Neo4Reach: Neo4j Reachability Index for Large Dynamic Graphs

Wael Farhan, *MS Student*
(wfarhan@eng.ucsd.edu)

Amarnath Gupta, PhD., *Advisor*
(a1gupta@ucsd.edu)

CSE291-G Project

University of California, San Diego

Neo4j is one of the world's leading graph database used today. Although a plethora of commercial companies adopted this tool, yet it seems to lack some basic features. One of the key elements that is lacking from Neo4j is the limited number of indexes available out of the box. There is a lot of research that is being done to develop novel graph indices (e.g. reachability, shortest path, distance ... etc.). In this project we implement a dynamic and scalable reachability index on top on Neo4j. We experimentally evaluate our implementation on demo datasets and we demonstrate that queries executed using our index runs faster than using plain Neo4j.

1 Introduction

Given a directed graph `G`, a source vertex `s` and a target vertex `t`, a reachability query checks if there exists a path from `s` to `t` in `G`. These sort of queries are important in various applications, for example social media graphs and road networks. Devising data structures for reachability index is non-trivial task, as it demands thorough analysis of how they are used from initializing the index, storage, querying and incremental updates. During our implementation we faced a number of technical issues that were not covered in previous work. Since we were trying to implement theory from two different paper we found major implementation gaps between those papers. In this paper we will discuss those issues and how we handled them.

2 Theory

In this project we aim to implement "Reachability Queries on Large Dynamic Graphs: A Total Order Approach" [1]. *total order labeling (TOL)* is an indexing framework that can be extended into various 2-hop labeling indices. Implementation of TOL demands a DAG G and a strict total order on all vertices. The level order of a vertex v, denoted as $l(v) \in [1, |V|]$, defines the rank of v. The lower the number the higher the level of the vertex. In other words, v has *higher* level than another vertex u, if $l(v) < l(u)$. L is defined as a 2-hop labeling index where each vertex is assigned an in-label set L_{in} and an out-label set L_{out}. To achieve the highest efficiency we need to choose an appropriate order level l. The optimal order should minimize the cost of the L_{in} and L_{out} of all V. More formally:

$$|L| = \sum_{v \in V} (|L_{in}(v)| + |L_{out}(v)|).$$

After constructing L for V, a reachability query from source vertex s to target vertex t is computed by finding the *witness set* as follows:

$$W(s,t) = (L_{out} \cup \{s\}) \cap (L_{in} \cup \{t\}).$$

If the *witness set* contains any vertex then that means there is a path from s to t through the nodes in the *witness set* and will return TRUE. Otherwise, if the set is empty then it will return FALSE. The reachability paper [1] describes in details two incremental update procedures: one for inserting new vertices, while the other to handle deleting a vertex. It also discusses how to compute initial l and construct L. We will discuss this in section 3 along with the modifications we introduce.

Reachability query paper assumes that our graph is a DAG, which is certainly not the case for most of Neo4j databases. We find a way to transform a graph to a DAG by implementing Kosaraju's algorithm. But the most crucial part is how to maintain the DAG while it is constantly changing by adding new edges, deleting edges, creating nodes and deleting them. We then incorporate "DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs" [2] which handles maintaining a DAG in a dynamic environment.

3 Implementation

In this section we will discuss how to implement a reachability index on Neo4j. There are 3 main components that have to be addressed. First we are going to talk about how to create a user interface that allows users to create new reachability index and execute reachability queries. Then, we will discuss what happens under the hood for each user call. And finally, we will discuss how we hook our index to the database and perform incremental updates. We implement these components using Neo4j Java API and eventually package them as a plugin that could be downloaded and installed on any Neo4j database. Throughout the development of this project we use Neo4j 3.0.1 which is the latest version to date of writing this paper.

3.1 Unmanage Extension API

This is a versatile tool, allowing developers to deploy arbitrary HTTP RESTful JAX-RS classes to the server. Just by adding the listing 1 to the *pom.xml* we are able to devise our own API.

Listing 1: Add RESTful interface to Unmanaged Extension

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
</dependency>
```

In this class we implement 4 different API calls:

- $GET/reach/reachability/create\_index$: This API call will build a new index by scanning the entire database to generate strongly connected components (SCC), which is then used to build a new DAG that represents the original graph. It will also compute l and L. This API call might take some time to execute, it depends on how much data is in the original graph. This API call is called only once, when the user needs to create a new index.
- $GET/reach/reachability/get\_scc/nodeId$: This API call is just for testing purposes. The user specifies the *nodeId* the response will be the SCC ID.
- $GET/reach/reachability/source/s/target/t$: Here we compute the *witness set* and return true or false depending on whether there is a path from s to t or not.
- $GET/reach/reachability/noindex/source/s/target/t$: This API call is used as a baseline for performance comparison.

3.2 Index Database Class

This class is the heart of the plugin. It creates, stores, queries and updates the index data. When creating a new index it creates a new separate Neo4j database. Where each node in this new database represents a SCC of the original graph. Here we will discuss the steps of creating a new index.

3.2.1 Create Strongly Connected Components

Here we implement Kosaraju's algorithm to compute SCCs. When the algorithm discovers a new SCC it creates a new vertex in the index database, find its vertex ID. Then, it updates all vertices that compose this SCC with a new property $_scc_id$ to store its location in the index database.

If we take a look at our index at this point, it is going to be a collection of disconnected vertices. So our next step is to add edges to connect SCC by looping through each edge (s,t) in graph and connecting them as follows:

- $if(s.\_scc\_id == t.\_scc\_id)$: Do nothing.
- $if(s.\_scc\_id \mathrel{!}= t.\_scc\_id)$ & $there\ is\ no\ edge\ connecting$ $(s.\_scc\_id, t.\_scc\_id)$: Create a new edge with a property *count* equals to 1.
- $if(s.\_scc\_id \mathrel{!}= t.\_scc\_id)$ & $and\ there\ exists\ edge$ $(s.\_scc\_id, t.\_scc\_id)$: Increment *count* of that edge.

Time complexity of this procedure is $O(|V| + |E|)$ where V is the vertices and E is the edges of the original graph.

3.2.2 Decide Vertex Level Order

After generating our basic index database structure, we apply various algorithms discussed in reachability paper [1]. The first algorithm 1 used to decide total order level (TOL) for each vertex.

```
for order in [1 … |V|]:
      for v in G nodes:
            calculate:
                  $S^{\top}_{in}(v) = \begin{cases} \sum_{u \in N_{in}(v)} (S^{\top}_{in}(u)+1), & \text{if } N_{in}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$
                  $S^{\top}_{out}(v) = \begin{cases} \sum_{u \in N_{out}(v)} (S^{\top}_{out}(u)+1), & \text{if } N_{out}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$
                  $f(v,G) = \frac{|S_{in}(v,G)| \cdot |S_{out}(v,G)| + |S_{in}(v,G)| + |S_{out}(v,G)|}{|S_{in}(v,G)| + |S_{out}(v,G)|}.$
      find v that has the maximum f(v,G)
      v.order = order
      remove v from G
```

Fig. 1: Computing Total Order Level

$S^T_{in}(v)$ denotes the in-score, $S^T_{out}(v)$ denotes the out-score of a vertex v. These values are computed by doing recursion on all v's neighbors and computing in-neighbors $N_{in}(u)$ and out-neighbors $N_{out}(u)$. We then compute the final score $f(v,G)$ for each node, extract the vertex with the maximum score and assign it with the label by setting the property *order* on that vertix. In the paper they suggest deleting the node v from G. But, we can not do that because we will lose valuable information such as the ID, node edges and order. Alternatively, we added this vertex to *visited* hash table. When a node is on that table we omit all calculations related to it, as if it does not exist in our graph. Time complexity for TOL algorithm is $O(|V| * (|V| + |E|))$ where V is the vertices and E is the edges of the *index* graph.

3.2.3 Labeling Algorithm - Butterfly

Here we compute the L_{in} and L_{out} sets for all vertices in index graph. We use the same algorithm presented by the reachability paper [1] which is listed below 2.

```
Algorithm 5: BUTTERFLY
   input  : $G$ and a level order $l$
   output: a TOL index $\mathcal{L}$
1  let $G_1 = G$ ;
2  for $k = 1, \cdots, |V|$ do
3      let $v$ be the vertex whose level is $k$ ;
4      identify the set $B^+(v)$ of vertices that $v$ can reach in $G_k$, using a
       BFS from $v$ that follows the outgoing edges of each vertex ;
5      identify the set $B^-(v)$ of vertices that can reach $v$ in $G_k$, using a
       BFS from $v$ that follows the incoming edges of each vertex ;
6      for each vertex $u$ in $B^+(v)$ do
7          if $L_{out}(v) \cap L_{in}(u) = \emptyset$ then
8              add $v$ to $L_{in}(u)$ ;
9      for each vertex $u$ in $B^-(v)$ do
10         if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then
11             add $v$ to $L_{out}(u)$ ;
12     remove $v$ and $G_k$ and denote the resulting graph as $G_{k+1}$ ;
13 return the label sets of all vertices in $G$ ;
```

Fig. 2: Labeling Algorithm - Butterfly. Source [1]

Time complexity is $O(V*(\bar{B^+}+\bar{B^-}))$, where $\bar{B^+}$ is the mean count of vertices reached by vertex v and $\bar{B^-}$ is the mean count of vertices that can reach vertex v. Here we also applied the trick of *visited* hash table because the algorithm suggests to remove the vertex from G. After executing the algorithm each vertex will be assigned a two properties: *L_in* a set of input labels and *L_out* a set of output labels. We added an extra script that computes inverse lists $I_{in}(u)$ and $I_{out}(u)$ such that:

$$I_{in}(u) = w | u \in L_{in}(w),$$

$$I_{out}(u) = w | u \in L_{out}(w).$$

In other words, if vertex u appears in $L_{in}(v)$ set then v is going to be in the $I_{in}(u)$. Same goes for L_{out} and I_{out}. Inverted lists are discussed in the reachability paper [1] to ease the procedure of incremental updates.

The above three steps discuss how we create an index and store it. Now we will explain how we perform a reachability query based on the data we stored in the index.

1. $HTTPGET/reach/reachability/source/s/target/t$
2. Query original database to get s and t.
3. Component Lookup: Get *_scc_id* for both nodes.
4. Query index database for both *_scc_id*.
5. Get *L_out* from source SCC node and *L_in* for target.
6. Compute *witnessset*.
7. Return *true* if there is a path, *false* if not.

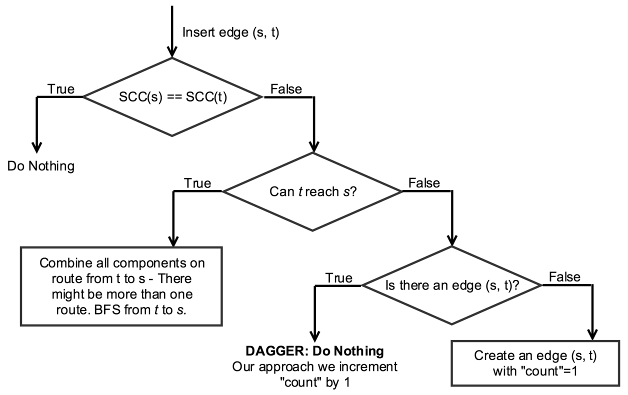
This class is implemented as a singleton class to maintain one connection to the index database. This way queries and updates will be performed much faster without the need to wait for a new instantiation of database connection.

3.3 Triggers using TransactionEventHandler

To this point we were able to create a fully functioning reachability index for *static* databases. In this section we will cover how to go about updating the index after each change committed to the database. Here we use another nifty tool provided by Neo4j, *TransactionEventHandler*, which invokes callback functions on three events: *afterCommit*, *afterRollback* and *beforeCommit*. Once *TransactionEventHandler* has been registered, it will receive events about what has happened in each transaction which is about to be committed and has any data that is accessible via *TransactionData*. This Data object in those callbacks has the following functions: *createdNodes*, *deletedNodes*, *createdRelationships*, and *deletedRelationships*. Using those functions we are able to collect all the changes applied to the original database graph and apply it to the index graph. In the below subsections we will tackle each update and how we implement it. All of these updates are based on "DAGGER"[2] which discuss DAG incremental updates.

3.3.1 Create Edge

When creating an edge on the database we need a way to propagate this change to our index database. Figure 3 illustrates the process of creating an edge.



Insert edge (s, t)
SCC(s) == SCC(t)
True
False
Do Nothing
Can t reach s?
True
False
Combine all components on route from t to s - There might be more than one route. BFS from t to s.
Is there an edge (s, t)?
True
False
DAGGER: Do Nothing
Our approach we increment "count" by 1
Create an edge (s, t) with "count"=1

Fig. 3: Flowchart of creating an edge

The first thing we do is check if both vertices belong to the same SCC. If they do then we do nothing to the index. If they do not, then we need to check if target t is able to reach source s using our reachability query. If there is a path that connects t to s then that means all vertices in this path should be merged into one SCC. So we need a way to create a procedure to merge those vertices. Else we check whether s to t edge already exist. If yes then we increment the *count* of that edge. Else we create a new edge (s,t) with *count* equals to 1 and perform TOL section 3.2.2 and butterfly algorithm section 3.2.3.

There is couple of things that are not covered in both papers. The first point is combining multiple DAG SCC and merge them into one big SCC. Hence using only 2 incremental algorithms provided by reachability paper we had to come up with a way of combining SCC while maintaining order l, in sets L_{in}, out sets L_{out}. One way of doing it is to run BFS starting from t, each vertex that reaches the source s means that its along the path, so we store the vertices that makes this SCC, outgoing and incoming edges in a data structure. Then we delete this vertex. When BFS is completely executed we create a new SCC, add edges stored in the data structure and we update *_scc_id* for all the nodes in the original database to match the new SCC. Listing 2 displays how the process is performed.

Listing 2: Merging Vertices

```
bfs_stack.push(targetScc);
while (!bfs_stack.empty()){
  Node node = (Node) dfs_stack.pop();
  if(instance.reachQuery(node, startScc)){
    containing += startScc.containing;
    for(e in node.outgoing_edges)){
      outgoing.add(e.getEndNode().getId());
      bfs_stack.push(e.getEndNode());
    }
    for(e in node.incoming_edges)){
      incoming.add(e.getEndNode().getId());
    }
    deleteSCC(node);
  }
}
addSCC(outgoing, incoming, containing);
```

The second point is that adding an edge is not covered in reachability paper [1]. It only covers adding new vertex with its edges and deleting the vertex with its edges. Thinking about we realize that we could delete the entire source vertex with its edges and create it again with the new edge. Alternatively, we create the new edge then run the *butterfly* algorithm to re-assign L_{in} and L_{out}.

3.3.2 Delete Edge

This subsection will deal with edge deletion. Similar to creating an edge we have to follow a simple flow chart 4. When an edge gets deleted we check the source and

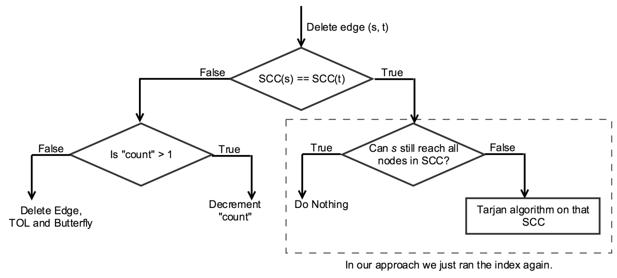

Delete edge (s, t)
SCC(s) == SCC(t)
False
True
Is "count" > 1
False
True
Can s still reach all nodes in SCC?
True
False
Delete Edge, TOL and Butterfly
Decrement "count"
Do Nothing
Tarjan algorithm on that SCC
In our approach we just ran the index again.

Fig. 4: Flowchart of deleting an edge

destination SCC nodes. If they are not the same then we check *count* property of the edge between s and t decrement *count* if it is more than one, else delete the edge and perform TOL section 3.2.2 and *butterfly* section 3.2.3. On the other hand, if SCC vertices are identical then we will need to split this SSC vertex into smaller ones if s can not reach t in other path. In DAGGER paper [2] they suggest executing tarjan's algorithm to find SCC nodes created strictly within the questioned SCC. But since we did not have enough time to do all this, we decided to rebuild the whole index starting from building SCC section 3.2.1, TOL 3.2.2 to executing butterfly 3.2.3.

3.3.3 Create Vertex

Creating vertices is straight forward. Initially when we create a new vertex on the database it is going to be disconnected from other vertices, which means that it is going to form its own SCC. Next we loop through all of its edges and create them one by one using the procedure presented in section 3.3.1.

3.3.4 Delete Vertex

Here we will do exactly the opposite of what we did in the previous section. We use the procedure presented in section 3.3.2 to delete all the edges attached to that particular vertex. By deleting the last edge this vertex should be disconnected from other vertices, hence it forms its own SCC. Finally, we delete the SCC.

One last thing to note about updates is that we need to maintain order to sustain consistency. We first handle adding vertices, then adding edges, third is deleting edges and finally deleting vertices.

4 Installation

To add reachability index to Neo4j the user has to follow few steps:

1. clone the code:
   git clone https://github.com/wael34218/Neo4Reach.git
2. Build: *mvn clean package*
3. Then copy the jar file created *target/neo4reach − 1.0.jar* to the *plugins/* directory of Neo4j server.
4. Configure Neo4j by adding the following line to *conf/neo4j − server.properties*:
   dbms.unmanaged_extension_classes = org.neo4j.reach.unmanagedextension = /reach
5. Start Neo4j server

After these steps are done, Neo4j should be responding to the RESTful HTTP API calls. In addition, it will be updating the index as the database changes.

5 Performance

We demonstrate the efficiency of reachability index by comparing time consumed on reachability query using the index against time consumed on the same query without using the index. Table 1 shows the results of our test measured on *Dr Who* dataset.

Type	Reachable	Not reachable
Using index	26ms	25ms
Not using index	121ms	26ms

Table 1: Time performance on *Dr Who* dataset

6 Limitations and Future Work

One of the major challenges that we faced while working on this project is testing. For example, it is little complicated to check whether TOL algorithm discussed in 3.2.2 achieved the optimal solution. We added Neo4j UNIQUE CONSTRAINT on *order* property to make sure that we do not have duplicate order. Which is able to pinpoint some bugs.

Incremental updates must handle a lot of special cases. Within the short time of implementing this project we were not able to cover and test all cases carefully. Up to this point this project has multiple errors that we were not able to fix. In future work this will be our first priority.

The two papers that we choose to implement do not integrate smoothly. There is lot of gaps that we had to fill in. For example, merging nodes is required by DAGGER [2] but is not discussed in reachability paper [1].

In future work, we are also looking to allow the user to define which type of relationship to use for the reachability index. Currently, we include all edges no matter what type they are.

7 Conclusion

Neo4j offers a lot of nifty tools that allow developers to create their own plugins. For example, *Unamanged Extensions* allow developers to add HTTP API to handle custom actions. Another tool is the *TransactionEventHandler* which allows developers to hook triggers gets invoked at different times during a commit.

The papers do not take advantage of Neo4j properties. We were able to use properties for two crucial procedures:

- *_scc_id*: For SSC lookup.
- *count*: It was used to avoid time consumed to check for another path when we delete an edge.

Using reachability index actually enhances performance and could be useful for many commercial applications.

References

[1] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. pages 1323–1334, 2014.

[2] H. Yildirim, V. Chaoji, and M. J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. 2013.