Analyzing Open Shortest Path First (OSPF) Networks with Neo4j and Cypher

William John Holden

7/26/2021

1 Introduction

Open Shortest Path First (OSPF) is a dynamic routing protocol used by network routers to communicate reachability information (Moy 1998). OSPF is considered an interior gateway routing protocol (IGP); it is used to route traffic within an organization or campus, but is not used for Internet-scale routing. There are multiple versions of OSPF, but OSPF version 2 is the most commonly used. OSPFv2 is specified the Internet Engineering Task Force's (IETF) Request for Comment (RFC) 2328.

A *distance-vector* routing protocol is constructed on a simple invariant: given multiple "directions" a packet might be forwarded, the router operating a distance-vector routing protocol forwards packets in the direction of least distance (Malkin 1998). Suppose a router seeking destination u_n has paths $P_1 = (u_i, u_j, \dots, u_n)$ and $P_2 = (u_k, u_l, \dots, u_n)$. If the router has information to show $\delta(P_1) < \delta(P_2)$, then there is no need to consider the structure of P_2: the router simply needs to forward the packet to u_i.

OSPF is a *link-state* routing protocol, *not* a distance-vector routing protocol (Moy 1998). A link-state routing protocol requires a complete topology table, known as a Link State Database (LSDB). Using the LSDB, the router computes the shortest path tree from itself to all destination networks. Traditionally, OSPF routers use some variation of Dijkstra's algorithm. The shortest path tree is then used to populate the router's routing table and forwarding table.

Cisco routers expose the raw contents of the LSDB to network managers on the Simple Network Management Protocol (SNMP). The management information base (MIB) for OSPFv2, named "OSPF-MIB," is specified in RFC 4750 (Joyal et al. 2006). Within OSPF-MIB, there is an object named `ospfLsdbTable` (1.3.6.1.2.1.14.4), which contains a list of `ospfLsdbEntry` objects (1.3.6.1.2.1.14.4.1) with the property `ospfLsdbAdvertisement` (1.3.6.1.2.1.14.4.1.8).

In this project, we use a Java program to:

1. Gather `ospfLsdbAdvertisement` entries from a router over SNMPv3.
2. Launch a Neo4j 4.3 graph database embedded in the application.
3. Insert the LSDB entries into the graph database.
4. Analyze the network graph using Neo4j's functions.

Areas of particular interest for the analysis include:

1. Network visualization.
2. Apply domain knowledge of common problems in computer networking and try to identify these problems using Cypher queries in Neo4j.
3. Automatic identification of "important" nodes, where importance is loosely defined here to signify that an individual router failure would cause a significant disruption or tear in a network.

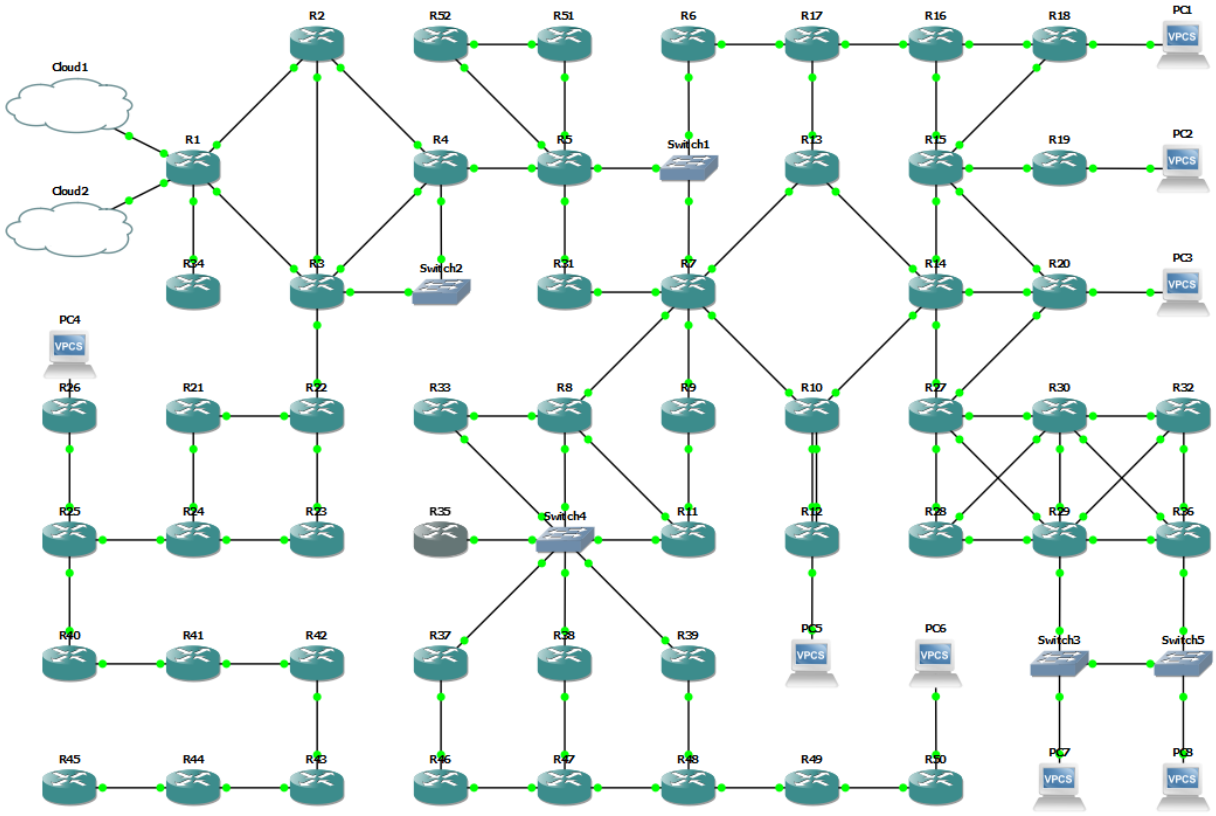

Figure 1: Network topology simulated for this project using GNS3

2 Project

```
library(neo4r)
library(tidyverse)
library(knitr)
```

```
con <- neo4j_api$new(url = "http://localhost:7474", user = "", password = "")
```

2.1 Embedded Neo4j Graph Database

The community edition of the Neo4j graph database system is freely available to Java developers everywhere using Maven. One method of including Neo4j in a Java project is to simply include the Neo4j `jar` files in the Java runtime environment's class path.

In this project, we include the following Maven libraries as dependencies using the IntelliJ IDE:

- `org.neo4j.gds:core:1.6.3`
- `org.neo4j.gds:proc:1.6.3`
- `org.neo4j:neo4j:4.3.2`

We also include the `org.reflections:reflections:0.9.12` library to assist loading the Graph Data Science (GDS) libraries.

A helper method creates a temporary directory for the database, starts the database server, and provides settings for the database server that are necessary for GDS.

```
public static GraphDatabaseService startDb() throws IOException {
    final Path databaseDirectory = Files.createTempDirectory(DEFAULT_DATABASE_NAME);

    Map<String, String> settings = new HashMap<>();
    settings.put("dbms.security.procedures.unrestricted", "jwt.security.*,gds.*,apoc.*");
    settings.put("dbms.security.procedures.whitelist", "gds.*");
    settings.put("dbms.connector.bolt.enabled", "true");
    settings.put("dbms.connector.http.enabled", "true");

    final DatabaseManagementService managementService = new DatabaseManagementServiceBuilder(
            databaseDirectory).
            setConfigRaw(settings).
            build();
    final GraphDatabaseService graphDb = managementService.database(DEFAULT_DATABASE_NAME);
    registerShutdownHook(managementService);

    return graphDb;
}
```

The `registerShutdownHook` method is copied verbatim from Neo4j embedded documentation. This method causes the JVM to invoke the `shutdown()` method on the Neo4j database management service as the main thread terminates.

```
private static void registerShutdownHook( final DatabaseManagementService managementService )
{
    // Registers a shutdown hook for the Neo4j instance so that it
    // shuts down nicely when the VM exits (even if you "Ctrl-C" the
    // running application).
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
```

```
        public void run()
        {
            managementService.shutdown();
        }
    } );
}
```

For the OSPF graph, a single `LINKED` edge is defined as a Neo4j relationship type.

```
private enum RelTypes implements RelationshipType {
    LINKED
}
```

Finally, graph algorithms from GDS are not included in the Neo4j embedded database automatically. The procedures (which extend `BaseProc`) and functions must be registered to the grap database service's procedures registry. This method is a simplified version of the methods discussed in GDS issue #91.

```
public static GraphDatabaseService registerGds(GraphDatabaseService graphDb)
        throws KernelException {
    final GlobalProcedures proceduresRegistry = ((GraphDatabaseAPI) graphDb)
            .getDependencyResolver().resolveDependency(GlobalProcedures.class,
                    DependencyResolver.SelectionStrategy.SINGLE);

    final Set<Class<? extends BaseProc>> procedures = new Reflections("org.neo4j.graphalgo").
            getSubTypesOf(BaseProc.class);
    procedures.addAll(new Reflections("org.neo4j.gds.embeddings").getSubTypesOf(BaseProc.class));
    procedures.addAll(new Reflections("org.neo4j.gds.paths").getSubTypesOf(BaseProc.class));

    for (Class<? extends BaseProc> procedureClass : procedures) {
        proceduresRegistry.registerProcedure(procedureClass);
    }

    final Class[] functionsToRegister = {
            AsNodeFunc.class,
            NodePropertyFunc.class,
            VersionFunc.class };

    for (Class f : functionsToRegister) {
        proceduresRegistry.registerFunction(f);
    }

    return graphDb;
}
```

The `registerGds` method uses the `Reflections` library to build a set of `BaseProc` classes in the `org.neo4j.graphalgo`, `org.neo4j.gds.embeddings`, and `org.neo4j.gds.paths` packages. It then adds each of these classes to the graph database service's procedures registry.

Functions must be handled differently, as they have no common supertype. Rather, a static list of three classes that are useful GDS functions (notably, the familiar `gds.util.asNode` function) are registered as functions. Finally, the mutated graph database service is returned to the caller.

2.2 Database Schema

The database uses three Neo4j labels to describe the objects of the OSPF LSDB:

- `ROUTER`: a router, which can be connected to other routers directly, transit networks, and stub networks. The `name` of a router is its `router-id`, which is usually chosen from the router's IP addresses but can

also be manually configured.

- `NETWORK`: a transit network that connects two or more routers on a multiple-access link. The `name` of a network is its prefix and subnet mask in classless inter-domain routing (CIDR) notation, such as `198.51.100.0/24`.
- `STUB`: any subnet that the router connects to, except for transit networks. Two good examples are loopback host addresses (only one router can be connected to a loopback) and customer-facing VLANs with no other router.

A `ROUTER` can connect to another `ROUTER`, a `NETWORK`, and a `STUB`. `NETWORK` and `STUB` nodes can only attach to `ROUTER` nodes. Interestingly, the `db.schema.visualization` procedure produces an incorrect result by showing relations that do not exist.

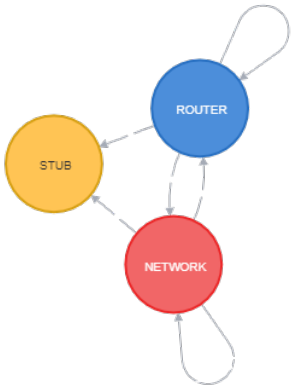

Figure 2: `CALL db.schema.visualization`

2.3 SNMPv3 Queries

This program uses a library called Tnm4j as a simplified API into SNMP4J. SNMP4J is a complex and difficult application to operate. Tnm4j chooses reasonable defaults and is much simpler for application developers to work with. The entire SNMP code is as follows:

```
private static List<Lsa> walkOspfLsdbMib(String address, String username,
                                         String authPassword, String privPassword)
                                         throws IOException {
    final Mib mib = MibFactory.getInstance().newMib();
    mib.load("SNMPv2-MIB");
    mib.load("OSPF-MIB");

    SecurityProtocols.getInstance().addAuthenticationProtocol(new AuthSHA());
    final SimpleSnmpV3Target target = new SimpleSnmpV3Target();
    target.setAddress(address);
    target.setSecurityName(username);
    target.setAuthType(SnmpV3Target.AuthType.SHA);
    target.setPrivType(SnmpV3Target.PrivType.AES128);
    target.setAuthPassphrase(System.getProperty("tnm4j.agent.auth.password", authPassword));
    target.setPrivPassphrase(System.getProperty("tnm4j.agent.priv.password", privPassword));

    final List<Lsa> lsdb = new ArrayList<>();
    try (SnmpContext context = SnmpFactory.getInstance().newContext(target, mib)) {
        final SnmpWalker<VarbindCollection> walker = context.walk(1, "sysName",
                "ospfLsdbAdvertisement");
        VarbindCollection row = walker.next().get();
```

```
        while (row != null) {
            final byte[] lsa = (byte[]) row.get("ospfLsdbAdvertisement").toObject();
            lsdb.add(Lsa.getInstance(lsa));
            row = walker.next().get();
        }
    }
    return lsdb;
}
```

This method gathers the `ospfLsdbAdvertisement` values as byte strings and instantiates an internal `Lsa` object. The `Lsa` object is subclassed as `RouterLsa` and `NetworkLsa`, which are distinguished by a byte within the `ospfLsdbAdvertisement` byte string.

2.4 Constraints

Three constraints guarantee that all three node types have distinct `name` properties. The constraints also implicitly define indicies, which should improve search times.

```
private static final String[] CONSTRAINTS = {
        "CREATE CONSTRAINT IF NOT EXISTS ON (r:ROUTER) ASSERT r.name IS UNIQUE",
        "CREATE CONSTRAINT IF NOT EXISTS ON (n:NETWORK) ASSERT n.name IS UNIQUE",
        "CREATE CONSTRAINT IF NOT EXISTS ON (s:STUB) ASSERT s.name IS UNIQUE"
};

private static void defineConstraints(GraphDatabaseService graphDb) {
    try (Transaction tx = graphDb.beginTx()) {
        Arrays.asList(CONSTRAINTS).forEach(c -> tx.execute(c));
        tx.commit();
    }
}
```

2.5 Inserting Data

Having gathered the OSPF router's LSDB using SNMPv3, the program finally inserts nodes and edges into the Neo4j graph database. The code is somewhat repetitive and is not shown in its entirety below, but representative exerpts are shown as follows.

First, the `ROUTER` nodes are created directly using the `createNode` Java method.

```
private static void createNetworks(GraphDatabaseService graphDb,
                                   Collection<NetworkLsa> networks) {
    try (Transaction tx = graphDb.beginTx()) {
        networks.forEach(lsa -> {
            Node node = tx.createNode();
            node.addLabel(NETWORK);
            final String prefix = lsa.getPrefix();
            node.setProperty("name", prefix);
        });
        tx.commit();
    }
}
```

Neo4j requires a *transaction* to be opened, committed, and closed when writing to the database. Closing the transaction is implicit in Java 7 and layer using the `try`-with-resources syntax. `NETWORK` nodes are created in a similar method, which is named `createNetworks`.

A more complicated `MERGE` statement relates `ROUTER` nodes to one another. The following method constructs

a parameterized `MERGE` statement which finds the two routers and creates a weighted relationship between them. Note that these relationships are intentionally directed; OSPF routers advertise their connection to one another as directed edges.

```
private static void connectRouters(GraphDatabaseService graphDb,
                                   Collection<RouterLsa> routers) {
    String queryString = "MATCH (src:ROUTER {name:$src})\n" +
            "MATCH (dst:ROUTER {name:$dst})\n" +
            "MERGE (src)-[:LINKED {cost:$cost}]->(dst)";
    try (Transaction tx = graphDb.beginTx()) {
        routers.forEach(src -> {
            src.getAdjacentRouters().forEach((dst, metric) -> {
                Map<String, Object> parameters = new HashMap<>();
                parameters.put("src", src.routerId.getHostAddress());
                parameters.put("dst", dst.getHostAddress());
                parameters.put("cost", metric);
                tx.execute(queryString, parameters);
            });
        });
        tx.commit();
    }
}
```

The `connectNetworks` similarly connects `NETWORK` objects to `ROUTER` objects.

The `connectTransport` method is long and inelegant. The `ROUTER` node is not easily matched to the `NETWORK` node. In an extremely large network, the `NETWORK` for a `ROUTER` could be identified in logarithmic time (using a binary search tree) or constant time (constant in relation to the length of an IP address, using a trie). For small networks, it is acceptably fast to perform a linear search through all `NetworkLsa` objects to identify the appropriate `NETWORK` node.

Finally, the `ROUTER` nodes are connected to their `STUB` networks. This Cypher query is interesting because it uses one `MATCH` clause and two `MERGE` clauses to identify the `ROUTER`, match or create the `STUB`, and relate the two.

```
private static void connectStubs(GraphDatabaseService graphDb,
                                 Collection<RouterLsa> routers) {
    String queryString = "MATCH (src:ROUTER {name:$src})\n" +
            "MERGE (dst:STUB {name:$dst})\n" +
            "MERGE (src)-[:LINKED {cost:$cost}]->(dst)";
    try (Transaction tx = graphDb.beginTx()) {
        routers.forEach(src -> {
            src.getStubs().forEach((dst, metric) -> {
                Map<String, Object> parameters = new HashMap<>();
                parameters.put("src", src.routerId.getHostAddress());
                parameters.put("dst", dst);
                parameters.put("cost", metric);
                tx.execute(queryString, parameters);
            });
        });
        tx.commit();
    }
}
```

The `main` method creates the database by invoking each of these methods.

```
final List<Lsa> lsdb = walkOspfLsdbMib(args[0], args[1], args[2], args[3]);
```

```
final List<RouterLsa> routers = lsdb.stream().filter(l -> l instanceof RouterLsa)
        .map(l -> (RouterLsa) l).collect(Collectors.toList());
final List<NetworkLsa> networks = lsdb.stream().filter(l -> l instanceof NetworkLsa)
        .map(l -> (NetworkLsa) l).collect(Collectors.toList());

createRouters(graphDb, routers);
createNetworks(graphDb, networks);
connectRouters(graphDb, routers);
connectNetworks(graphDb, networks);
connectTransport(graphDb, routers, networks);
connectStubs(graphDb, routers);
```

2.6 Execution

When the `OspfAnalyzer` program runs, it accepts the router IP address and SNMPv3 username, authorization password, and privacy passwords as four command-line arguments. The program initializes the Neo4j graph database, gathers the OSPF LSDB over SNMPv3, inserts the network data into the database, and then pauses. The embedded Neo4j graph database is accessible using the Bolt protocol, which is most easily accessed from Neo4j Browser. Simply connect to `neo4j://localhost:7687` in Neo4j Browser and leave both the username and password blank. Alternatively, any Neo4j driver (such as `neo4r`, used in this document) can connect to the graph database using Bolt or HTTP.

The project is developed in Java 11 but compiled with a language level of Java 8. This is for backward-compatibility to commonly-used Java Runtime Environments (JRE) seen in large enterprises.

3 Analysis

3.1 Network Visualization

Neo4j Browser provides a powerful graph visualization feature. Nodes can be colored by label. Both nodes and relationships can show text labels based on property values. The user can drag vertices to manually arrange the graph. Neo4j Browser's graph visualization is easier to use than GraphViz, although possibly not as rich in features.

The network visualization of all three node types (`ROUTER`, `NETWORK`, and `STUB`) is too large and noisy to comprehend.

A simpler visualization is to match only `ROUTER` and `NETWORK` labels. This gives a smaller graph that makes more sense to network operators.

Network documentation is a tedious, difficult, and error-prone activity for network operators everywhere. Networks frequently change, and the diagram for a large network may already become incorrect before a technician completes the drawing. Though the Neo4j graph visualization is a good start, it fails to capture information that network operators would typically annotate on a node, such as:

- Make, model, and serial number
- Hostname
- Building, floor, and room numbers
- Interface names
- Circuit names

Network diagrams are most commonly built in tools like Microsoft Visio. Though automatic network diagrams are not likely to displace manual diagrams, it can still be a useful troubleshooting and verification tool. Also, the automatic network diagram could be constructed as the basis for drawing a manual network diagram in a tool like Visio.

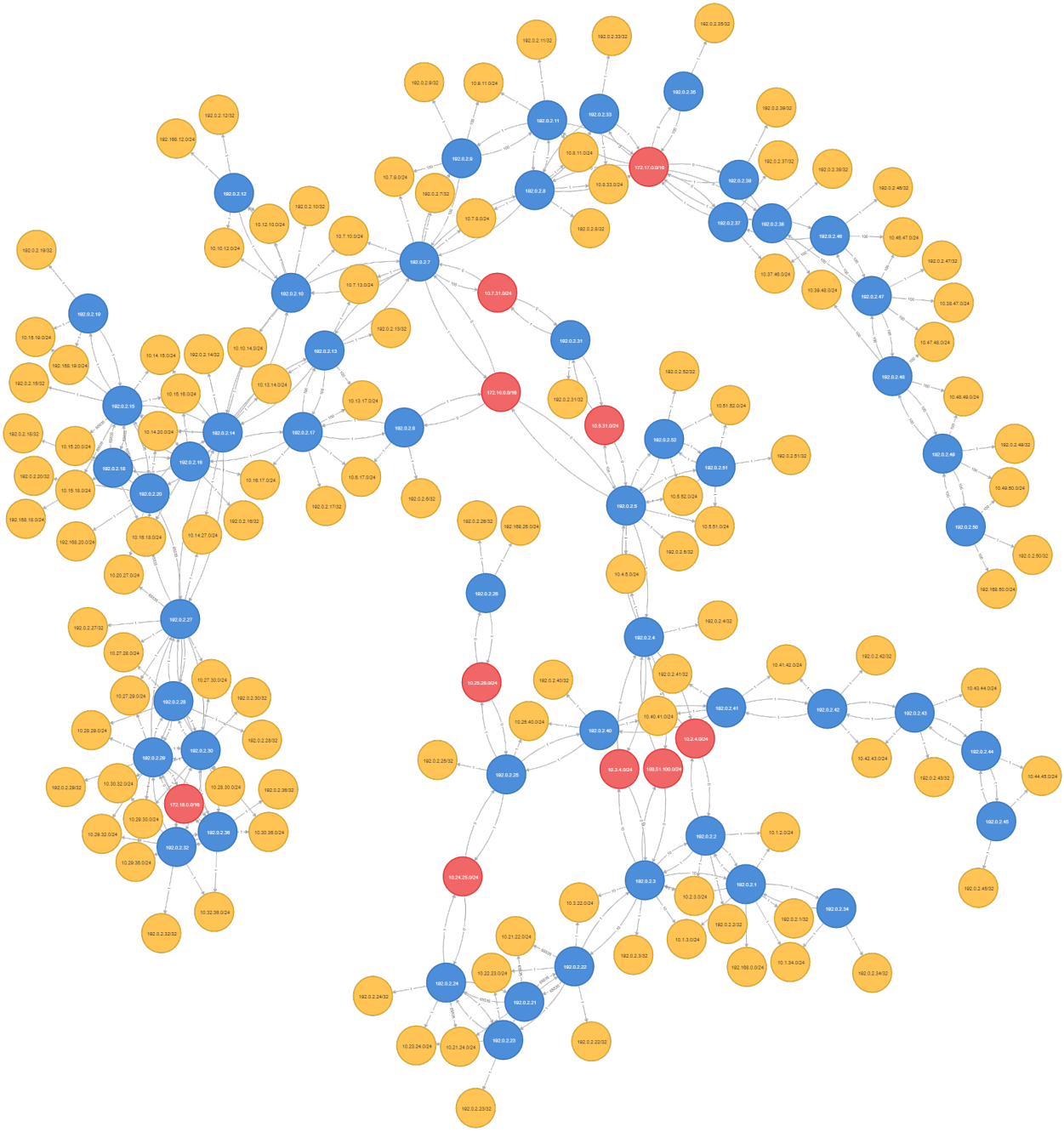

Figure 3: `MATCH n RETURN n`

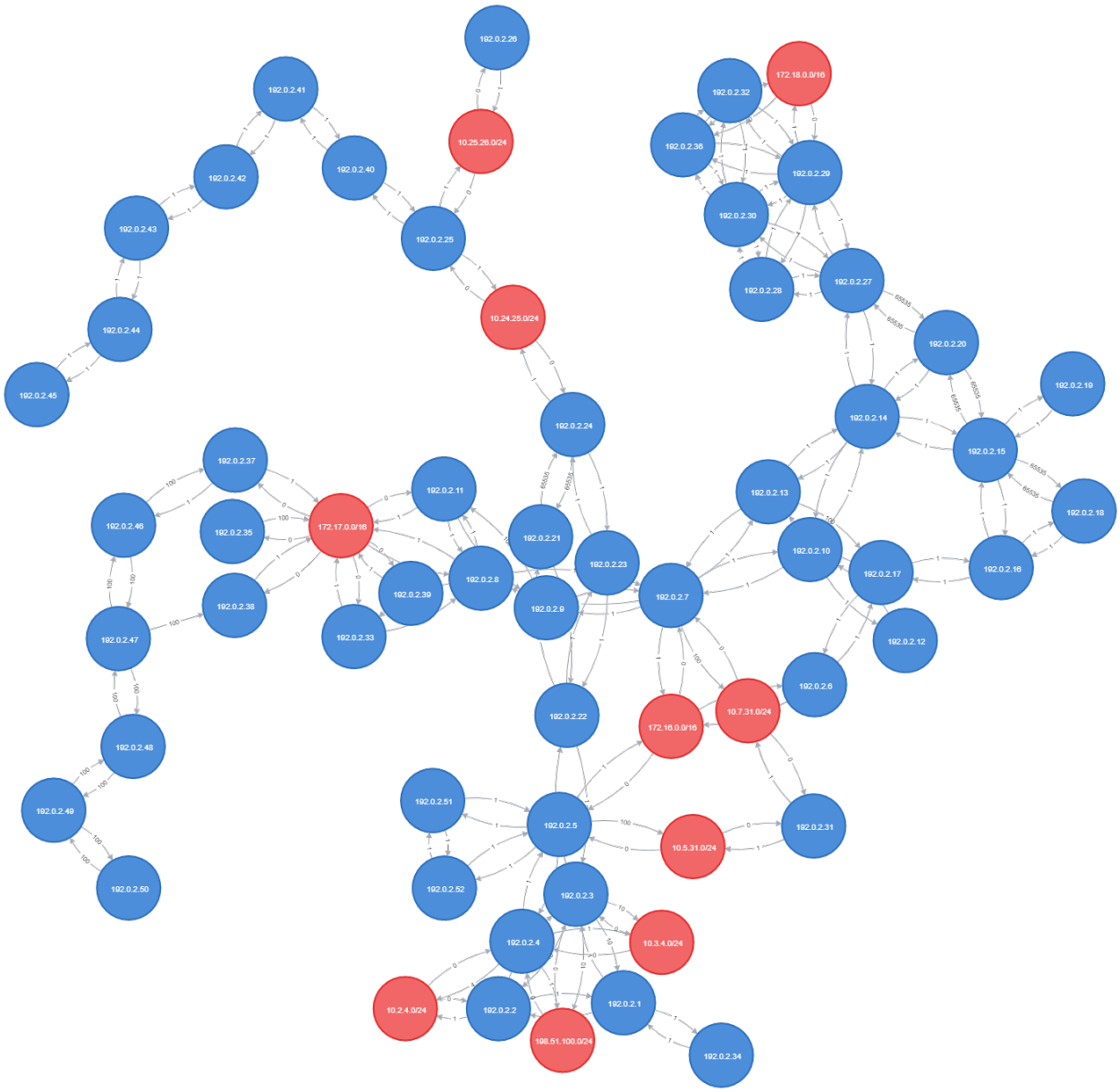

Figure 4: `MATCH (n) WHERE n:ROUTER OR n:NETWORK RETURN n`

3.2 Domain Knowledge

The graph database provides many opportunities for a network technician to discover problems that might have otherwise been quite difficult.

The OSPF LSDB represents a direct connection between two routers u and v with two edges: $e_1 = (u, v)$ and $e_2 = (v, u)$. The costs, $w(e_1)$ and $w(e_2)$, may or may not be equal. Mismatched costs may lead to *asymmetric routing*, where the path $p_1 = u \rightsquigarrow v$ is not the reverse of $p_2 = v \rightsquigarrow u$. Asymmetric routes can cause many problems on computer networks. If only half of a connection traverses a firewall, then the firewall incorrectly deny traffic. Also, there may be a performance-related reason to prefer one path over others.

The following Cypher query finds point-to-point connections between routers that have unequal cost.

```
tmp = "
MATCH (u:ROUTER)-[e1:LINKED]->(v:ROUTER)
MATCH (v)-[e2:LINKED]->(u)
WHERE e1.cost <> e2.cost AND id(u)<id(v)
RETURN u.name, v.name, e1.cost, e2.cost
" %>% call_neo4j(con)
kable(tibble(u = tmp$u.name$value, v = tmp$v.name$value,
             e1 = tmp$e1.cost$value, e2 = tmp$e2.cost$value))
```

u	v	e1	e2
192.0.2.1	192.0.2.3	1	10
192.0.2.2	192.0.2.3	1	10
192.0.2.3	192.0.2.22	10	1
192.0.2.7	192.0.2.9	1	100
192.0.2.9	192.0.2.11	100	1
192.0.2.37	192.0.2.46	1	100

```
rm(tmp)
```

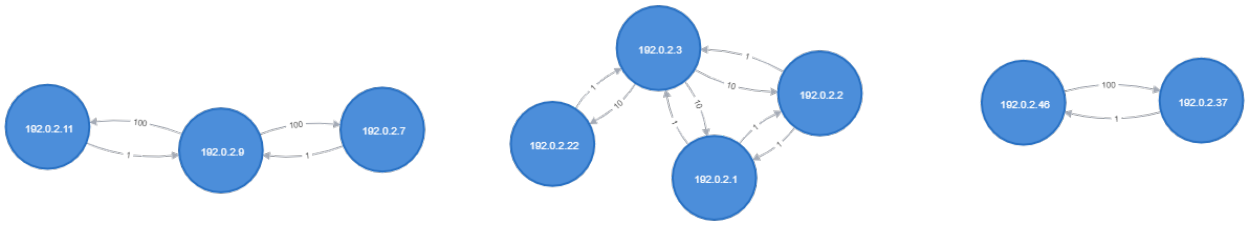

Figure 5: `MATCH (u:ROUTER)-[e1:LINKED]->(v:ROUTER) MATCH (v)-[e2:LINKED]->(u) WHERE e1.cost <> e2.cost AND id(u)<id(v) RETURN u, v`

Similarly, the costs of the many routers attached to a broadcast, multiple-access transit network need not be equal. The following Cypher query uses the standard deviation to test if all costs are equal; if the standard deviation is nonzero, then at least one cost is unequal.

```
tmp = "
MATCH (n:NETWORK)<-[e:LINKED]-(:ROUTER)
WITH n.name AS Network, COUNT(e) AS Connections, stDev(e.cost) AS s
WHERE s <> 0
RETURN Network, Connections
" %>% call_neo4j(con)
kable(tibble(Network = tmp$Network$value, Connections = tmp$Connections$value))
```

Network	Connections
10.2.4.0/24	2
10.3.4.0/24	2
10.5.31.0/24	2
10.7.31.0/24	2
172.17.0.0/16	7
198.51.100.0/24	2

```
rm(tmp)
```

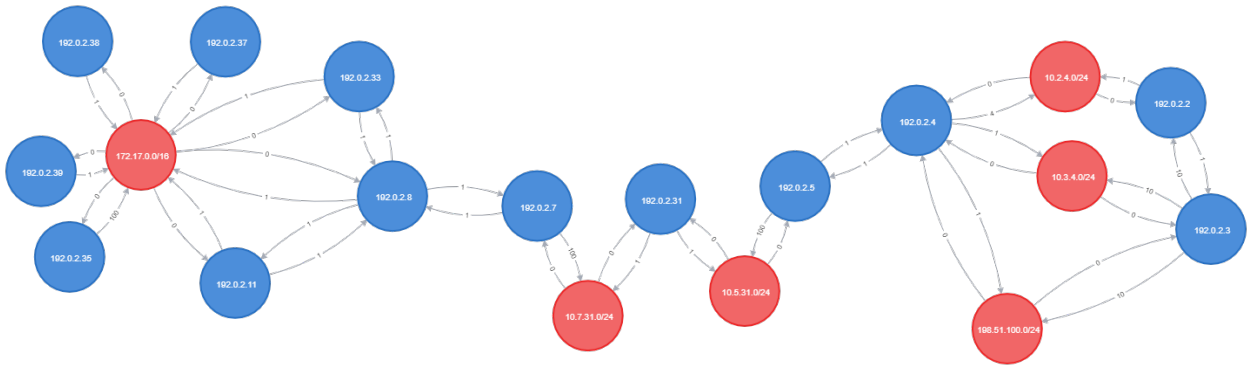

Figure 6: `MATCH (n:NETWORK)<-[e:LINKED]-(r:ROUTER) WITH n as n, stDev(e.cost) AS s, collect(r) as r WHERE s <> 0 RETURN n, r`

A much less common error is for the `ip ospf network` type to be mismatched. On point-to-point networks, the router interface should be configured as `ip ospf network point-to-point` ("Initial Configurations for OSPF over a Point-to-Point Link" 2007). For broadcast networks, this configuration should be left at its default value, `ip ospf network broadcast`.

```
tmp = "
MATCH (u:ROUTER)-[]->(v:ROUTER)
WHERE NOT (v)-[]->(u)
RETURN u, v
" %>% call_neo4j(con)
kable(tibble(u = tmp$u$name, v = tmp$v$name))
```

u	v
192.0.2.47	192.0.2.38

```
rm(tmp)
```

Figure 7: `MATCH (u:ROUTER)-[]->(v:ROUTER) WHERE NOT (v)-[]->(u) RETURN u, v`

3.3 Important Nodes

Finally, we attempt to discover "important" nodes using tools from the Graph Data Science (GDS) library.

First, we project the database to memory.

```
"
CALL gds.graph.create('myGraph', ['ROUTER', 'NETWORK', 'STUB'], 'LINKED', {
relationshipProperties: 'cost'
})
" %>% call_neo4j(con)
```

We compute the closeness centrality of all vertices in the graph.

```
tmp = "
CALL gds.alpha.closeness.stream('myGraph')
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).name as name, centrality
ORDER BY centrality DESC
LIMIT 15
" %>% call_neo4j(con)
kable(tibble(Node = tmp$name$value, Centrality = tmp$centrality$value))
```

Node	Centrality
192.0.2.7	0.5765472
172.16.0.0/16	0.5709677
192.0.2.5	0.5566038
192.0.2.4	0.5267857
192.0.2.13	0.5236686
192.0.2.10	0.5205882
192.0.2.8	0.5160350
192.0.2.6	0.5057143
192.0.2.9	0.4985915
198.51.100.0/24	0.4903047
10.3.4.0/24	0.4903047
10.7.31.0/24	0.4876033
192.0.2.17	0.4876033
192.0.2.14	0.4770889
192.0.2.31	0.4770889

```
rm(tmp)
```

Closeness centrality provides a few good results, but also a few poor results. The closeness centrality correctly shows that nodes 7 (identified by router ID 192.0.2.7), 5, 4, and 13 are very central to the network. Unfortunately, closeness centrality overestimates the importance of nodes 51 and 52. These nodes form a small triangle with node 5, which really is very central to the network, but the loss of either or both nodes 51 and 52 would not cause a partition in the network.

PageRank fares even worse in this application.

```
tmp = "
CALL gds.pageRank.stream('myGraph', {
    maxIterations: 20,
    dampingFactor: 0.85,
    relationshipWeightProperty: 'cost'
})
```

```
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name as name, score
ORDER BY score DESC
LIMIT 15
" %>% call_neo4j(con)
kable(tibble(Node = tmp$name$value, PageRank = tmp$score$value))
```

Node	PageRank
172.17.0.0/16	0.5288720
10.13.17.0/24	0.4089846
192.0.2.15	0.4013324
192.0.2.20	0.3764594
192.0.2.21	0.3762511
10.7.31.0/24	0.3730326
10.5.31.0/24	0.3707725
10.20.27.0/24	0.3578942
10.21.22.0/24	0.3476399
10.15.18.0/24	0.3430826
10.21.24.0/24	0.3385179
192.0.2.17	0.3248360
10.15.20.0/24	0.3152772
192.0.2.27	0.3009566
192.0.2.13	0.2997742

```
rm(tmp)
```

PageRank nonsensically scores nodes 15, 20, 21 highly. It also seems to favor certain transit networks. The PageRank scoring of node centrality for this network topology does not align with a technician's expectation.

Finally, we use GDS's betweenness centrality procedure in search of bridge nodes.

```
tmp = "
CALL gds.betweenness.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC
LIMIT 15
" %>% call_neo4j(con)
kable(tibble(Node = tmp$name$value, Betweenness = tmp$score$value))
```

Node	Betweenness
192.0.2.7	6371.089
192.0.2.5	5442.522
172.16.0.0/16	5062.522
192.0.2.4	4750.000
192.0.2.3	4144.333
192.0.2.22	3507.500
192.0.2.14	3361.089
192.0.2.8	3183.667
192.0.2.24	2808.500
172.17.0.0/16	2766.833
10.24.25.0/24	2582.000
192.0.2.25	2472.000

Node	Betweenness
192.0.2.27	2225.000
192.0.2.10	2075.544
192.0.2.13	2010.211

```
rm(tmp)
```

Betweenness centrality correctly identifies nodes 7, 5, 4, 3, 22, 14 and 8 as important nodes that are central to the network. Many of these nodes (5, 4, and 3) would cause a partition in the network if removed.

Among closeness centrality, PageRank, and betweenness centrality, the betweenness centrality appears to give the most accurate prediction of network node importance.

4 Conclusion

Graph databases are a compelling option for analyzing any data set that can be represented as a network. An important question to ask is whether *transitivity* makes sense in the data set. A graph database could be used to hold, for example, `(:STORE)-[:HAS]->(:ITEM)` relationships, but the graph database may not be particularly useful if there is no `(:STORE)-[]->(:STORE)` or `(:ITEM)-[]->(:ITEM)` relationships.

The OSPF protocol for network routing uses a graph directly to forward packets. A graph database's syntax, visualization features, and analytical functions offer network technicians a natural way to explore, study, document, and optimize their routers.

This research project presents a standalone Java application that gathers the OSPF LSDB using SNMPv3. The program also hosts an embedded Neo4j graph database, which can be access through Neo4j Browser or any Neo4j driver using the Bolt or HTTP protocols. We show that Neo4j effortlessly produces a competent network diagram, though the diagram is missing many details that a network technician would ordinarily need. Finally, we evaluate the closeness centrality, PageRank, and betweenness centrality procedures and found that betweenness centrality predicts bridge nodes most accurately.

Java source code is posted on GitHub under wjholden/OSPF-Analyzer.

References

"Initial Configurations for OSPF over a Point-to-Point Link." 2007. *Cisco*. Cisco. https://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/13687-15.html.

Joyal, Dan, Fred Baker, Spencer Giacalone, Dan Joyal, and Piotr Galecki. 2006. "OSPF Version 2 Management Information Base." Request for Comments. RFC 4750; RFC Editor. https://doi.org/10.17487/RFC4750.

Malkin, Gary S. 1998. "RIP Version 2." Request for Comments. RFC 2453; RFC Editor. https://doi.org/10.17487/RFC2453.

Moy, John. 1998. "OSPF Version 2." Request for Comments. RFC 2328; RFC Editor. https://doi.org/10.17487/RFC2328.