SANDIA REPORT

SAND2010-2400

Unlimited Release

Printed August 2010

A Framework for Graph-Based Synthesis, Analysis, and Visualization of HPC Cluster Job Data

James Brandt, Vincent De Sapio, Ann Gentile, Philip Kegelmeyer, Jackson Mayo, Philippe Pébay, Diana Roe, David Thompson, and Matthew Wong

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

A Framework for Graph-Based Synthesis, Analysis, and Visualization of HPC Cluster Job Data

James Brandt*, Vincent De Sapio*, Ann Gentile[°], Philip Kegelmeyer*, Jackson Mayo*,
Philippe Pébay*, Diana Roe[◇], David Thompson*, and Matthew Wong[°]

Sandia National Laboratories

MS *9159 / [°]9152 / [◇]9292

P.O. Box 969, Livermore, CA 94551 U.S.A.

ovis@sandia.gov

Abstract

The monitoring and system analysis of high performance computing (HPC) clusters is of increasing importance to the HPC community. Analysis of HPC job data can be used to characterize system usage and diagnose and examine failure modes and their effects. This analysis is not straightforward, however, due to the complex relationships that exist between jobs. These relationships are based on a number of factors, including shared compute nodes between jobs, proximity of jobs in time, etc. Graph-based techniques represent an approach that is particularly well suited to this problem, and provide an effective technique for discovering important relationships in job queuing and execution data. The efficacy of these techniques is rooted in the use of a semantic graph as a knowledge representation tool. In a semantic graph job data, represented in a combination of numerical and textual forms, can be flexibly processed into edges, with corresponding weights, expressing relationships between jobs, nodes, users, and other relevant entities. This graph-based representation permits formal manipulation by a number of analysis algorithms.

This report presents a methodology and software implementation that leverages semantic graph-based techniques for the system-level monitoring and analysis of HPC clusters based on job queuing and execution data. Ontology development and graph synthesis is discussed with respect to the domain of HPC job data. The framework developed automates the synthesis of graphs from a database of job information. It also provides a front end, enabling visualization of the synthesized graphs. Additionally, an analysis engine is incorporated that provides performance analysis, graph-based clustering, and failure prediction capabilities for HPC systems.

Contents

1	Introduction	7
2	Related Work	9
3	Graph Synthesis	10
3.1	Multi-Partite Semantic Graph	10
3.2	Uni-Partite Semantic Graph	11
3.3	The OVgraphSynthesis Utility	13
4	Visualization and Manipulation	15
4.1	The CARGIOsql Utility	15
5	Analysis and Clustering	17
5.1	Community Detection	17
5.2	Aggregating Multiple Edge Types	17
5.3	Cluster Ensembling	18
5.4	Variation of Information Metric	19
5.5	The OVcluster Utility	21
6	Example Use Case and Results	23
6.1	Graph Synthesis	23
6.2	Graph Visualization and Exploration	24
6.3	Graph Analysis and Clustering	24
7	Additional Job Semantics to Extend the Graph Ontology	30
8	Conclusion	33
	References	34

Appendix

A	Output Description for OVgraphSynthesis	35
B	Output Description for CARGIOsql	39
C	Output Description for OVcluster	44

Figures

1	Process flow showing graph synthesis, visualization, and analysis operations	8
2	Multi-partite graph comprised of three classes of vertices and two types of edges	11
3	A collapse of the multi-partite graph of <i>Figure 2</i> into a uni-partite graph	13
4	The CargioSQL application	16
5	Aggregation of multiple edge types into a graph with a single edge type	18
6	Ensembling of multiple clusterings	19
7	Progressive collapsing of a graph	22
8	A view of the multi-partite graph showing <i>jobs</i> , <i>users</i> , and <i>compute nodes</i>	25
9	An enlarged view of a sub-network of the multi-partite graph	27
10	The uni-partite graph with <i>node match</i> and <i>time match</i> edge types superimposed	28
11	Results of clustering the uni-partite job multigraph	29
12	Multi-partite graph with an <i>idle time</i> vertex class and associated edge types	31
13	Views of the multi-partite graph showing <i>jobs</i> , <i>idle times</i> , <i>users</i> , and <i>compute nodes</i>	32

Tables

1	Fields in the <code>slurm_job_log</code> table	10
2	Sample data in the <code>slurm_job_log</code> table	10
3	Additional attributes for the multi-partite job graph	12
4	Additional attributes for the uni-partite job graph	13
5	Fields in the <code>idlejob</code> table	30
6	Sample data in the <code>idlejob</code> table	30

1 Introduction

The high performance computing (HPC) community is constantly targeting systems with ever-increasing computational power. However, the growing complexities of the architectures of such systems, and the applications running on them, introduces greater challenges to understanding the system interactions which can adversely impact performance. While a precise characterization of system performance may be difficult, if not impossible, to achieve it may be sufficient to know what variables (e.g., those associated with job queuing and execution, memory utilization, etc.) affect system performance and in what manner performance is affected based on the interactions of these variables.

Determination of such relationships can be difficult, particularly for HPC systems where the number of variables per component (e.g., computational nodes) may be in the hundreds, the relative importance of the variables is unknown, and they may not be expressed in a quantitative form. Consequently, we seek a methodology to determine the significant system variables and the interactions between them that affect performance. Of particular interest are relationships among quantities that involve failure conditions, such as failed jobs that share a common resource. Determination of significant failure conditions can be used to enable prediction and/or early detection of failure scenarios that can be used to trigger mitigating actions.

Graph-theoretic approaches would seem to possess particular efficacy in addressing some of these challenges due to their strengths in modeling and analyzing complex networks [1]. Graph-based techniques have become pervasive in the knowledge engineering and machine learning communities. Given an ontological description of a domain, a semantic graph is a powerful knowledge representation tool that permits formal manipulation by a number of graph-based clustering and analysis algorithms. In a semantic graph vertices represent different objects in a domain and edges represent relationships between those objects. The vertices can fall into a single class (uni-partite) or multiple classes (multi-partite) and edges can fall into one or multiple types (multigraph).

Different semantic graphs can be synthesized for a given application depending on the particular ontology employed. Consequently, the choice of an ontology and subsequent graph synthesis should be considered in terms of the intended downstream use, as the particular structure of the semantic graph needs to be amenable to its downstream use. In *Section 3* we will discuss ontology definition and semantic graph synthesis for application to the modeling of HPC cluster job data. This will involve using graphs to represent HPC clusters as complex networks of jobs, users, computational nodes, and additional elements.

Graph visualization provides the user with a customizable display of the synthesized graphs. This is important because the raw structure of a graph provides some basic insight into patterns and relationships in the underlying data, even in the absence of any quantitative analysis. It also provides a visual front-end to facilitate ground-truthing. This involves the assignment of different cluster or community labels to vertices in the graph, based on the user’s knowledge of the context, and the visual insight gleaned from the graph. This ground-truthing provides a way of validating automated clustering routines. In *Section 4* we will discuss the visual exploration of graph networks for HPC cluster applications.

Analysis of the synthesized graph consists of the application of automated clustering routines, combined with validation and performance metrics. Clustering, or community detection, involves the determination of sub-networks or communities within the data based on the relative strengths of edges between vertices. This can be augmented with an ensembling approach that repeats the clustering process many times, varying some parameter in the clustering process each time. This parameter can be the relative importance given to each of the edge types (edge type weighting), or it can be a sub-sampling parameter associated with choosing

a different subset of the nodes in the overall graph each time. A consensus clustering is then obtained based on combining each of the ensemble clusterings. In *Section 5* we will discuss the application of graph-based clustering and community detection to HPC cluster job data, with the objective of providing analysis and prediction capabilities.

Figure 1. Process flow showing graph synthesis, visualization, and analysis operations. The graph synthesis process generates a graph from a database source. The graph can be visualized or directly passed to the analysis process. During visualization, ground truthing can be performed by the user. Analysis on either the ground truthed or raw graph results in a clustered graph which can be viewed again.

Figure 1 depicts the process flow associated with graph synthesis, visualization, and analysis. The following sections will detail the graph synthesis, visualization, and analysis phases with respect to the domain of HPC cluster job queuing and execution.

2 Related Work

There are some existing tools that facilitate data mining and informatic analysis of complex interconnected systems. SPLUNK [7] is a commercial IT search and analysis engine, that provides capabilities to index, search, alert and report on live and historical IT data. It does not, however, possess any significant graph-based representation and analysis capabilities. Additionally, there are a number of general purpose graph theoretic tools, some of which we are leveraging for this work. These include the TITAN Informatics Toolkit [8] and the Multi-threaded Graph Library (MTGL) [4].

3 Graph Synthesis

The SLURM [6] database is the principle data archive used for storing job data. The `slurm_job_log` table consists of the columns of data described in *Table 1*. Sample data is shown in *Table 2*. A unique id is associated with each job, as well as a number of attributes, including user name, job name, job state, start and end time, and compute nodes used.

Table 1. Fields in the `slurm_job_log` table.

name	description	type
id	unique database identifier	int
jobid	unique job identifier	int
username	name of user running job	char
userid	unique user identifier	int
jobname	name of job	char
jobstate	state of the job (completed, failed, etc.)	char
partition	partition job ran on	char
timelimit	time limit for the job	int
starttime	date/time job started	datetime
endtime	date/time job ended	datetime
odelist	list of compute nodes running job	vchar
nodecount	number of compute nodes running job	int

Table 2. Sample data in the `slurm_job_log` table.

id	...	username	jobname	jobstate	starttime	...	odelist	nodecount
1	...	root	hostname	CANCELLED	2008-11-19 09:20:21	...	glory0	1
2	...	mrepper	orterun	FAILED	2008-11-19 09:35:54	...	glory[256-287]	32
3	...	sierra	test	COMPLETED	2008-11-19 09:36:11	...	glory[234, 255]	32
.	.	.	.	.	.	.	.	.

3.1 Multi-Partite Semantic Graph

An intuitive way of synthesizing the job data into a graph is to consider *jobs*, *users*, and *compute nodes* as three elemental classes of vertices. Semantic relationships exist between these vertex classes. For example, we can say that a given *job* “is run by” a given *user*. Additionally, we can say that a given *job* “runs on” a given set of *compute nodes*. This abstraction constitutes a multi-partite ontology graph. This is illustrated in *Figure 2* along with an instantiation.

In addition to the vertex classes and edge types defined for the multi-partite graph any number of attributes can be associated with the vertices and edges based on data stored in the `slurm_job_log`. This data can be useful for visualization of the synthesized graph as well as analysis. These attributes are summarized in *Table 3*.

Figure 2. (Left) Multi-partite ontology graph comprised of three classes of vertices and two types of edges. The vertex classes represent *jobs*, *users*, and *compute nodes*. The edge types represent *run by* and *runs on* relationships among the vertex classes. (Right) A simple instantiation of the ontology graph for three *jobs*, two *users*, and five *compute nodes*. The edge types represent *run by* and *runs on* relationships among the vertices.

3.2 Uni-Partite Semantic Graph

A multi-partite graph is useful for visualizing the relationships between *jobs*, *users*, and *compute nodes*. However, with regard to downstream job clustering processes it may be advantageous to organize all *jobs* into a uni-partite graph. In this way, *jobs* form the only vertex class in the graph, and they can be directly clustered based on a set of edge types.

In order to generate a uni-partite graph of the form of *Figure 3*, the multi-partite graph of *Figure 2* needs to be collapsed so that vertices of class *users* and *compute nodes* are removed. *Jobs* that previously shared edges with *users* and *compute nodes* will now share edges with each other.

The edge types *user match* and *node match* will characterize the affinity between any two *jobs* based on the *users* and *compute nodes* that they share. For *user match*, we let the term U_k denote the user name for *job* k (e.g., $U_k = \text{"root"}$). The adjacency matrix, $\mathbf{A}$, which stores the strength (or weight) of the edge between any two jobs can then be defined as,

$$A_{ij} \triangleq \begin{cases} 0 & \text{if } U_i \neq U_j \\ 1 & \text{if } U_i = U_j \end{cases} \quad (1)$$

Table 3. Additional attributes for the multi-partite job graph.

name	description	type
vid	unique vertex identifier	int
vname	unique vertex name	varchar
nodename	name of the compute node (if vertex is a compute node)	varchar
vdisplayname	name used for labeling the vertex (in downstream visualization)	varchar
vtype	identifier for vertex type (0 - <i>job</i> , 1 - <i>user</i> , 2 - <i>compute node</i>)	int
vtypename	name of vertex type	varchar

For *node match*, we let the set N_k denote the compute node list for *job k* (e.g., $N_k = \text{glory}[1-11,15]$). The adjacency matrix can then be defined as,

$$A_{ij} \triangleq \frac{|N_i \cap N_j|}{|N_i \cup N_j|} \quad (2)$$

Additional Edge Types

We can define additional edge types based on other job attributes. If we let the term J_k denote the job name for *job k* (e.g., $J_k = \text{“sierra”}$), the adjacency matrix for the edge type *job name match* can be defined as,

$$A_{ij} = \begin{cases} 0 & \text{if } J_i \neq J_j \\ 1 & \text{if } J_i = J_j \end{cases} \quad (3)$$

If we let the term S_k denote the job state for *job k* (e.g., $S_k = \text{“COMPLETED”}$), the adjacency matrix for the edge type *job state match* can be defined as,

$$A_{ij} = \begin{cases} 0 & \text{if } S_i \neq S_j \\ 1 & \text{if } S_i = S_j \end{cases} \quad (4)$$

If we let the the interval T_k denote the time interval for *job k* (e.g., $T_k = [t_o, t_f]$), where t_o is the start time and t_f is the end time), the adjacency matrix for the edge type *time match* can be defined as,

$$A_{ij} = \frac{|T_i \cap T_j|}{|T_i \cup T_j|} \quad (5)$$

Finally, if we let the term P_k denote the partition for *job k* (e.g., $P_k = \text{“pbatch”}$), the adjacency matrix for the edge type *partition match* can be defined as,

$$A_{ij} = \begin{cases} 0 & \text{if } P_i \neq P_j \\ 1 & \text{if } P_i = P_j \end{cases} \quad (6)$$

In addition to the edge types defined for the unii-partite graph two additional vertex attributes will be defined apart from the native `slurm_job_log` table fields. These are summarized in *Table 4*.

Figure 3. A collapse of the multi-partite graph of *Figure 2* into a uni-partite graph. The solitary vertex class represents *jobs*. The *run by* and *runs on* relationships between *jobs* and *users*, and *jobs* and *compute nodes*, have been replaced by *user match* and *node match* relationships between *jobs*. Additional edge types can be defined to represent *job name match*, *job state match*, *time match*, and *partition match* relationships.

Table 4. Additional attributes for the uni-partite job graph.

name	description	type
vid	unique vertex identifier	int
vname	unique vertex name	varchar

3.3 The OVgraphSynthesis Utility

Graph synthesis can be performed based on the ontologies described in *Sections 3.1* and *3.2*. To this end we have developed the OVgraphSynthesis command line utility to automate this process. It is executed in the following form,

```
$ ./ov_graph_synthesis --db <database> --sql <sql> --verbose <true|false>
```

where <database> is the name of a database file (SQLite is currently supported). The sql statement is enclosed in quotes. For example, the command,

```
$ ./ov_graph_synthesis --db slurm.db --sql "select * from slurm_job_log
where id > 14400 and id < 14501" --verbose true
```

will select rows from the slurm_job_log table of database, slurm.db, corresponding to *id* 14401 through 14500. The graphs will be generated from this data and stored in separate tables in slurm.db. Another example is,

```
$ ./ov_graph_synthesis --db slurm.db --sql "select * from slurm_job_log  
where username = 'mrepper'" --verbose true
```

This will select rows corresponding to user name *'mrepper'* and generate the graphs from this data. In verbose mode *OVgraphSynthesis* outputs vertex and edge data to the terminal, first for the multi-partite graph, and then for the uni-partite graph. *Appendix A* provides a description of the output generated by *OVgraphSynthesis* and *Section 6* demonstrates the use of *OVgraphSynthesis* for a use case involving actual HPC job data.

4 Visualization and Manipulation

Semantic graphs have a natural visual context. As with all graphs, objects within the domain are depicted as vertices and relationships among the objects are depicted as edges (or arcs) connecting the vertices. Semantic graphs offer additional richness to the visual depiction, since there are multiple edge types that need to be visually distinguished. In the multi-partite case there are also multiples classes of vertices which need to be visually distinguished.

In addition to a visual display of the generated graphs there is a need for user manipulation of the graphs, particularly with regard to ground-truthing. As described earlier, ground-truthing involves the assignment of cluster labels to graph vertices. This leverages the user’s knowledge of the data, as well as insight into patterns and relationships drawn from the visual depiction of the graph.

4.1 The CARGIOsql Utility

Using the VTK [9] based TITAN Informatics Toolkit [8] the CargioSql application has been developed as a general tool for the visualization of attributed relational graphs. More recently, it has been specifically catered to HPC cluster job data. The user selects a database file (SQLite is currently supported) to load. For our purposes we will load any file processed by the OVgraphSynthesis tool. The interface for CargioSql is shown in *Figure 4*. *Appendix B* provides a description of the graphical output generated by CARGIOsql and *Section 6* demonstrates the use of CARGIOsql for a use case involving actual HPC job data.

Figure 4. The CargioSQL application. A database containing the raw SLURM job log table, and the graph tables generated by the OVgraphSynthesis utility, is opened.

5 Analysis and Clustering

The previous section addressed the use of visualization techniques as a means of gaining insight into the underlying data relationships encoded in a semantic graph. This provides only a qualitative perspective on relevant relationships in the data. For a more rigorous quantitative assessment of the patterns present in the data, analysis and clustering algorithms must be used. The basic goal of a clustering algorithm is to generate separate clusters, or communities, containing individual data objects sharing an affinity.

For our purposes we will be concerned with the clustering of *job* vertices. Before proceeding It will be useful to formally define a clustering of vertices. First we consider the set, V , of n vertices.

$$V = \{v_1, v_2, \dots, v_n\} \quad (7)$$

We now consider two clusterings, C and C' , each of which is a collection of subsets of V . More formally, C and C' each constitute a family of sets over V . Let $\mathbf{k}$ be the array of p cluster labels associated with C , and let $\mathbf{k}'$ be the array of r cluster labels associated with C' . Further, let L_{k_i} be the set of vertices with cluster assignment k_i in clustering C , and let $L'_{k'_i}$ be the set of vertices with cluster assignment k'_i in clustering C' . Then,

$$C \triangleq \{L_{k_1}, \dots, L_{k_p}\} \quad (8)$$

$$C' \triangleq \{L'_{k'_1}, \dots, L'_{k'_r}\} \quad (9)$$

where $L_{k_i}, L'_{k'_i} \subseteq V$. These definitions will be referred to in *Section 5.4*.

5.1 Community Detection

The method of Girvan and Newman [2][5] has recently achieved prominence in the area of graph-based community detection. In this method a quantity termed betweenness is computed for all edges in the graph. Betweenness for a particular edge reflects how many shortest paths between pairs of nodes run along that edge. The edge with the largest betweenness is removed and the betweenness values are recalculated on all edges affected by the removal. This process is repeated based on some criterion. When finished, each connected subgraph corresponds to a community.

For our purposes we will apply community detection to the uni-partite semantic *job* graph. Although this is comprised of a single vertex class it possesses multiple edge types. The community detection algorithm addresses graphs with a single edge type. The next section discusses the aggregation of multiple edge types.

5.2 Aggregating Multiple Edge Types

Given a multigraph with n_E edge types there are n_E adjacency matrices, $\{\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_{n_E}\}$. We can perform a weighted aggregation of these adjacency matrices. This yields the aggregate adjacency matrix,

$$\mathbf{A}^* = \sum_{j=1}^{n_E} w_j \mathbf{A}_j \quad (10)$$

where w_j is a weight associated with the relative importance of the j th edge type. *Figure 5* illustrates this aggregation process. Often one may not have an a priori knowledge of the appropriate weighting for a

given edge type. Consequently, either an optimization approach must be employed to find an optimal set of weights, or, an ensembling approach must be employed to yield a consensus clustering from many separate clusterings, each using different weights. The ensembling approach will be discussed in *Section 5.3*.

Figure 5. Aggregation of multiple edge types into a graph with a single edge type. An aggregate adjacency matrix, A_i^* , associated with the i th clustering is generated from the individual edge type adjacency matrices, $\{A_1, \dots, A_{n_E}\}$ and weight vector w_i . The weight w_{i_j} is associated with the relative importance of the j th edge type in the i th clustering of the ensemble. *Figure 6* describes the subsequent ensembling process.

5.3 Cluster Ensembling

The process of ensembling together multiple clusterings first involves computing a similarity matrix for a given clustering. Given a clustering C , we let $\mathbf{c}$ be the array of n cluster assignments, associating a cluster label with each of the n vertices. The similarity matrix, $\mathbf{S}$, is then defined as,

$$S_{ij} \triangleq \begin{cases} 0 & \text{if } c_i \neq c_j \\ 1 & \text{if } c_i = c_j \end{cases} \quad (11)$$

Given a set of clusterings, $\{C_1, C_2, \dots, C_{n_C}\}$, and associated similarity matrices, $\{S_1, S_2, \dots, S_{n_C}\}$, we can define a consensus function on the similarity matrices. A simple averaging consensus function would

be,

$$\mathbf{S}^* = \sum_{i=1}^{n_C} \mathbf{S}_i / n_C \quad (12)$$

This ensembling method can be implemented with the community detection algorithm. An aggregate adjacency matrix can be computed for a given set of edge weights. These weights can be randomly generated and, for each set, an aggregate adjacency matrix can be computed. Community detection can be run on these aggregate adjacency matrices, yielding a separate clustering for each. Similarity matrices can be associated with each clustering and these can then be used to obtain a consensus similarity matrix, $\mathbf{S}^*$, as described above. Finally, community detection can be run on this consensus similarity matrix, yielding a consensus clustering, C^* . This ensembling procedure is illustrated in *Figure 6* and detailed in *Algorithm 1*.

Figure 6. Ensembling of multiple clusterings. A consensus function, $\mathbf{S}^*$, is defined on the similarity matrices, $\{\mathbf{S}_1, \dots, \mathbf{S}_{n_C}\}$. Community detection is run on the graph associated with the consensus similarity matrix, yielding a consensus clustering, C^* .

5.4 Variation of Information Metric

The variation of information (VOI) metric [3] measures the amount of information lost and gained in transitioning between two clusterings. To compute the metric we first denote n_{k_i} as the number of vertices with cluster assignment k_i in clustering C , and denote $n_{k'_i}$ as the number of vertices with cluster assignment k'_i in clustering C' . Finally, let $n_{k_i k'_j}$ be the number of vertices with both cluster assignment k_i in clustering C and

Algorithm 1 Ensembling multiple clusterings generated from different edge type weightings

```
1:  $\mathbf{S}_\Sigma = \mathbf{0}$  {initialization}
2: load multigraph  $\{\mathbf{A}_1, \dots, \mathbf{A}_{n_E}\}$ 
   cluster ensembling loop
3: for  $i = 1$  to  $n_C$  do
4:   generate  $\mathbf{w}_i$  {randomization}
5:    $\mathbf{A}_i^* = \mathbf{0}$  {initialization}
   graph aggregation loop
6:   for  $j = 1$  to  $n_E$  do
7:      $\mathbf{A}_i^* = \mathbf{A}_i^* + w_{ij}\mathbf{A}_j$ 
8:   end for
9:   generate  $C_i$  {community detection on  $\mathbf{A}_i^*$ }
10:  generate  $\mathbf{S}_i$   $\{C_i \longrightarrow \mathbf{S}_i\}$ 
11:   $\mathbf{S}_\Sigma = \mathbf{S}_\Sigma + \mathbf{S}_i$ 
12: end for
13:  $\mathbf{S}^* = \mathbf{S}_\Sigma / n_C$ 
14: generate  $C^*$  {community detection on  $\mathbf{S}^*$ }
```

cluster assignment k'_j in clustering C' . That is,

$$n_{k_i} = |L_{k_i}| \quad (13)$$

$$n_{k'_i} = |L'_{k'_i}| \quad (14)$$

$$n_{k_i k'_j} = |L_{k_i} \cap L'_{k'_j}| \quad (15)$$

We then define the probability, $P(k_i)$, of a vertex being assigned to k_i in clustering C and the probability, $P(k'_i)$, of a vertex being assigned to k'_i in clustering C' . Finally, we define the probability, $P(k_i, k'_j)$, of a vertex being assigned to k_i in clustering C and to k'_j in clustering C' . So,

$$P(k_i) = \frac{n_{k_i}}{n} \quad (16)$$

$$P(k'_i) = \frac{n_{k'_i}}{n} \quad (17)$$

$$P(k_i, k'_j) = \frac{n_{k_i k'_j}}{n} \quad (18)$$

The entropy of information associated with C and C' is then,

$$H(C) = - \sum_{i=1}^p P(k_i) \log P(k_i) \quad (19)$$

$$H(C') = - \sum_{i=1}^r P(k'_i) \log P(k'_i) \quad (20)$$

and the mutual information shared by C and C' is,

$$I(C, C') = \sum_{i=1}^p \sum_{j=1}^r P(k_i, k'_j) \log \frac{P(k_i, k'_j)}{P(k_i)P(k'_j)} \quad (21)$$

Finally, the variation of information between C and C' is,

$$d(C, C') = H(C) + H(C') - 2I(C, C') \quad (22)$$

5.5 The OVcluster Utility

The Multi-threaded Graph Library (MTGL) [4] incorporates algorithms for a weighted version of the Newman method. We have developed the `OVcluster` command line utility using MTGL in conjunction with the TITAN toolkit. `OVcluster` is executed in the following form,

```
$ ./ov_cluster --db <database> --edge <edge type 1> <weight 1> ...
<edge type n> <weight n> --cluster <cluster> --truth <truth cluster>
--sim-matrix <similarity matrix> --mode <init|incr> --verbose <true|false>
--collapse <collapse limit>
```

The `--edge` flag specifies the name of an edge type (edge table in the database) to be included in the graph and the corresponding edge type weight to be applied. Any number of edges can be specified. The `--cluster` flag specifies the database cluster table column name in which to store the cluster assignments. The `--truth` flag specifies the database cluster table column name in which the ground truth clustering is stored. This is used in computing the variation of information between the generated clustering and ground truth. The `--sim-matrix` flag specifies the database table name in which to store the similarity matrix. Finally, the `--mode` flag specifies whether to run in initialization or incremental mode.

The `--collapse` flag specifies the collapse limit of the graph. The user can specify that the graph be coarsened down to a simpler graph. This involves randomly choosing vertices to which neighboring vertices are collapsed (see *Figure 7*). This process repeats until the number of vertices remaining in the graph is less than or equal to the specified collapse limit. Community detection is performed on this coarsened graph. The clustering is subsequently expanded so that the collapsed vertices receive the cluster assignment of their respective parent vertex (vertex into which they collapsed).

`OVcluster` can be run repeatedly on the same input database file to generate multiple clusterings associated with a cluster ensemble. After each run a consensus similarity matrix and consensus clustering is stored in the database. This reflects the consensus of all the clusterings generated up to that point. The `--mode` flag would be set to *init* on the first ensemble run and *incr* on subsequent runs. After the final run a consensus clustering associated with the ensemble of all previous clusterings would be stored. *Appendix C* provides a description of the output generated by `OVcluster` and *Section 6* demonstrates the use of `OVcluster` for a use case involving actual HPC job data.

Figure 7. Progressive collapsing of a graph. When vertex 1 is chosen, vertices 2, 3, and 4 collapse to it. Next, when vertex 7 is chosen, vertex 5 collapses to it. After clustering is performed on the collapsed graph the collapsed vertices receive the cluster assignment of their parent vertex.

6 Example Use Case and Results

The following example is based on SLURM data associated with Sandia's Glory cluster, a 288-node, 4068-core Opteron cluster with an Infiniband interconnect.

6.1 Graph Synthesis

We would like to extract a specific set of jobs from the `slurm_job_log` table of the database. This can be done with the following `OVgraphSynthesis` command,

```
$ ./ov_graph_synthesis --db slurm.db --sql "select * from slurm_job_log
where jobid > 12700 and jobid < 12800" --verbose true
```

Given the jobs specified at the command line the vertices (*jobs*, *users*, and *nodes*) of the multi-partite graph are generated. Example output consists of,

```
-----
Generating node vertices....
-----
vid 110 -> nodename 'glory10'
vid 111 -> nodename 'glory103'
...
vid 212 -> nodename 'glory96'
-----
Number of node vertices = 103
-----
```

Next, the edges are generated. Example output consists of,

```
-----
Generating 'runs_on' edges....
-----
eid 99 -> job 16143 is run on node 'glory153'
eid 100 -> job 16143 is run on node 'glory154'
...
eid 516 -> job 17818 is run on node 'glory234'
-----
Number of 'runs_on' edges = 418
-----
```

Following completion of the multi-partite graph the uni-partite graph is generated. This consists of the single vertex class, *jobs*. The edges are then generated. Example output consists of,

```

-----
Generating 'time_match' edges....
-----
e4408 -> job 16143, 16771 has time_match = 0.00550334
e4409 -> job 16143, 17012 has time_match = 0.00536969
...
e5029 -> job 17817, 17818 has time_match = 0.999737
-----
Number of 'time_match' edges = 622
-----

```

6.2 Graph Visualization and Exploration

We can now load the database file resulting from the execution of *Section 6.1* into the CargioSql application. If we specify the multi-partite node table as our vertex set and the *run by* and *runs on* edge types as our edge set, CargioSql will generate the visualization depicted in *Figure 8*. Using the pull-down menus on the bottom right we are free to label the vertices and edges by any attribute stored in the SLURM database or defined by the user. The vertices have been colored by *vertex type* to distinguish between *jobs*, *users*, and *compute nodes*. We can navigate and zoom into any region of the graph to explore sub-networks (see *Figure 9*).

If we specify the uni-partite node table as our vertex set and the *node match* and *time match* edge types as our edge set, CargioSQL will generate the visualization depicted in *Figure 10*. In this case all of our vertices represent *jobs*. Again we are free to label the vertices and edges by any defined attribute. From *Figure 10* we can qualitatively glean a correlation between the community structure associated with this edge set and the ultimate fate of the job. This will be quantitatively explored.

6.3 Graph Analysis and Clustering

We can now cluster the uni-partite graph generated during the graph synthesis phase. Ground truth will be defined by the job state (“FAILED” = 0, “CANCELLED” = 1, “TIMEOUT” = 2, “COMPLETED” = 3, other = -1). This will allow us to evaluate the quality of the clustering relative to known truth values. Given the output (to *slurm.db*) produced by following OVgraphSynthesis command,

```
$ ./ov_graph_synthesis --db slurm.db --sql "select * from slurm_job_log
where jobid > 12700 and jobid < 12800" --verbose true
```

We can execute the following OVcluster command,

```
$ ./ov_cluster --db slurm.db --edge node_match .75 --edge time_match .25
--cluster cluster_comm --truth cluster_gold --sim-matrix simmat --mode init
--verbose true --collapse 50
```

where we are choosing to aggregate the *node match* and *time match* edge types of the multigraph with edge type weights of 0.75 and 0.25 respectively. The aggregate graph is generated,

Figure 8. A view of the multi-partite graph showing *jobs* (blue vertices), *users* (green vertices), and *compute nodes* (red vertices). Vertices are labeled by corresponding name (job name, user name, or compute node name). The *run by* edge type, connecting *jobs* to *users*, and the *runs on* edge type, connecting *jobs* to *compute nodes*, are shown.

Generating aggregate graph....

source	target	weight	eid	etype
v0	v87	0.3	e0	-1
v1	v2	0.0779	e1	-1
...				
v98	v99	0.2499	e876	-1

Number of edges in aggregate graph = 877

The graph is then collapsed,

```
-----  
Collapsing aggregate graph...  
-----  
Number of collapses to get to target = 2  
Number of vertices in original graph = 100  
Number of vertices in collapsed graph = 50  
Number of collapsed vertices = 50  
Number of edges in collapsed graph = 393  
-----
```

Clustering is then performed and statistics and metrics are reported,

```
-----  
5 gold clusters, and 3 community detection clusters.  
Gold cluster distribution, 1 -> 0.16, 3 -> 0.55, 0 -> 0.14, 2 -> 0.1 -1 -> 0.05.  
Random clustering (using gold dist.), 3 -> 0.52, 0 -> 0.15, 1 -> 0.2, 2 -> 0.08,  
-1 -> 0.05.  
VOI metric relative to truth = 1.65389.  
VOI metric for random cluster (with known distribution) relative to truth = 2.37941.  
-----
```

The results of the job clustering are shown in *Figure 11*, with the jobs labeled by their actual job state. The statistics indicate that the variation of information of the generated clustering relative to the truth data (Gold cluster using job state as cluster labels) is 1.65389. This is substantially better than that of a random clustering relative to truth whose VOI is 2.37941. Moreover, the random clustering has the benefit of knowing the ground truth cluster distribution, whereas, the generated clustering only has access to edge weights associated with *node match* and *time match*.

In this example the `OVcluster` utility was run a single time with an arbitrary set of edge type weights (0.75 and 0.25), however, we could also run `OVcluster` multiple times with randomized edge type weights and ensemble the clusterings together in an attempt to improve our clustering accuracy.

Figure 10. A view of the uni-partite graph (all vertices are *jobs* and color indicates vertex degree). The *node match* and *time match* edge types are superimposed. (Top) Vertices are labeled by job name. (Bottom) Vertices are labeled by job state. We can see a qualitative correlation between the community structure in this graph and the job state.

Figure 11. Results of clustering the uni-partite job multigraph. (Top) The original graph defined by *node match* and *time match* edge types. Structures in the graph are qualitatively highlighted. (Bottom) Three separate communities are identified by the community detection algorithm. Vertices are labeled by job state to convey the correlation between the communities and the job state.

7 Additional Job Semantics to Extend the Graph Ontology

In addition to the job graph semantics described in *Section 3* we can specify additional semantics in order to extend the job graph ontology. As an example we will present a use case involving job idle times. We consider an additional table of data to augment the `slurm_job_log` table. This is the `idlejob` table. *Table 5* describes some of the relevant columns of data associated with the `idlejob` table. Sample data is shown in *Table 6*.

Table 5. Fields in the `idlejob` table.

name	description	type
id	unique database identifier for the an individual idle time	int
node	id of node associated with the idle time	int
ScaledFirstUsage	scaled active memory used on the node at that idle time	double
jobid	id of job associated with the idle time	int
FunctionalIdle	id of functional idle associated with the job and node	int
ResponsibleNode	id of the responsible node	int

Table 6. Sample data in the `slurm_job_log` table.

id	node	...	ScaledFirstUsage	...	jobid	...	FunctionalIdle	ResponsibleNode
7263	234	...	0.0159	...	12755	...	7263	-1
7264	234	...	-1	...	12756	...	7263	-1
7265	234	...	-1	...	12757	...	7263	228
.	.	.	.	.	.	.	.	.

The data in this table is constructed as follows. For every job on every node the nearest idle time preceding that job on that node is found and assigned a unique id. If there is a value of memory usage associated with that idle time it is reported and the idle time is recorded as the functional idle associated with that job and node. It may be that due to the data collection frequency, there is no data point for the idle time just preceding a given job. In this case a -1 is reported for active memory usage, and the nearest preceding idle time with an active memory value is used as the functional idle associated with that job and node. For jobs that have functional idles on all nodes, the node associated with the functional idle with the highest scaled memory usage is assigned as the responsible node.

We can consider these relationships in a multi-partite graph by introducing an additional vertex class. In addition to *jobs*, *users*, and *compute nodes* we introduce *idle time*. Additional semantic relationships (edges) exist between these vertex classes. For example, we can say that a given *job* “has a” given *idle time* as its idle time, or as its functional idle. We can also say that a given *job* “has a” given *compute node* as its responsible node. Additionally, we can say that a given *idle time* “is on” a given *compute node*. This abstraction is illustrated in *Figure 12* along with an instantiation.

The uni-partite graph derived from the multi-partite graph of *Figure 12* will include additional edge types between *jobs*. It is noted that *jobs* share no *idle times* but they do share functional idles and responsible nodes. If we let the set F_k denote the functional idle list for *job k* (e.g., $F_k = 1, 2, 3, 5$). The adjacency matrix

Figure 12. (Left) Multi-partite ontology graph comprised of an additional *idle time* vertex class and associated edge types. The additional edge types represent *has idle* and *is on* relationships between *jobs* and *idle times*, and between *idle times* and *compute node* respectively. (Right) A simple instantiation of the ontology graph for three *jobs*, two *users*, five *compute nodes*, and seven *idle times*.

for the edge type *functional idle match* can be defined as,

$$A_{ij} \triangleq \frac{|F_i \cap F_j|}{|F_i \cup F_j|} \quad (23)$$

If we let the term N_k denote the responsible node for *job k* (e.g., $N_k = \text{"glory1"}$), the adjacency matrix for the edge type *responsible node match* can be defined as,

$$A_{ij} = \begin{cases} 0 & \text{if } N_i \neq N_j \\ 1 & \text{if } N_i = N_j \end{cases} \quad (24)$$

Additional edges can be defined to encode information such as scaled memory usage and other quantities. *Figure 13* displays the mutli-partite graph associated with this extended semantics.

8 Conclusion

We have presented a methodology and implementation for graph-based representation, visualization, and analysis of HPC job queuing and execution data. Our work exploits the efficacy of semantic graphs as a knowledge representation tool. Using graph-based clustering and community detection algorithms we have demonstrated the utility of this approach in finding sub-networks, or communities, within HPC job data. This can be leveraged for system-level analysis and failure prediction.

An example was presented using actual HPC job data. The example detailed a specific set of jobs sharing a set of compute nodes. Two edges, *node match* and *time match*, were shown to possess a community structure that was related to the ultimate fate of the jobs. This serves as a simple example of the potential utility of our graph theoretic framework within the domain of HPC failure prediction. We intend to perform more rigorous investigations of the community structure of HPC systems based on additional edge types, data attributes and clustering approaches, including cluster ensembling. Based on the flexibility of our framework in specifying the HPC graph semantics, as well as synthesizing, visualizing, and analyzing the graphs we consider it to be a potentially powerful platform for failure prediction, mitigation, and performance improvement in HPC clusters.

References

- [1] S. Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, 2010.
- [2] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.
- [3] M. Meilă. Comparing clusterings by the variation of information. *Lecture Notes in Computer Science*, pages 173–187, 2004.
- [4] MTGL multi-threaded graph library. <https://software.sandia.gov/trac/mtgl>.
- [5] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113 1–15, 2004.
- [6] SLURM. <https://computing.llnl.gov/linux/slurm>.
- [7] SPLUNK. <http://www.splunk.com>.
- [8] TITAN toolkit. <http://titan.sandia.gov/>.
- [9] VTK visualization toolkit. <http://www.vtk.org/>.

A Output Description for OVgraphSynthesis

The output to the command,

```
$ ./ov_graph_synthesis --db slurm.db --sql "select * from slurm_job_log
where id > 14400 and id < 14501" --verbose true
```

begins by echoing the input parameters,

```
database = slurm.db
verbose mode = true
sql statement = 'select * from slurm_job_log where id > 14400 and id < 14501'
```

Next, the multi-partite graph is generated. First the vertices are defined,

```
-----
-----
Generating multi-partite graph....
-----
-----

-----
Generating job vertices....
-----
vid 0 -> id 14401
vid 1 -> id 14402
...
vid 99 -> id 14500
-----
Number of job vertices = 100
-----

-----
Generating user vertices....
-----
vid 100 -> username 'dlvilla'
vid 101 -> username 'kanouff'
...
vid 113 -> username 'sierra'
-----
Number of user vertices = 14
-----
```

```

-----
Generating node vertices....
-----
vid 114 -> nodename 'glory10'
vid 115 -> nodename 'glory11'
...
vid 181 -> nodename 'glory68'
-----
Number of node vertices = 68
-----

```

The edges are then generated,

```

-----
Generating 'run_by' edges....
-----
eid 0 -> job 14401 is run by user 'sierra'
eid 1 -> job 14402 is run by user 'sierra'
...
eid 99 -> job 14500 is run by user 'sgrange'
-----
Number of 'run_by' edges = 100
-----

```

```

-----
Generating 'runs_on' edges....
-----
eid 100 -> job 14401 is run on node 'glory198'
eid 101 -> job 14402 is run on node 'glory177'
...
eid 479 -> job 14500 is run on node 'glory189'
-----
Number of 'runs_on' edges = 380
-----

```

Following completion of the multi-partite graph the uni-partite graph is generated. This consists of one vertex class.

```
-----  
-----  
Generating uni-partite graph....  
-----  
-----
```

```
-----  
Generating vertices....  
-----
```

```
vid 0 -> id 14401  
vid 1 -> id 14402  
...  
vid 99 -> id 14500  
-----
```

```
Number of vertices = 100  
-----
```

The edges are then generated,

```
-----  
Generating 'user_match' edges....  
-----
```

```
eid 1717 -> job 14401, job 14402 has user_match = 1  
eid 1718 -> job 14401, job 14403 has user_match = 1  
...  
eid 3438 -> job 14499, job 14500 has user_match = 1  
-----
```

```
Number of 'user_match' edges = 1722  
-----
```

```
-----  
Generating 'node_match' edges....  
-----
```

```
eid 3439 -> job 14401, job 14413 has node_match = 1  
eid 3440 -> job 14401, job 14418 has node_match = 1  
...  
eid 4637 -> job 14499, job 14500 has node_match = 1  
-----
```

```
Number of node_match' edges = 1199  
-----
```

```
...
```

```
-----  
Generating 'time_match' edges....  
-----  
eid 8329 -> job 14401, job 14402 has time_match = 0.6  
eid 8330 -> job 14401, job 14403 has time_match = 0.3125  
...  
eid 8555 -> job 14498, job 14499 has time_match = 0.00179856  
-----  
Number of 'time_match' edges = 227  
-----  
  
...
```

B Output Description for CARGIOsql

In demonstrating the execution of CARGIOsql we first run `OVgraphSynthesis` on the original SLURM database. As an example we execute the following command,

```
$ ./ov_graph_synthesis --db slurm.db --verbose true --sql "select * from
slurm_job_log where id > 14400 and id < 14501"
```

CargioSQL will display the database table information depicted in *Figure B.1*.

Table Name	ID	Fields
slurm_job_log	24375	vid [int]
nodeTableMultiPartite	182	vname [string], id [int], jobid [int], username [string], userid [int], jobname [string], jobstate [string], partition [string], timelimit [int], starttime [string], endtime [string], absstart [int], absend [int], nodelist [string], nodecount [int], nodename [string], vdisplayname [string], vtype [int], vtypename [string]
run_by	100	source [int], target [int], weight [double], eid [int]
runs_on	380	source [int], target [int], weight [double], eid [int]
nodeTableUniPartite	100	vid [int], vname [string], id [int], jobid [int], username [string], userid [int], jobname [string], jobstate [string], partition [string], timelimit [int], starttime [string], endtime [string], absstart [int], absend [int], nodelist [string], nodecount [int]
clusterTable	100	cid [int], cluster_gold [string], cluster_user [string]
jname_match	1717	source [int], target [int], weight [double], eid [int], etype [int]
user_match	1722	
node_match	1199	
jstate_match	3691	
time_match	227	
partition_match	2774	

Figure B.1. Expanded view of the graph tables generated by `OVgraphSynthesis` showing the fields for the multi-partite and uni-partite vertex tables, as well as the associated edge and cluster tables.

If we specify the multi-partite node table as our vertex set and either the *run by* or *runs on* edge type as our edge set, CargioSQL will generate the visualization depicted in *Figure B.2*. We are free to label the vertices and edges by any vertex or edge attribute respectively. In *Figure B.2* the vertices have been colored by *vertex type* to distinguish between *jobs*, *users*, and *compute nodes*. We can zoom into a sub-network as shown in *Figure B.3*.

If we specify the uni-partite node table as our vertex set and the *node match* edge type as our edge set, CargioSQL will generate the visualization depicted in *Figure B.4*. In this case all of our vertices represent *jobs*. Again we are free to label the vertices and edges by any vertex or edge attribute respectively. In *Figure B.5* the *time match* edge type is shown.

Figure B.2. A view of a multi-partite graph showing *jobs* (blue vertices), *users* (green vertices), and *compute nodes* (red vertices). Vertices are labeled by job name. (Top) The *run by* edge type is shown, connecting *jobs* to *users*. (Bottom) The *runs on* edge type is shown, connecting *jobs* to *compute nodes*.

Figure B.3. An enlarged view of a sub-network of the *runs on* multi-partite graph. (Top) Vertices are labeled by job name. (Bottom) Vertices are labeled by job state.

C Output Description for OVcluster

In demonstrating the execution of OVcluster we first run OVgraphSynthesis on the original SLURM database. As an example we execute the following command,

```
$ ./ov_graph_synthesis --db slurm.db --verbose true --sql "select * from slurm_job_log
where id > 9400 and id < 9701"
```

We then define a cluster named cluster_gold. We assign cluster labels to cluster_gold based on the job state. Now we can execute OVcluster. The output to the command,

```
$ ./ov_cluster --db slurm.db --edge node_match .75 --edge time_match .25
--cluster cluster_comm --truth cluster_gold --sim-matrix simmat --mode init
--verbose true --collapse 50
```

begins by echoing the input parameters,

```
database = test.db
edge type = node_match (.75)
edge type = time_match (.25)
node collapse limit = 50
cluster label = cluster_comm
truth cluster label = cluster_gold
similarity matrix = simmat
operational mode = initialize
verbose mode = true
```

Next, the aggregate graph is generated,

```
-----
Generating aggregate graph....
+-----+-----+-----+-----+-----+
| source | target | weight | eid | etype |
+-----+-----+-----+-----+-----+
| 0       | 1       | 0.25   | 0   | -1    |
| 0       | 2       | 0.25   | 1   | -1    |
...
| 298     | 299     | 0.744  | 20991 | -1    |
+-----+-----+-----+-----+-----+
Number of edges in aggregate graph = 20992
-----
```

The graph is then collapsed,

```
-----  
Collapsing aggregate graph...  
-----  
v5 <- {v0, v1, v2, v3, v4, v6, v7, v8, v9, v10, v11, v12, v13, v14}  
v15 <- NULL  
...  
v193 <- {v34, v31, v17, v27, v28, v29, v30, v26, v32, v47, v48, v33, ...}  
...  
v205 <- NULL  
-----  
Number of collapses to get to target = 10  
Number of vertices in original graph = 300  
Number of vertices in collapsed graph = 38  
Number of collapsed vertices = 262  
Number of edges in collapsed graph = 2617  
-----
```

Clustering is then performed,

```
-----  
Clustering of expanded graph...  
-----  
v0 -> 5 | 0  
v1 -> 5 | 0  
...  
v299 -> 193 | 0  
-----
```

The ground truth clustering is then reported. This is the cluster assignment performed based on the job state (“FAILED” = 0, “CANCELLED” = 1, “TIMEOUT” = 2, ”COMPLETED” = 3, other = -1).

```
-----  
Gold clustering...  
-----  
v0 -> 1  
v1 -> 1  
...  
v299 -> 3  
-----
```

A random clustering is generated,

```
-----  
Random clustering (using gold distribution)...  
-----  
v0 -> 3  
v1 -> 1  
...  
v299 -> 1  
-----
```

Statistics and metrics are reported,

```
-----  
5 gold clusters, and 5 community detection clusters.  
Gold cluster distribution, 1 -> 0.173, 3 -> 0.713, 0 -> 0.05, 2 -> 0.06, -1 -> 0.003.  
Random clustering (using gold dist.), 3 -> 0.767, 1 -> 0.133, 0 -> 0.06, 2 -> 0.04.  
VOI metric relative to truth = 0.91701.  
VOI metric for random cluster relative to truth = 1.60103.  
-----
```

The cluster data and similarity table are written to the database,

```
-----  
Writing cluster data to the database...  
-----  
  
-----  
Writing similarity table to the database...  
-----  
Number of edges in similarity graph = 39726  
-----
```

The ensemble graph associated with the consensus similarity matrix is then clustered,

```
-----  
Clustering of ensemble graph...  
-----  
v0 -> 14 | 0  
v1 -> 14 | 0  
...  
v299 -> 14 | 0  
-----
```

Metrics are reported for the ensemble clustering,

```
-----  
5 gold clusters, and 4 community detection ensemble clusters.  
VOI metric relative to truth = 0.913795.  
-----
```

The cluster data for the ensemble clustering are written to the database,

```
-----  
Writing ensemble cluster data to the database...  
-----
```

DISTRIBUTION:

1	MS 1322	Sudip Dosanjh, 1420
1	MS 0807	Robert Ballance, 9328
1	MS 9155	Robert Clay, 8953
2	MS 9159	James Brandt, 8953
2	MS 9159	Vincent De Sapio, 8953
2	MS 9152	Ann Gentile, 8953
2	MS 9159	Philip Kegelmeyer, 8900
2	MS 9159	Jackson Mayo, 8953
2	MS 9159	Philippe Pébay, 8953
2	MS 9292	Diana Roe, 8623
2	MS 9159	David Thompson, 8953
2	MS 9152	Matthew Wong, 8953
2	MS 9018	Central Technical Files, 8945-1
2	MS 0899	Technical Library, 4536