

Listing 1.5. *Divide* query identifying the lowest common ancestor

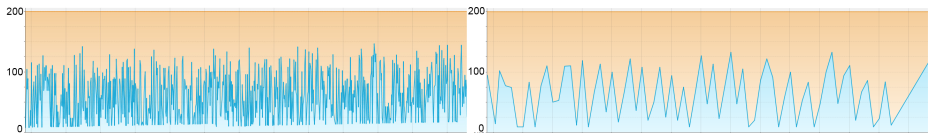
```
Term findTheLowestAncestor (Long firstTermId, Long secondTermId) {
  Set<Term> commonAncestors = intersection(ancestors(firstTermId), ancestors(
      secondTermId));
    Integer theBestPathLength = Integer.MAX_VALUE;
    Term theLowestAncestor = null;
    for(Term commonAncestor : commonAncestors) {

    int totalPathLength = shortestPath(firstTermId, commonAncestor.getId()) +
        shortestPath(secondTermId, commonAncestor.getId())

    if ( theBestPathLength < totalPathLength ) {
      theBestPathLength = totalPathLength; theNearestAncestor = commonAncestor;
      }
    }
    return theLowestAncestor;
  }
}

@Query("START firstTerm=node({0}), secondTerm=node({1}) MATCH p = shortestPath(
     firstTerm-[*]->secondTerm) RETURN p")
Path shortestPath (Long firstTerm, Long secondTerm);
```

The results of the execution time comparison of the queries 1.4 and 1.5 are presented in tab. 3.

Table 3. Execution time [ms] of *single* (listing 1.4) and *divide* (listing 1.5) queries

Single Divide	GO:0039542	GO:0039559	GO:0039551	GO:0039544	GO:0039558	GO:0039575	GO:0039546	GO:0039583	GO:0039555	GO:0039543
GO:0039542		9734	10228	10002	9020	10391	9235	10268	9739	9716
GO:0039559	192		9164	9206	8301	9529	8526	9662	9151	9129
GO:0039551	178	201		9203	8296	9535	8533	9655	9151	9141
GO:0039544	196	184	167		8278	9497	8518	9592	9125	9129
GO:0039558	234	176	162	157		8636	7745	8742	8287	8278
GO:0039575	250	179	166	168	145		8872	10022	9481	9491
GO:0039546	235	173	155	159	137	151		8947	8504	8525
GO:0039583	226	178	169	159	137	159	142		9636	9632
GO:0039555	240	178	169	168	149	163	158	160		9056
GO:0039543	236	177	163	171	144	165	155	159	166	

The results obtained show unambiguously that the average time of *divide* method is more than 52 times shorter. The average execution time of a *single* approach (1.4) was 9178.2 ms, whereas in case of *divide* approach (1.5) it was only 174.6 ms.



(a) *Single* query (1.4) (b) *Divide* query (1.5)

Fig. 3. Level of memory usage [MB] for method of finding the lowest common ancestor

Fig. 3 shows that the maximum value of the memory usage for the *single* approach is equal 150 MB. Whereas for *divide* approach only 130 MB, what is about 13% less comparing to the *single* approach to this issue.

3.3 Finding Disjunctive Common Ancestors

The last issue considered in this paper is the problem of searching for disjunctive common ancestors. This issue is encountered in the GraSM algorithm. Analogously to the two previous problems, also in this case, the two distinct ways to achieve the goal were verified. The first of them, *single* is shown below:

Listing 1.6. *Single* query identifying the disjunctive common ancestors

```
START firstTerm=node({0}), secondTerm=node({1}) MATCH firstTerm -[firstTermPaths
      *]->ancestor<-[secondTermPaths*]-secondTerm WITH firstTermPaths ,
      secondTermPaths , ancestor WHERE NOT ANY(path IN firstTermPath WHERE path IN
      secondTermPath) RETURN distinct ancestor
```

This query finds common ancestors, memorizes a path connecting the first term with the ancestor and the second term with the ancestor. In the next step, the ancestors for which there is identical path between the first term and the common ancestor, and the second term and the common ancestor, are filtered out.

The *divide* method (listing 1.7) relies on finding common ancestors, next finding all paths between the term and the ancestor, and finally determining whether any path is included both for the first and for the second term.

The results of the execution time comparison of the queries 1.6 and 1.7 are presented in tab. 4.

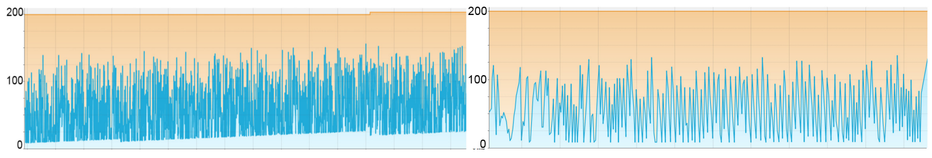
The average time of finding the common disjunctive ancestors is equal to 24726.1 ms in case of a *single* query (1.6) and 744.9 ms in case of *divide* query (1.7). It means that an average *divide* query was executed in over 33 times shorter time then a *single* query.

Listing 1.7. *Divide* query identifying the disjunctive common ancestors

```
Set<Term> findDisjunctiveAncestors(Long firstTermId, Long secondTermId) {
    Set<Term> disjunctiveCommonAncestors = {};
    Set<Term> commonAncestors = intersection(ancestors(firstTermId), ancestors(
         secondTermId));
   for(Term commonAncestor : commonAncestors) {
       Set<Path> pathsBetweenFirstTermAndAncestor = findAllPaths(commonAncestor.
            getId(), firstTermId);
       Set<Path> pathsBetweenSecondTermAndAncestor = findAllPaths(commonAncestor.
            getId(), secondTermId);
       if(!doTheyHaveACommonPath(pathsBetweenFirstTermAndAncestor,
            pathsBetweenSecondTermAndAncestor) {
          disjunctiveCommonAncestors.add(commonAncestor);
        }
  }
  return disjunctiveCommonAncestors;
}

@Query("START firstTerm=node({0}), secondTerm=node({1}) MATCH p = (firstTerm-[*]->
     secondTerm) RETURN p")
Set<Path> findAllPaths(Long firstTerm, Long secondTerm);
```

Table 4. Execution time [ms] of *single* (listing 1.6) and *divide* (listing 1.7) queries

Single / Divide	GO:0039542	GO:0039559	GO:0039551	GO:0039544	GO:0039558	GO:0039575	GO:0039546	GO:0039583	GO:0039555	GO:0039543
GO:0039542		27386	27341	18163	25310	30355	23362	30408	25973	18162
GO:0039559	787		23883	25892	18265	26221	23117	26042	25978	25976
GO:0039551	743	965		25865	21748	26241	23117	25936	25742	25751
GO:0039544	920	684	674		24109	28892	21797	28634	24381	16449
GO:0039558	793	848	828	628		23810	21024	23669	23357	23382
GO:0039575	896	737	718	680	661		25345	27492	28204	28209
GO:0039546	906	649	640	867	576	622		25306	21120	21069
GO:0039583	794	853	852	676	686	746	618		27982	27945
GO:0039555	949	690	669	854	603	650	775	642		24264
GO:0039543	963	678	661	854	604	657	770	644	813	



(a) *Single* query (1.6) (b) *Divide* query (1.6)

Fig. 4. Level of memory usage [MB] for method of finding disjunctive common ancestors

The results presented on Fig. 4 show that the level of memory usage for *divide* query slightly exceeds 125 MB, whereas for *single* query it slightly exceeds 150 MB, what is about 13% difference.

3.4 Average Execution Time and Its Standard Deviation

All of the previously presented experiments were repeated for 5% of randomly selected terms for the three Gene Ontology subgraphs: Biological Process (1267 terms), Mollecular Function (481 terms) oraz Cellular Component (164 terms). For each pair of terms belonging to the same subgraph of GO the average execution time and standard deviation were measured.

Table 5. Average time [ms] (Avg) and standard deviation [ms] (St. dev.) of execution time for 5% populations of three GO subgraphs : Biological Process (BP), Cellular Component (CC), Molecular Function (MF)

		Common ancestors		Lowest Common Ancestor		Disjunctive common ancestors	
		divide	single	divide	single	divide	single
BP	Avg	9.38	53.66	9.84	58.07	11.99	149.95
	St. dev.	21.13	388.72	21.37	433.77	26.52	1317.15
CC	Avg	6.58	40.86	7.04	48.29	11.05	123.41
	St. dev.	10.01	142.66	10.26	174.60	17.54	512.70
MF	Avg	11.52	42.42	12.07	48.45	14.83	130.42
	St. dev.	52.26	741.01	52.59	934.97	68.57	2435.99

The obtained results are consistent with the results from the previous experiments. The results presented in Tab. 5 clearly indicate, that the *divide* method has a much smaller execution time and standard deviation. It is worth noting that the standard deviation for the *single* method is many times larger, so this approach can be considered as less stable and more sensitive to the location of the terms in the GO graph.

4 Conclusions

The article presented two different approaches, applied in graph database Neo4j, to solve the issues related to Gene Ontology term similarity analysis. The issues that were taken into consideration cover finding common ancestors, the nearest common ancestor and disjunctive common ancestors.

The first approach was based on a single query of Cypher language, which is integral part of Neo4j database. The second approach divided the query into subqueries and additional merging processing.

The experiments performed proved that *divide* queries perform significantly better. The execution time analysis showed that they can be from 33 to 54 times faster then *single* queries. It was also shown that *divide* queries are more stable (in terms of execution time) and less dependent on the terms' depth in GO graph. Additionally, the memory usage was verified in the experiments. Increase of the memory usage is often observed during the time optimization in a classic optimization approach. However, this trend was not observed in the experiments performed. Moreover, the figures presented show that the maximal memory usage is always reduced when *divide* queries are executed.

This work presents how to implement efectively the three different classes of problems that can be encountered during Gene Ontology analysis. The conclusions can be generalized, as we can point a certain type of query that is a common feature of the analysed issues. This query that has the following form in Cypher language "->common_node<-" is faster realised by Neo4j when it is divided into separate queries. This conclusion does not disqualify graph database systems and Neo4j being their representative in this work as an interesting and profitable environment that can be applied to GO-based analysis, as it was pointed in other works [5].

Acknowledgments. The work was supported by National Science Centre (decision DEC-2011/01/D/ST6/07007), by Ministry of Science and Higher Education as a Statutory Research Project (decision 8686/E-367/S/2013) and by POIG.02.03.01-24-099/13 grant: "GCONiI - Upper-Silesian Center for Scientific Computation".

References

1. Al Mubaid, H., Nagar, A.: Comparison of four similarity measures based on go annotations for gene clustering. In: IEEE Symposium on Computers and Communications, ISCC 2008, pp. 531–536. IEEE (2008)
2. Ashburner, M., et al.: Gene Ontology: tool for the unification of biology. Nat. Genet. 25(1), 25–29 (2000)
3. Couto, F.M., Silva, M.J., Coutinho, P.M.: Measuring semantic similarity between gene ontology terms. Data & Knowledge Engineering 61(1), 137–152 (2007)
4. Jiang, J., Conrath, D.: Semantic similarity based on corpus statistics and lexical ontology. In: Proc. on International Conference on Research in Computational Linguistics, pp. 19–33 (1997)
5. Kozielski, M., Stypka, Ł.: Gene ontology based gene analysis in graph database environment. Studia Informatica 34(2A), 111 (2013)
6. Lin, D.: An information-theoretic definition of similarity. In: ICML, vol. 98, pp. 296–304 (1998)
7. Neo4j: Graph database: `http://www.neo4j.org`
8. Pesquita, C., Faria, D., Falcao, A.O., Lord, P., Couto, F.M.: Semantic similarity in biomedical ontologies. PLoS Computational Biology 5(7), e1000443 (2009)
9. Resnik, P.: Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. Journal of Artificial Intelligence Research 11, 95–130 (1999)