Applying Graph Partitioning Techniques to Modularize Large Ontologies

Ana Carolina Garcia, Letícia Tiveron, Claudia Justel,
Maria Cláudia Cavalcanti

Seção de Engenharia de Computação. Instituto Militar de Engenharia

Praça General Tibúrcio, 80 Praia Vermelha – Urca, Rio de Janeiro, RJ – Brazil

{carolina.acgg, leticiativeronbt}@gmail.com, {cjustel, yoko}@ime.eb.br

Abstract. *Nowadays, it is difficult to reuse ontologies, especially those that cover a large domain. It is in this context that ontology modularization can be useful. The goal of this work is to investigate graph partitioning techniques and their application on the modularization of large ontologies, typically, biomedical ontologies. Such investigation may be divided in two steps: (i) how to convert an ontology, represented in OWL or RDF languages into a graph; (ii) which partitioning algorithm would be suitable. More specifically, this work focus is on how to preserve certain ontology properties/relationships in the generated modules. Therefore, a single way of graph conversion was adopted and user-defined edge weights were taken into account. Five graph partitioning algorithms were used for the present investigation, but just three of them were used to verify their behavior in face of edge weight variations. A case study was conducted using a toy-ontology on the pizza domain, and showed preliminary but interesting results.*

1. Introduction

The constant growth of data and publications in the biomedical area has been pushing the creation and reuse of domain ontologies in that area, not only for structured data annotation, but also for text indexation and annotation. Examples of such reusage are: Genbank[1], Pubmed[2] and NCBO Portal[3]. Genbank is the most popular comprehensive database that contains publicly available nucleotide sequences, for more than 380,000 organisms. Each sequence feature (genes, repetitive areas, etc.) is usually annotated with ontology references. Pubmed is one of the most popular digital biomedical citation reference (more than 21 million). Each text citation is associated (indexed) using the MeSH[4] thesaurus. A more detailed indexation, also known as text annotation, associates text expressions to ontology terms. The NCBO (*The National Center for Biomedical Ontology*) BioPortal provides the Annotator tool, specially created to support biomedical text annotations.

The Open Biological and Biomedical Ontologies (OBO) Foundry[5] *and the* NCBO BioPortal provide together more than 300 ontologies. How can a biomedical annotator deal with such a variety of ontologies? In addition, it is even more difficult

[1] http://www.ncbi.nlm.nih.gov/genbank/
[2] http://www.ncbi.nlm.nih.gov/pubmed/
[3] http://bioportal.bioontology.org/
[4] http://www.nlm.nih.gov/mesh/
[5] http://obofoundry.org/

to reuse them taking into account that typical biomedical ontologies have more than five hundred terms.

It is in this context that ontology modularization can be useful. However, in order to put modularization into practice greatly depend on the goals that are pursued, and thus, the splitting of ontologies into smaller modules has to follow some criteria [Parent and Spaccapietra, 2009]. It is worth noting that ontologies are semantic-based structures, in which each class and each property have different meanings. Such differences should be taken into account in the process of modularization, i.e., certain classes and/or properties (relationships) may be more relevant than others in the generated modules. For instance, for a user of the Gene Ontology (GO) [GO Consortium, 2000] it may be more important to generate modules where part-of relationships between selected nodes are all included. Therefore, a good criterion for generating reusable ontology modules is to rank properties, so that they are not discarded during modularization.

The task of modularizing large ontologies, typically, biomedical ontologies, can be divided in two steps: (i) how to convert an ontology, represented in OWL or RDF languages into a graph; and (ii) which partitioning algorithm would be suitable. With respect to (i), there are different ways of doing such conversion, as stated in [Coskun *et al.*, 2011], but either one should represent property ranking values in the graph. With respect to (ii), there are many graph partitioning algorithms, such as, *Spin Glass, Fast Greedy, Walktrap, Leading Eigenvector* and *Edge Betweenness.* However, just some of them take into account edge weights: *Spin Glass, Walktrap* and *Fast Greedy*. In this context, a question still remains: which of these algorithms would be more suitable for the ontology property-aware modularization problem?

There are previous works that evaluate partitioning algorithms applied to ontologies [Coskun *et al.*, 2011][Oh and Yeom, 2012], but they did not take into account edge weight variations.

This paper investigates the behavior of graph partitioning algorithms with respect to edge weight variations, and describes a case study that shows some initial but interesting results. In order to conduct this study it was necessary to adapt and use existing tools. For (i), the PATO[6] tool was used because it includes an ontology-graph conversion mechanism that adds weights to two types of properties (is-a and other domain properties). It had to be adapted in order to assign a different weight to each distinct domain property. Then, for (ii), the iGraph[7] tool was chosen as it includes implementations of three edge weight aware partitioning algorithms.

The case study was carried out using a toy-ontology on the pizza domain. Although the modularization targets are large ontologies, it would be very difficult to analyze and evaluate the resulting modules for these ontologies. Therefore, the idea of using a small ontology was to facilitate the analysis of the modularization results. Moreover, the choice for a common knowledge domain, such as pizza, was to avoid the need for biologists or domain specific specialists in the initial tests. The pizza case study showed that the variation of properties weights led (according to the user needs) to different partitioning results. It was noted that some algorithms kept most of the

[6] http://web.informatik.uni-mannheim.de/anne/Modularization/pato.html
[7] http://igraph.sourceforge.net/

highest weight properties within the modules and did not use them as cut edges (between the modules).

The rest of this paper is organized as follows. The following section describes the biomedical scenario, which motivates this investigation. The third section describes briefly some of the main graph partitioning techniques. The fourth section describes the experiment and discusses its results. The fifth section concludes the paper, pointing to future work.

2. Biomedical Ontologies

Nowadays there are many ontologies on the biomedical domain that can be found at *The Open Biological and Biomedical Ontologies (OBO) Foundry* and at the NCBO BioPortal. The OBO Foundry is maintained by a group of researchers that establish a set of principles for ontology development with the goal of creating a suite of orthogonal interoperable reference ontologies in the biomedical domain [Smith et al. 2007]. Besides being a repository, the OBO plays the role of a reference organism, which reviews and certifies a set of ontologies on the biomedical domain. At the time of the writing of this paper, there were around 113 ontologies, from which only 8 were considered OBO ontologies, while the other 105 were still candidate ontologies. The GO (Gene Ontology) is one of the most popular among OBO ontologies.

The NCBO BioPortal [Noy et al. 2009] is an ontology repository that feeds a text annotation service. At the time of this writing there were 306 ontologies. After a quick analysis about their size, it was possible to conclude that approximately 3% have more then 100,000 classes; 8% have more than 10,000 classes; and more then 50% have more then 500 classes. Then, it is fair to say that in the biomedical domain, ontologies are typically of medium and large size. How can a biomedical (ontology-driven) annotator deal with such a variety of large ontologies?

This scenario motivates us on the investigation of alternative solutions for reducing the complexity of ontology reuse. The next section summarizes some of the graph partitioning techniques that could be useful to facilitate their reuse.

3. Graph Partitioning Techniques

The graph partitioning problem has been used to model problems of different areas. The growth and rapid evolution of real networks, created from technological and social networks, resulted in an increasing volume of data sets. These data can be used to extract information of the network elements. A network consists of a combination of elements and relationships between pairs of elements. Given this definition, we can construct an associated graph $G = (V,E)$, in which the vertices $(v \in V)$ represent the network elements and the edges $(e \in E)$ represent some kind of relationship between the elements.

The techniques for solving the graph partitioning problem try to divide the set V (the vertices of G) in clusters (also communities or partitions) that optimize a certain criterion. For instance, each cluster must have edges between internal vertices with high weight and edges between different clusters with low weight. Following this optimization criterion, the graph partitioning problem is NP hard and can be formally defined as:

Given a graph $G=(V,E)$, find p subsets $V_1, V_2, \ldots V_p$ such that:

i. $\bigcup_{i=1}^{P} V_i = V$ *and* $V_i \cap V_j = \emptyset$, for any $i \neq j$.

ii. $W(i)$ and W represent the sums of the weights of the edges between vertices inside the sets V_i and V, respectively.

iii. The sum of the weights of the edges that connect the vertices into two subsets V_i and V_j, for all pair i,j must be minimal.

There are several approximate algorithms to solve the graph partitioning. In this paper we consider the partitioning algorithms implemented in the library iGraph. This library provides an implementation of 5 (five) different algorithms for graph partitioning: *Edge Betweenness Community*, *Walktrap Community*, *Fast Greedy Community*, *Spin Glass Community* and *Leading Eigenvector Community algorithms*.

The *Edge Betweenness Community* is a divisive algorithm. It removes recursively edges of the graph until determine communities [Coskun et al., 2011]. From a non-weighted graph $G=(V, E)$ the *betweenness* of an edge $e \in E$, is the number of shortest paths that connect any two vertices v_1 and v_2 of V passing through the edge e. Note that there may be more than one shortest path between two vertices. In this case, if there are k shortest paths between the vertices v_1 and $v_2 \in E$, then each one will have a weight $1/k$ to calculate the edge's *betweenness* of these paths [Schaeffer, 2007].

This algorithm computes the values of the *betweenness* of each edge. And is based on the following observation: edges with higher value of *betweenness* must be connecting vertices of two different partitions, i.e., they are not inner edges in a partition. Then the algorithm divides the graph into clusters, removing one by one the edges with the highest value of *betweenness*. If more than one edge has the highest value, one is chosen randomly. After each removal, the *betweenness* is recalculated for each edge. This process is repeated until a stop criterion.

The *Walktrap Community* is an algorithm based on the following statement: "random walks in a graph tend to get trapped in dense parts of the graph, corresponding to the communities" [Coskun et al., 2011]. That is, by drawing a random path between two nodes, the nodes that belong to the path are more likely to belong to the same community. It is a hierarchical agglomerative algorithm, because the communities are built step by step through the union of vertices to form communities. Initially the algorithm treats all vertices of the graph as communities of a single node. Then, at each step, two communities are joined, until the stopping criterion.

The *Fast Greedy Community* is an algorithm widely used to determine communities for non-directed and sparse graphs $G=(V,E)$ [Eom et al., 2009]. This algorithm is based on the concept of modularity of a partition $C=\{C_1, \ldots C_p\}$, $Q(C)=\Sigma_{1\leq i\leq p}\, a_{ii} - a_i^2$, where a_{ii} represents the edges of the graph inside the set C_i and a_i the number of edges with one endpoint in the set C_i The *Fast GreedyCommunity* algorithm maximizes the value of the modularity in a greedy fashion.

Initially, the algorithm considers each vertex of the graph as a unitary community. Then, the algorithm finds the pair of communities C_p and C_q having the maximum value $\Delta Q_{ij} = e_{ij} - e_{ji} - 2a_i\, a_j = 2(e_{ij} - a_i\, a_j)$, where e_{ij} is the number of edges between C_i and C_j. Next, the algorithm combines these two communities C_p and C_q in only one community. The process is repeated while ΔQ is a positive number. During the process of union of two communities into one, the algorithm updates the values corresponding to neighboring communities (internal and external edges of the communities affected by the union of C_p and C_q).

The *Spin Glass Community* is an algorithm based in a thermodynamics technique to model the graph-partitioning problem. The meta-heuristic *Simulated Annealing* is used to solve the minimization energy problem considering in this model. In this context, the spin states consider the vertices of the graph and the structure of the partition in the graph is interpreted as the spin configuration that minimizes the energy of the Spin Glass.

The *Leading Eigenvector Community* algorithm uses the concept of modularity in a different way to perform the partitioning. In this case, the algorithm finds the eigenvector corresponding to the most positive eigenvalue of a modularity matrix, defined from values ΔQ_{ij} and divide the network into two groups, according to the signs of this vector elements [Newman, 2006].

Table 1 surveys 3 characteristics of the five algorithms mentioned before: type, worst-case complexity, possibility of use of weighted edges. Note that just three of them take into account edge weights. The complexities of these algorithms were obtained from the iGraph API documentation. We note the graph *G=(V,E),* where $|V|$ is the number of vertices and $|E|$ is the number of edges.

Table 1 - Comparison between the five algorithms.

Algorithm	**Weighted Edges**	**Type**	**Complexity**
Edge Betweenness	No	Divisive	$O(\|V\|^3))$
Walktrap	Yes	Agglomerative	$O(\|V\|^2 log(\|V\|))$
Fast Greedy	Yes	Greedy	$O(\|E\| + \|V\| * log^2(\|V\|))$
Spin Glass	Yes	Aproximate	Not Found
Leading Eigenvector	No	Spectral	$O(\|E\| + \|V\|^3)$

4. Case Study

As stated in [Parent and Spaccapietra, 2009], each module is expected to show a similar unit of purpose, gluing together those elements that participate on a given goal. A module should make sense to ontology engineers seeking to (re)use them [Grau et al., 2006]. For instance, a module should represent an agreed conceptualization of a sub-domain of the domain of the ontology.

Evaluating ontology modules is not an easy task. A recent related work [Oh and Yeom, 2012] proposes a new evaluation framework for selecting an appropriate ontology modularization tool. In their work, modularization tools were evaluated as use cases according to the proposed framework, which takes into account three aspects: tool performance, data performance, and usability. Data performance includes verifying the cohesion of a module, which means asking if a module contains plenty of concepts, relationships, and axioms. However, although different partitioning methods were compared, the property ranking was not used as a criterion for modularization evaluation.

Another related work [Stuckenschmidt and Klein, 2004] identifies ontology partitions depending on property weights. These weights are calculated based on graph dependencies (e.g. subclass, domain and range restrictions). However, a method purely based on the structure of the ontology may not be able to capture semantics of such properties.

The present work adopts a user-based, but probably not scalable approach. According to the user preference (weights assigned to object properties), a set of modules is generated. A module makes sense if it is cohesive, meaning if it includes the user-preferred properties, and the related concepts. Based on this criterion it was possible to evaluate the generated modules.

In order to verify the behavior of the selected five algorithms taking into account weighted ontology properties, a case study scenario was prepared: support tools were configured and/or adapted to execute such algorithms, having as input a chosen ontology. The following subsections detail the case study scenario and the subsequent executions of the algorithms, discussing the cohesion of the resulting modules.

4.1 Scenario Preparation

The modularization of a given ontology was divided into two tasks: the first one consists in representing a given ontology as a graph, and the second consists in partitioning this graph.

With respect to the first task, although it is a non-trivial task, it was not in the scope of this work to focus on this problem. There are different and richer ways of converting an OWL/RDF ontology into a graph representation [Coskun et al., 2011], but in the context of this work it was performed as follows: given an OWL file, converts it into a graph *G=(V,E),* where each OWL class or RDF resource corresponds to a vertex of *V*, and each OWL/RDF object property corresponds to an edge of *E*. In this type of conversion, usually there is no distinction between the edges, and therefore the different relationship types are lost. As we stated before, since ontologies are semantic-based structures and have different domain properties (object properties), the edge-weight variation is meaningful to their modularization.

The literature pointed to some tools to perform the first task. The Jena[8] java library and the PATO tool were alternatives. Their outputs are graphs in graphML and Pajek formats, respectively. Although Jena allows three distinct representations for the graph, PATO was more suitable as it allows assigning weight values to graph edges.

The PATO tool could be useful for the second task as well, since it performs the complete modularization of an ontology. However, this tool is poorly documented and it was not possible to identify the algorithm behind its partitioning algorithm. On the other hand, there were two known partitioning C++ libraries available: SNAP[9] and iGraph. SNAP provides the implementation of two different partition algorithms, *edge betweenness* and *fast greedy,* while iGraph provides three others besides those two: *spin glass, walktrap* and *Leading Eigenvector.* Furthermore, iGraph admits Pajek input format, allowing its use in conjunction with the PATO tool. Therefore, iGraph was chosen for its coverage and compatibility.

The PATO tool had to be adapted in two ways. First, it was removed all its unnecessary functionalities for our goal. Second, since PATO's original code only differentiates the subclass relationship and the domain properties, assigning the same weight to all the domain properties, it was necessary to adapt the code in order to allow assigning distinct weights to different domain properties.

[8] http://jena.sourceforge.net
[9] http://snap.stanford.edu/snap/

In addition to iGraph and PATO, the graphic interface of the R-tool[10] from iGraph was used in order to visualize the results. Also, it was developed a Java procedure whose input is the output of the iGraph library, in text format (.txt). For a partitioning that generates *k* communities, this routine returns k files (communities) in Pajek format.

To facilitate the analysis of the edge weight variation it was necessary to choose a domain, small but sufficient, ontology example. As most of the biomedical ontologies are large, they were initially discarded. Furthermore, it would be difficult to extract a reduced size module of it to work with. Therefore, work with a toy-ontology on a common knowledge domain would be a wise choice. A well-known example of toy ontology on the pizza domain (used in knowledge representation tutorials and courses) was chosen. However due to limitations of the PATO tool on dealing with OWL format, an RDF smaller version of the pizza ontology[11] was adapted and used.

4.2 Partitioning Results

The output graph obtained with PATO from the Pizza ontology is shown in Figure 1. This figure was generated with the aid of R-tool and Power Point. The Pizza ontology used has three types of properties: "*isa*", "*consists_of*" and "*consumes*" and for this study it was planned 5 (five) combinations of weight distribution for the edges, as described in the Table 2. From each choice of the edge weights combinations, 5 weighted graphs corresponding to the Pizza ontology were created.

Table 2 - Weight combinations of the edges assigned to the Pizza graph

Graph	consists_of	Consumes	Isa
0	1	1	1
1	1	1	2
2	1	2	1
3	2	1	1
4	2	2	1

Vertex	
0	Non_Vegetarian_Ingredient
1	Raw_Ingradient
2	Vegetarian_Ingredient
3	Vegan_Customer
4	Vegetarian_Customer
5	Pizza
6	Edible_Thing
7	Customer
8	Vegan_Ingredient
9	Small_Pizza
10	Medium_Pizza
11	Large_Pizza
12	Non_Vegetarian_Pizza
13	Vegetarian_Pizza
14	Vegan_Pizza
15	Non_Vegetarian_Customer

Graph

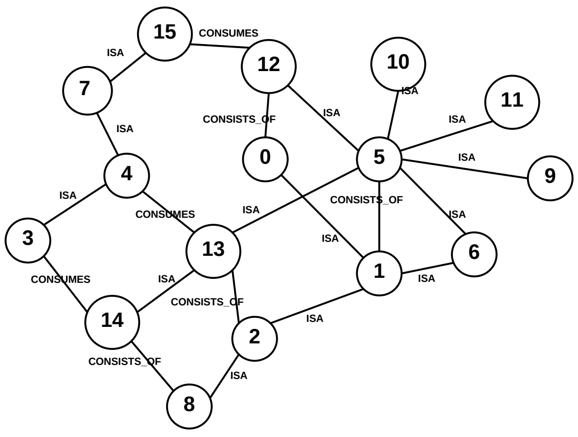


Figure 1 - Vertices of the graph corresponding to the Pizza ontology

[10] http://www.R-project.org/
[11] http://www.heiko-stoermer.net/teaching/2006-models-and-techniques-of-knowledge-representation

Each of the five resulting graphs was partitioned by *Fast Greed, Spin Glass* and *Walktrap* algorithms, which allows weighted graphs as input. The other two algorithms, *Leading Eigenvector* and *Edge Betweeness*, were executed only to the graph with constant weights (graph 0).

For graph 0, the *Spin Glass* and *Fast Greedy* algorithms obtained the same result (the same 3 communities). The *Walktrap* and *Edge Betweenness* algorithms obtained the same result (the same 3 communities). But the difference between the partitions obtained in these two cases is the allocation of vertices 0 and 12 (Non-Vegetarian Pizza and Non-Vegetarian Ingredient). Figure 2 shows the output by the 5 algorithms for graph 0. In this case, constant weights for the edges, the algorithms seem to have similar or slightly different behavior. However, the communities obtained with *Fast Greedy* and *Spin Glass* algorithms seem to be better.

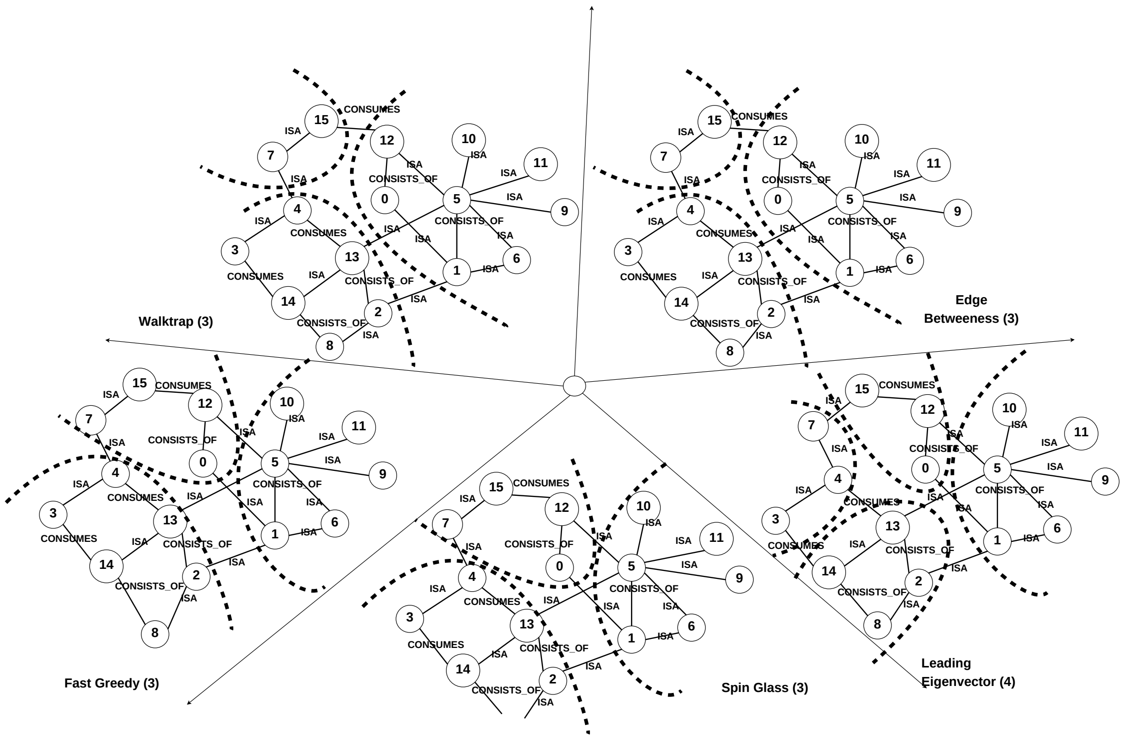


Figure 2 - Partitioning for the first graph (graph 0).

In what follows, we will discuss the content of each community generated with the algorithms. Table 3 shows the vertices allocated in each of the 3 communities by *Fast Greedy* and *Spin Glass* algorithms for graph 0. Both algorithms included vegan/vegetarians on one community and non-vegetarians in another. Furthermore, a third community of generic nodes, which did not fit in either first two communities, was generated. Note that vertex 7 is not allocated in the generic nodes community. However, vertex 7 is not adjacent to any other vertex in the third community, and therefore the obtained result makes sense.

For the graph 0, *Edge betweenness* and *Walktra*p algorithm obtain 3 communities. In this case, vertices 0 and 12 (Non-Vegetarian Ingredient and Non-Vegetarian Pizza) belong to the partition correspondent to general classes (Non-Vegetarian Customer). Both algorithms focus on paths (one is deterministic and the other is probabilistic), and this approach is different from the first two algorithms analyzed. However, more tests should be performed in order to obtain general conclusions.

Table 3 - Communities generated with *Fast Greedy* and *Spin Glass* algorithms for graph 0.

Community	**Vertices**	**Description**
Vegan/Vegetarian	2, 3, 4, 8, 13, 14	Vertices that represent Vegans and Vegetarians Pizzas, Ingredients and Customers
Non-Vegetarian	0, 7, 15, 12	Vertices that represent Non-Vegetarian Ingredient, Non-vegetarian Pizza and Non-Vegetarian Customer, and yet Customer (vertex 7)
Pizza	5, 11, 9, 10, 1, 6	Vertices that represent General classes (Pizza, Raw Ingredient, Edible Thing) and pizza sizes.

It is also worth noting that, different from the other algorithms, the *Leading Eigenvector* algorithm generated an extra partition with different types of customer (Vegan and Vegetarian Customer and Customer itself). Vertex 15 (Non-Vegetarian Customer), which seemed to be semantically closer to this community, was allocated in another related community, which includes Non-Vegetarian Pizza and Non-Vegetarian Ingredient.

With respect to the other graphs (graphs 1-4), *Fast Greedy* algorithm showed the best performance in terms of vertex clustering (communities). *Fast Greedy*'s partitioning results are shown in Figure 3, while Figures 4 and 5 show the partitioning results for the *Spin Glass* and *Walktrap* algorithms, respectively.

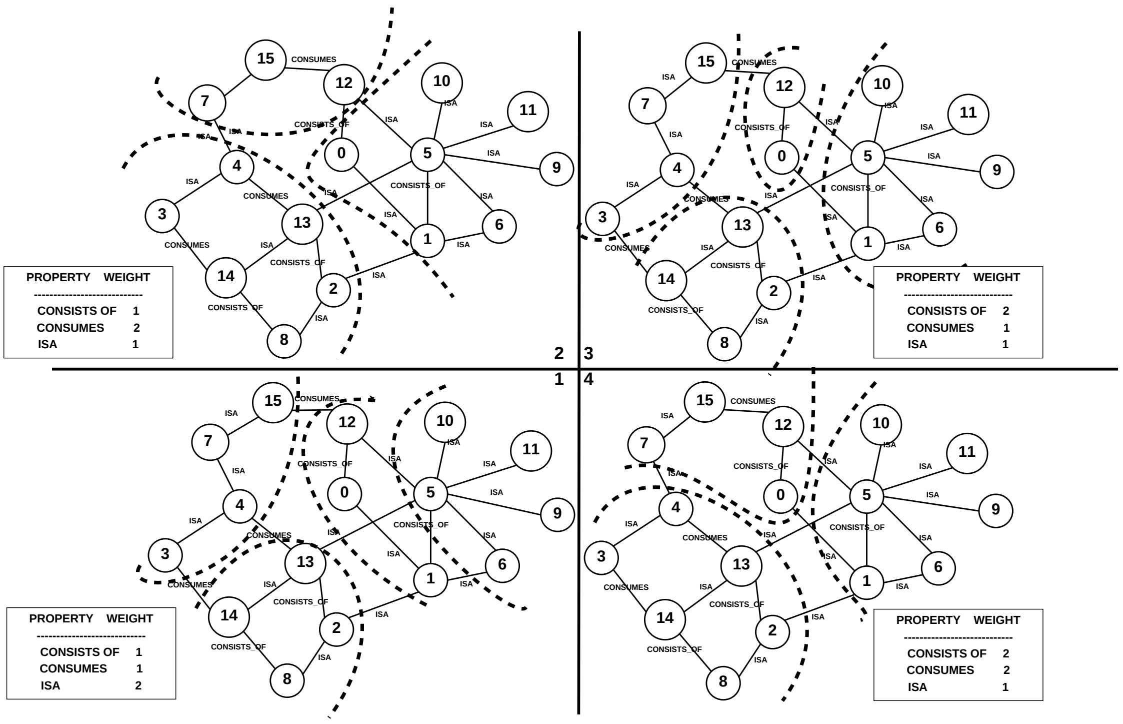


Figure 3 – *Fast Greedy* communities for different weight configurations (the numbers at the center correspond to graph numbers of Table 2).

Graph 3, which assigns the highest weight for the "*consists_of*" property, is the one for which *Fast Greedy* showed the best partitioning. This partitioning seems to make more sense than the one generated by graph 0 (analyzed previously). The four generated communities may be easily described, as follows: (i) Consumers (3,4,7,15); (ii) Non-vegetarians (12,0); (iii) Generic classes (1,5,6,9,10,11); (iv) Vegan and vegetarians (2,8,13,l4). Note that vertex 7 (Customer) now belongs to a community that includes the other customers. The partitioning for graph 4 is equal to graph 0. In the other graphs (1 and 2) there are some out of place vertices. For instance, vertex 0 (non-vegetarian ingredient) is separated from vertex 12 (non-vegetarian pizza).

The partitioning results for Spin Glass algorithm do not show much variation. Graphs 2-4 partitioning results are equal to graph 0 results. Graph 1, which assigns the highest weight for the "isa" property, is the only one whose results are different, but interesting. Note that they are similar to graph 3 results of the Fast Greedy algorithm, with the difference that communities (ii) and (iii) are merged.

Similarly, partitioning results for Walktrap algorithm show variation only for graph 1, and one of its communities (0,1 and 6) do not make much sense, joining together vertices with not much in common.

Another important analysis is if the generated modules attend the property ranking criterion, i.e., if the module includes the properties and concepts according to the user priority assignment. As stated before, the idea is to prioritize one type of property, in order to maintain them in the resulting partition. Figure 3 shows that the partitioning results for graph 1, which assigns highest weight value to the "*isa*" property, shows that most of the edges that represent this property are "inside" the community, i.e., they are not between two different communities (cut edges). Similarly, partitioning results for graph 2, which assigns highest weight value to the "*consumes*" property, shows that all the edges that represent the "*consumes*" property are inside the communities. In other words, the vertices joined by edges of greater weight tended to remain in the same community, and these edges were then maintained in some partition. *Fast Greedy* algorithm also showed the best results, varying according to the different configurations of edge values. Table 4 summarizes its results. Note that when the "*consumes"* property is prioritized, it does not appear as a cut edge (0) in the graph. The same occurs with the *"consists_of"* property.

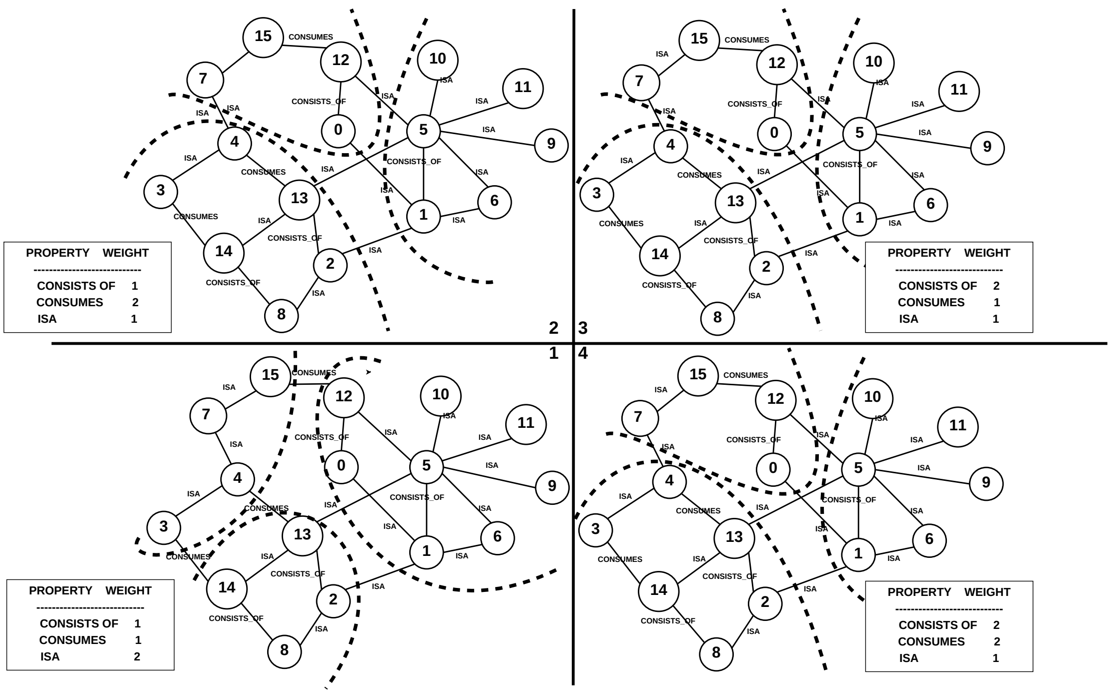
Table 4 – Property Priority (highest weight) versus the number of cut edge and inside properties for the *Fast Greedy* algorithm (analysis from Figure 3 graphs)

Priority (> weight)	**cut edge properties**			**inside properties**		
	isa	**consists_of**	**consumes**	**isa**	**consists_of**	**consumes**
Isa (graph 1)	3	2	3	11	2	0
Consists of (graph 3)	4	0	3	10	4	0
Consumes (graph 2)	4	1	0	10	3	3

5. Conclusion

This paper described an experiment that showed some initial but interesting results on how partitioning algorithms behave for ontology modularization, with focus on edge weight variations. In the context of ontologies, it makes sense to identify priorities for ontology object properties before partitioning. This can lead to more useful ontology modules from the user point of view.

The focus of this work was on the data performance evaluation, one of the aspects of the evaluation framework proposed in [Oh and Yeom, 2012]. More specifically, the focus was on the cohesion of the resulting modules. Five graph partitioning algorithms were executed for the graph representation of a toy-ontology on Pizza domain, but only three of them allowed the generation of weighted graphs. Among these three, *Fast Greedy* algorithm had the best preliminary results, showing "sensibility" with respect to the domain property ranking. However, further tests should be executed with larger and different ontologies. The suggested assumptions stated in this work can be refuted or confirmed by future work.



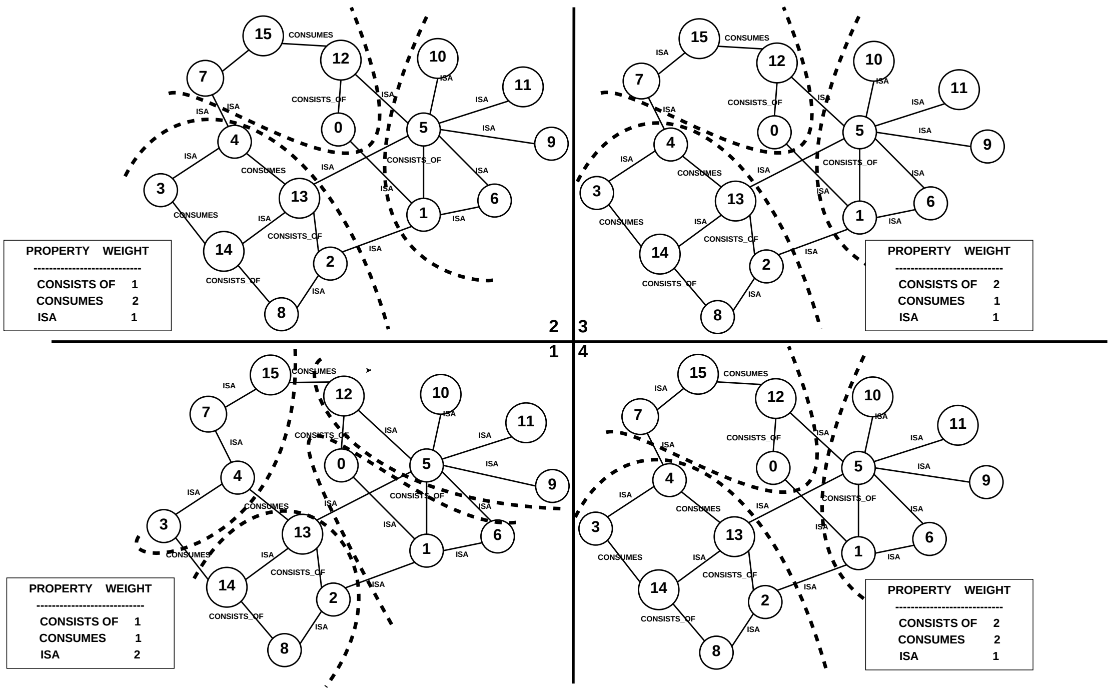
PROPERTY	WEIGHT
CONSISTS OF	1
CONSUMES	2
ISA	1

PROPERTY	WEIGHT
CONSISTS OF	2
CONSUMES	1
ISA	1

PROPERTY	WEIGHT
CONSISTS OF	1
CONSUMES	1
ISA	2

PROPERTY	WEIGHT
CONSISTS OF	2
CONSUMES	2
ISA	1

Figure 4 – *Spin Glass* communities for different weight configurations.



PROPERTY	WEIGHT
CONSISTS OF	1
CONSUMES	2
ISA	1

PROPERTY	WEIGHT
CONSISTS OF	2
CONSUMES	1
ISA	1

PROPERTY	WEIGHT
CONSISTS OF	1
CONSUMES	1
ISA	2

PROPERTY	WEIGHT
CONSISTS OF	2
CONSUMES	2
ISA	1

Figure 5 – *Walktrap* communities for different weight configurations.

Acknowledgements

The authors would like to thank CNPq (309307/2009-0; 305516/2010-8 486157/2011-3; PIBIC scholarships) and FAPERJ (E-26/111.147/2011) for partially funding their research projects.

References

Coskun, G.; Rothe, M.; Teymourian, K.; Paschke, A. (2011). “Applying Community Detection Algorithms on Ontologies for Identifying Concept Groups”. Proc. of the Fifth International Workshop on Modular Ontologies, (WoMO 2011), p. 12-24.

GO Consortium. (2000) “Gene ontology: Tool for the Unification of Biology”. Nat. Genet., 25(1), p. 25-29.

Grau, B.C., Parsia, B., Sirin, E., Kalyanpur, A. (2006) “Modularity and web ontologies”. In: Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006), p. 198-209.

Eom, Young-Ho; Choi, Yoonchan; Jeong, Hawoong; Kwak, Haewoon; Moon, Sue (2009) “Mining Communities in Networks: A Solution for Consistency and Its Evaluation”. In: Proc. of the Internet Measurement Conf. (IMC 2009), p. 301-314.

Newman, M. E. J. (2006) “Finding community structure in networks using the eigenvectors of matrices”. Physical Review E 74(3).

Noy, N. F., Shah, N.H., Whetzel, P.L. et al. (2009) “BioPortal: Ontologies and Integrated Data Resources at the Click of a Mouse”. Nucleic Acids Research 37 (Web-Server-Issue), p. 170-173.

Parent, C. and Spaccapietra, S. (2009) “An Overview of Modularity”. In: Stuckenschmidt, H., Parent, C.; Spaccapietra, S. (Eds) Modular Ontologies: Concepts, Theories and Techniques for Knowledge Modularization. Lecture Notes on Computer Science 5445, Springer, p. 5-23.

Schaeffer, S.E. (2007) “Graph Clustering”. Computer Science Review 1(1), p. 27-64.

Smith, B., Ashburner, M., Rosse, C., et al. (2007) “The OBO Foundry: Coordinated Evolution of Ontologies to Support Biomedical Data Integration”. Nature Biotechnology 25, p.1251-1255.

Stuckenschmidt, J. and Klein, M. (2004). Structure-based partitioning of large concept hierarchies. In: Proc. Int. Semantic Web Conference (ISWC).

Oh, S. and Yeom, H.Y. (2012) “A comprehensive framework for the evaluation of ontology modularization”. Journal Expert Systems with Applications 39(10), p. 8547-8556