Master thesis project proposal

Graph Queries

Meda, NagaCharan nagac@student.chalmers.se

Suggested Supervisor at CSE : Krasimir Angelov

January 12, 2021

1 Introduction

Graph structured data is enjoying an increasing popularity as web technology and archiving techniques advance. Numerous emerging applications need to work with graph-like data due to its expressive power to handle complex relationships among objects. Bioinformatics, social science, link analysis, citation analysis, and social networks are some of the examples where Graph Databases are used[CP10]. Graph queries, for the most part, attempt to identify an explicit pattern within the graph database. Graph queries have an expressive power to return something at the level of an analytic in a normal data processing system. And to be fair, many analytics that you find in the normal world are really just good SQL queries, so this makes sense. What graph queries usually do is find a known subset of the important nodes in the graph haystack. Here are some typical examples of queries [Woo]:

- Find me all friends Dan and Kevin share:

```
MATCH (:Person (name:'Dan')) -(mutualFriends:Person)

-(:Person name:'Kevin')

RETURN mutualFriends;
```

- Find me all friends that Dan has that Kevin doesn't (these anti-patterns are often used to create recommendation engines):

```
MATCH (kev:Person)-(dan:Person name:'Dan')

-(newFriend)

WHERE NOT (kev) -(newFriend)

RETURN newFriend;
```

Figure 1: Examples from Neo4j Graph Databases

Graph Queries are implemented using Graph reachability (or simply reachability) queries, to test whether there is a path from a node v to another node u in a large directed graph. Consider a semantic network that represents people as nodes in the graph and relationships among people as edges in the graph. There are needs to understand whether two people are related for security reasons . On biological networks, where nodes are either molecules, or reactions, or physical interactions of living cells, and edges are interactions among them, there is an important question to "find all genes whose expressions are directly or indirectly influenced by a given molecule" . All those questions can be mapped into reachability queries. The needs of such a reachability query can be also found in XML when two types of links (document-internal links and cross-document links) are

treated the same. Reachability queries are so common that fast processing is mandatory[CP10].

2 Problem

Daison is a database where Haskell is used for programming both the database and the application. The database stores ordinary Haskell values, table definitions are written as Haskell data types, and querying is similar to list comprehension. Daison is already used for storing real data but without using any Graph queries so far. The problem is to extend Daison with Graph queries using efficient algorithms.

3 Goals and Challenges

The goal of the project is to implement **Graph Queries** - is there a path between two vertices in a graph. Here the graph usually represents some kind of ontology and is part of the database. The search should work in close to constant time by using appropriate indices.

Efficient reachability is important because it allows us to quickly find a path between two vertices. There are two possible opposite implementations.

- The first is to search in the graph from scratch every time, which takes $O(n^2)$ time. This gives us no benefits.
- The second is to precompute the reachability once and store the result in a cache. After that we can just consult the cache which takes O(1) time. This will allow faster path searches but it will consume a lot of space ($O(n^2)$ space complexity). Furthermore, the cache must be recomputed every time when the graph changes.

Algorithms like AI Labels [Shu+15] and others are somewhere in between. They use some cache but not of size n^2 and answer reachability queries in time which is usually close to $O(1)$ but not always. When you have the reachability query in $O(1)$, you can easily find a path in $O(m)$ where m is the length of the path. The challenge is exactly that to find a good compromise and build a practical implementation.

4 Approach

Recently, reachability queries on large graphs have attracted much attention. Many state-of-the-art approaches leverage spanning tree to construct indexes. However, almost all of these work require indexes and original graph in memory simultaneously, which will limit the scalability. In AILabel algorithm each node is labelled with a quadruple. Index construction time of AILabel is O(m+n),

which requires only one traversal through the graph. Be- sides, AILabel only needs index to answer the queries. A directed graph can be transformed into Directed Acyclic Graph after folding each Strongly Connected Components into a single node[Shu+15]. Algorithms like Tarjans Algorithm, Kosarajus Algorithm can be used to find Strongly Connected Components. Both the algorithms have the same time complexity of O(m+n) unless an adjacency matrix is used where Kosaraju's Algorithm will take O(m^2) in the respective case.

AILabel is working on static graphs. Nevertheless, graphs are not always static in practical applications such as twitter and Facebook. Hence, how to deal with graph update operations is a great challenge. To deal with this problem algorithms like D-AILabel [Shu+15], [HST08], [BPW19] might be considered. D-AILabel is an approach that can handle dynamic graphs efficiently based on AILabel, where each node is assigned a quintuple [Shu+15].

For fully-dynamic graphs, conditional lower bounds provide evidence that the update time cannot be improved by polynomial factors over recomputing the SCCs from scratch after every update. Nevertheless, substantial progress has been made to find algorithms with fast update time for decremental graphs, i.e. graphs that undergo edge deletions. [BPW19] is the first algorithm for general decremental graphs that maintains the SCCs in total update time $O^{\tilde{}}$ (m) [1] , thus only a polylogarithmic factor from the optimal running time .

In [HST08], the algorithm maintains a topological order of a directed acyclic graph as arcs are added, and detects a cycle when one is created. The algorithm takes O($m^{1/2}$) amortized time per arc, where m is the total number of arcs.

References

[BPW19] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. *Decremental Strongly-Connected Components and Single-Source Reachability in Near-Linear Time*. 2019. arXiv: 1901.03615 [cs.DS].

[CP10] Jing Cai and Chung Keung Poon. "Path-Hop: Efficiently Indexing Large Graphs for Reachability Queries". In: *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. CIKM '10. Toronto, ON, Canada: Association for Computing Machinery, 2010, pp. 119–128. ISBN: 9781450300995. DOI: 10.1145/1871437.1871457. URL: https://doi.org/10.1145/1871437.1871457.

[HST08] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. *Incremental Topological Ordering and Strong Component Maintenance*. 2008. arXiv: 0803.0792 [cs.DS].

[1] We use $O^{\tilde{}}$(f(n)) notation to suppress logarithmic factors, i.e. g(n) = $O^{\tilde{}}$(f(n)) if g(n) = O(f(n)polylog(n)).

[Shu+15] Feng Shuo et al. "AILabel: A Fast Interval Labeling Approach for Reachability Query on Very Large Graphs". In: *Web Technologies and Applications*. Ed. by Reynold Cheng et al. Cham: Springer International Publishing, 2015, pp. 560–572. ISBN: 978-3-319-25255-1.

[Woo] Dan Woods. *Improve Your Graph IQ: What Are Graph Queries, Graph Algorithms And Graph Analytics?* URL: https://www.forbes.com/sites/danwoods/2018/04/30/improve-your-graph-iq-what-are-graph-queries-graph-algorithms-and-graph-analytics/.