

MASTER THESIS PROJECT PLANNING  
REPORT

# Graph Queries

Meda, NagaCharan nagac@student.chalmers.se

Supervisor : Krasimir Angelov

Relevant completed courses:

(TDA452, Functional Programming)

April 7, 2021

# 1 Introduction

With an increase in web technologies and advance archiving techniques the need for Graph structured data has seen an exponential growth in the recent years. In this context, graph like data for its expressive power to handle complex relationships among objects, has become utility for various emerging applications such as bioinformatics, link analysis, social networks etc. [CP10]. Moreover, the graph queries can help us identify an explicit pattern within the graph database. This flexibility along with user controlled ability of Graph Databases allow us to exploit its utility. These graph queries find an underlying known subset of the important nodes in the graph haystack. Some examples of working of these graph queries have been reported in [Woo].

To illustrate the working principle of these graph queries one can see the functioning of a social network. The examples have been derived from the Neo4j Graph Database. Say that the aim of the graph query is to find all the mutual friends between Dan and Kevin, one can write the query as follows:

```
MATCH (:Person (name:'Dan'))  --(mutualFriends:Person)
--(:Person {name:'Kevin'})
RETURN mutualFriends;
```

As can be seen above, the graph query identifies the pattern of matching common friends between Dan and Kevin and returns them under 'mutualfriends'. Another such example where these graph queries have a higher functionality can be illustrated if the aim of the graph query is all the friends that Dan and Kevin doesn't share. One can write the query as follows:

```
MATCH (kev:Person)  --(dan:Person {name:'Dan'})
--(newFriend)
WHERE NOT (kev:Person {name:'Kevin'})  --(newFriend)
RETURN newFriend;
```

In terms of understanding how the implementation of Graph queries is concerned one can often use the precursor of implementing the graph reachability queries. Reachability queries tell us if there is a path between two nodes (say node  $v$  and node  $u$ ) in a large directed graph, without actually finding the path itself.

In this project, using reachability queries as a foundation, efficient implementation of Graph queries will be looked into. The project aims to implement various existing algorithms for the mentioned purpose.

## 2 Thesis Arena

### 2.1 Daison

In order to illustrate the implementation of Graph Queries in a database, Daison (DATA lISt comprehensiON) [Ang] has been chosen. Daison is an already existing database wherein the language for data management is Haskell instead of Structured Query Language (SQL). This particular speciality allows one to use Haskell's List Comprehensions generalized to Monads by utilizing the Monad Comprehension extension. Also, this functionality can replace the SELECT statements in SQL. Moreover, the database that is selected can store any serializable datatype defined in Haskell, which avoids the need to convert between Haskell types and SQL types for every query. Furthermore, Daison supports algebraic datatypes which are difficult to handle in conventional relational databases.

The main problem statement that this project aims to answer is the implementation of Graph queries in Daison. Before we address the main problem, it is important to note that the backend storage in Daison is SQLite where all the SQL related features are removed. The resultant is a simple key-value storage with a Haskell API on top of it which substitutes the SQL language, further giving better efficiency and reliability of an established database without the need of SQL Interpreter. The implementation of the SQLite backend is composed of B-Trees. As of now, Daison is utilized for storing real data using Graph Queries based on static graphs. The project is to extend Daison with Graph Queries that can also work for Dynamic Graphs using various algorithms. A typical example of how a graph can be represented in Daison is mentioned below:

```
type Distance = Int

data City = City { name :: String,
                  id :: Int,
                  population :: Int,
                  paths :: [(Key City, Distance)]
                } deriving (Typeable, Data )

instance Show City where
    show (City a b c d) = a ++ " - " ++ show b

cities :: Table City
cities = withIndex (table "cities") cityPaths

cityPaths :: Index City [(Key City, Distance)]
cityPaths = index cities "paths" paths
```

The above code represents the network of cities, each of which denoted by the node in the graph and the path between these cities represented by an edge.

## 2.2 Aim

The goal of the project is to implement Graph queries on databases. For the same, we try to first establish the reachability queries as a foundation, so as to show that various graph queries can be implemented from there. Reachability Queries show if there is any path between two vertices without actually finding the path itself.

Here, there are two methods of implementation that are different in terms of their approach:

- The first approach is to run a search for the path from the initial step everytime. However, the time complexity for the approach would be  $O(n^2)$  time which makes it a computationally costly approach.
- To mitigate the cost of computation incurred in the first approach, one can precompute the reachability and store it in a cache. Now, one can access the cache in  $O(1)$  time. Though the cost of computation is reduced, the space complexity is high which is  $O(n^2)$ . In addition to the space complexity, the cache must be recomputed when the graph is dynamic.

Also, note that since the project runs on SQLite Database which is based on B-Trees, there can be an overhead of  $O(\log N)$  for the above mentioned approaches. The main challenges of both the approaches are that they require high time or space complexity to solve it. Also when the graph is dynamic, one needs to recompute all over again which is costly in computation and difficult to achieve. Therefore, a suitable algorithm needs to be utilized to mitigate both the above mentioned challenges.

## 3 Methodology

### 3.1 Approach

In order to implement the reachability queries in graphs, there have been several methods that have already been proposed. Many of these methods leverage spanning trees to connect indices. This is however cumbersome, because both the indices and the graph needs to be in memory simultaneously. However, the algorithm proposed by Shu et al. which is the AILabel mitigates this problem. The AILabel [Shu+15] searches the path by decomposing the graph into subtrees and separately storing its edges which join the subtrees.

### 3.2 Limitations

Though the above mentioned approach is a viable one, there are some limitations to the algorithm. The algorithm only works for an acyclic graph and not for a cyclic graph. Hence, a directed cyclic graph must be transformed into directed acyclic graph. To solve this problem, one can use Tarjans [Tar72], Kosaraju [Sha81] algorithms which folds the Strongly Connected Components (SCC) into a single node. By this a cyclic graph converts into acyclic graph.

However, the entire approach is valid for only graphs which are static. This will not be the case during the practical applications where the graph is dynamic. With every addition/removal of an edge in dynamic graphs, the strongly connected components must be recomputed. Doing this recomputation from the initial step is a highly uneconomical process and also nullifies the optimizations that D-AILabel [Shu+15] provides which is a variant of AILabel that can handle dynamic graphs.

### 3.3 Related Work

Algorithms like [BPW19] and [HST08] can maintain the Strongly Connected Components at a minimum cost, even after insertion/deletion of new edge. But using these algorithms complicate the final implementation and also cannot leverage the indexing that AILabel provides.

In [BPW19], the algorithm preserves a hierarchy of graphs inorder to detect if a connected component must be split into two or more sub components. By using this method, there is no need of recomputation of all components for the whole graph. The hierarchy arises obviously. Say, for example an edge is added, one might have to look at merging several nodes into a single component. However the component maintains information about its internal structure which can be seen as hierarchy of graphs.

The algorithm proposed in [HST08], sustains topological ordering to detect cycles. Thereafter, the algorithm identifies the sub-graph for which the connected components must be recomputed. Here, the topological ordering can be substituted by using the labeling provided by AILabel.

### 3.4 Sub Goals

The main goal can be divided into the following components:

1. Implementing Reachability Queries for Acyclic Static graphs
2. Implementing Reachability Queries for Acyclic Dynamic graphs
3. Implementing Reachability Queries for Cyclic Static graphs
4. Implementing Reachability Queries for Cyclic Dynamic graphs

## 5. Implementing Graph Queries in above mentioned graphs.

As already mentioned in the approach section, since AILabel and D-AILabel algorithms only work for acyclic graphs, these algorithms can be implemented for acyclic static and dynamic graphs. Hence, the cyclic graphs need to be converted into acyclic graphs using compression/folding algorithms such as Tarjan, Kosaraju. These folding algorithms convert the Strongly Connected Components of a graph into a node, therefore any cycle in the graph will be compressed into a node. Then the AILabel algorithm can be implemented on the output of these algorithms. The resultant of this process will be reachability queries. Further, after the implementation of reachability queries in cyclic and acyclic graphs, Graph Queries can be implemented.

### 3.5 Timeline

A visualization of the project timeline is illustrated in the below figure:

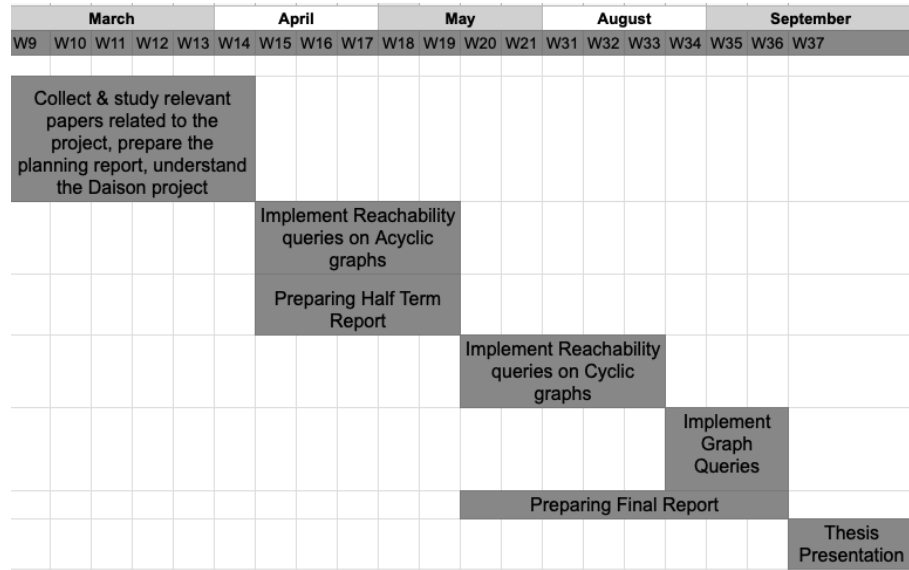


Figure 1: Timeline of the entire Thesis.

## References

- [Ang] Krasimir Angelov. *Daison*. URL: <https://github.com/krangelov/daison/blob/master/doc/tutorial.md>.
- [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].
- [Sha81] M. Sharir. “A strong-connectivity algorithm and its applications in data flow analysis”. In: *Computers Mathematics with Applications* 7.1 (1981), pp. 67–72. ISSN: 0898-1221. DOI: [https://doi.org/10.1016/0898-1221\(81\)90008-0](https://doi.org/10.1016/0898-1221(81)90008-0). URL: <https://www.sciencedirect.com/science/article/pii/0898122181900080>.
- [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.
- [Tar72] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *Society for Industrial and Applied Mathematics* 1.2 (1972), pp. 146–160.
- [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/>.