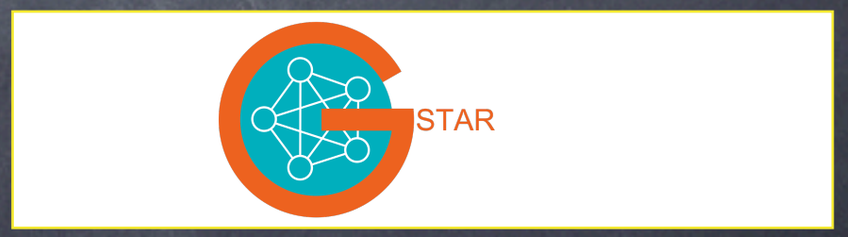
Data Gauge

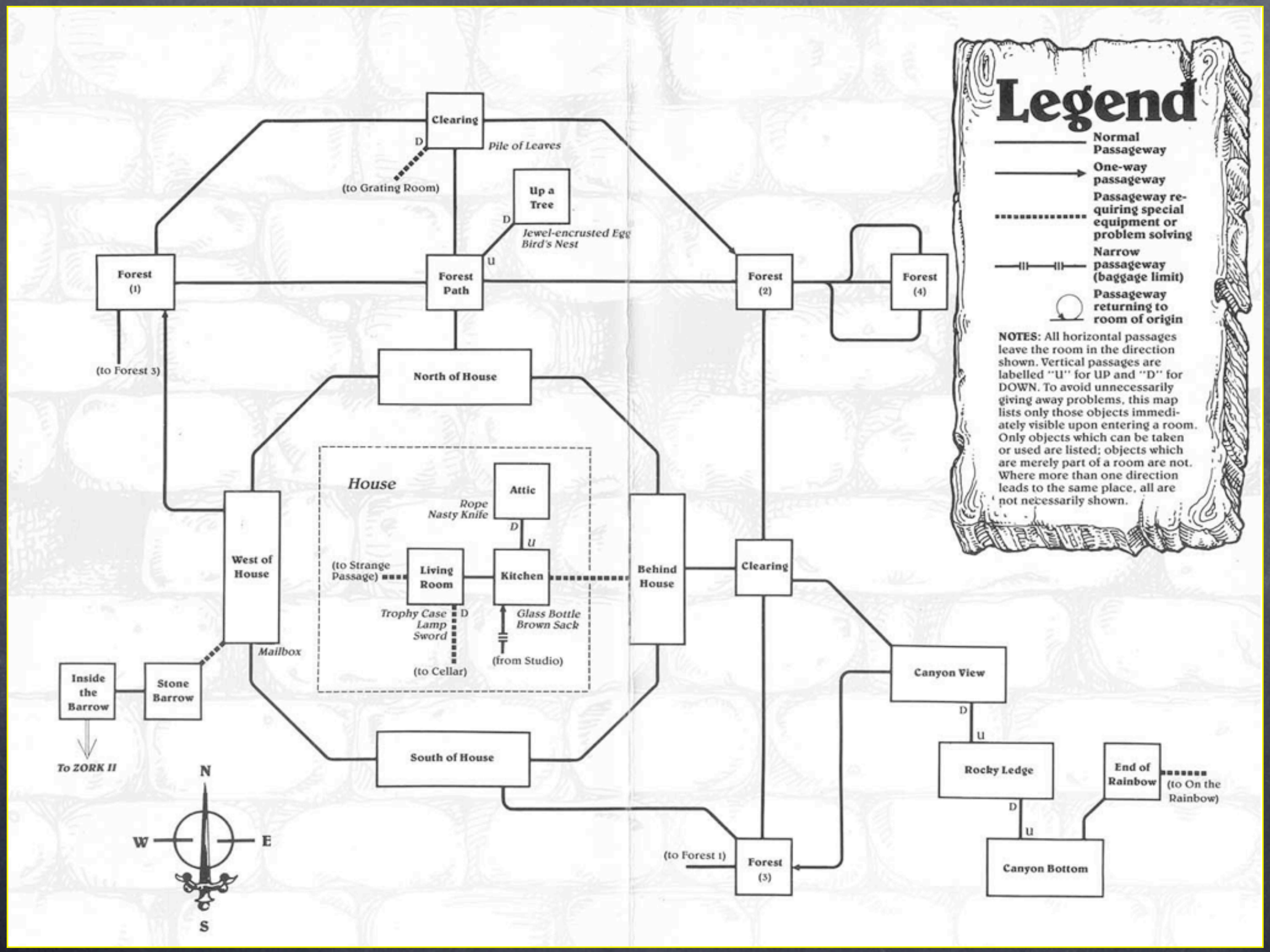
An experimental validation framework for G* using PostgreSQL and Neo4j

by
Jak Akdemir (jakdemir@gmail.com) and Alan Labouseur (alan@Labouseur.com)

May 3, 2012



Validation Data: Map of the Great Underground Empire



Some traits of the validation graph.

Node ID	*Name*	*In-degree*	*Out-degree*	*Vertex-degree (sum of in-degree and out-degree)*
0	West of House	3	4	7
1	North of House	3	3	6
2	Behind House	4	4	8
3	South of House	3	3	6
4	Kitchen	3	3	6
5	Attic	1	1	2
6	Living Room	1	1	2
7	Clearing #1	4	4	8
8	Canyon View	2	3	5
9	Rocky Ledge	2	2	4
10	End of Rainbow	1	1	2
11	Canyon Bottom	2	2	4
12	Forest #3	4	3	7
13	Forest #1	4	3	7
14	Forest Path	5	5	10
15	Clearing #2	2	3	5
16	Up a Tree	1	1	2
17	Forest #2	4	3	7
18	Forest #4	1	1	2
19	Stone Barrow	2	2	4
20	Inside the Barrow	1	1	2
		53	53	106

```
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
```

See Galaxy.sql for PostgreSQL script.

Testing PostgreSQL - Setting up the graph

```
create table Graphs (
   gid  integer not null,                                  -- Should be serial eventually.
   name text,
 primary key (gid)
);

create table Nodes (
    gid  integer not null references Graphs(gid),          -- Should be serial eventually.
    nid  bigint  not null,                                 -- Should be bigserial eventually.
    name text,
    data text,
 primary key (gid, nid)
);

create table Edges (
    gid      integer not null references Graphs(gid),      -- Should be serial eventually.
    fromNode bigint  not null,                             -- Should be bigserial eventually.
    toNode   bigint  not null,                             -- Should be bigserial eventually.
    weight   real,
    data     text,
 primary key (gid, fromNode, toNode),
 foreign key (gid, fromNode) references Nodes(gid, nid),
 foreign key (gid, toNode)   references Nodes(gid, nid)
);
```

```
--Add the nodes to this new graph.
insert into Nodes(gid, nid, name) values(0,  0, 'West of House');
insert into Nodes(gid, nid, name) values(0,  1, 'North of House');
insert into Nodes(gid, nid, name) values(0,  2, 'Behind House');
insert into Nodes(gid, nid, name) values(0,  3, 'South of House');
insert into Nodes(gid, nid, name) values(0,  4, 'Kitchen');
insert into Nodes(gid, nid, name) values(0,  5, 'Attic');
insert into Nodes(gid, nid, name) values(0,  6, 'Living Room');
insert into Nodes(gid, nid, name) values(0,  7, 'Clearing #1');
insert into Nodes(gid, nid, name) values(0,  8, 'Canyon View');
insert into Nodes(gid, nid, name) values(0,  9, 'Rocky Ledge');
```

```
-- Add the edges to the new nodes in the new graph.
insert into Edges(gid, fromNode, toNode) values(0,  0,  1);
insert into Edges(gid, fromNode, toNode) values(0,  1,  0);
insert into Edges(gid, fromNode, toNode) values(0,  1,  2);
insert into Edges(gid, fromNode, toNode) values(0,  2,  1);
insert into Edges(gid, fromNode, toNode) values(0,  2,  3);
insert into Edges(gid, fromNode, toNode) values(0,  3,  2);
insert into Edges(gid, fromNode, toNode) values(0,  3,  0);
insert into Edges(gid, fromNode, toNode) values(0,  0,  3);
insert into Edges(gid, fromNode, toNode) values(0,  2,  4);
insert into Edges(gid, fromNode, toNode) values(0,  4,  2);
```

Testing PostgreSQL - Dist. of Vertex Degrees

```
-- Create View to display the in-degree of all nodes in graph 0.
create view Graph0indegrees
  as
select n.nid,
       n.name,
       count(e.toNode) as "In_degree"
from Graphs g,
     Nodes  n,
     Edges  e
where g.gid = 0
  and n.gid = g.gid
  and e.gid = g.gid
  and e.toNode = n.nid      -- We need this to ensure we have a valid node.
group by n.nid, n.name
order by n.nid;
```

There's another view for out-degree .

```
-- Create a view to display the vertex degree of all out nodes in graph 0.
-- Vertex degree = in-degree + out-degree. (See http://reference.wolfram.com/
create view Graph0vertexdegrees
 as
select gin.nid,
       gin.name,
       ("In_degree" + "Out_degree") as "Vertex_degree"
from Graph0indegrees  gin,
     Graph0outdegrees gout
where gin.nid = gout.nid
order by nid;
-- 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"
;
```

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

Testing PostgreSQL - Clustering Coefficients

```sql
-- Create a view to compute and display the local clustering coefficients for graph 0.
create view Graph0clusteringcoefficients
 as
select np.nid,
       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 Nodes.nid, "neighbors", ( ("neighbors" * ("neighbors" - 1) / 2) ) as "possible"
     from Nodes ,
       (select n.nid, count(*) as "neighbors"
        from Nodes n,
             Edges e
        where n.gid = 0
          and e.gid = 0
          and e.fromNode = n.nid
        group by n.nid
       ) as np
     where Nodes.nid = np.nid
       and "neighbors" > 1      -- Clustering coefficient is not defined for nodes with fewer than one neighbor.
    ) as np
left outer join
    -- Actual neighbor connections
    (select n.nid,
            count(*) as "actual"
     from Nodes n,
          Edges e1,
          Edges e2,
          Edges e3
     where n.gid = 0
       and e1.gid = 0
       and e1.fromNode = n.nid
       and e2.fromNode = e1.toNode
       and e3.fromNode = e2.toNode
       and e3.toNode = n.nid
     group by n.nid
    ) as na
on np.nid = na.nid
order by np.nid
;
```

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

Testing PostgreSQL - Single-source Shortest Path

First, our own validation graph:
the Bellman-Ford example from CLRS 3ed page 652.

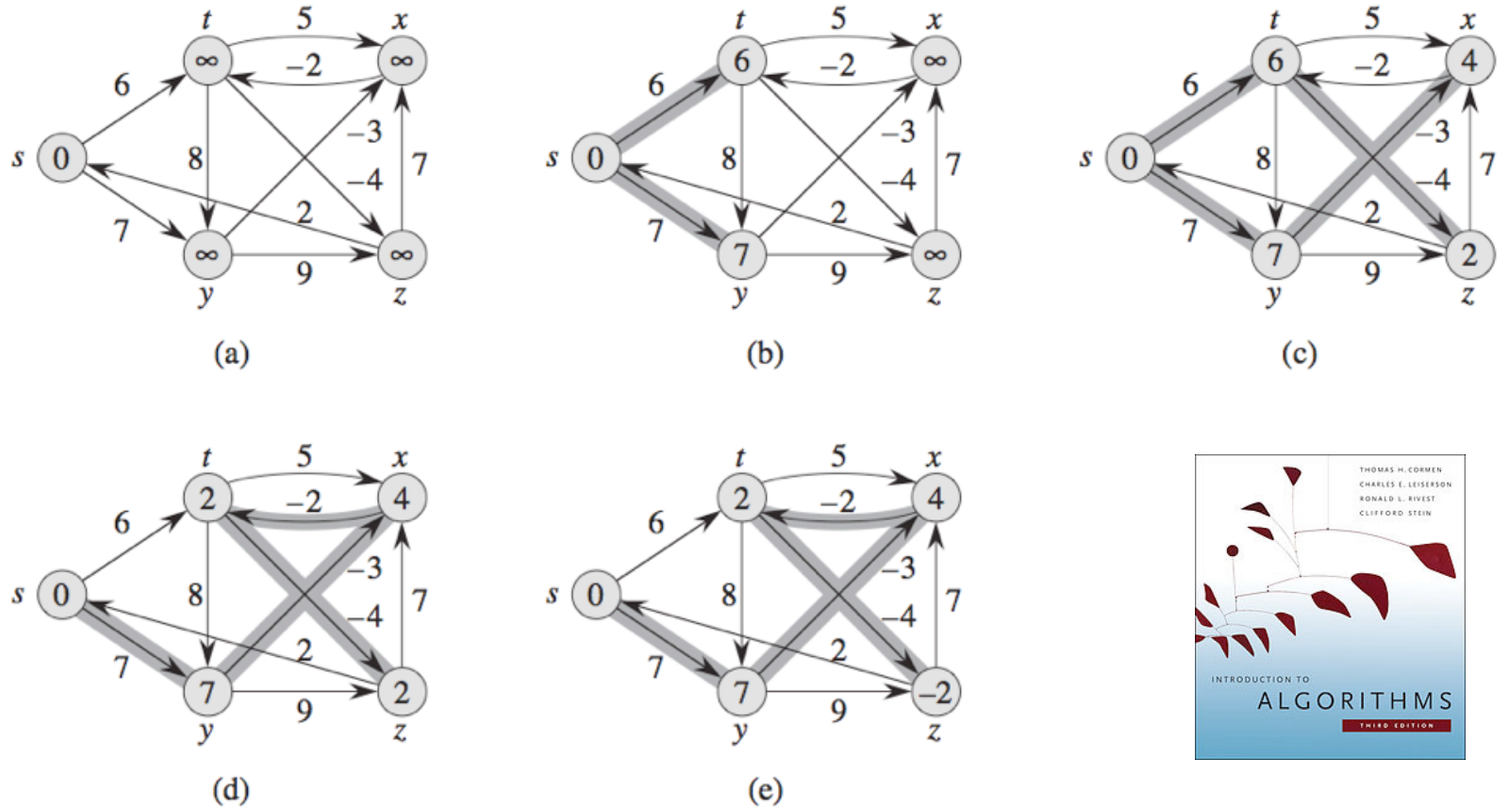
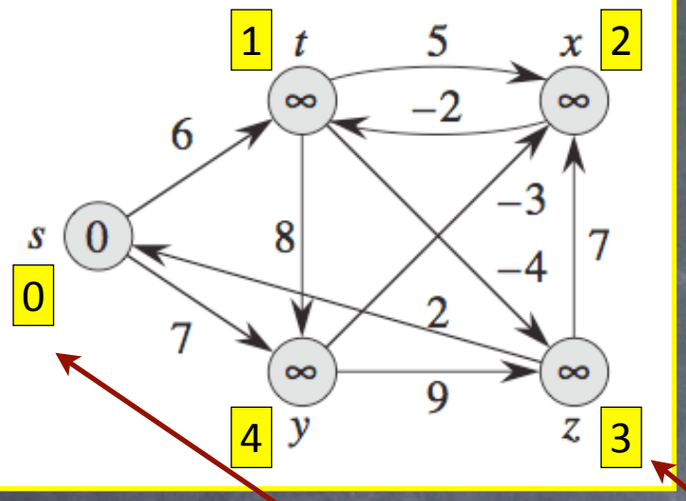


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Testing PostgreSQL - Single-source Shortest Path

Test graph start state

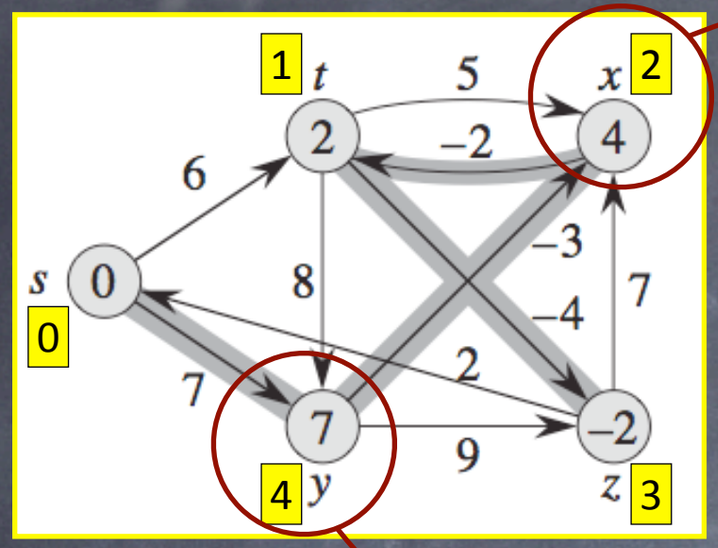



```
-- Graph id=1: The Bellman-Ford example graph from CLRS 3ed page 652.
insert into Graphs(gid, name) values(1, 'CLRS Bellman-Ford example');

--Add the Vertexes to this new graph.
insert into Vertexes(gid, vid, name) values(1,  0, 's');
insert into Vertexes(gid, vid, name) values(1,  1, 't');
insert into Vertexes(gid, vid, name) values(1,  2, 'x');
insert into Vertexes(gid, vid, name) values(1,  3, 'z');
insert into Vertexes(gid, vid, name) values(1,  4, 'y');

-- Add the edges to the new Vertexes in the new graph.
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  0,  1,  6);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  0,  4,  7);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  1,  4,  8);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  1,  2,  5);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  1,  3, -4);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  2,  1, -2);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  3,  2,  7);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  3,  0,  2);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  4,  3,  9);
insert into Edges(gid, fromVertex, toVertex, weight) values(1,  4,  2, -3);
```

id bigint	From text	To text	id bigint	weight real
0	s	t	1	6
0	s	y	4	7
1	t	x	2	5
1	t	z	3	-4
1	t	y	4	8
2	x	t	1	-2
3	z	s	0	2
3	z	x	2	7
4	y	x	2	-3
4	y	z	3	9

Testing PostgreSQL - Single-source Shortest Path

Test graph final state



vid bigint	sssp_est integer	sssp_pred integer
0	0	<NULL>
1	2	2
2	4	4
3	-2	1
4	7	0

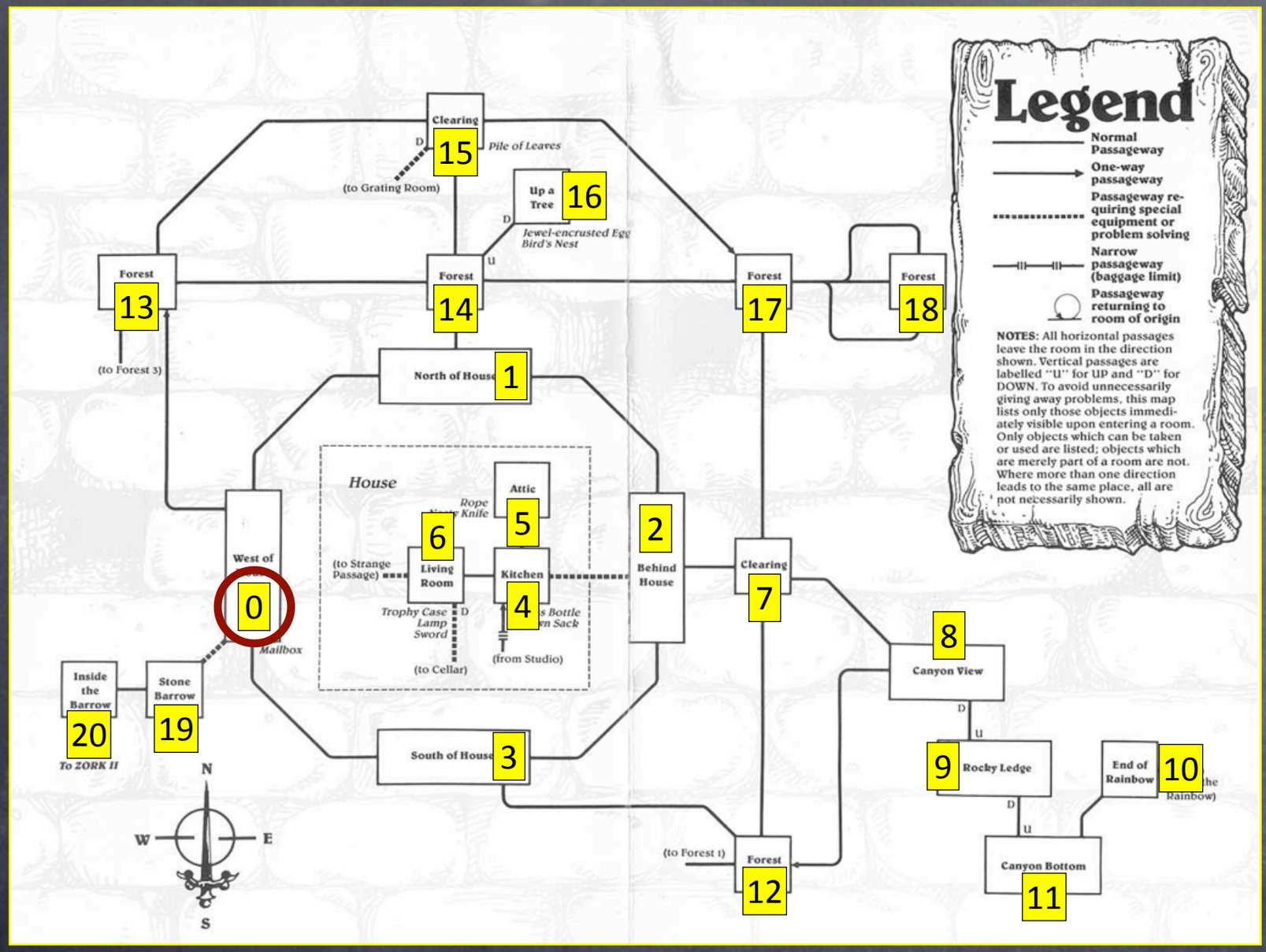
It Works!

```
create or replace function shortestPath(graphId integer,
                                        fromVertex integer) returns boolean as $$
declare
    numVertexesInGraph integer;
    e record;
    fromVertexEst, toVertexEst, currentEdgeWeight, newEst real;
begin
    update Vertexes
    set sssp_est  = 32768, sssp_pred = NULL where Vertexes.gid = graphId; -- TODO: Use a more natural stand-in for infinity.
    update Vertexes set sssp_est = 0 where Vertexes.gid = graphId and Vertexes.vid = fromVertex;
    select count(*) into strict numVertexesInGraph from Vertexes where Vertexes.gid = graphId;
    for i in 1 .. numVertexesInGraph - 1 loop
        for e in select Edges.fromVertex, Edges.toVertex, Edges.weight from Edges where Edges.gid = graphId loop
            select Vertexes.sssp_est into strict fromVertexEst from Vertexes where Vertexes.gid = graphId and Vertexes.vid = e.fromVertex;
            select Vertexes.sssp_est into strict   toVertexEst from Vertexes where Vertexes.gid = graphId and Vertexes.vid = e.toVertex;
            currentEdgeWeight = coalesce(e.weight,1); -- NULL edge weights are coalesced to 1.
            if toVertexEst > (fromVertexEst + currentEdgeWeight) then
                newEst := (fromVertexEst + currentEdgeWeight);
                update Vertexes set sssp_est = newEst, sssp_pred = e.fromVertex where Vertexes.gid = graphId and Vertexes.vid = e.toVertex;
            end if;
        end loop;
    end loop;
    -- TODO: Check for negative cycles (CLRS 3ed page 651) and ROLLBACK if there are any. Else COMMIT.
    return true;
end;
$$ language plpgsql;
```

This is minified code. The actual code has comments and is nicely formatted.

Testing PostgreSQL - Single-source Shortest Path

SSSP on the validation graph. Source = Vertex 0



vid bigint	sssp_est integer	sssp_pred integer
0	0	<NULL>
1	1	0
2	2	1
3	1	0
4	3	2
5	4	4
6	4	4
7	3	2
8	4	7
9	5	8
10	7	11
11	6	9
12	2	3
13	1	0
14	2	13
15	2	13
16	3	14
17	3	14
18	4	17
19	1	0
20	2	19

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

```
0:X
1:XXXX
2:XXXXX
3:XXXX
4:XXXX
5:X
6:X
7:X
```

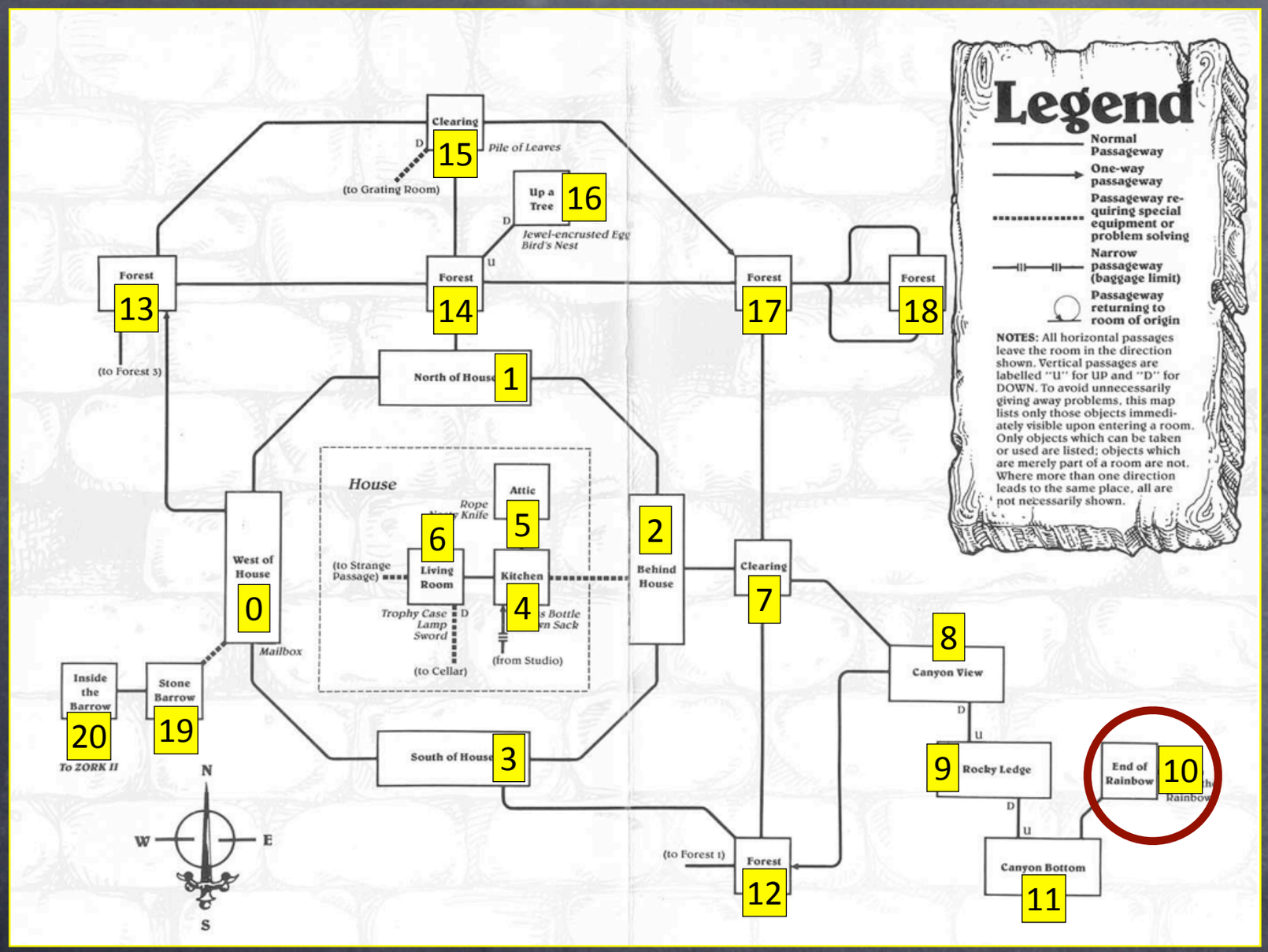
All edges are have uniform weight = 1.

Path Lengths and Dist.

Query time = 66ms

Testing PostgreSQL - Single-source Shortest Path

SSSP on the validation graph. Source = Vertex 10



vid bigint	sssp_est integer	sssp_pred integer
0	6	3
1	6	2
2	5	7
3	5	12
4	6	2
5	7	4
6	7	4
7	4	8
8	3	9
9	2	11
10	0	<NULL>
11	1	10
12	4	8
13	5	12
14	6	13
15	6	13
16	7	14
17	5	7
18	6	17
19	7	0
20	8	19

sssp_est integer	count bigint
0	1
1	1
2	1
3	1
4	2
5	4
6	6
7	4
8	1

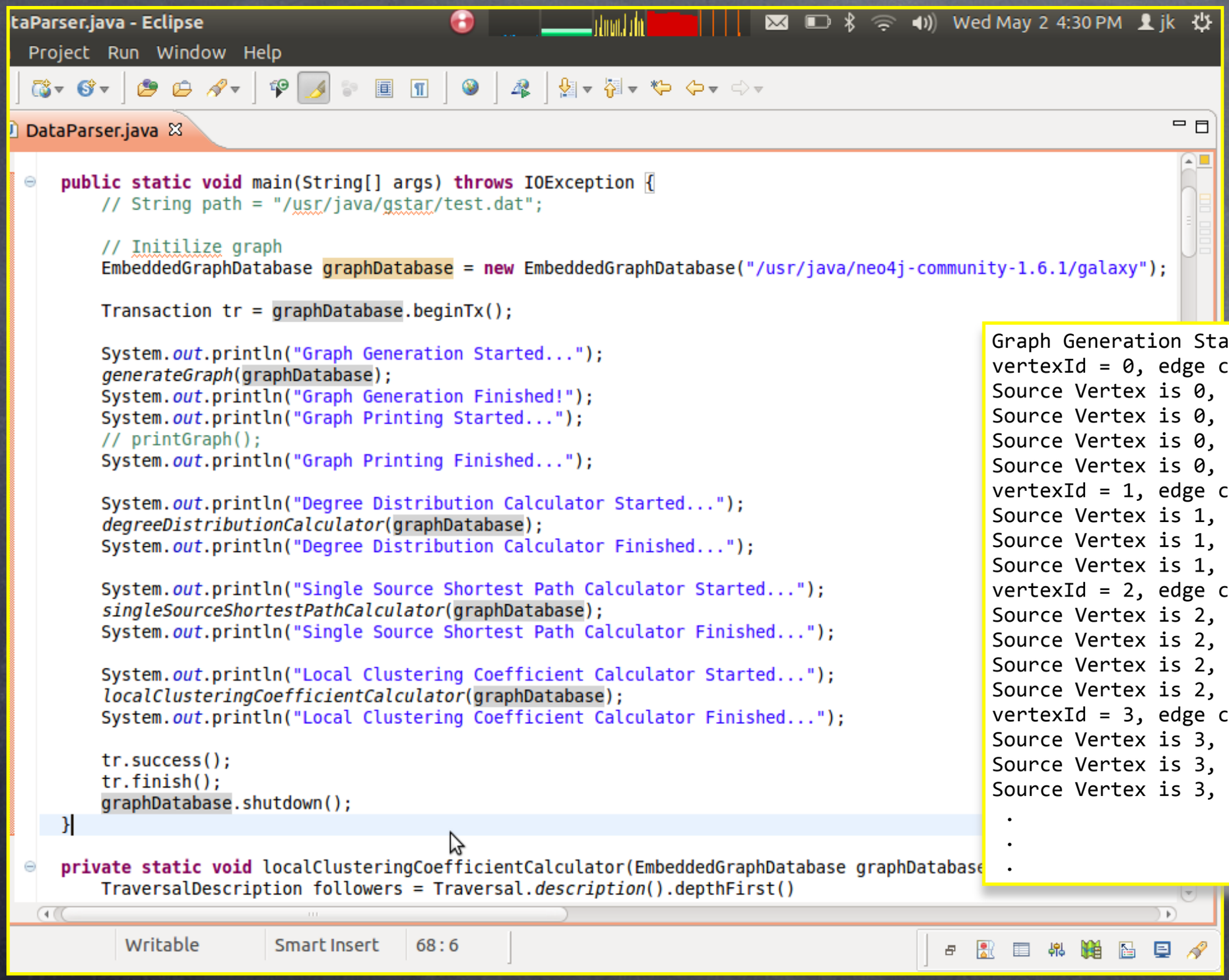
```
0:X
1:X
2:X
3:X
4:XX
5:XXXX
6:XXXXXX
7:XXXX
8:X
```

All edges are have uniform weight = 1.

Path Lengths and Dist.

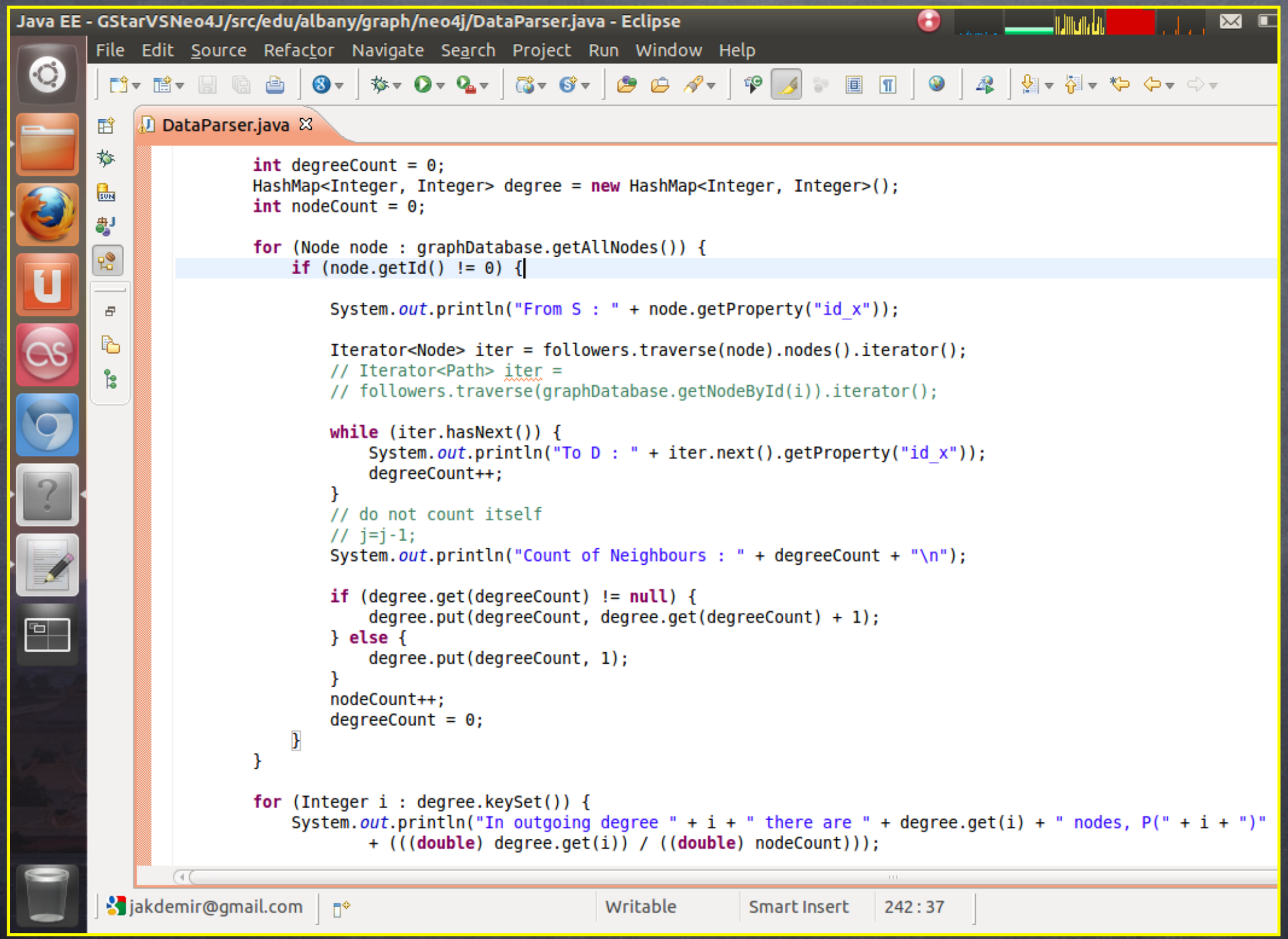
Query time = 65ms

Testing Neo4j - Setting up the graph




```
Graph Generation Started...
vertexId = 0, edge count = 4
Source Vertex is 0, destination vertex is 1
Source Vertex is 0, destination vertex is 3
Source Vertex is 0, destination vertex is 13
Source Vertex is 0, destination vertex is 19
vertexId = 1, edge count = 3
Source Vertex is 1, destination vertex is 0
Source Vertex is 1, destination vertex is 2
Source Vertex is 1, destination vertex is 14
vertexId = 2, edge count = 4
Source Vertex is 2, destination vertex is 1
Source Vertex is 2, destination vertex is 3
Source Vertex is 2, destination vertex is 4
Source Vertex is 2, destination vertex is 7
vertexId = 3, edge count = 3
Source Vertex is 3, destination vertex is 0
Source Vertex is 3, destination vertex is 2
Source Vertex is 3, destination vertex is 12
 .
 .
 .
```

Testing Neo4j - Distribution of Vertex Degrees



Testing Neo4j - Distribution of Vertex Degrees

Neo4j

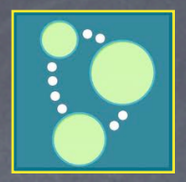
```
Degree Distribution Calculator Started...
From S : 0
To D : 1
To D : 3
To D : 13
To D : 19
To D : 1
To D : 3
To D : 19
Count of Neighbours : 7

From S : 1
To D : 0
To D : 2
To D : 14
To D : 0
To D : 2
To D : 14
Count of Neighbours : 6
 .
 .
 .
In outgoing degree  2 there are 6 nodes, P(2)0.2857142857142857
In outgoing degree  4 there are 3 nodes, P(4)0.14285714285714285
In outgoing degree  5 there are 2 nodes, P(5)0.09523809523809523
In outgoing degree  6 there are 3 nodes, P(6)0.14285714285714285
In outgoing degree  7 there are 4 nodes, P(7)0.19047619047619047
In outgoing degree  8 there are 2 nodes, P(8)0.09523809523809523
In outgoing degree 10 there are 1 nodes, P(10)0.047619047619047616

Degree Distribution Calculator Finished...
```

They Match!

PostgreSQL

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

Testing Neo4j - Clustering Coefficients



VSNeo4J/src/edu/albany/graph/neo4j/DataParser.java - Eclipse

Edit Source Refactor Navigate Search Project Run Window Help

DataParser.java

```
        for (Node node : graphDatabase.getAllNodes()) {
            if (node.getId() != 0) {
                int countOfPathsBetweenNeighbour = 0;
                int countOfNeighbourNodes = 0;

                System.out.println("For S : " + node.getProperty("id_x"));

                Iterator<Node> neighbourSource = followers.traverse(node).nodes().iterator();

                while (neighbourSource.hasNext()) {
                    countOfNeighbourNodes++;
                    Node sourceNode = neighbourSource.next();

                    Iterator<Node> sourceAdjIter = followers.traverse(sourceNode).nodes().iterator();
                    while (sourceAdjIter.hasNext()) {
                        Node sourceAdjNode = sourceAdjIter.next();

                        //Undirected paths between neighbours
                        TraversalDescription followersSpecialForAlan = Traversal.description().depthFirst()
                                .relationships(RelTypes.FOLLOWS, Direction.BOTH).uniqueness(Uniqueness.RELATIONSHIP_GLOBAL)
                                .evaluator(Evaluators.atDepth(1));


                        Iterator<Node> neighbourDestination = followersSpecialForAlan.traverse(node).nodes().iterator();

                        while (neighbourDestination.hasNext()) {
                            Node destinationNode = neighbourDestination.next();

                            if (sourceAdjNode.getProperty("id_x") == destinationNode.getProperty("id_x")) {
                                countOfPathsBetweenNeighbour++;
                                break;
                            }
                        }

                    }
```

kdemir@gmail.com | Writable | Smart Insert | 70 : 61

Testing Neo4j - Clustering Coefficients

Neo4j

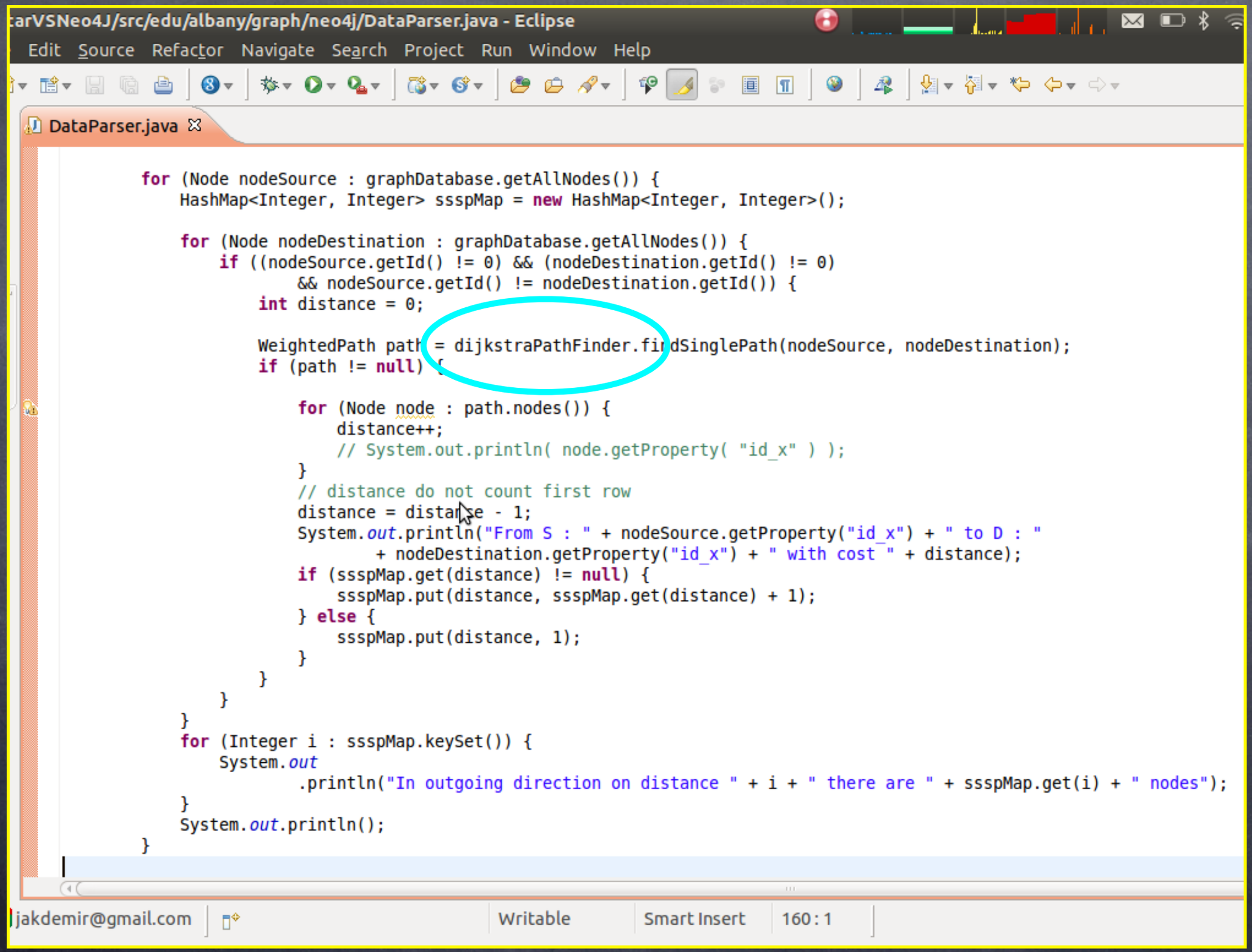
```
Local Clustering Coefficient Calculator Started...
For S :  0 there are 4 neighbours with 0 external paths=> LCC 0.0
For S :  1 there are 3 neighbours with 0 external paths=> LCC 0.0
For S :  2 there are 4 neighbours with 0 external paths=> LCC 0.0
For S :  3 there are 3 neighbours with 0 external paths=> LCC 0.0
For S :  4 there are 3 neighbours with 0 external paths=> LCC 0.0
For S :  5 there are 1 neighbours with 0 external paths=> LCC 0.0
For S :  6 there are 1 neighbours with 0 external paths=> LCC 0.0
For S :  7 there are 4 neighbours with 1 external paths=> LCC 0.25
For S :  8 there are 3 neighbours with 2 external paths=> LCC 0.6666666666666666
For S :  9 there are 2 neighbours with 0 external paths=> LCC 0.0
For S : 10 there are 1 neighbours with 0 external paths=> LCC 0.0
For S : 11 there are 2 neighbours with 0 external paths=> LCC 0.0
For S : 12 there are 3 neighbours with 1 external paths=> LCC 0.3333333333333333
For S : 13 there are 3 neighbours with 2 external paths=> LCC 0.6666666666666666
For S : 14 there are 5 neighbours with 3 external paths=> LCC 0.6
For S : 15 there are 3 neighbours with 4 external paths=> LCC 1.3333333333333333
For S : 16 there are 1 neighbours with 0 external paths=> LCC 0.0
For S : 17 there are 3 neighbours with 1 external paths=> LCC 0.3333333333333333
For S : 18 there are 1 neighbours with 0 external paths=> LCC 0.0
For S : 19 there are 2 neighbours with 0 external paths=> LCC 0.0
For S : 20 there are 1 neighbours with 0 external paths=> LCC 0.0
Local Clustering Coefficient Calculator Finished...
```

PostgreSQL

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

They're close.

Testing Neo4j - Single Source Shortest Path



Testing Neo4j - Single Source Shortest Path

SSSP on the validation graph. Source = Vertex 0

Neo4j (Dijkstra)

```
Single Source Shortest Path Calculator Started...
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
```

PostgreSQL (Bellman-Ford)

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

They Match!

SSSP on the validation graph. Source = Vertex 10

Neo4j (Dijkstra)

```
Single Source Shortest Path Calculator Started...
In outgoing direction on distance 1 there are 1 nodes
In outgoing direction on distance 2 there are 1 nodes
In outgoing direction on distance 3 there are 1 nodes
In outgoing direction on distance 4 there are 2 nodes
In outgoing direction on distance 5 there are 4 nodes
In outgoing direction on distance 6 there are 6 nodes
In outgoing direction on distance 7 there are 4 nodes
In outgoing direction on distance 8 there are 1 nodes
```

PostgreSQL (Bellman-Ford)

sssp_est integer	count bigint
0	1
1	1
2	1
3	1
4	2
5	4
6	6
7	4
8	1

They Match!

Validating G* - The Graph

Galaxy.dat

```
0 4
 1
 3
 13
 19

1 3
 0
 2
 14

2 4
 1
 3
 4
 7

3 3
 0
 2
 12

4 3
 2
 5
 6

5 1
 4

6 1
 4

7 4
```

Servers.txt

127.0.0.1:12001

Loading the test data into G*

```
Alan:~ >./import_graph.sh -input-data-file galaxy.dat -server-list servers.txt -servers 1 -new-edges-per-graph 1000 -graphs 1
% "*(10.0.0.7:10000)" started!
% servers are ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% "*(10.0.0.7:10000)" is initialized to "0(10.0.0.7:10000)"
% GraphManager@0 is loading data.
% 1335380095536: "0(10.0.0.7:10000)" is trying to start ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335380096174: "0(10.0.0.7:10000)" broadcasts information about peers ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335380096186: "0(10.0.0.7:10000)" requests to remove all data at ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335380096224: "0(10.0.0.7:10000)" is collecting statistics.
operator statistics:
server statistics:
server 1 - cpu utilization: 0.00%, memory used: 10MBs, actual caching ratio: 0.0
% 1335380096260: "0(10.0.0.7:10000)" requests to add a graph at ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335380096265: "0(10.0.0.7:10000)" is adding 73 vertices.
0.0% done
edges: 1000
% 1335380096323: "0(10.0.0.7:10000)" requests to chekcpoint the in-memory state of ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335380096323: "0(10.0.0.7:10000)" checkpoints its in-memory state.
% started serialization
% ended serialization in 2 ms
% [id] [graphs per cgi] [index size] [data size] [graphs] [subgraphs]
0 -1 463 0 1 0
% 1335380096326: "0(10.0.0.7:10000)" has completed checkpointing its in-memory state.
% edges added: 1000
% [graphs per cgi] [# edges] [insertion time] [total time]
16 1000 60 66
% 1335380096329: "0(10.0.0.7:10000)" is collecting statistics.
operator statistics:
server statistics:
server 1 - cpu utilization: 5.55%, memory used: 13MBs, actual caching ratio: 1.0
% 1335380096336: "0(10.0.0.7:10000)" is shutting down ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
Coordinator % 1335380096706: "0(10.0.0.7:10000)" has shutdown all.
% 1335380096706: "0(10.0.0.7:10000)" is shutting down.
% 1335380096728: "0(10.0.0.7:10000)" has shut down.
```

Why 73 vertices? There are 21 in the data file.

Validating G* - Distribution of Vertex Degrees

Distribution Degrees Query (.plan) in G*sql

```
%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]);
```

Distribution Degrees execution in G*

```
Alan:~ >./run_query.sh -query distribution_degrees.plan -server-list servers.txt -servers 1
% "*(10.0.0.7:10000)" started!
% servers are ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% "*(10.0.0.7:10000)" is initialized to "0(10.0.0.7:10000)"
% GraphManager@0 is loading data.
% index loaded, type = class graphdb.index.ExtendedCompactGraphIndex
% GraphManager 0 finished loading data.
% 1335384727056: "0(10.0.0.7:10000)" is trying to start ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335384727692: "0(10.0.0.7:10000)" broadcasts information about peers ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335384727720: "0(10.0.0.7:10000)" has launched [vertex@*, degree@*, histogram@*, union@1, histogramMerge@1]
% query results:
[[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, 100):0, [graph0]]
% all results in 24 ms.
% 1335384727744: "0(10.0.0.7:10000)" is collecting statistics.
operator statistics:
degree@1 (completed) - input operators: [[vertex, 1]], input record counts: {0=21}, output record count: {0=21}
histogram@1 (completed) - input operators: [[degree, 1]], input record counts: {0=21}, output record count: {0=1}
histogramMerge@1 (completed) - input operators: [[union, 1]], input record counts: {0=1}, output record count: {0=1}
union@1 (completed) - input operators: [[histogram, 1]], input record counts: {0=1}, output record count: {0=1}
vertex@1 (completed) - input operators: [], input record counts: {0=21}, output record count: {0=21}
server statistics:
server 1 - cpu utilization: 0.00%, memory used: 15MBs, actual caching ratio: 1.0
% 1335384727782: "0(10.0.0.7:10000)" is shutting down ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
Coordinator % 1335384728229: "0(10.0.0.7:10000)" has shutdown all.
```

Query Results.

Validating G* - Distribution of Vertex Degrees

Distribution Degrees Query results from G*

```
% query results:
[[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, 100):0, [graph0]]
% all results in 24 ms.
```

Distribution Degrees Test Data 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
```

This one is correct!

```
[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
```

and PostgreSQL results

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

and Neo4j results

```
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
```

Validating G* - Clustering Coefficients

Clustering Coefficients query (.plan) in G*sql

```
%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]);
```

Execution

```
Alan:~ >./run_query.sh -query distribution_coefficients.plan -server-list servers.txt -servers 1
% "*(10.0.0.7:10000)" started!
% servers are ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% "*(10.0.0.7:10000)" is initialized to "0(10.0.0.7:10000)"
% GraphManager@0 is loading data.
% index loaded, type = class graphdb.index.ExtendedCompactGraphIndex
% GraphManager 0 finished loading data.
% 1335386380509: "0(10.0.0.7:10000)" is trying to start ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335386381145: "0(10.0.0.7:10000)" broadcasts information about peers ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335386381174: "0(10.0.0.7:10000)" has launched [vertex@*, c_coeff@*, histogram@*, union@1, histogramMerge@1]
% query results:
[[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, [graph0]]
% all results in 141 ms.
% 1335386381315: "0(10.0.0.7:10000)" is collecting statistics.
operator statistics:
c_coeff@1 (completed) - input operators: [[vertex, 1]], input record counts: {0=21}, summary message count: 52, output record count: {0=20}
histogram@1 (completed) - input operators: [[c_coeff, 1]], input record counts: {0=20}, output record count: {0=1}
histogramMerge@1 (completed) - input operators: [[union, 1]], input record counts: {0=1}, output record count: {0=1}
union@1 (completed) - input operators: [[histogram, 1]], input record counts: {0=1}, output record count: {0=1}
vertex@1 (completed) - input operators: [], input record counts: {0=21}, output record count: {0=21}
server statistics:
server 1 - cpu utilization: 0.00%, memory used: 17MBs, actual caching ratio: 1.0
% 1335386381353: "0(10.0.0.7:10000)" is shutting down ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
Coordinator % 1335386381777: "0(10.0.0.7:10000)" has shutdown all. . .
```

Query Results.

Validating G* - Clustering Coefficients

Clustering Coefficients results from G*

```
% query results:
[[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, [graph0]]
% all results in 141 ms.
```

Clustering Coefficient Test Data Traits

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

← This one is correct! →

```
[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
```

and PostgreSQL results

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

Validating G* - Single Source Shortest Path

SSSP query (.plan) in G*sql

```
%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]);
```

Execution

```
Alan:~ >./run_query.sh -query distribution_distances.plan -server-list servers.txt -servers 1
% "*(10.0.0.7:10000)" started!
% servers are ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% "*(10.0.0.7:10000)" is initialized to "0(10.0.0.7:10000)"
% GraphManager@0 is loading data.
% index loaded, type = class graphdb.index.ExtendedCompactGraphIndex
% GraphManager 0 finished loading data.
% 1335984026622: "0(10.0.0.7:10000)" is trying to start ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335984027279: "0(10.0.0.7:10000)" broadcasts information about peers ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
% 1335984027309: "0(10.0.0.7:10000)" has launched [min_dist@*, histogram@*, union@1, histogramMerge@1]
% query results:
% 1335984032454: "0(10.0.0.7:10000)" has recevied a vote on operator min_dist from server 1.
[[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, [graph0]]
% all results in 6052 ms.
% 1335984033359: "0(10.0.0.7:10000)" is collecting statistics.
operator statistics:
histogram@1 (completed) - input operators: [[min_dist, 1]], input record counts: {0=21}, output record count: {0=1}
histogramMerge@1 (completed) - input operators: [[union, 1]], input record counts: {0=1}, output record count: {0=1}
min_dist@1 (completed) - input operators: [], input record counts: {}, summary message count: 35, output record count: {0=21}
union@1 (completed) - input operators: [[histogram, 1]], input record counts: {0=1}, output record count: {0=1}
server statistics:
server 1 - cpu utilization: 0.00%, memory used: 16MBs, actual caching ratio: 1.0
% 1335984033409: "0(10.0.0.7:10000)" is shutting down ["0(10.0.0.7:10000)", "1(127.0.0.1:12001)"]
Coordinator % 1335984033778: "0(10.0.0.7:10000)" has shutdown all.
```

Query Results.

This looks right. Why was it 73 before?

Validating G* - Single Source Shortest Path

SSSP results from G*sql - SSSP from vertex 0

```
% query results:
% 1335984032454: "0(10.0.0.7:10000)" has recevied a vote on operator min_dist from server 1.
[[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, [graph0]]
% all results in 6052 ms.
```

PostgreSQL and Neo4j results (from vertex 0)

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

```
 0 - 1:X
 1 - 2:XXXX
 2 - 4:XXXXXXXXX
 4 - 8:XXXXXXX
 8 -16:
16-32:
32-64:
64-128:
```

```
 0 - 1:X
 1 - 2:XXXX
 2 - 4:XXXXXXX
 4 - 8:XXXXXXXX
 8 -16:X
16-32:
32-64:
64-128:
```

```
[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
```

These distributions are pretty close.

Query time = 66ms

Query time = 6052ms

But these times are not!

Validating G* - Single Source Shortest Path

SSSP results from G*sql - SSSP from vertex 10

```
% query results:
% 1335990154975: "0(10.0.0.7:10000)" has recevied a vote on operator min_dist from server 1.
[[0.0, 1.0):1, [1.0, 2.0):1, [2.0, 4.0):2, [4.0, 8.0):14, [8.0, 16.0):3, [16.0, 32.0):0, [32.0, 64.0):0, [64.0, 128.0):0, [graph0]]
% all results in 6050 ms.
```

PostgreSQL and Neo4j results (from vertex 0)

sssp_est integer	count bigint
0	1
1	1
2	1
3	1
4	2
5	4
6	6
7	4
8	1

```
 0 - 1:X
 1 - 2:X
 2 - 4:XX
 4 - 8:XXXXXXXXXXXXXXX
 8 -16:X
 16-32:
 32-64:
64-128:
```

```
 0 - 1:X
 1 - 2:X
 2 - 4:XX
 4 - 8:XXXXXXXXXXXXXX
 8 -16:XXX
 16-32:
 32-64:
64-128:
```

```
[0.0,    1.0): 1
[1.0,    2.0): 1
[2.0,    4.0): 2
[4.0,    8.0):14
[8.0,   16.0): 3
[16.0,  32.0):
[32.0,  64.0):
[64.0, 128.0):
```

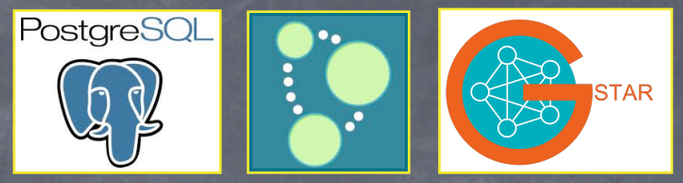
These distributions are pretty close.

Query time = 65ms

Query time = 6050ms

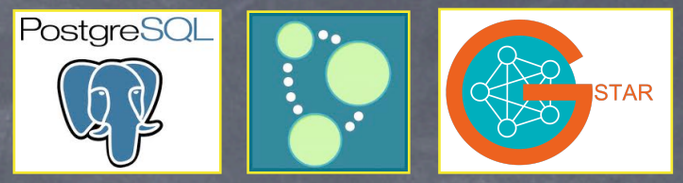
But these times are not!

Summary



- We provided an easy-to-validate test graph.
- We validated three graph metrics (distribution of degrees, clustering coefficients, and single-source shortest paths) against our test graph by hand and in two other environments: PostgreSQL and Neo4j.
- For SSSP we also validated our code against a CLRS reference graph to be extra careful.
- The PostgreSQL and Neo4j results matched each other and our manual results.
- G* results did not match PostgreSQL, Neo4j, or manual results.

Conclusions



- That G* results did not match PostgreSQL, Neo4j, or manual results demonstrates the need for a validation framework in order to measure G*'s accuracy as it evolves.
- This project is a good start, but it's only the beginning of such a framework.
- Future Work

More and different validation graphs: larger, dense, sparse

More trait algorithms

Automation, Logging, and Analysis over time

Investigate OpenStreetMap data with PostGIS -> G*