U.S. DEPARTMENT OF ENERGY

PNNL-20317

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Graph-O-Scope Concept

S al-Saffar
C Joslyn

February 2011

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle *Since 1965*

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service,
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

This document was printed on recycled paper.
(9/2003)

Graph-o-scope Concept*

Sinan al-Saffar, Cliff Joslyn, and Alan Chappell

February 14, 2011

Abstract

This document presents the requirements and specification for Graph-oscope, a fast semantic graph analyzer.

1 Introduction

It is increasingly common in research efforts in Semantic Graph Databases (SGDs) that there is a need to do initial anlysis of an existing SGD. An SGD may be provided from web crawls, be curated by a data management group, be extracted from structure or unstructured data, be generated from a model, or provided from some other source. When a new SGD is available, questions immediately arise, not just about raw size, but about the distribution of semantic and structural properties, and the nature and extent not just of the base data, but also the semantic meta-data and ontological typing information present (if any), and the extent of reification.

A facility to flexibly and scalably provide such reporting is clearly in order, and within multiple implementation environments (e.g. against SPARQL, SQL, applications programming languages, or within a high performance computing environment). We have both been gathering requirements for such a facility, and developing initial capabilities for use in curent projects.

This document is intended to provide a specification of an overal software capability to support such analysis of semantic graphs in a rapid fashion. This capability is intended to be implemented in a variety of formats for application against a variety of SGDs. In practice, implementations and deployments will be made as needed to satisfy the needs of multiple projects. For example, there may be the facility of a stand-alone tool for semantic graph analysis, and also a module in graph querying and mining engines. Versions of Graph-o-scope are anticipated which are serial or parallel, to be deployed in medium- or high-performance envrionments, or potentially involving proof-of-principle research prototype algorithms.

It should be noted that Graph-o-scope is scoped to be a reporting tool for fast data analyses of an overall SGD, and not as a general graph query or mining

*PNNL Technical Report PNNL-20317.

engine. Basically, the input is a whole SGD, and the outputs are a variety of user-selected descriptive statistical and semantic features, potentially in different formats. In particular Graph-o-scope is not intended to support general graph search nor is it intended to perform graph mining or summarization operations whose implementations require exponential resources.

Sinan: What do you mean by this in the original? "This high level of specialization enables the design to exploit properties specific to the specialization in order to achieve enhanced performance. "

In some implementations, the design will be able to exploit properties specific to large SGDs to achieve enhanced performance. While some of the statistics generated by Graph-o-scope's functions can be implemented in query languages such as SPARQL or SQL, such implementations would be slower against large SGDs as those languages cater to a general query capability. In effect, these high performance Graph-o-scope implementations are a calculated trade-off that sacrifices generality to achieve significantly greater performance.

We also refer the reader to additional work where some Graph-o-scope capabilities are described or used [3, 4, 5, 6, 7, 8].

2 Semantic Graphs

We presume that the readers are familiar with graph theory and at least the rudiments of semantic graphs, databases, and the OWL/RDF/SPARQL paradigm. In this section we introduce basic concepts for clarity.

Semantic Graphs are realized as mathematical objects which are directed graphs with labels on both nodes and edges. Labels carry meta-data about node and edge types, and are frequently values in ontological typing structures. In a particular edge of the source node is referred to as a subject while the target node is referred to as the object. Subjects and objects together are resources. The directed edges are referred to as properties or predicates. Thus a labeled edge of type p from resource subject s to resource object o is represented as a triple $\langle s, p, o \rangle$, corresponding in logic to a binary predicate $p(s, o)$.

The following restrictions apply on the labels:

URI Labels : A Uniform Resource Identifier is simply a string that serves as a globally unique identifier. URI labels are the only kind of labels permitted on edges. They also may appear in subject and object nodes.

Blank Nodes : A blank node is labeled with a locally unique identifying string. The purpose of such blank nodes is to represent there exists relations so care must be taken when merging different datasets containing blank nodes in order to avoid name clashes.

Literals : Literals are labels on terminal nodes, nodes that are only object nodes in the graph and hence the edges leading into those nodes can be thought of as node properties and the literal labels on those nodes are the values of those properties. A literal may be a string representing a name or some length text, an abstract, or an int value.

We think of a knowledgebase as having three main components based on what is being described in each of these components. Basically we have $KB = IG \cup ON$ or the knowledgebase is comprised of the ontologies plus the instance graph that may contain reified edges which may be thought of as a third component.

Ontology and Types : An ontology provides the typing and semantic references for the semantic graph and is represented as a semantic graph itself. Nodes in an ontology represent classes or types. Within the ontology, relations between the different classes such as subclassOf and other semantically valid predicates are presented with the edges. When given a set of files, say in triple RDF format representing a semantic graph, they may or may not contain ontological information embedded in them. Typically noisy sources such as data sets resulting from web crawls [1] will contain ontologies or fragments of ontologies. Cleaner sources with uniform data [2] typically have their ontologies provided in separate files with import statements pointing to those files. In either case when analyzing the data it is important to be able to identify ontologies.

Instance Graph : This is the main data component where nodes represented by URI's have directed relationships expressed with the directed labeled edges between these nodes. The edges themselves are labeled with URI's possibly drawn from the predicates defined by an ontology. Some nodes are labeled with literals and we think of those as node properties (the edges going in those literal nodes do not represent a relationship). Instance Graphs are connected to ontologies through edges labeled with rdf:type properties specifying the ontologically defined types of the nodes.

Reified Information : Because an edge is an ontologically defined predicate, edges in the knowledge base are not instances. Rather, they define a relationship between two entities. Thus, it is not meaningful to annotate or otherwise describe an edge. However, it often is meaningful to describe the combination of the subject, object, and predicate, i.e., the full sentence. When we want to represent information about an edge and its two end nodes then we need to use reification. With reification four new edges are needed to state information about the edge in question. One edge points to the subject node, one to the object, and one to the predicate connecting the two. The subject of these three edges is one new node representing the statement we are reifying. The fourth reficiation edge is used to express that this node is actually of type Statement so its edge label is rdf:type with the target node representing the Statement type.

3 Ontological Separation

At a conceptual level ontologies play a distinct role in datasets. An ontology expresses relationships between classes and properties as opposed to the relations

between instances in the instance graph portion of the knowledge base. The typing provided by ontologies contributes to the understanding of the roles that instances of those types play in the instance graph. Furthermore, ontologies along with inference serve as a factory for additional knowledge beyond that explicitly asserted.
A semantic graph dataset may contain zero or more ontologies. Noisy datasets such as the Billion Triples Challenge [1] typically contain many ontologies or even fragments of ontologies whereas cleaner and more uniform datasets such as those generated through computer programs typically contain a single ontology. A good example of a dataset with a billion triples accompanied by a single ontology is the LUBM8k dataset [2]. Currently, most instance data links only to one ontology. However as the complexity of multi-ontology systems grows, instance data that mixes ontologies and even instances with multiple typing from different ontologies will likely become more prevalent. Given this, the analysis of any semantic dataset needs to explain their ontological components distinctly from the instance data to more fully support understanding that data set. To accomplish this distinct analysis, Graph-o-scope must provide the functionality of separating the ontologies from the instance graph. This is not a trivial task as ontologies and instance graphs are both represented using the same semantic graph representation and are not always deployed in separation. Beyond separation, ontologies need to be complete. Noisy datasets may only instantiate portions of an ontology or may only include the specific type and relation definitions used. Without the complete ontology, implied additional information may be lost. Hence, ontologies should be obtained from their sources in order to avoid using erroneous versions or fragments that may be inadvertently present in the data as a side effect of the production process.

It is difficult to separate the instance graph from the ontology using SPARQL alone as there is not one consistent property that belongs to one and not the other especially in noisy data. For example, the ontological rdf:type predicate links instances to the ontology, but is also used within the ontology as well. In our experience with noisy data, many classes have no class type (so we could not query on nodes that have a type "owl:class" for example to identify the ontology).

An alternative algorithm is to identify instances based on the fact that instances would have a type but not themselves be a type to other nodes. Instances then are the leaves of the ontological class hierarchy (even if we don't have that entire heirarchy). So we may try some logic as expressed in the following SPARQL code:

```
SELECT *  {
    ?x ?a ?class.
    optional{?s ?a ?x}.
       FILTER (!bound(?s))
};
```

However, the above logic does not work on real-world noisy data as there are cases where intermediate ontological nodes have a type but are not proper

types to other classes but rather super classes to those other classes.

One viable, conceptually sound, and clean way to separate the ontology is to identify the ontologies from their URI's in the ontological triples in the dataset and then discard all those ontological triples. A single triple for each ontology is then added to "import" it from the sources as identified by those ontological URI's. We also need to think about blank nodes for intersection and union.

Below is a sketch of such a separation algorithm:

{ALGORITHM: Find Ontological Triples}

$Classes \leftarrow$ SELECT distinct ?class where {?s a ?class} //List of all classes
$Prefixes \leftarrow$ getUniquePrefixes($Classes$) //List of all ontological prefixes
$Triples \leftarrow$ SELECT ?s ?p ?o where {?s ?p ?o} //List of all triples

for all $t_i \in Triples$ **do**
  $ont_i \leftarrow false$ //flag
  if $prefix(subject(t_i)) \in Prefixes$ AND $prefix(object(t_i)) \in Prefixes$ **then**
    $ont_i \leftarrow true$ //t_i is an ontological triple
  end if
end for

4 Library Functions

Where appropriate we give the SPARQL lines that return the desired results. The tool will not be implemented in SPARQL as the whole purpose is to achieve rapid stating of the data and SPARQL operates against slower though more general databases. The inclusion of SPARQL is for communicating the semantics of the functions. In all the following functions we assume that ontological triples are already removed from the semantic graph and replaced with import RDF statements indicating the sources of those ontologies as described in the previous section.

4.1 Graph Properties

These are single-variables descriptions of the semantic graph

4.1.1 unsigned int countKBNodes()

Returns the number of nodes in the graph

```
SELECT COUNT( distinct ?s ) AS ?no  {
    { ?s ?p ?o  } UNION { ?o ?p ?s }
};
```

4.1.2 int unsigned countKBEdges()

Returns the number of edges in the graph.

```
SELECT COUNT(?p) AS ?no  {
    ?s ?p ?o
};
```

4.1.3 double getKBDensity()

The density of a graph $G = (V, E)$ measures how many edges are in set E compared to the maximum possible number of edges between vertices in set V. Density is calculated from the number of edges and number of nodes obtained from calling the two functions countKBNodes() and countKBEdges() and then:

$$Density_{kb} = |E_{kb}|/(|V_{kb}|^2)$$

4.1.4 unsigned int countIGNodes()

Returns the number of instance graph nodes in the KB. An instance graph node is defined as a node that is in the knowledgebase but not in the ontology and is also not a literal node. The purpose is to understand how many nodes are involved in instance relations in the knowledgebase.

4.1.5 int unsigned countIGEdges()

Returns the number of edges in the instance graph (knowledgebase portion of the graph but not in the ontology) and not used to describe node properties (exclude edges leading to literal nodes).

4.1.6 double getIGDensity()

$|V_{ig}| = countIGNodes()$ and $|E_{ig}| = countIGEdges()$ then:

$$Density_{ig} = |E_{ig}|/(|V_{ig}|^2)$$

4.1.7 unsigned int countOnNodes()

Returns the number of ontological nodes in the graph. This should be greater than the number of direct classes as it contains classes that are not super types but not proper types of other nodes for example. This count should be based on the ontologies imported from their original sources through following the RDF import statements.

4.1.8 int unsigned countOnEdges()

Returns the number of edges in the ontological portion of the graph but not in the instance graph (edges connecting classes but not individuals). This count should be based on the ontologies imported from their original sources through following the RDF import statements.

4.1.9 double getOnDensity()

$|V_{on}| = countOnNodes()$ and $|E_{on}| = countOnEdges()$ then:

$$Density_{on} = |E_{on}|/(|V_{on}|^2)$$

4.1.10 unsigned int countTypedNodes()

Returns the number nodes that at least have one type in the instance graph.

```
SELECT  COUNT(distinct ?s) AS ?count {
    ?s a ?class
} ;
```

4.1.11 unsigned int countURINodes()

Returns the number of URI nodes in the graph

```
SELECT COUNT(distinct ?s ) AS ?no  {
    { ?s ?p ?o  } UNION { ?o ?p ?s }
    FILTER(isURI(?s)).
};
```

4.1.12 unsigned int countBlankNodes()

Returns the number of blank nodes in the instance graph.

```
SELECT COUNT(distinct ?s ) AS ?no  {
    { ?s ?p ?o  } UNION { ?o ?p ?s }
    FILTER(isBlank(?s)).
};
```

4.1.13 unsigned int countLiteralNodes()

Returns the number of literal nodes in the instance graph. Note that we could just interrogate the objects nodes if that is results in a faster implementation as a literal should always be a terminal object node.

```
SELECT COUNT(distinct ?s ) AS ?no  {
    { ?s ?p ?o  } UNION { ?o ?p ?s }
```

```
    FILTER(isLiteral(?s)).
};
```

4.2 S, P, O Counts

4.2.1 unsigned int countTotalSubjects()

Returns the number of nodes that are subjects in the graph. Subject and object nodes are not disjoint sets.

```
SELECT COUNT(distinct ?s) AS ?no  {
    ?s ?p ?o
};
```

4.2.2 unsigned int countURISubjects()

Returns the number of nodes that are URI subjects in the graph.

```
SELECT COUNT(distinct ?s) AS ?no  {
    ?s ?p ?o .
    FILTER (isURI(?s)).
};
```

4.2.3 unsigned int countBlankSubjects()

Returns the number of nodes that are blank subjects in the instance graph.

```
SELECT COUNT(distinct ?s) AS ?no  {
    ?s ?p ?o .
    FILTER (isBlank(?s)).
};
```

4.2.4 unsigned int countTotalObjects()

Returns the number of nodes that are objects in the instance graph. Subject and object nodes are not disjoint setes.

```
SELECT COUNT(distinct ?o) AS ?no  {
    ?s ?p ?o
};
```

4.2.5 unsigned int countURIObjects()

Returns the number of nodes that are URI objects in the graph.

```
SELECT COUNT(distinct ?o) AS ?no  {
    ?s ?p ?o .
    FILTER(isURI(?o)).
};
```

4.2.6 unsigned int countBlankObjects()

Returns the number of blank objects nodes in the instance graph.

```
SELECT COUNT(distinct ?o) AS ?no  {
    ?s ?p ?o .
    FILTER(isBlank(?o))
};
```

4.2.7 unsigned int countTotalPredicates()

Returns the number of predicates (distinct edges by their labels) in the instance graph. Edges with the same URI are considered duplicates and are counted once.

```
SELECT COUNT(distinct ?p) AS ?no  {
    ?s ?p ?o
};
```

4.2.8 unsigned int countRelationalPredicates()

Returns the number of different predicates (distinct edges by their labels) in the instance graph that connect resources (non literal nodes) and hence represents relationships between instances. Edges with the same URI are considered duplicates and are counted once. It is not trivial to separate the edges in the instance graph from the ontological ones using SPARQL (see the algorithmic sketch to do this in section 4). Here is the code to ignore the data properties:

```
SELECT COUNT(distinct ?p) AS ?no  {
    ?s ?p ?o.
    FILTER (!isLiteral(?o)).
};
```

4.2.9 unsigned int countDataPredicates()

Returns the number of predicates (distinct edges by their labels) in the instance graph that point to literal nodes and hence represents node data properties. Edges with the same URI are considered duplicates and are counted once.

```
SELECT COUNT(distinct ?p) AS ?no  {
    ?s ?p ?o.
    FILTER (isLiteral(?o)).
```

```
};
```

Note that:
$countDataPredicates() + countResourcePredicates() = countTotalPredicates()$

4.3 S, P, and O Distributions

4.3.1 dist getPredicateDistribution()

Returns a pointer to a class (or struct if implemented in C) that contains three slots. The first two are char and int pointers to two arrays:

```
    struct distStruct {
            char* uri;
            unsigned int* count;
            unsigned int size;
    };  typedef struct distStruct * dist;
```

The third slot contains the size of the distribution which should be equal to the value returned by calling countTotalPredicates().
Example truncated output produced from the uri and count pointers:

```
http://www.battelle.org/TAI/base#name                    11621
http://www.w3.org/1999/02/22-rdf-syntax-ns#type          11558
http://www.battelle.org/TAI/base#involves                3391
http://www.battelle.org/TAI/base#has_source              1065
http://www.battelle.org/TAI/terrorism#has_target         922
http://www.battelle.org/TAI/base#has_receiver            888
http://www.w3.org/2000/01/rdf-schema#label               569
http://www.battelle.org/TAI/terrorism#has_attacker       559
http://www.battelle.org/TAI/base#has_acquired_item       370
http://www.battelle.org/TAI/base#has_time                224
http://www.battelle.org/TAI/base#has_location            130
```

```
SELECT ?p  (COUNT(?s) AS ?count1)  {
          ?s ?p ?o.
} ORDER BY desc(?count1) ;
```

4.3.2 dist getSubjectDistribution()

The URIs are pretty meaningless here, forcing the user to start searching for those URIs to try to understand. If there is a name/label data property, I'd much prefer to see that.

Returns a pointer to a dist struct as in the previous function. This call basically returns graph nodes and their out-degrees. The size of the distribution is equal to the value returned by calling countTotalObjects(). Example output produced from the uri and count pointers:

```
http://akea.pnl.gov/extracted#fee032ff-dbd1-441b-b7ab          41
http://akea.pnl.gov/extracted#23032b16-0dcc-4563-b12f          24
```

```
http://tai.pnl.gov/extracted#8c156695-6d29-4c6c-86e6          24
http://akea.pnl.gov/extracted#35e4a4b8-8e25-4319              23
http://akea.pnl.gov/extracted#1521324b-050c-4a8d              23
http://akea.pnl.gov/extracted#1b1b961d-1da9-4032              22
http://akea.pnl.gov/extracted#44e4b1f3-7df1-4b07              22
http://akea.pnl.gov/extracted#d63f0474-9122-477d              18
```

```
SELECT ?s COUNT(?p) AS ?count1 {
    ?s ?p ?o.
}  ORDER BY desc (?count1) ;
```

4.3.3 dist getObjectDistribution()

This must be old. You said earlier that we were pulling out the ontology, now its back in?

Returns a pointer to a dist struct. This call basically returns nodes and their in-degrees. The size of the distribution is equal to the value returned by calling countTotalObjects(). Example output produced from the uri and count pointers:

```
http://www.battelle.org/TAI/base#Entity              6364
http://www.battelle.org/TAI/base#Communicate         1134
http://www.battelle.org/TAI/terrorism#Attack         1131
http://www.battelle.org/TAI/base#Acquire             438
http://www.battelle.org/TAI/base#Publish             399
posted                                               342
http://www.battelle.org/TAI/base#Person              303
http://akea.pnl.gov/extracted#05edc840-850e          204
```

Usually the nodes with the highest in-degree are ontological. And here is the sparql implementation:

```
SELECT ?o COUNT(?p) AS ?count1 {
    ?s ?p ?o .
}  order by desc (?count1) ;
```

4.4 Typing

4.4.1 unsigned int countKBClasses()

Returns the number of classes used in the graph.

```
SELECT COUNT(distinct ?o) AS ?no {
        ?s a ?o
};
```

4.4.2 dist getClassDistribution()

Returns a pointer to a dist struct. Lists classes in the graph and the count of the number of nodes from each of those classes. The size of the distribution is

equal to the value returned by calling countTotalClasses(). Example truncated output produced from the uri and count pointers in the dist structure:

```
http://www.battelle.org/TAI/base#Entity              6362
http://www.battelle.org/TAI/terrorism#Attack         1130
http://www.battelle.org/TAI/base#Communicate         1130
http://www.battelle.org/TAI/base#Acquire             438
http://www.battelle.org/TAI/base#Publish             399
http://www.battelle.org/TAI/base#Person              301
http://www.battelle.org/TAI/terrorism#PoliceAction   186
```

Usually the nodes with the highest in-degree is ontological. And here is the sparql implementation:

```
SELECT  ?class (COUNT(?s) AS ?count ) {
               ?s a ?class.
 } GROUP BY ?class ORDER BY desc(?count)
```

4.5 Reification

Reified information is information about the graph edges as opposed to the graph nodes. To be able to describe graph edges within a graph model a new node is created to represent the target edge. This node is hooked to the target edge with three extra edges pointing to the subject, predicate, and object of the edge. In addition a fourth edge is used to declare that node to be of type statement. Thus four edges are used to create a new node in the graph to which other properties can be attached for provenance information for example. It is useful to know what percentage of the semantic graph is there to declare these reification hooks vs. the portion that is actually used for reification (has extra edges containing pointing to edge properties). Furthermore there are rare situations where reification hooks are declared and edge information such as provenance exist but the edge itself is missing. The meaning of such statements can be thought of as the target statement has been stated by a source but its truth value is not yet known. It is useful to obtain the percentage of those motifs as well. Here is a summary:

- portion of sentences reified (number of edges that have a reification hook defined to number of total edges)
- portions of reifications used beyond just definition (number of reified edges that have edges beyond reification definition leading in or out of those statements to the number of edges defined for reification)
- portion of reifications without the corresponding sentence materialized
- proportion of non-data to data (number of sentences that include a Statement as S or O / number of sentences that do not have a Statement as S or O)

- proportion of metadata to data (number of sentences that include a Statement as S or O except for the 3 types that define the reifications / number of sentences that do not have a Statement as S or O)

4.6 Ontology

The following may be additional functions we wish to implement to stat the ontologies. Note that by ontologies we mean the original ontologies imported from their source URI's.

- Deepest level of subclassing
- Deepest level of property subclassing
- Amount of multiple inheritance
- Number of distinct ontologies present
- Potential amount of ontological inflation (present/potential information)

4.7 Name Spaces

The namespaces in the kb and the ontology tell us (potentially) a lot about the thinking of the designer. It should be easy to draw out a namespace tree based on decomposing the URIs down to the shorthand prefixes (rdf:) and then information about what is in that namespace, would result in a small forest with the leaves being the various prefixes. Then much of the analysis above is relevant to subsets of the data in the various prefixes. Note that such name space analyses may be useful for practical purposes as it should have little significance conceptually (URI's are meant to be unique identifiers for example and it is irrelevant what their source domain really is).

5 Additional Capabilities

There are additional semantic graph analysis capabilities which we have considered, and not provided exposition about here in order to provide an early scoping document which was focused on rapid analyses. This tool is aimed to achieve the fastest analysis possible for a given dataset. Accommodating extended requirements is considered however the implementation will invoke different code paths for functionality requiring different implementations in order to customize and optimize for each scenario. As an example there is no need to use a full blown graph data structure to implement count and distribution analyses. In fact such generality will hinder the high performance (measured in time and space consumed) of the tool which is a high priority. Many of the following capabilities have been developed in this context, and are under consideration for deployment within this or similar capabilities:

Extant Ontology: A diagrammatic representation of the statistical ontological structure of a semantic graph [6], listing classes connected by their predicates, and labeled by frequency and/or relative frequency of predicate.

n-grams of Link Types: Analysis of existing short linear motifs of predicate types [6].

Indegree, Outdegree Distribution

Histogram of connected component sizes: Considering the instance graph as an undirected graph

Histogram of same-as clique sizes

Distributions of leaves, roots: Identification of both initial and terminal nodes in the instance graph.

Distribution of cycles and strongly connected components: Identification of cyclic structures within the instance graph.

References

[1] The 2010 ISWC Billion Triples Dataset. `http://challenge.semanticweb.org`. Visited January 2011.

[2] The Lehigh University Benchmark. `http://swat.cse.lehigh.edu/projects/lubm`. Visited January 2011.

[3] al Saffar, S., Joslyn, C. A., and Chappell, A. Extant Ontological Scaling and Descriptive Semantics for Semantic Structure Discovery in Large Graph Datasets. In *IEEE/WIC/ACM Int. Conf. on Web Intelligence* (2011).

[4] Goodman, E., Jimenez, E., Mizell, D., al-Saffar, S., Adolf, B., and Haglin, D. High-performance computing applied to semantic web databases. In *The 8th Extended Semantic Web Conference* (2011).

[5] Joslyn, C., Adolf, B., al-Saffar, S., Feo, J., Goodman, E., Haglin, D., Greg Mackey, and Mizell, D. High performance descriptive semantic analysis of semantic graph databases. *1st Workshop on High-Performance Computing for the Semantic Web at ESWC*.

[6] Joslyn, C., Adolf, B., al-Saffar, S., Feo, J., Goodman, E., Haglin1, D., Mackey, G., and Mizell, D. High performance semantic factoring of giga-scale semantic graph databases. In *Semantic Web Challenge* (November 2010), The 9th International Semantic Web Conference.

[7] Joslyn, C., Adolf, B., al-Saffar, S., Feo, J., and Haglin, D. Report on april, 2011, workshop on semantic graph database search patterns. *1st Workshop on High-Performance Computing for the Semantic Web at ESWC*.

[8] Joslyn, C., al Saffar, S., Haglin, D., and Holder, L. Combinatorial Information Theoretical Measurement of the Semantic Significance of Semantic Graph Motifs. In *Proceedings of the Mining Data Semantic Workshop (MDS 2011), SIGKDD 2011* (2011).