Graph Clustering in SPARQL

Kevin Deweese*  John R. Gilbert*  Adam Lugowski*  Steve Reinhardt†

Abstract

SPARQL is a powerful engine for querying graphs based on local patterns. We present a method to find a global graph metric, clustering using the PeerPressure algorithm, using RDF/SPARQL. Our approach is extensible to other classes of algorithms such as searches and centrality metrics.

1 Introduction

SPARQL is a powerful query language similar to SQL which operates on graphs specified in the RDF format. RDF graphs are composed of triples, where each triple consists of a subject, predicate, and object and specifies a directed edge from subject to object with attribute predicate.

SPARQL provides a rich way to query local neighborhoods. Our motivation is to find a way to combine this with a global graph metric: clustering.

The driving application is clustering large clinical datasets, to help identify potential disease causes. Autism researchers need to understand the underlying causes of autism spectrum disorders, based on data from genetic (e.g., SNPs in the GABA and glutamate pathways), medical history (diagnoses, prescriptions, provider visits, including pre-natal/infant, esp. infant brain MRIs), environmental (e.g., carcinogens, household chemicals), family medical history (i.e., parental psychiatric history), and early-childhood intervention-strategy domains. In practice, the patient base consists of thousands of individuals, with roughly 1M relationships per patient.

Clinical autism datasets are in general proprietary and protected by privacy laws. Therefore, as a surrogate, we target our code for a dataset from the Mayo Clinic “Smackdown” project. We have chosen to benchmark our code using the cluster-realistic synthesis work of Pinar et al. [4].

2 Our Selected Clustering Algorithm

Peer Pressure [5] (pp. 59-68 in [2]) is a clustering algorithm based on the observation that for a given graph clustering the cluster assignment of a vertex will be the same as that of most of its neighbors.

The algorithm starts with an initial cluster assignment, such as each vertex being in its own cluster. Each iteration performs an election at each vertex to select which cluster that vertex should belong to at the end of the iteration. The votes are the cluster assignments of its neighbors. Ties are settled by selecting the lowest cluster ID to maintain determinism, but can be settled arbitrarily. The algorithm converges when two consecutive iterations have a (tunably) small difference between them. Typically this means on the order of five to ten iterations on well-clustered graphs.

This algorithm is also known by the name Label Propagation [3] in the physics literature. Boldi et. al. [1] extend that work with Layered Label Propagation which accepts a parameter $\gamma$ which selects between large relatively sparse clusters and small relatively dense clusters.

3 Peer Pressure in SPARQL

Our SPARQL implementation of Peer Pressure keeps only the cluster assignment of each vertex as its state. We create RDF triples to represent an “inCluster” relationship for each vertex. A vertex is said to be in a particular cluster if there is an RDF “inCluster” triple between the vertex and the cluster ID.

The cluster election at a vertex is equivalent to counting the number of length-two paths between that vertex, one of its neighbors, and that neighbor’s cluster ID (via an “inCluster” edge). The winner of the election is found by grouping these paths by cluster ID, counting them, and selecting the cluster ID with the maximum count. These operations are done using nested subqueries which tally the votes using a GROUP BY and COUNT, and find the winner using a MAX.

In our implementation the election query operates on edges that represent a “hasLink” relationship. This relationship marks edges that passed the similarity metric, as embodied in an initialization query.

*Dept. of Computer Science, University of California, Santa Barbara
†YarcData LLC, A Cray Company.
defined by the scientists. For benchmark purposes we randomize similarity values and set a threshold. The “hasLink” edges are stored in a separate named graph, and the result of each iteration of the algorithm is stored in its own named graph as well. Each stored iteration result consists of only the “inCluster” edges. We compare results of consecutive iterations to determine convergence.

Since Peer Pressure is an iterative algorithm, the election and assignment need to be performed multiple times to reach convergence. Our approach is to construct RDF triples for each cluster assignment, then store them in a named graph for retrieval by the election query of the next iteration. Once convergence is reached the clustering can be read from the final named graph.

3.1 Implementation in JavaScript We have implemented our workflow using a driver webpage with the steps defined as JavaScript functions. This allows us to target multiple SPARQL backends, such as YarcData’s uRIKA or Apache Fuseki. Additional queries return aggregate information about the resulting clusters. This approach also enables visualization methods that are too intensive to run client-side.

4 Performance Results
We benchmark on a dataset with 10k vertices and 232k edges (RDF triples). We compared an 8-socket AMD Opteron 8378 system (32 cores) with 128 GB RAM running Apache Fuseki against a 64-processor YarcData uRIKA. The uRIKA is a dedicated SPARQL appliance. It uses highly-multithreaded ThreadStorm processors descended from the Cray XMT architecture. uRIKA’s architecture makes it very memory latency tolerant, which makes it very good at handling JOINs.

The uRIKA’s architecture really shines on our JOIN intensive query. One iteration of PeerPressure on our dataset took between 69s and 17s to execute, for a total of about 200s for the 5 iterations before convergence. Later iterations are faster due to partially clustered data which results in smaller groups.

Our Opteron system required about 9 hours for each iteration on the same data. Note that most current SPARQL implementations, including Fuseki, execute each query single threaded. On COTS hardware, then, we depend on multiple independent experiments to be run concurrently for parallelism.

5 Discussion
We note that our approach to the Peer Pressure algorithm, i.e. keeping algorithm state and iterating, makes possible a wide variety of other graph algorithms to be performed using SPARQL queries. For example, breadth-first search could be implemented using a similar approach, with state maintained by adding links to “nFrontier” and “discovered” edges as the queries traverse the graph. More complex state schemes could enable algorithms such as PageRank and Betweenness Centrality.

References