Triangle Finding: How Graph Theory can Help the Semantic Web

Edward Jimenez, Eric L. Goodman

Sandia National Laboratories, Albuquerque, NM, USA
{esjimen,elgoodm}@sandia.gov

Abstract. RDF data can be thought of as a graph where the subject and objects are vertices and the predicates joining them are edge attributes. Despite decades of research in graph theory, very little of this work has been applied to RDF data sets and it has been largely ignored by the Semantic Web research community. We present a case study of triangle finding, where existing algorithms from graph theory provide excellent complexity bounds, growing at a significantly slower rate than algorithms used within existing RDF triple stores. In order to scale to large volumes of data, the Semantic Web community should look to the many existing graph algorithms.

1 Introduction

The Semantic Web continues to evolve and grow with data sizes becoming increasingly large and unwieldy. As such, we need to utilize the most efficient and effective algorithms in order to scale to meet the growing data requirements. We believe that the graph theory body of research has much to offer in terms of formal analysis and understanding of the Semantic Web. Also, graph theory has much to offer in terms of efficient algorithms that can be employed for the Semantic Web. In particular we examine triangle finding. For several decades there have been algorithms for finding triangles that have a temporal complexity of $O(m^{3/2})$ where m is the number of edges [5]. As such, we believe it incumbent of any SPARQL engine to use these methods whenever a triangle appears as part of a query.

In Section 2 we introduce formally the notion of triangle finding and the associated notation. In Section 3 we discuss the frequency of triangles in SPARQL query benchmarks. In Section 4 we discuss how SPARQL can be used to find all the triangles in a graph. We then outline the particular triangle finding algorithm we employ in Section 5 that is $O(m^{3/2})$. Section 6 compares experimentally the triangle finding method we employ against Jena[1] and Sesame[2], two common open-source RDF/SPARQL engines. Section 7 outlines related work. We then conclude in Section 8.

[1] jena.apache.org
[2] www.openrdf.org

2 Formalisms

RDF data can be modeled as a graph, $G = (V, E)$ where V is a set of vertices and E is a set of edges connecting the vertices $v \in V$. We use n to refer to the number of vertices, $|V|$, and m as the number of edges, $|E|$. For RDF, the subjects and objects are vertices in V. The predicates can be thought of as attributes associated with each edge. We enumerate the vertices so that each has a unique id in $[1, n]$. Also, we enumerate the edges of G to be in $[1, m]$. The notation e_i refers to the i^{th} edge under the enumeration. Since RDF is described directionally, i.e. there is a subject uni-directionally related to an object, edges in the graph are also directed. We will use $source(e_i)$ to refer to the source or subject of the edge. $target(e_i)$ refers to the target or object of the edge. We call two vertices v_1, $v_2 \in V$ *adjacent* if there is an edge $e \in E$ where $(source(e) = v_1 \wedge target(e) = v_2) \vee (source(e) = v_2 \wedge target(e) = v1)$. We use $\delta(v_i)$ to denote the total degree, both incoming and outgoing, of a vertex. Below we formally define the notion of a triangle.

Definition 1. *A* triangle *in a graph G is a set of three edges, e_i, e_j, $e_k \in E$ such that the set of vertices $\hat{v} = \{v | v = source(e_i) \vee target(e_i) \vee v = source(e_j) \vee v = target(e_j) \vee v = source(e_k) \vee v = target(e_k)\}$ have the following properties:*

1. $|\hat{v}| = 3$
2. *All $v \in \hat{v}$ are adjacent to one another.*

We also refer to *triads*, which we define as pair of edges with a common vertex.

It is also useful to define the notion of triangle equality.

Definition 2. *Two triangles are considered* equal *if their edge sets are the same.*

While triangle equality may seem obvious, it is important point to consider when generating result sets via SPARQL. Without care, duplicate triangles can be generated that are not prunable with the `DISTINCT` keyword. Duplicate triangles can be produced that differ in the order in which the nodes and edge labels are presented (i.e. to which variables they are bound), and thus `DISTINCT` will have no effect on these duplicates. More of this will be discussed in Section 4.

3 Applicability to Current Benchmarks

An obvious question when optimizing for triangles in SPARQL queries is how often triangles occur in practice. We examine two popular benchmarks, LUBM [11] and SP2Bench [15]. Two out of the 14 queries in LUBM have triangles in them, namely queries 2 and 9. For brevity and to emphasize the triangle portion of the queries, we omit the prefixes and the type constraints on each of the variables.

Query 2

```
SELECT ?X, ?Y, ?Z WHERE
{?X ub:memberOf ?Z .
 ?Z ub:subOrganizationOf ?Y .
 ?X ub:undergraduateDegreeFrom ?Y}
```

Query 9

```
SELECT ?X, ?Y, ?Z WHERE
{?X ub:advisor ?Y .
 ?Y ub:teacherOf ?Z .
 ?X ub:takesCourse ?Z}
```

These two queries are also some of the more complicated and time intensive ones as reported by various vendors and researchers (e.g. AllegroGraph[3] OWLIM[4]), pointing to the need for efficient processing.

SP2Bench does not have any queries with triangles in them, but there are several queries that with a natual extension of one more constaint form a triangle. For example, Query 4 requests *all distinct pairs of article author names for authors that have published in the same journal.* A natural extension is to define a constraint between the two authors, either directly, forming a triangle, or to another common vertex. The latter case forms what is called a quadrangle, and the algorithmic approach is similar and has the same complexity as finding triangles [5]. In conclusion, there appears to be a small but significant portion of queries that contain triangle structures within them.

4 Finding Triangles with SPARQL

Expressing a SPARQL query to find all triangles in a graph is surprisingly convoluted. Figure 1 is a query that finds all unique triangles in an RDF graph, but perhaps more importantly it finds the triangles with no duplication, meaning that no two solution triangles in the result set are equal. We will refer to this query as the *Triangle-finding SPARQL query.* Note that this query only works on data involving only IRIs as the subjects and objects. According to the standard the `STR` function is not defined for blank nodes and the function also removes typing and language modifiers on literals which may cause the comparison filter to be incorrect.

Before we discuss how this query finds the triangles without duplication, we must first discuss two concepts, that of *graph isomorphism* and *graph automorphism.* Two graphs G and H are isomorphic if there is a bijection, f, mapping the vertex sets of G and H such that two vertices v_i and v_j are adjacent in G if and only if $f(v_i)$ and $f(v_j)$ are also adjacent in H. A graph automorphism is

[3] `http://www.franz.com/agraph/allegrograph/agraph_bench_lubm.lhtml`
[4] `http://www.ontotext.com/owlim/benchmark-results/lubm`

```
SELECT ?X ?Y ?Z
WHERE {
  { ?X ?a ?Y .
    ?Y ?b ?Z .
    ?Z ?c ?X
    FILTER (STR(?X) < STR(?Y))
    FILTER (STR(?Y) < STR(?Z))
  }
  UNION
  {
    ?X ?a ?Y .
    ?Y ?b ?Z .
    ?Z ?c ?X
    FILTER (STR(?Y) > STR(?Z))
    FILTER (STR(?Z) > STR(?X))
  }
  UNION
  {
    ?X ?a ?Y .
    ?Y ?b ?Z .
    ?X ?c ?Z
  }
}
```

Fig. 1. The above query finds all unique triangles and each is represented once in the result set.

when there is a isomorphism of a graph G onto itself, and generally we will be concerned with non-identity mappings.

The reason this is important can be understood from the query below:

```
SELECT ?X ?Y ?Z
WHERE {
  { ?X ?a ?Y .
    ?Y ?b ?Z .
    ?Z ?c ?X }
```

For this query, every triangle found will appear three times. For instance, a triangle with solution $\langle s_1, s_2, s_3 \rangle$ will also appear as $\langle s_2, s_3, s_1 \rangle$ and $\langle s_3, s_1, s_2 \rangle$. The reason for this is that each of the vertices s_1, s_2, and s_3 each bind to each of the three variables ?X, ?Y, and ?Z under different circumstances. All three solutions satisfy the query, but each solution is the same triangle. `DISTINCT` does not help because the bindings are different for each duplicate solution. The problem arises because the query graph represented above is automorphic. Each non-trivial automorphism is a mapping to translate from one solution to another solution using the same set of edges.

Thus we arrive at the convoluted nature of the query in Figure 1. When constructing the SPARQL query, we need to account for all possible triangles

but not generate duplicates. There are eight types of triangles, shown in Figure 2. However, all eight of these types can be collapsed down to the three unioned clauses in Figure 1. We present this formally as a proof.

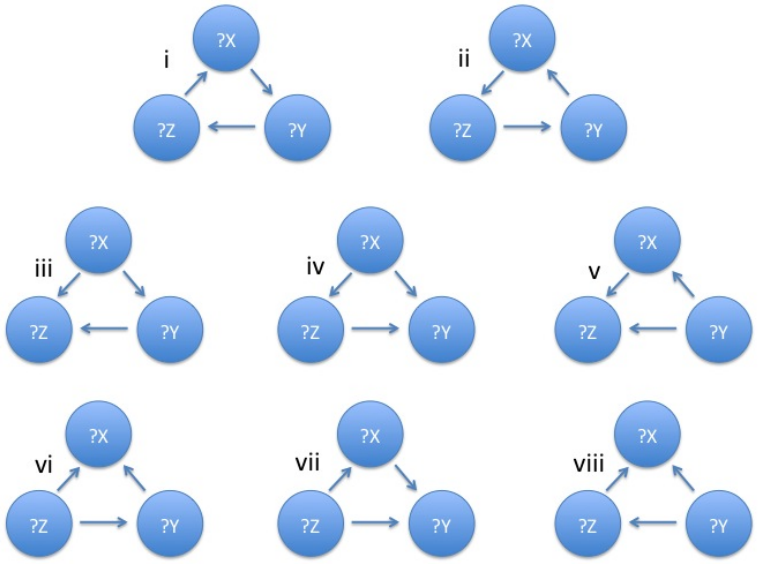


Fig. 2. The eight possible triangle patterns.

Theorem 1. *The* Triangle-finding SPARQL query *finds all triangles in the queried graph and each solution is unique.*

Proof. First consider that triangle type *iii* is isomorphic to types *iv* through *viii*. The below table outlines the functions needed to show the isomorphisms. As they are isomorphic, type *iii* is sufficient to represent all the other types *iv* - *viii*. Also, type *iii* is not automorphic, and thus will not produce any duplicate triangles. Finally, the solutions resulting from *iii* are disjoint from *i* and *ii* as there is not mapping from *iii* to either *i* or *ii*. Thus we can deal with the solution sets separately.

Mapping *iii* to ...	?X	?Y	?Z
iv	?X	?Z	?Y
v	?Y	?X	?Z
vi	?Z	?Y	?X
vii	?Z	?X	?Y
vii	?Y	?Z	?X

Concerning types i and ii, they are isomorphic under the mapping $f(?X) = ?Z$, $f(?Y) = ?Y$, and $f(?Z) = ?X$. Thus, we need only include one of the patterns in the query. We arbitrarily select type i. However, type i is automorphic (as is ii), and we must concoct a way of avoiding duplicate triples with the available SPARQL language features. We solve the issue by enforcing an ordering on the bindings. The first clause of the Triangle-finding SPARQL query enforces that the string representation of the bindings must obey $?X < ?Y < ?Z$ under an alphanumeric ordering. The second clause enforces $?Y > ?Z > ?X$. It remains to show that these two orderings will find all triangles of type i without duplication. The table below outlines all possible relations between the variables assuming no self loops.

?X ? ?Y	?Y ? ?Z	?Z ? ?X	
$<$	$<$	$<$	Empty since $?X < ?Y \wedge ?Y < ?Z \implies ?X < ?Z$, contradicting the third constraint.
$<$	$<$	$>$	Fulfilled directly by first clause
$<$	$>$	$<$	Use automorphism $f(?X) = ?Y, f(?Y) = ?Z, f(?Z) = ?X$. Fulfilled by first clause.
$<$	$>$	$>$	Fulfilled directly by second clause.
$>$	$<$	$<$	Use automorphism $f(?X) = ?Z, f(?Y) = ?X, f(?Z) = ?Y$. Fulfilled by first clause.
$>$	$<$	$>$	Use automorphism $f(?X?) = ?Z, f(?Y) = ?X, f(?Z) = ?Y)$. Fulfilled by second clause.
$>$	$>$	$<$	Use automorphism $f(?X) = ?Y, f(?Y) = ?Z, f(?Z) = ?X$. Fulfilled by second clause.
$>$	$>$	$>$	Empty since $?X > ?Y \wedge ?Y > ?Z \implies ?X > ?Z$, contradicting the third constraint.

Two of the possibilities are invalid because the constraints are contradictory. Another two possibilities directly match the first two clauses of the triangle-finding query. The remaining four cases match the two clauses through an automorphism. Thus, we may conclude that the Triangle-finding SPARQL query does in fact find all triangles in the graph with no duplicates. □

5 An $O(m^{3/2})$ Triangle Finding Algorithm

There are many triangle finding algorithms that are $O(m^{3/2})$. For the experiments we employ an algorithm presented by Cohen [6]. This algorithm has the benefit of already being described in a parallel fashion in terms of mappers and reduces of the MapReduce paradigm [8]. Also, there is a version implemented in the MultiThreaded Graph Library[5] (MTGL) [2]. However, a formal complexity analysis was not outlined in [6], so we perform that here.

[5] `https://software.sandia.gov/trac/mtgl`

Cohen's algorithm operates on what he calls a *simplified graph*. Namely, a graph in which self-loops are eliminated, directionality is ignored, and there are no duplicate edges. In our later experiments we do not allow self-loops, but we account for directionality and do allow duplicate edges. This is due to the fact that RDF is directional and multiple edges can be defined between vertices with different edge types (predicate types). We do not want to collapse all of these edge types down into one edge. We do not formally account for these different assumptions in our analysis.

Theorem 2. *Given a simplified graph G, Cohen's triangle finding algorithm is $O(m^{3/2})$.*

Proof. We assume the worst case, that G is completely connected. Relating m to n, we have

$$m = \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} \tag{1}$$

Cohen's algorithm is composed of two MapReduce phases. The input to the first map is a list of edges. Each edge has been previously been augmented with the degree of each vertex. If one were to include this preprocessing step in the overall complexity, it is $O(m)$ which is also in $O(m^{3/2})$.

Map 1 Map each edge to its low-degree vertex. According to our assumptions, $\delta(v_x) = \delta(v_y)\ \forall x, y \leq n$. Cohen suggests a tie-breaker based on vertex ordering; we'll use $v_1 < v_2 < \cdots < v_n$. Below is the composition of the bins. Since directionality is ignored, we'll use a canonical representation of each edge, $\langle v_i, v_j \rangle$, such that $i < j$.

Bin 1	Bin 2	$\cdots$	Bin $n-1$
$\langle v_1, v_2 \rangle$	$\langle v_2, v_3 \rangle$	$\cdots$	$\langle v_n - 1, v_n \rangle$
$\langle v_1, v_3 \rangle$	$\langle v_2, v_4 \rangle$		
$\vdots$	$\vdots$		
$\vdots$	$\langle v_2, v_n \rangle$		
$\langle v_1, v_n \rangle$			

Thus, for m edges, perform m mappings; hence, $O(m)$.

Reduce 1 Emit a record for each pair of edges in a bin (one for every open triad). For the graph G, the first Map phase created $n-1$ bins, and bin i contains

$n-i$ edges. Therefore the number of triads created is

$$\sum_{i=1}^{n-2}\binom{n-i}{2} = \sum_{i=1}^{n-2}\frac{(n-i)!}{2(n-(i+2))!} \tag{2}$$

$$= \frac{1}{2}\sum_{i=1}^{n-2}(n-i)(n-(i+1)) \tag{3}$$

$$< \frac{1}{2}\sum_{i=1}^{n-2}(n-1)(n-(i+1)) \tag{4}$$

$$= \frac{n-1}{2}\sum_{i=1}^{n-2}(n-(i+1)) \tag{5}$$

$$= \frac{n-1}{2}\sum_{i=1}^{n-2} i \tag{6}$$

$$= \frac{n-1}{2}\frac{(n-1)(n-2)}{2} \tag{7}$$

$$< \frac{\sqrt{2}(n-1)^3}{4} \tag{8}$$

$$= \left(\frac{(n-1)^2}{2}\right)^{3/2} \tag{9}$$

$$< \left(\frac{n(n-1)}{2}\right)^{3/2} \tag{10}$$

$$= m^{3/2} = O(m^{3/2}) \tag{11}$$

Hence, the first MapReduce task has complexity $O(m^{3/2})$.

For the second MapReduce phase the input is the emitted records of the first MapReduce phase ($O(m^{3/2})$) as well as the augmented edge list that was used as the input for the first MapReduce phase ($O(m)$). For the second MapReduce phase, let p be the size of the combined input, note:

$$O(p) = O(m) + O(m^{3/2}) = O(m^{3/2}).$$

Map 2 Combine degree-augmented file and output from Reduce 1. For the augmented edge list we have the following mapping to remove the vertex valences:

$$\begin{array}{c} \overline{key_1 = [e_1,\ \delta(v_x),\ \delta(v_y)]} \\ \overline{key_2 = [e_2,\ \delta(v_w),\ \delta(v_z)]} \\ \vdots \\ \overline{key_n = [e_n,\ \delta(v_u),\ \delta(v_t)]} \end{array} \quad \Rightarrow \quad \begin{array}{c} \overline{key_{e_1} = [e_1]} \\ \overline{key_{e_2} = [e_2]} \\ \vdots \\ \overline{key_{e_n} = [e_n]} \end{array},$$

and for the records emitted by Reduce 2, we have the identity operation. Therefore, this task is $O(p)$.

Reduce 2 Each bin corresponds with a vertex pair. A bin will contain at most one edge record and any number of triad records. With our assumptions of a completely connected graph, bin i contains $\binom{n-i}{2}$ triad records and one edge record, and the reducer will emit $\binom{n-i}{2}$ triangles. From our previous analysis, this is $O(m^{3/2})$ and therefore the overall complexity is $O(m^{3/2})$. □

6 Experiments

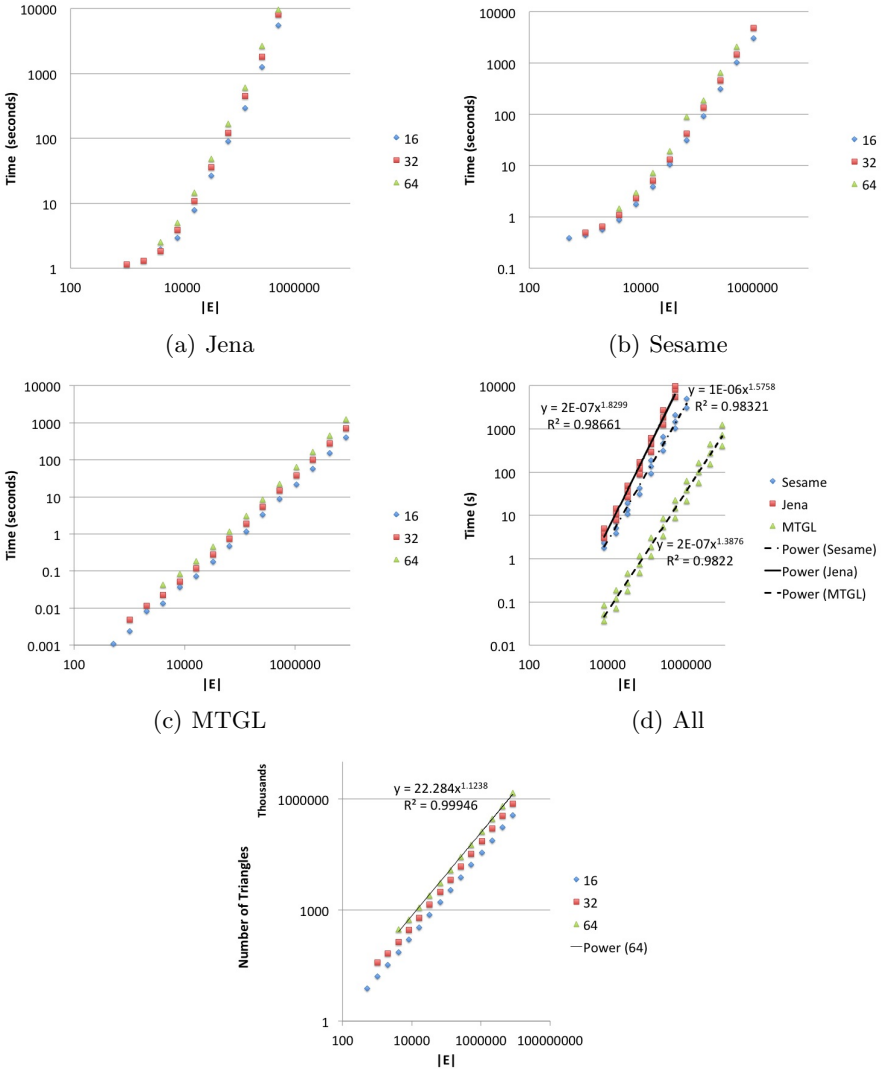
We experimentally compare open source RDF/SPARQL engines, Jena, version 2.7.2, and Sesame, version 2.6.8, with MTGL's implementation of Cohen's algorithm. For both Jena and Sesame we use the in-memory backend version of each. We did need to make some modifications to the MTGL version in order to allow for directionality and duplicate edges. Namely, we created a multimap that gives a the list of edges connecting any two vertices. This allowed us to create multiple triangles from single instances of a triad in the last reduce phase. We used a workstation with 8 GB of memory and a 2.2 GHz Intel Core i7 processor. Detailed times for our experiments can be found in the Appendix.

For our experiments we create R-MAT [4] graphs to simulate real-world graph properties such as power-law distributions on degree, small-world graphs, and small diameter. We varied the size of the graph from between $n = 2^5$ to $n = 2^{19}$. Also, we tried three different edge factors (average degree per vertex), namely 16, 32, and 64. R-MAT has four other parameters, a, b, c, and d. These four parameters are probabilities used recursively to determine where edges exist within the adjacency matrix. We set these to the values of the Graph500[6] search benchmark: $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$. To enable the SPARQL engines the ability to process the data, we created IRI's of the form *<http://i>* where i is the vertex id given by the R-MAT generator. Also we made all edges of the same type.

Figures 3(a), 3(b), and 3(c) show the individual performance of each of the three platforms as a function of the number of edges. All of the plots have a log-log scale. Figure 3(d) shows the three platforms side by side. For this Figure, we exclude graphs below 8192 edges to give a better idea of the scaling behavior for larger graphs. Both Jena and Sesame exhibit a fair amount of constant overhead per query that dominates the times in the smaller graphs. When excluding this data, the fit of trendlines using power regression is quite good, with R^2 all exceeding 0.98. As can be seen from Figure 3(d), Jena has a complexity of around $O(m^{1.83})$, Sesame has $O(m^{1.58})$, and MTGL's version of Cohen's algorithm is around $O(m^{1.39})$. The best possible for this data would be around $O(m^{1.12})$, which is rate of growth of triangles for the data as determined experimentally and shown in Figure 3(e).

It is clear that both Jena and Sesame are not employing a triangle finding algorithm as their rate of growth is significantly larger than $O(m^{1.5})$. While the differences in powers may seem small between the three, consider that extrapolating out to a billion edges, the difference in times between an $O(m^{1.39})$

[6] www.graph500.org



(a) Jena (b) Sesame

(c) MTGL (d) All

(e) Triangle Growth

Fig. 3.

algorithm and one with $O(m^{1.58})$ with the same constant is about a 50x difference. And the difference between $O(m^{1.39})$ and $O(m^{1.83})$ is around 9000x. While not the focus of this paper, it is interesting to note that the MTGL version computed the triangles in about two-orders of magnitude less time than either Jena or Sesame.

7 Related Work

There is much work within the graph theory community where there is a straightforward application to the Semantic Web. For the most part, interfacing with the Semantic Web takes place through SPARQL. This largely confines interactions to subgraph matching tasks. Under this limitation, we can find work on clique-finding [5,10], or generalizations of cliques such as trusses [6]. An excellent overview of research in subgraph pattern matching over a thirty year timespan can be found in Conte et al. [7].

There are also algorithms outside of subgraph matching that can be of help in analyzing the Semantic Web. Probably one of the most fundamental algorithms is breadth-first search. In recent years, the Graph500 list has significantly increased the competitiveness and visibility of the task, and scalability and performance have increased concomitantly with the greater attention. Other algorithms include single source shortest path [13], betweeness centrality [1], connected components [12], and many others.

Related to all of these efforts is the task of graph partitioning. Graphs have historically been difficult to partition in a distributed memory setting, where the interconnectedness of the graph make it difficult to divide the data in such a way to minimize communication overhead. Notable efforts include Buluç and Madduri [3] and Devine et al. [9].

Programming models have also been created to aid development of graph-centric algorithms. Notable among them are Google's Pregel [14] and Signal/Collect by Stutz et al. [16]. These may prove to be valuable paradigms for implementing efficient graph-oriented code that can scale on large distributed systems.

8 Conclusions

In order for Semantic Web applications to scale, the community needs to adopt efficient algorithms to use as the computational kernels underlying analytics. In this paper we've demonstrated how long existing triangle finding algorithms can be employed to speed up SPARQL queries. We believe there are other algorithms and lessons from graph theory that can utilized to speed up Semantic Web applications and also open up other avenues for analysis.

References

1. D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Proceedings of the 5th international conference on Algorithms and models for the web-graph*, WAW'07, pages 124–137, Berlin, Heidelberg, 2007. Springer-Verlag.
2. J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. *Parallel and Distributed Processing Symposium, International*, 0:495, 2007.

3. A. Buluç and K. Madduri. Graph partitioning for scalable distributed graph computations. In *Proc. 10th DIMACS Implementation Challenge Workshop – Graph Partitioning and Graph Clustering*, Feb. 2012.
4. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *SDM*. SIAM, 2004.
5. N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.
6. J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29 –41, july-aug. 2009.
7. D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, pages 265–298, 2004.
8. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
9. K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.
10. F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, Oct. 2004.
11. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
12. A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994, volume 30 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1994.
13. K. Madduri, D. Bader, J. Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.
14. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
15. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.
16. P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg, 2010. Springer-Verlag.

A Experimental Data

Below is the data we collected running the triangle query on Jena, Sesame, and MTGL. We divide the data into three tables, one for each edge factor. Times are in seconds.

A.1 Edge Factor = 16

$\|V\|$	$\|E\|$	Num Triangles	Jena	Sesame	MTGL
2^5	512	7,645	0.66	0.39	0.0011
2^6	1,024	16,159	0.96	0.44	0.0024
2^7	2,048	33,263	1.23	0.57	0.0081
2^8	4,096	72,357	1.96	0.88	0.0132
2^9	8,192	156,716	2.96	1.78	0.0365
2^{10}	16,384	333,174	7.79	3.86	0.0721
2^{11}	32,768	739,951	26.38	10.67	0.1798
2^{12}	65,536	1,648,301	89.72	31.12	0.4696
2^{13}	131,072	3,450,520	291.65	92.78	1.1767
2^{14}	262,144	7,573,624	1263.91	317.21	3.3691
2^{15}	524,288	16,864,063	5550.93	1022.38	8.7734
2^{16}	1,048,576	35,286,039		3062.77	21.6454
2^{17}	2,097,152	74,837,468			57.1703
2^{18}	4,194,304	168,767,188			153.9390
2^{19}	8,388,608	357,383,850			409.6070

A.2 Edge Factor = 32

$\|V\|$	$\|E\|$	Num Triangles	Jena	Sesame	MTGL
2^5	1,024	37,561	1.151	0.496	0.0048
2^6	2,048	66,132	1.288	0.643	0.0115
2^7	4,096	137,725	1.845	1.083	0.0225
2^8	8,192	292,062	3.878	2.299	0.0514
2^9	16,384	624,619	10.916	5.064	0.1197
2^{10}	32,768	1,388,020	35.841	13.330	0.2796
2^{11}	65,536	3,033,157	122.567	42.596	0.7512
2^{12}	131,072	6,537,422	448.030	135.857	1.8940
2^{13}	262,144	14,955,653	1829.724	459.467	5.4287
2^{14}	524,288	32,521,939	8067.696	1486.286	14.8550
2^{15}	1,048,576	73,026,129		4916.445	38.6545
2^{16}	2,097,152	155,196,692			100.6970
2^{17}	4,194,304	342,379,527			275.3078
2^{18}	8,388,608	746,302,590			720.8690

A.3 Edge Factor = 64

$\|V\|$	$\|E\|$	Num Triangles	Jena	Sesame	MTGL
2^6	4096	303,016	2.510	1.464	0.0432
2^7	8,192	554,405	5.053	2.937	0.0843
2^8	16,384	1,151,110	14.524	7.235	0.1872
2^9	32,768	2,481,009	48.234	19.189	0.4539
2^{10}	65,536	5,452,648	168.198	91.224	1.1790
2^{11}	131,072	12,051,583	612.075	189.095	3.0500
2^{12}	262,144	26,614,815	2674.014	657.880	8.4796
2^{13}	524,288	57,835,285	9738.723	2103.770	22.5987
2^{14}	1,048,576	131,190,442			63.0300
2^{15}	2,097,152	290,104,980			166.7070
2^{16}	4,194,304	634,855,745			451.3670
2^{17}	8,388,608	1,420,402,577			1250.280