Data Gauge

An experimental validation framework for G* using PostgreSQL and Neo4j

Jak Akdemir
University at Albany
jakdemir@gmail.com

Alan Labouseur
University at Albany
alan@Labouseur.com

ABSTRACT

In this paper we describe the basis of a validation framework for G* ("gee star"), a research graph database currently under development at the University at Albany under Prof. Jeong-Hyon Hwang (jhh@cs.albany.edu). We develop a validation graph small enough to evaluate our test metrics by hand (and with Ms-Excel). We then implement our test metrics in the PostgreSQL open source relational database management system and also in the open source Neo4j graph database. After showing that our PostgreSQL and Neo4j results match our manually calculated traits, we examine the accuracy of G* in this context.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and Networks
G.2.2 [**Discrete Mathematics**]: Graph Theory – *graph algorithms*
H.2 [**Database Management**]: Systems

General Terms

Measurement, Experimentation, Theory, Verification.

Keywords

Graph Database, PostgreSQL, Neo4j, validation, G*, G-star

1. INTRODUCTION

A long time ago in a galaxy far, far away... Oops, wrong saga! This is not a tale of good and evil battling it out in a universe of FTL (Faster Than Light travel as well as an elegant proof of Fermat's Last Theorem). Neither is this a tale of two cities, although we originally set out to measure the best of times and the worst of times. But we found some issues with G* so following the sage advice that in software development one should get it right before one worries about getting it fast (or beautiful), we pivoted to building a validation framework instead of benchmarking. We present here a tale of three databases: PostgreSQL, Neo4j, and G*.

2. TEST METRICS

We use three (3) graph algorithms in this validation framework: the distribution of vertex degrees, distribution of clustering coefficients, and the distribution of single-source shortest paths. Each of these metrics is of crucial importance in many aspects of graph processing (clustering, summarizing, searching) so it's worth talking just a little about each of them.

2.1 Distribution of Vertex Degrees

Distribution of Vertex Degrees is useful to understand as it measures whether the graph's connectivity is strong or not. The Vertex Degree in a graph stands for the count of incoming and outgoing edges of vertex to and from its neighbors. The degree distribution is probability, `P(k)`, which describes portion of nodes which have a certain number of neighbors. `P(k) = nk/n` where there are `nk` nodes on degree `k` in a graph of `n` nodes.

2.2 Distribution of Clustering Coefficients

Distribution of Clustering Coefficients (CC) is one of the most important metrics used to analyze graph heterogeneity. To calculate this measure we first calculate all possible edges among neighbors of a vertex, which is `(n*(n-1))/2` if there are `n` neighbors. Then we find actual number of connections, `m`, between these neighbors. Each vertex has its own CC, and for node `x`, `CC(x)= m/((n*(n-1))/2)`. For further explanation and examples please see [4] and [5].

2.3 Distribution of Single-source Shortest Paths

Distribution of Single-source Shortest Paths (SSSP) is a metric that measures the distribution of the lengths of shortest paths to all reachable vertexes from a given source vertex. We apply two techniques to find shortest paths: the well-known shortest path search algorithm named for (and by) Dijkstra and the Bellman-Ford algorithm. On Neo4j we apply Dijkstra repeatedly and in PostgreSQL we use Bellman-Ford. This shows that we preserve consistency of results between these two different algorithms.

3. VALIDATION GRAPH

In order to bootstrap our chain of trusted validation cases we began with a simple 21-vertex graph. Our base case had to be small enough that we could calculate our validation metrics by hand or with Ms-Excel.

3.1 The Great Underground Empire

We chose the first (ground) level of the Great Underground Empire from the seminal interactive fiction game Zork by Infocom. (What computer scientist didn't play some form of interactive fiction game? And why not go back to one of the originals?[1]) This 21-vertex graph is almost but not entirely undirected. Our graphs are all directed for greater generality, so we added sets of edges between all undirected vertexes. For the few directed edges we added only one edge (of course). Since edges in Zork are not labeled with weight we assume a uniform weight of one (1) for all of them. Figure 1 shows our validation map as it appeared in Zork documentation a few decades ago.

[1] Yes, we know that William Crowther's Colossal Cave game predates Zork, but that's too obscure for our proposes here.

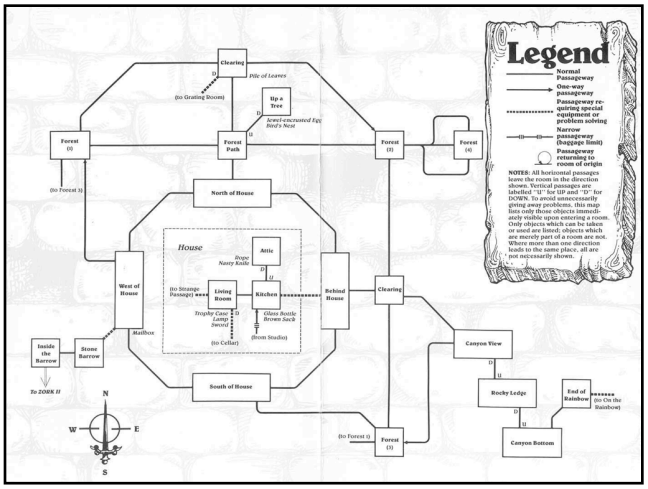

Figure 1. Zork map level 1 - Our validation graph

3.2 Validation Traits

There are 53 edges in our graph. We counted up the in-degree and out-degree for each vertex. They both sum to 53, which is nice, since there are 53 edges and a vertex on both ends of every one. Thus the sum of the total vertex degrees (in-degree plus out-degree) is 106, or twice the number of edges in the graph. Good and consistent.

Figure 2 shows our manual calculations for the distribution of vertex degrees and clustering coefficients traits.

```
vertex degree:          0  1  2  3  4  5  6  7  8  9 10
degree distribution = {0, 0, 6, 0, 3, 2, 3, 4, 2, 0, 1 }
Stem and leaf:
 0:
 1:
 2:XXXXXX
 3:
 4:XXX
 5:XX
 6:XXX
 7:XXXX
 8:XX
 9:
10:X

clustering coefficient | count
0.00                   |  8
0.16                   |  1
0.30                   |  1
0.33                   |  3
0.67                   |  1
1.00                   |  1
```

Figure 2. Manually calculated validation traits

4. POSTGRES RESULTS

4.1 Modeling the Graph

We created a relational database in PostgreSQL (version 9) to model a collection of graphs. The identifier names are self-explanatory. See Figure 3 for the code. Note the careful use of foreign keys on lines 8, 18, 24, and 25 to ensure that there is no "bleed over" among separate graphs. The `sssp_` fields on lines 12 and 13 are used in and store the results of our single-source shortest path computation.

```
 1  create table Graphs (
 2     gid  integer not null,
 3     name text,
 4   primary key (gid)
 5  );
 6
 7  create table Vertexes (
 8     gid       integer not null
                          references Graphs(gid),
 9     vid       bigint  not null,
10     name      text,
11     data      text,
12     sssp_est  int,
13     sssp_pred int,
14   primary key (gid, vid)
15  );
16
17  create table Edges (
18     gid        integer not null
                          references Graphs(gid),
19     fromVertex bigint  not null,
20     toVertex   bigint  not null,
21     weight     real,
22     data       text,
23   primary key (gid, fromVertex, toVertex),
24   foreign key (gid, fromVertex)
                 references Vertexes(gid, vid),
25   foreign key (gid, toVertex)
                 references Vertexes(gid, vid)
26  );
```

Figure 3. PostgreSQL representation of graphs

Several insert statements later we have a populated database and can begin calculating our validation metrics.

4.2 Distribution of Vertex Degrees

To compute the distribution of vertex degrees we began by creating two views on the data: one for in-degree and one for out-degree. (The full source code available is online [6] and in these appendices.) Given those two views, we computed the distribution of vertex degrees with the code in Figure 4.

```
create view Graph0vertexdegrees
 as
select gin.vid,
       gin.name,
       ("In_degree" + "Out_degree") as "Vertex_degree"
from Graph0indegrees  gin,
     Graph0outdegrees gout
where gin.vid = gout.vid
order by vid;
-- Test it.
select *
from Graph0vertexdegrees;

-- Compute the distribution of vertex degrees.
select "Vertex_degree", count("Vertex_degree")
from Graph0vertexdegrees
group by "Vertex_degree"
order by "Vertex_degree";
```

Figure 4. PostgreSQL distribution of vertex degrees code

Running the code in Figure 4 against our validation graph we get the following results, presented in Figure 5.

Vertex_degree bigint	count bigint
2	6
4	3
5	2
6	3
7	4
8	2
10	1

Figure 5. PostgreSQL Distribution of vertex degrees results

Comparing our PostgreSQL results in Figure 5 to those we manually computed in Figure 2 we see that they match. So far, so good.

4.3 Distribution of Clustering Coefficients

We created another view, this time for calculating clustering coefficients. The rather complex (but hopefully readable) clustering coefficient code is given in Figure 6.

```
create or replace view Graph0clusteringcoefficients
 as
select np.vid,
       np.neighbors,
       coalesce(na.actual, 0) as "actual_connections",
       np.possible as "possible_connections",
       ( cast(coalesce(na.actual, 0) as real) /
cast(np.possible as real) ) as "clustering coefficient"
from
    -- Possible neighbor connections
    (select Vertexes.vid, "neighbors", ( ("neighbors" *
("neighbors" - 1) / 2) ) as "possible"
     from Vertexes ,
       (select v.vid, count(*) as "neighbors"
        from Vertexes v, Edges e
        where v.gid = 0 and e.gid = 0
          and e.fromVertex = v.vid
        group by v.vid
       ) as nptemp
     where Vertexes.vid = nptemp.vid
       and "neighbors" > 1
    ) as np
left outer join
    -- Actual neighbor connections
    (select v.vid,
            count(*) as "actual"
     from Vertexes v, Edges e1,
          Edges e2,   Edges e3
     where v.gid = 0 and e1.gid = 0
       and e1.fromVertex = v.vid
       and e2.fromVertex = e1.toVertex
       and e3.fromVertex = e2.toVertex
       and e3.toVertex = v.vid
     group by v.vid
    ) as na
on np.vid = na.vid
order by np.vid;
```

Figure 6. PostgreSQL clustering coefficients code

Running the code in Figure 6 against our validation graph we get the following results, presented in Figure 7.

nid bigint	neighbors bigint	actual_ bigint	possible_ bigint	clustering coef real
0	4	0	6	0
1	3	0	3	0
2	4	0	6	0
3	3	0	3	0
4	3	0	3	0
7	4	1	6	0.166667
8	3	1	3	0.333333
9	2	0	1	0
11	2	0	1	0
12	3	1	3	0.333333
13	3	2	3	0.666667
14	5	3	10	0.3
15	3	3	3	1
17	3	1	3	0.333333
19	2	0	1	0

Figure 7. PostgreSQL clustering coefficients results

Counting up our PostgreSQL results in Figure 7 and comparing to those we manually computed in Figure 2 we see that they are the same. Still good, at least so far.

4.4 Distribution of Single-source Shortest Paths

Single-source shortest path (SSSP) is the most complex of our three validation metrics. Though our validation graph is indeed small and easy to analyze by hand, and while individual shortest paths are relatively easy to see, recognizing shortest paths from a single vertex to all others is not readily obvious through casual observation. Since we cannot "just see it" we need a more formal justification for our baseline results if we are to believe them later when evaluating G*.

Time to hit the books. Page 652 of the third edition of *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein (CLRS) [1] includes an even simpler graph showing the execution of the Bellman-Ford SSSP algorithm. We built that graph in our PostgreSQL model to test our implementation of Bellman-Ford. Details of this meta-experiment are included in the appendices and available online [6]. As shown in Figure 8, we successfully validated our PostgreSQL implementation against the CLRS reference.

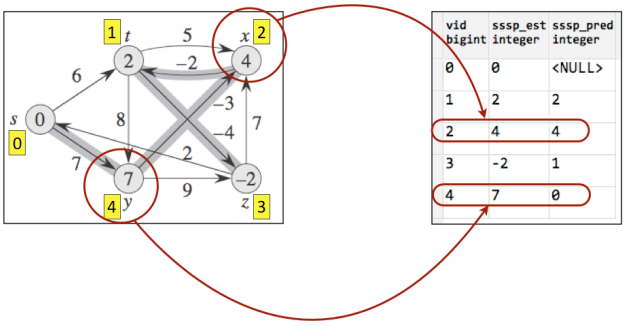

Figure 8. Validating our SSSP code against CLRS reference

Our pg/plsql code for Bellman-Ford is given in Figure 9.

```
create or replace function shortestPath(graphId integer,
                                        fromVertex integer) returns boolean as $$
declare
    numVertexesInGraph integer;
    e                  record; -- Index variable used to iterate over the edges.
    fromVertexEst, toVertexEst, currentEdgeWeight, newEst real;
begin
    -- Initialize the vertex data. (CLRS 3ed, page 648)
    update Vertexes
    set sssp_est = 32768, sssp_pred = NULL  -- TODO: Use a more natural stand-in for infinity. (MaxInt() or something.)
        where Vertexes.gid = graphId;
    -- Set our fromVertex's sssp_est to 0, since we're already there.
    update Vertexes set sssp_est = 0 where Vertexes.gid = graphId and Vertexes.vid = fromVertex;
    -- Iterate over all of the count(verticies in our target graph)-1 times
    select count(*) into strict numVertexesInGraph  from Vertexes where Vertexes.gid = graphId;
    for i in 1 .. numVertexesInGraph - 1 loop
        -- Iterate over all of the edges in our target graph.
        for e in select Edges.fromVertex, Edges.toVertex, Edges.weight from Edges where Edges.gid = graphId loop
            -- Relax (CLRS 3ed, page 649)
            select Vertexes.sssp_est into strict fromVertexEst from Vertexes
                   where Vertexes.gid = graphId and Vertexes.vid = e.fromVertex;
            select Vertexes.sssp_est into strict   toVertexEst from Vertexes
                   where Vertexes.gid = graphId and Vertexes.vid = e.toVertex;
            currentEdgeWeight = coalesce(e.weight,1); -- NULL edge weights are coalesced to 1.
            if toVertexEst > (fromVertexEst + currentEdgeWeight) then
                -- Update the new path estimate and predecessor vertex for the current vertex (v).
                newEst := (fromVertexEst + currentEdgeWeight);
                update Vertexes set sssp_est = newEst, sssp_pred = e.fromVertex
                       where Vertexes.gid = graphId and Vertexes.vid = e.toVertex;
            end if;
        end loop;
    end loop;
    -- TODO: Check for negative cycles (CLRS 3ed page 651) and ROLLBACK if there are any. Else COMMIT.
    return true;
end;
$$ language plpgsql;
```

Figure 9. PostgreSQL single-source shortest path code

Now that were were confident in our SSSP code we ran it against our validation graph with source vertex = 0. Note the results in Figure 10.

sssp_est integer	count bigint
0	1
1	4
2	5
3	4
4	4
5	1
6	1
7	1

Figure 10. PostgreSQL SSSP from vertex 0 results

5. NEO4J RESULTS

5.1 Modeling the Graph

Neo4j has a simple API for graph generation, update, and graph - wide calculations. We use community edition, version 1.6.1.

In graph modeling, we used same data as PostgreSQL (and later G*) for input. Each vertex has directed or undirected edges. Our validation application initializes the graph with the `generateGraph` method. To create a relation between two vertexes we create a new relation type as `RelTypes.Follows`. Also, for each relation we assign a default cost of one (1). We'll use this in the shortest path metric. Additionally, we used a hash map to store vertex IDs and corresponding IDs given by Neo4j. A portion of this code is shown in Figure 11.

```
Relationship relationship =
    sourceVertex.createRelationshipTo(destinationVertex,
                                      RelTypes.FOLLOWS);
relationship.setProperty("cost", 1);
 ¦
HashMap<Integer, Long> nodeListMap =
                            new HashMap<Integer, Long>();
 ¦
EmbeddedGraphDatabase graphDatabase =
          new EmbeddedGraphDatabase(".../neo4j-galaxy");
Transaction tr = graphDatabase.beginTx();
generateGraph(graphDatabase);
tr.success();
tr.finish();
graphDatabase.shutdown();
```

Figure 11. Creating the graph in Neo4j

5.2 Distribution of Vertex Degrees

For this metric the `degreeDistributionCalculator` method calculates the total vertex degree for each vertex. First we define `TraversalDescription` as followers then we iterate over all vertices and explore their neighbors. Our full source code is in the appendices and online [6]. Our results are given in Figure 12.

```
In outgoing degree 2 there are 6 nodes
In outgoing degree 4 there are 3 nodes
In outgoing degree 5 there are 2 nodes
In outgoing degree 6 there are 3 nodes
In outgoing degree 7 there are 4 nodes
In outgoing degree 8 there are 2 nodes
In outgoing degree 10 there are 1 nodes
```

Figure 12. Neo4j Distribution of vertex degree results

Comparing our Neo4j results in Figure 12 to those we manually computed in Figure 2 and those from PostgreSQL in Figure 5 we see that they are all consistent and match. Excellent.

5.3 Distribution of Clustering Coefficients

Neo4j has some difficulties like reaching collection size and randomAccess to vertexes according to property. For this part these problems emerged in multiple iterations stated in each other which caused polynomial performance problems. Nonetheless, our validation application first retrieved all vertices then for each vertex, all neighbors. After that, for all neighbors we sum actual edges among them and divide as noted in section 2.2. Our results are given in Figure 13.

```
 0 - 4 neighbors with 0 external paths=> LCC 0.0
 1 - 3 neighbors with 0 external paths=> LCC 0.0
 2 - 4 neighbors with 0 external paths=> LCC 0.0
 3 - 3 neighbors with 0 external paths=> LCC 0.0
 4 - 3 neighbors with 0 external paths=> LCC 0.0
 5 - 1 neighbors with 0 external paths=> LCC NaN
 6 - 1 neighbors with 0 external paths=> LCC NaN
 7 - 4 neighbors with 1 external paths=> LCC 0.1667
 8 - 3 neighbors with 2 external paths=> LCC 0.6667
 9 - 2 neighbors with 0 external paths=> LCC 0.0
10 - 1 neighbors with 0 external paths=> LCC NaN
11 - 2 neighbors with 0 external paths=> LCC 0.0
12 - 3 neighbors with 1 external paths=> LCC 0.3333
13 - 3 neighbors with 2 external paths=> LCC 0.6667
14 - 5 neighbors with 3 external paths=> LCC 0.3
15 - 3 neighbors with 4 external paths=> LCC 1.3333
16 - 1 neighbors with 0 external paths=> LCC NaN
17 - 3 neighbors with 1 external paths=> LCC 0.3333
18 - 1 neighbors with 0 external paths=> LCC NaN
19 - 2 neighbors with 0 external paths=> LCC 0.0
20 - 1 neighbors with 0 external paths=> LCC NaN
```

Figure 13. Neo4j clustering coefficients results

Counting up our Neo4j results in Figure 13 and comparing them to those we manually computed in Figure 2 and those from PostgreSQL in Figure 7 we see some differences. There are some issues to be addressed in future work here. See section 8.

5.4 Distribution of Single-source Shortest Paths

For this metric we use the `singleSourceShortestPathCalculator` method. We directly used Neo4j's built-in method `GraphAlgoFactory.dijkstra` to find shortest path between two vertices. For `n` vertexes, it iterates on n^2 cases from all vertexes to all vertexes, calculating the all shortest paths.

```
for (Node nodeSource : graphDatabase.getAllNodes()) {
    HashMap<Integer, Integer> ssspMap = new HashMap<Integer, Integer>();

    for (Node nodeDestination : graphDatabase.getAllNodes()) {
        if ((nodeSource.getId() != 0) && (nodeDestination.getId() != 0)
                && nodeSource.getId() != nodeDestination.getId()) {
            int distance = 0;

            WeightedPath path = dijkstraPathFinder.findSinglePath(nodeSource, nodeDestination);
            if (path != null) {

                for (Node node : path.nodes()) {
                    distance++;
                    // System.out.println( node.getProperty( "id_x" ) );
                }
                // distance do not count first row
                distance = distance - 1;
                System.out.println("From S : " + nodeSource.getProperty("id_x") + " to D : "
                        + nodeDestination.getProperty("id_x") + " with cost " + distance);
                if (ssspMap.get(distance) != null) {
                    ssspMap.put(distance, ssspMap.get(distance) + 1);
                } else {
                    ssspMap.put(distance, 1);
                }
            }
        }
    }
```

Figure 14. Neo4j SSSP code

Executing the code in Figure 14 we get our results, presented here in Figure 15.

```
In outgoing direction on distance 0 there are 1 nodes
In outgoing direction on distance 1 there are 4 nodes
In outgoing direction on distance 2 there are 5 nodes
In outgoing direction on distance 3 there are 4 nodes
In outgoing direction on distance 4 there are 4 nodes
In outgoing direction on distance 5 there are 1 nodes
In outgoing direction on distance 6 there are 1 nodes
In outgoing direction on distance 7 there are 1 nodes
```

Figure 15. Neo4j SSSP from vertex 0 results

Comparing these results to those from PostgreSQL in Figure 10 we see that they are the same. Extra cool is the fact that we've also cross-validated the Bellman-Ford algorithm and Dijkstra's.

6. EVALUATING G*

6.1 Modeling the Graph

G* is a research graph database currently under development at the University at Albany under Prof. Jeong-Hyon Hwang (jhh@cs.albany.edu). See [2], [3] for details. We express our validation graph as one source vertex ID followed by one or more destination vertex IDs in a plain text file. (Our file, `galaxy.dat`, is available online [6].)

When executing `import_graph.sh` (which makes a call to `graphdb.GraphImporter` from the G* JAR) with `galaxy.dat` we observe that G* reports the presence of 73 vertices. There are only 21 in the our graph and in the data file, so this is not a good sign. But let's press on and see what happens when we look at our validation metrics.

6.2 Distribution of Vertex Degrees

We formulate this query in G* with the code shown in Figure 16. Note how precise and compact this code is compared to PostgreSQL and Neo4j.

```
%select graph.id, histogram(degree(vertex).degree, [0,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100])
%from graph(*) group by graph.id

vertex@* = VertexOperator([], *);
degree@* = DegreeOperator([vertex@local]);
histogram@* = HistogramOperator([degree@local], degree,
Integer, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100]);
union@1 = UnionOperator([histogram@*]);
histogramMerge@1 =
HistogramMergeOperator([union@local]);
```

Figure 16. G* Distribution of vertex degrees code

Executing the G* code in Figure 16 we get the results, presented here in Figure 17.

```
[0,  1):1
[1,  2):5
[2,  3):3
[3,  4):8
[4,  5):3
[5,  6):1
[6,  7):0
[7,  8):0
[8,  9):0
[9, 10):0
[10+  ):0
```

Figure 17. G* Distribution of vertex degree results

Comparing these results from G* to those we computed manually (Figure 2) and those from PostgreSQL (Figure 5) and Neo4j (Figure 12) we are forced to observe that G* is wrong.

6.3 Distribution of Clustering Coefficients

We formulate this query in G* with the code noted in Figure 18. Note once again how precise and compact this code is compared to PostgreSQL and Neo4j. That's pretty cool.

```
%select graph.id, histogram(c_coeff(vertex).c_coeff, [0,
0.16, 0.3, 0.33 0.67, 1.0])
%from graph(*) group by graph.id

vertex@* = VertexOperator([], *);
c_coeff@* = CCoeffOperator([vertex@local]);
histogram@* = HistogramOperator([c_coeff@local],
c_coeff, Double, [0, 0.16, 0.3, 0.33 0.67, 1.0]);
union@1 = UnionOperator([histogram@*]);
histogramMerge@1 =
HistogramMergeOperator([union@local]);
```

Figure 18. G* Distribution of clustering coefficients code

Executing the G* code in Figure 18 we get our results, presented here in Figure 19.

```
[0.0, 0.16):12
[0.16, 0.3):0
[0.3, 0.33):0
[0.33, 0.67):3
[0.67, 1.0):0
```

Figure 19. G* Distribution of clustering coefficients results

Comparing these results from G* to those we computed manually (Figure 2) and those from PostgreSQL (Figure 7), we are forced to observe that G* is once again wrong. Bummer.

6.4 Distribution of Single-source Shortest Paths

We formulate this query (from vertex 0) in G* with the code noted in Figure 20.

```
%select graph.id, histogram(distance(vertex,
0).distance, [0, 1, 2, 4, 8, 16, 32, 64, 128])
%from graph(*) group by graph.id

min_dist@* = MinDistOperator([], 0, *);
histogram@* = HistogramOperator([min_dist@local],
min_dist, Double, [0, 1, 2, 4, 8, 16, 32, 64, 128]);
union@1 = UnionOperator([histogram@*]);
histogramMerge@1 =
HistogramMergeOperator([union@local]);
```

Figure 20. G* SSSP code

Executing the G* code in Figure 20 we get our results, presented here in Figure 21.

```
[0.0,    1.0):1
[1.0,    2.0):4
[2.0,    4.0):7
[4.0,    8.0):8
[8.0,   16.0):1
[16.0,  32.0):0
[32.0,  64.0):0
[64.0, 128.0):0
```

Figure 21. G* SSSP from vertex 0 results

Comparing these results from G* to those from PostgreSQL (Figure 10) and Neo4j (Figure 15), we are pleased to observe that G* is very close. For reasons that are left to explore in future work (see section 8) we see that G* is far more accurate for SSSP than for the other two validation metrics we measured. Interesting.

7. CONCLUSIONS

Though G* failed to perform accurately in two of the three validation metrics we used, that the third, SSSP, performed so well as give us hope that getting the others into compliance may not be terribly difficult nor take terribly long once the source code is analyzed. Overall, our analysis of G* yields hope for the future.

8. FUTURE WORK

- Improve our Neo4j clustering coefficient implementation, since it's at least a little broken here.
- We shall be examining G*'s implementation of vertex degree distribution and clustering coefficient distribution, especially in comparison with SSSP, as SSSP is far more accurate than the first two.
- We should also like to implement more graph algorithms for validation.
- Automation and logging is necessary for long-term testing and measurement of progress as G* evolves.

There is much to do.

9. ACKNOWLEDGMENTS

Our thanks go out to Prof. Jeong-Hyon Hwang for introducing us to this fascinating topic as well as for his enthusiasm and support.

10. REFERENCES

[1] Cormen, Leiserson, Rivest, Stein - 2009. Introduction to Algorithms. *The MIT Press.* ISBN 978-0-262-53305-8

[2] Hwang, Spillane - 2012. G*: A Parallel System for Efficiently Managing Large Graphs.

[3] http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=1149372 accessed on May 11, 2012

[4] Izquierdo and Hanneman. Introduction to the Formal Analysis of Social Networks Using Mathematica. Section 7.6.2. 2005 http://library.wolfram.com/infocenter/TechNotes/6638/Izquierdo_Hanneman.pdf accessed May 12, 2012

[5] Watts and Strogatz. 1998 Collective Dynamics of "Small-world" Networks. Nature vol 393 - 4 June 1998

[6] http://www.labouseur.com/projects/dataGauge/