CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Implementing Graph Queries in Acyclic Graphs

Master's thesis in Computer science and engineering

Naga Charan Meda

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Master's thesis 2021

Implementing Graph Queries in Acyclic Graphs

Naga Charan Meda

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Implementing Graph Queries in Acyclic Graphs

Naga Charan Meda

Supervisor and Advisor: Krasimir Angelov, Department of Computer Science and Engineering, Chalmers University of Technology
Examiner: Andreas Abel, Department of Computer Science and Engineering, Chalmers University of Technology

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Implementing Graph Queries in Acyclic Graphs

Naga Charan Meda
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Graph structured data has become prevalent in multiple applications in the current day scenario. Applications like social media networks has hundreds of millions of nodes and the sizes of these graphs grows dynamically. In order to handle these huge graph modelled data we need efficient techniques and methods. In this context graph queries are implemented using reachability queries as foundation. Through reachability queries we can easily find if a node is reachable to another node without actually traversing the graph. In this Master thesis we try to evaluate three different algorithms to implement graph queries using reachability queries.

Keywords: computer science, engineering, graph queries, reachability queries, acyclic graphs, functional programming, SQLite.

Acknowledgements

I would like to thank my supervisor Krasimir Angelov for his continuous support and guidance during the project.

Naga Charan Meda, Gothenburg, August 2021

Contents

List of Figures

List of Tables

1

Project Status

1.1 Current Status

The project aims to implement Graph Queries in Acyclic Graphs using Reachability Queries as a foundation. As a first step towards implementing reachability queries, the following algorithms have been investigated for smooth generation of labels.

- AILabel[4] (Augmented Interval Label) Algorithm
- D-AILabel[4] (Dynamic Augmented Interval Label) Algorithm
- D-AILabel using Smart Relabelling.

This half-term report reports the successful implementation of the AILabel, D-AILabel, D-AILabel using Smart Relabeling algorithms. However, there are some issues identified during the implementation for large graphs and the work is in progress in rectifying the same.

1.2 Deviation from the Original Plan

The original plan was to implement Graph Queries for both cyclic and acyclic graphs. Instead, Graph Queries will be implemented for only acyclic graphs. A modified version of D-AILabel algorithm for acyclic graphs using a smart relabelling technique will also be implemented. While handling dynamic graphs there are instances where the whole graph needs to be relabelled which is a cumbersome process, the smart relabelling technique alleviates the problem by relabelling only a fraction of all nodes.

1.3 Remaining Work

As of now there are some instances where the algorithms are not working as expected, after rectifying the same results will be collected by comparing depth first search, AILabel, D-AILabel and D-AILabel using smart relabelling technique and reported in the final report.

1.4 Time plan

As per the time plan, I am supposed to work on the Reachability Queries for Cyclic Graphs from May 16 - May 31 and August 1 - August 21. As the direction of thesis changed, I will focus on implementing the D-AILabel algorithm using Smart Relabeling technique and collect the results comparing the algorithms that are implemented. After fixing the issues mentioned in the previous section, I am expecting to complete the Final report during the Third week of September and present my thesis in the Fourth week of September.

2
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 [6]. The ability to handle data has become a challenge today and graph theory brings its own perks in solving this everlasting problem. 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 bio-informatics, link analysis, citation analysis, social networks etc.., [3]. Moreover, the graph queries can help us identify an explicit pattern within the graph database. These graph queries find an underlying known subset of the important nodes in the graph haystack. Some examples of working on these graph queries have been reported in [5]. The true nature and applications of graph queries can be evidently witnessed during data processing.

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'. Note, that the graph queries are often used as functional units which can have multiple functionalities. Typically, graph algorithms are based on a similar principle, wherein the functionality that the graph query offers is used to process the data in a particular fashion. 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;
```

Here, the anti-patterns were also identified using a graph query. Similar anti-patterns are often used to create recommendation engines. 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.

To understand reachability queries better, let us consider a semantic network where people are represented as nodes and relationships among them are represented as edges in the graph. Let us consider a graph query where one would want to know whether two people are related or not. One can just run a reachability query which will tell us whether people are related without telling the relation between them. Similarly, we can also understand reachability queries in terms of biological networks wherein nodes are either molecules, or reactions, or physical interactions of living cells, edges are interactions among them. Graph reachability queries can help us address the important question of finding all genes whose expressions are directly or indirectly influenced by a molecule.

Similar needs of reachability queries can also be found in XML where both the document-internal links and cross-document links are treated as the same. The utility of these reachability queries is so multidimensional that these are fundamentally required for faster processing [3].

After this introduction about reachability queries, let us understand what the acyclic graphs are. In simple words, an acyclic graph is a graph when there is no path from a vertex to itself. Class hierarchies in computer science is one such example of an acyclic graph. In the following figure we show a class hierarchy of various vehicle categories.

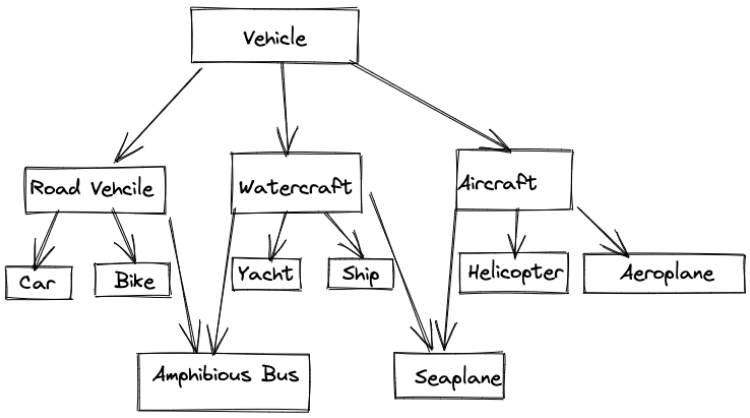

Figure 2.1: A class hierarchy depicting various vehicle categories.

2.1 Aim

The goal of the project is to implement graph queries on acyclic graphs. 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.

The traditional approaches for reachability computation are breadth first search (BFS) and depth first search (DFS), where each approach takes O(m+n) in the worst case. Another approach that can be considered is to compute a Full Transitive Closure but it requires a quadratic space complexity. Owing to these challenges the above approaches cannot handle large graphs.

In this context, the other approaches that are still utilized are as follows:

- Hop labeling approach: In Hop labeling approach, the reachability between two nodes is found with the help of intermediate nodes. Every node u stores two node sets u_{in} and u_{out}, where u_{in} is a list of intermediate nodes that can reach u and u_{out} is a list of intermediate nodes that u can reach. Subsequently, the reachability between u and v can be determined by a join operation between u_{out} and v_{in}, if there is a non-empty subset this means u can reach v.

- Interval labeling approach: In Interval labeling approach, each node stores an interval label that is generated by a spanning tree of the graph. There are multiple approaches to generate these interval labels. In this Masters thesis, AILabel (Augmented Interval Label) [4] and D-AILabel (Dynamic Augmented Interval Label) Interval labeling [4] approaches are used that will work efficiently on very large graphs.

2.2 Sub Goals

The main goal can be divided into the following sub-goals where we evaluate the following three algorithms in this thesis:

1. Implementing Reachability Queries for Acyclic Static graphs using AILabel algorithm
2. Implementing Reachability Queries for Acyclic Dynamic graphs using D-AILabel algorithm
3. Implementing Reachability Queries for Acyclic Dynamic graphs using D-AILabel algorithm with smart relabelling technique

3
Daison

Reachability queries are typically considered to be a precursor to establishing graph queries. In line of our main objective, i.e to establish graph queries in large graph data networks, we try to first establish reachability queries. The queries are implemented on top of *Daison* database which is explained in the current chapter.

Daison (DAta lISt comprehensiON) [1] 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 and also gives reliability of an established database without the need of SQL Interpreter. As of now, Daison is utilized for storing real data using Graph Queries based on static graphs. Implementation of some of the basic queries is listed below.

3.1 Transactions

Databases can be modified using Transactions. Since SQLite allows multiple reader - single writer access, the database has two kinds of access modes.

```
data AccessMode = ReadWriteMode | ReadOnlyMode
```

The database transaction is started using.

```
runDaison :: Database -> AccessMode -> Daison a -> IO a
```

The function signature makes it clear that it needs a database, access mode and an operation in the Daison Monad. The operation can be read/insert/update/delete data, create/drop tables. If there are any exceptions during the transaction, it will be rolled back, otherwise the transaction is committed.

3.2 Tables

In Daison, a table is a sequence of rows, where each row stores one Haskell value. Since the value can be of record type, the tables still can have columns. The data type for the rows can be defined as mentioned below.

```
data City = City { name :: String,
                   id :: Int,
                   population :: Int
                 }
```

After the declaration of necessary data types, the respective tables and indices are defined:

```
cities :: Table Cities
cities = table "cities"
            `withIndex` city_name
            `withIndex` city_id

city_name :: Index City String
city_name = index cities "name" name

city_id :: Index City Int
city_id = index cities "id" id

city_population :: Index City Int
city_population = index cities "population" population
```

It can be deduced from the example that *table* takes a *String* which is the table name and returns a value of type *Table*. Also indices can be added using *withIndex*

```
table :: String -> Table a
withIndex :: Data b => Table a -> Index a b -> Table a
```

The index can be defined in three different ways

- The easiest way to create an index is:

```
index :: Table a -> String -> (a -> b) -> Index a b
```

  The *index* function takes a table, index name and an arbitrary function that takes a value from a row and returns the value to be used in the index. If the row value is of record type, it is obvious that some indices can be over a particular field.

- It is also possible to index a row by more than one value using.

```
listIndex :: Table a -> String -> (a -> [b]) -> Index a b
```

- There can also be a condition where you need to index only some rows of the table instead of all. The following function will cater this condition:

```
maybeIndex :: Table a -> String -> (a -> Maybe b) -> Index a b
```

  If the indexing function returns *Nothing*, then the current row is omitted from the index.

After the definitions are defined, tables must be created using a read-write transaction as it is a database operation which changes the database. *createTable* will fail if there is a table with the same name in the database whereas *tryCreateTable* creates a table only if it is not created already.

```
runDaison db ReadWriteMode $ do
  tryCreateTable cities
```

Once a table is created, it can be renamed using *renameTable:*

```
runDaison db ReadWriteMode $ do
  towns <- renameTable cities "towns"
```

Now the table *cities* is renamed to *towns*, and can be accessed using the new name only.
The type of the table can also be changed using *alterTable*:

```
runDaison db ReadWriteMode $ do
  alterTable cities towns modify
where
  modify = ...
```

Given the definitions of *cities :: Table a*, *towns :: Table b*, the function*modify :: a->b* modifies the values of the table into the new ones.
A table can be removed using *dropTable* and *tryDropTable*

3.3 Data Access

3.3.1 Insert

Data can be inserted into the table using:

```
insert_ :: Data a => Table a -> a -> Daison (Key a)
```

The above function takes a table and a value to be inserted. The function returns the primary key number of the inserted value. *Key a* is a synonym for a 64-bit integer:

```
type Key a = Int64
```

When one wants to insert multiple values in the table, one can use the following function that takes a query as input that would extract the data from other tables and then insert the extracted data in the target table. In this process, the returned value is a pair of initial and final primary keys of the inserted rows.

```
insert :: Data a => Table a -> Query a -> Daison (Key a, Key a)
```

An example is mentioned below, where values from the table *towns* are taken and are modified using a function *f* and then the results are inserted into the table *cities*

```
runDaison db ReadWriteMode $ do
  (initial,final) <- insert cities [f x | x <- from towns]
  ...
```

3.3.2 Select

This is the main function for extracting the data from the database

```
select :: QueryMonad m => Query a -> m [a]
```

The *select* function takes only one argument *Query*, generally written as a monad comprehension. Here *QueryMonad* is a type class which allows the *select* query to be used either standalone from the *Daison* monad or as a nested query from the *Query* monad.

3.3.3 Queries

The main function for querying is:

```
from :: From s r => s -> r (K s) -> Query (V s r)
```

The *from* function is complicated because it is overloaded to these four types:

```
from :: Data a => Table a -> At (Key a) -> Query a
from :: Data a => Table a -> Restriction (Key a) -> Query (Key a, a)
from :: Data b => Index a b -> At b -> Query (Key a)
from :: Data b => Index a b -> Restriction b -> Query (b, Key a)
```

It is evident that the first argument is either a *table* or an *index*. The second argument *At* locates the value at the respective primary key or index. Finding an element at a given primary key value can be fetched using:

```
runDaison db ReadMode $ do
  select [c | c<- from cities (at 1)]
```

The equivalent query in SQL is:

```
SELECT * FROM cities WHERE id=1
```

Finding all the elements that match with the given range can be done using:

```
runDaison db ReadMode $ do
  select [(id,population) | (population,id) <-
  from city_population (everything ^> 50000 ^< 100000)]
```

whereas the query can be written in SQL as

```
SELECT id, population FROM cities
    where population >50000 and population < 100000
```

Unlike the previous example, an index is used here, the function returns the primary key instead of a row. Similar to relational databases, a table contains the data while an index contains a mapping between the value and the respective primary key of the table.

Restriction is another type that can be used along with *from* as mentioned in the examples above, it filters the data depending on the given condition. *everything, asc and desc* are different types of restrictions, which actually return all the rows. These restrictions are modified using (^<), (^<=), (^>) or (^>=) operators which actually filter the data. If there are other kind of constraints, they can be introduced using guards. For example:

```
runDaison db ReadMode $ do
  select [n | n <- from numbers (everything ^> 1), isPrimeNumber n]
```

3.3.4 Aggregation

Similar to SQL, there are aggregate functions in Daison to aggregate the results. Aggregations can be used in a query when the selected query needs to be post-processed, e.g. to compute the averages, sums, minimums, maximums, to group results, etc..,

```
query :: QueryMonad m => Aggregator a b -> Query a -> m b
```

Here, *Aggregator* is a function that takes a sequence of rows of type a and then transforms them into b in an incremental fashion. The list of built-in aggregators is listed below. Some aggregators are self explanatory:

- `listRows :: Aggregator a [a]`

  collects the rows into a list. In fact *select* is also defined as

```
select = query listRows
```

- `distinctRows :: Ord a => Aggregator a (Set a)`

  returns the unique rows as a *Set*

- `firstRow :: Aggregator a a`

- `lastRow :: Aggregator a a`

- `topRows :: Int -> Aggregator a [a]`

- `bottomRows :: Int -> Aggregator a [a]`

- `sumRows :: Num a => Aggregator a a`

- `averageRows :: Fractional a => Aggregator a a`

- `countRows :: Aggregator a Int`

- `foldRows :: (b -> a -> b) -> b -> Aggregator a b`

  returns the top-to-bottom fold of the rows.

  ```
  foldRows1 :: (b -> a -> b) -> Aggregator a b
  ```

returns the same values as *foldRows*, but uses the first row as the initial value.

- `groupRows :: Ord a => Aggregator (a,b) (Map.Map a [b])`

  groups rows of pairs by the first element in the pair

- ```
  groupRowsWith :: Ord a => (b -> c -> c) -> c
              -> Aggregator (a,b) (Map.Map a c)
  ```

  works similar to *groupRows* but also uses a function to combine the grouped values.

- `groupRowsBy :: Ord b => (a -> b) -> Aggregator a (Map.Map b [a])`

  works similar to *groupRows* but the rows can be of arbitrary type and uses a function to compute the value by which the grouping is done.

- ```
  groupRowsByWith :: Ord b => (a -> b) -> (a -> c -> c) -> c
                      -> Aggregator a (Map.Map b c)
  ```

  Combination of *groupRowsBy* and *groupRowsWith*.

- `sortRows :: Ord a => Aggregator a [a]`

  works as *listRows* but also sorts the values

- `sortRowsBy :: (a -> a -> Ordering) -> Aggregator a [a]`

  works similar to *sortRows* but uses and ordering function.

3.3.5 Update

Data can be updated using *update*:

```
update :: Data a => Table a -> Query (Key a, a) -> Daison [(Key a, a)]
```

The function takes a table on which the updates need to be performed and a query that returns a pair of key and value and the corresponding rows in the table are updated with the new values.

```
runDaison db ReadWriteMode $ do
  update cities
         [(city_id,city{population=0})| (city_id,city,_)
          <- fromIndex  city_population (everything ^<= 500)]
```

The above code updates the population of all the cities to zero if the population of a city is less than or equal to 500. The function returns the list of all key,value pairs that were updated. There is also another option if you don't need the results.

```
update_ :: Data a => Table a -> Query (Key a, a) -> Daison ()
```

The function always assumes that there are many rows that needs to be updated depending on the result of a query. Single row updations can be done using return:

```
update_ cities
 (return (1,City {name="Gothenburg", id=400, population= 556892}))
```

3.3.6 Store

This function is a combination of both *insert* and *update*. Consider an instance, where a document needs to be edited and then stored in the database. So, if this is a new document, it need to be added with *INSERT*, otherwise the previous document need to be updated using *UPDATE* statement. To handle this case, *store* can be used.

```
store :: Data a => Table a -> Maybe (Key a) -> a -> Daison (Key a)
```

The functions takes a table, key value, the updated value as input, if the key value is *Nothing*, then the value is inserted and the function returns the new primary key. Otherwise, if the key value is *Just key* then the value is updated with new value and the original key is returned again.

3.3.7 Delete

```
delete  :: Data a => Table a -> Query (Key a) -> Daison [Key a]
delete_ :: Data a => Table a -> Query (Key a) -> Daison ()
```

Similar to the *update* operation that has been explained above, the deletion operation takes a table and a query which selects the keys that are deleted. The function *delete* returns the list of all deleted keys where *delete_* doesn't return anything. Single row deletion can be done using the *return* function.

```
delete_ cities (return 20)
```

4

Methodology

4.1 AILabel and D-AILabel [4]

4.1.1 Definitions

In this subsection, we will specify important definitions that one would need in order to understand the algorithm.

- Spanning Tree: For a given graph G, spanning tree is a subset of the graph which is a tree and spans over all nodes.
- Spanning Forest: For a given graph, when a spanning tree is generated for each component of the graph, the collection of all these spanning trees is called as spanning forest.
- Tree Edge and Non Tree Edge: Given a spanning forest of a graph, the edges that are present in the spanning forest are Tree Edges and vice versa.
- Interval Label: As the name indicates, every node is assigned an interval. These intervals are generated by spanning tree. The important interval labels are [*pre, post*] which are explained through an example graph H in the following Figure 4.1. During DFS traversal of a graph following the tree edges, the number generated during the first visit to a node is assigned to *pre* where as the number generated during the second visit to the same node is assigned to *post*. As visible from the Figure 4.1, starting from node a, the directed path represented by the dotted line is the path in consideration. The *pre,post* labels of a node are assigned as described earlier. The numbers that are generated should be in a sequence, but need not be consecutive, however for our example, we have considered generating a sequence of consecutive numbers. If a particular node doesn't have an outgoing tree edge, then the *post* interval will be updated to the next number generated in the sequence.
- Hop node and Special node of a Graph: Given a non-tree edge, the starting node is defined as a special node whereas the ending node is called a hop node.
- Hops of node u: Given a special node u, all the nodes connected to u via the non-tree edges are called u's *hops*.
- Directs of node u: The *directs* of u are all special nodes reachable from it via tree edges.

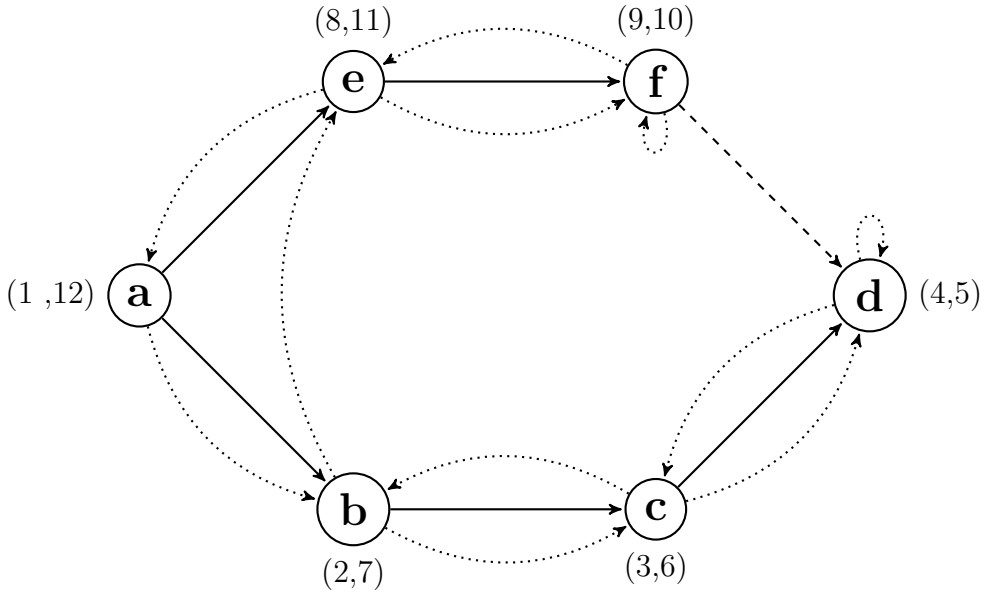

Figure 4.1: A graph H explaining the *pre* and *post* order

The definitions are demonstrated using an example in Fig. 4.2 (taken from the [4]) after applying a DFS on the graph:

- *Tree Edges:* All the solid lines in the graph.
- *Non-Tree Edges:* All the dashed lines.
- *Special Nodes:* F,I,J,H and C
- *Hop Nodes:* G,H and K
- *Hops of B:* null, since B is not a special node
- *Hops of F:* G
- *Directs of A* : C,F,I,J and H

4.1.2 AILabel Index

In AILabel, each node is assigned a quadruple <*pre, post, hops, directs*>, where the definition of each label is explained in the previous section. To get an AILabel index for a graph, we start DFS from the root node of the graph, update the *pre, post* of each node as explained in the example mentioned in Figure 4.1. Simultaneously, *hops* and *directs* are also updated with the respective values. Once the traversal is completed and the AILabel index is created for all the nodes as mentioned in the table below, reachability between two nodes can be found using the process which is explained in the next sub-section.

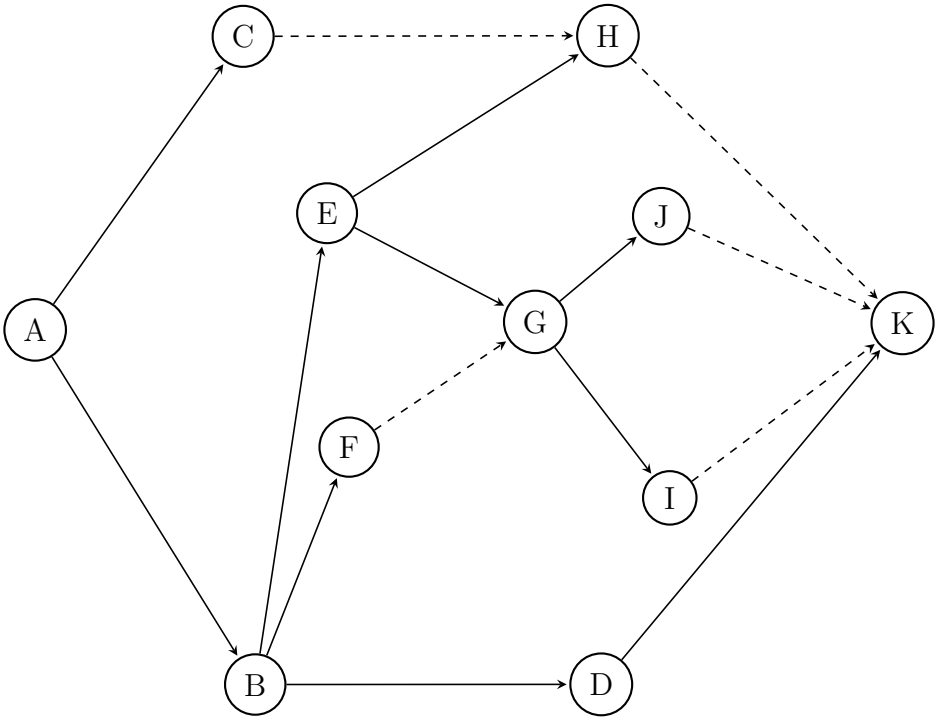

Figure 4.2: A sample Directed Acyclic Graph G

node	pre	post	hops	directs
A	0	21	[]	[I,J,H,F,C]
B	1	18	[]	[I,J,H,F]
C	19	20	[H]	[]
D	2	5	[]	[]
E	6	15	[]	[I,J,H]
F	16	17	[G]	[]
G	7	12	[]	[I,J]
H	13	14	[K]	[]
I	8	9	[K]	[]
J	10	11	[K]	[]
K	3	4	[]	[]

Table 4.1: AILabel index for the graph G

4.1.3 Reachability querying using AILabel

Given a graph g, to find if a node u is reachable to v, there are two steps to be followed.

1. After generating AILabel indices for all the nodes, we can confirm that u is reachable to v if $u_{pre} < v_{post} \leq u_{post}$, otherwise we have to perform the next step.

2. This step is further divided into two sub-steps where new queries needs to be generated in each sub-step. If the result of any of these sub-step is positive, u is reachable to v, otherwise it is not reachable.

- In this sub-step, we need to find if u's *hops* can reach v by using both steps (1 and 2) .
- In this case, we need to find if u's *directs* can reach v. Unlike the first sub-step, here we use only the step 2 to determine the reachability.

Let us consider an example to understand this. In the graph G from Figure 4.2, if we want to find whether B can be reachable to C, the query that needs to be generated is *Q(B,C)*. Since $(B_{pre} = 1) < (C_{post} = 20) \leq (B_{post} = 18)$ is not true, we need to proceed to the next step. Since B doesn't have any *hops*, we need to generate *Q(I,C),Q(J,C),Q(H,C),Q(F,C)* from B's *directs* by using only Step 2. After evaluating all these queries we get the final result that B is not reachable to C.

4.1.4 D-AILabel Index

Since AILabel works only on static graphs, D-AILabel is introduced which can handle dynamic graphs that are useful in practical applications like social media, genome sequences. The principle of D-AILabel is that AILabel is appended with *tree_parent* of a node to form D-AILabel. However, the main difference between the AILabel and D-AILabel is the approach how indices are created. The process of creating D-AILabel indices is not so simple as AILabel and we discuss the same in the next sub-section.

4.1.5 Index Maintenance in DAILabel

A graph can be updated in four possible ways, which are node insertion, node deletion, edge insertion and edge deletion. However, node insertion is setting the index on the new node and node deletion is deleting all the edges of the respective node, it is very trivial to study those. Hence we study only Edge Insertion and Edge Deletion which are explained below.

4.1.5.1 Edge Insertion

Depending on the type of nodes that an edge is inserted between, there are four different cases that need to be considered.

- Case 1: (u- not isolated, v- not isolated)

1. Say v has no parent node and it is not a special node either. Then, we can say that the *directs* of v can be added to u and u's ancestor nodes also. If this is not the case, then v instead of its directs, would be added into directs of u and u's ancestor nodes. Furthermore, v's *tree_parent* is set to u. Since the added edge is a tree edge, the intervals of v and all its descendants are updated as sub-intervals of u.

2. Another sub case, would be that if v has a parent node And if u is a special node. Then v is added into u's *hops*. Otherwise, u becomes a special node, when the new edge would be added in the original graph. Now u is added into directs of u's ancestor nodes and v can be added into u's hops

- Case 2: (u- not isolated, v- isolated)

  If one wants to insert an edge between two nodes u and v, such that u is not an isolated node but v is, then first would to acknowledge the fact that the new edge would actually be a tree edge. The tree edge would then be added to the spanning tree, that would further require all the nodes to be relabeled with a new set of updated intervals. However, relabeling the whole graph is inefficient and can actually be of a very broad based issue in terms of computing and cost. Therefore, this case is highlighted to be one of the cases that actually adds an overhead. To mitigate the same, an algorithm is introduced to optimize the process which will be discussed in the coming sections.

- Case 3: (u- isolated, v- not isolated)

  If one wants to insert an edge between two nodes u and v, such that u is an isolated node but v is not, then first v is added to the hops of u, if v has a parent node. Further, the intervals of u are updated to *[maximumid+1, maximumid+2]* respectively. Here, *maximumid* refers to the maximum id among all the intervals. However, the computation algorithm is different if v has no parent node. Then the same process can be repeated which is mentioned in Case 1.

- Case 4: (u- isolated, v- isolated)

  In the case where both nodes u and v are isolated, then the new edge that is being inserted becomes a standalone edge in the spanning tree. The *pre* and *post* labels of the nodes u and v are then updated to [*maximumid*+1,*maximumid*+4] and [*maximumid*+2, *maximumid*+3] respectively. Furthermore, as it becomes standalone edge, v's tree parent can actually be set to u.

4.1.5.2 Edge Deletion

The edge deletion case between u and v nodes can actually be classified into two separate cases:

One would be if the edge that is to be deleted is a tree edge. For this case, the node v would need to be removed from the *directs* of both u and u's ancestor nodes. Furthermore, v would not have parent node once the tree edge is deleted. To introduce that change in the spanning tree, v's tree_parent is set to -1 and the entire spanning tree is relabelled starting from v.

If the edge that is being deleted is not a tree edge. Then we would need to carefully see if the edge that we want to deletes is the only non tree edge from u.

If that is so, then u will no longer be a special node. Rather, it would have itself removed from the *directs* of its ancestral nodes. The *directs* for u will be appended to the *directs* of its ancestors nodes. Then the process becomes simple and the node v would only be deleted from u's *hops*.

4.1.5.3 Challenges

As can be seen from the algorithm there are three instances of relabelling the whole graph, which adds a huge overhead for dynamic graphs. This problem can be mitigated using order maintenance algorithms where only fraction of total number of nodes are relabelled, which reduces the overhead by a huge scale. We introduce the algorithm[2] in the next section for the same purpose.

4.2 Order Maintenance

In this section we discuss the algorithm[2] that would reduce the number of elements to be relabelled by a huge fraction.

4.2.1 Principle of the Algorithm

While handling dynamic graphs, the maintenance of depth-first traversal of the spanning forest of the graph is of primary importance. Especially, as we would be studying the functionality of D-AILabel, we need to find and implement an algorithm that can preserve the order of *pre, post* interval labels of a graph, even after addition, deletion of nodes and edges. In this scenario, the algorithm[2] can be helpful to optimize the existing process in D-AILabel.

In this approach, a list of n elements is taken as input, which are labelled sequentially to a set of integers from 1 to u called *tags*. We call the range of elements that these tags cover as tag universe size. The tag universe size should be a power of two so that the bits of the tags implicitly form a virtual binary tree as depicted in the Figure 4.3. This tree is a *trie* of bit strings, where the leaves correspond to the tags and the edges represent either 0 or 1. The leaves are labelled with the respective root-to-leaf path which is the binary representation of the tags. Any new element that is inserted into the list is assigned a new tag forming the new leaf of the virtual tree. Let us understand this through an example as depicted in the figure below. Consider a list of elements { *e,h,k,m,p,s* }, which are placed in the leaves of the tree. The respective tags (labels) are { 0011, 0100, 0101, 1000, 1011, 1101 }. As visible from the tree the edges are labelled with bits whereas the leaves are labelled with the sequence of bits from root to the respective leaf. Each internal node represents a range of tags that are present in the leaves of respective node, these internal nodes are important in order to find the range of elements that needs to be relabelled.

Before going into the relabelling process, we need to understand some definitions here in the current context.

- Enclosing tag ranges: As we already know that the formed tree is a *trie* of bit strings, any internal node will represent a sublist of the given elements that

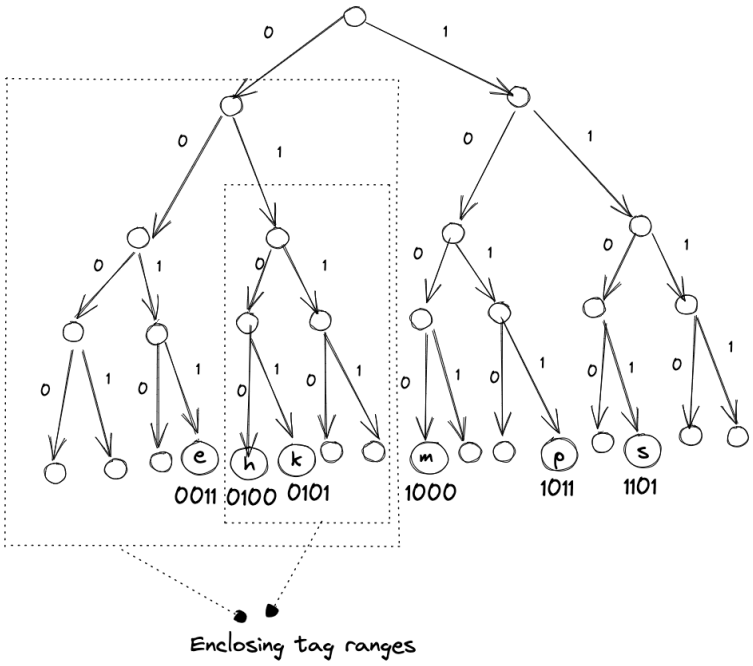

Figure 4.3: A sample list of elements l forming a virtual tree

reside in the leaves. Since the depth of the tree is log u, every leaf has log u *enclosing tag ranges*

- Density: For a given node, density is the fraction of its descendant leaves that are occupied.

During the insertion of a new element y between x and z, we can chose any tag that is available between the tags of x and z. When the elements (x,z) are adjacent to each other leaving no available tag for the incoming element, we then relabel a sublist of the given list so that the new element can be accommodated here. In this process, the smallest enclosing tag range with low-enough density is obtained for x and all the leaves in this range are relabelled. The algorithm to find the range of elements that needs to be relabelled is mentioned below:

Algorithm 1: Procedure to find the range of elements to be relabelled

$d \leftarrow 3$
$count \leftarrow 2$
while $d < max$ **do**
  while $label(prevOf(begin)) \geq (v \ \& \ negate \ \ d)$ **do**
    $begin \leftarrow prevOf(begin)$
    $count \leftarrow count + 1$
  while $label(nextOf(end)) \leq (v \mid d)$ **do**
    $end \leftarrow nextOf(end)$
    $count \leftarrow count + 1$
  if $count * count < (d + 1)$ **then**
    break
  $d \leftarrow d \mid (d \ << \ 1)$

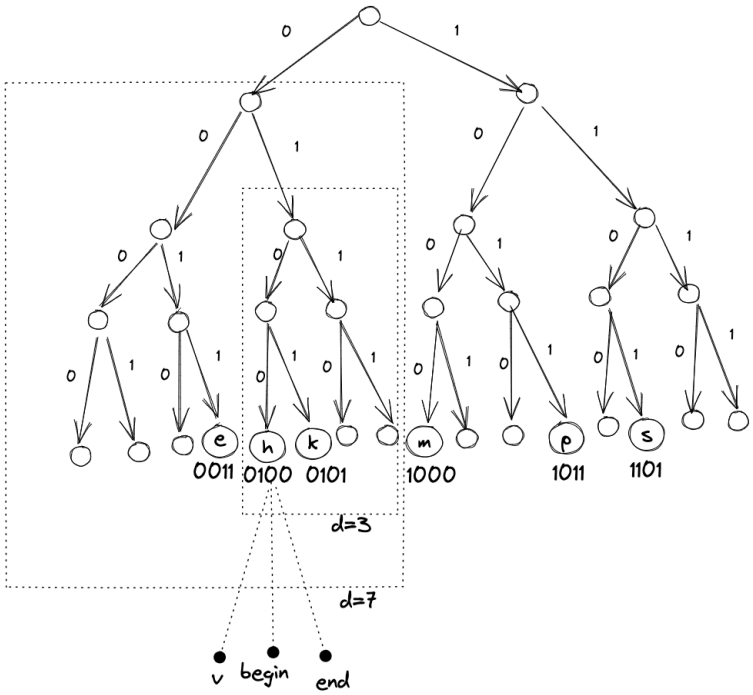

Figure 4.4: The elements before relabelling

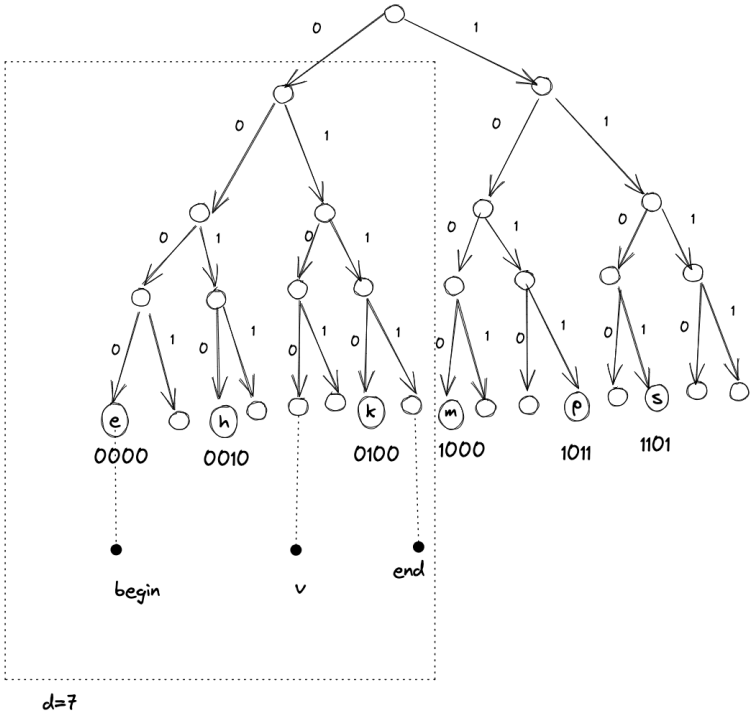

Figure 4.5: The elements after relabelling

Let us consider an example to understand how the relabelling works. As already mentioned the tree is virtual and we see how bit masking can be used in this process. The given elements are { e,h,k,m,p,s} and the respective labels (tags) are {0011(3),

0100(4), 0101(5), 1000(8), 1011(11), 1101(13)}. Now we want to insert an element between *h* and *k*, since there is no gap available here, we need to find the range of elements that needs to be relabelled using the algorithm which is already mentioned. In this process, *begin, end, v* are initialized to the tag 0100, after which the new element is being inserted, *max* is the *tag universe size*. Here, *d* is initialized to sequence of 0's and 1's (0011), the number of 1's represent the number of levels to go up in the tree to find the common root. The main purpose of the algorithm is to find that common root and then relabel all the tags that are present in the leaves. In the first inner *while* loop, a logical conjunction operation of *v* and complementing all the bits of *d* gives you the left most child whereas the second *while* loop gives you the right most child when a logical disjunction is performed on *v* and *d*. The *if* condition represents the stopping condition to find the enclosing tag range with low-enough density, where *d+1* represents the upper limit of the number of leaves present in the sub-tree. Finally, the first 0 bit is set to 1 in *d* after every iteration using bit masking to find the common root. As the value of *d* increases, the tag range widens and the number of elements to be relabelled increases. In this example the final value of *d* is 0111 which means we need to update all the leaves of the respective internal node. Once the range of elements is identified these elements are relabelled by having sufficient gap in incremental way between the elements. The elements after relabelling can be seen in the Figure 4.5.

In order to understand how the order maintenance algorithm is embedded into D-AILabel algorithm, we have to proceed to the next section.

4.3 D-AILabel with smart relabelling

We already know from Figure 4.1, that *pre,post* labels are a sequence of numbers. From Section 4.1.3, it is evident why the order of interval labels is important while querying. For the same reason, we are relabelling the whole graph in three instances while adding or deleting an edge from the graph, as already mentioned in Sub-section 4.1.5.3. We need to devise a process in order to avoid these frequent relabellings. In this context, we have to understand the following strategies:

- Average Inserts: If the label for the new element is the average of the neighbours, then the strategy is called Average Inserts.
- Consecutive Inserts: If the label for the new element is just a consecutive number of the previous element, then the strategy is called Consecutive Inserts.

Implementing the Average Insert strategy for the interval labelling process will reduce the number of relabels that might occur because of overflow. In order to implement this strategy, for a given interval, we need to maintain *previous* and *next* intervals. For this purpose, the DAILabel is extended with four new fields to maintain the details of children and siblings namely {*firstChild, lastChild, nextSibling, previousSibling*}. Using these new fields the previous and next elements can be identified using the process mentioned in Algorithm 2. An extra (new) root node is added where *pre* and *post* labels are initialized with minimum and maximum range

of the respective data type, which means the values of *pre* with zero, and the *post* with a max value which would be the upper bound of respective data type of node. Then the child nodes are labelled with the average of parent nodes which means the *pre, post* values of a first child node can be updated to *average (0,max), average(max/2, max)* respectively.

Algorithm 2: Procedure to find the next and previous elements of a given element

```
data PrePostRef = PreLabel Nd | PostLabel Nd

nextOf (PreLabel node) =
    case firstChild node of
        Just child -> PreLabel child
        Nothing -> PostLabel node
nextOf (PostLabel node) =
    case nextSibling node of
        Just next -> PreLabel next
        Nothing -> PostLabel (parentOf node)

prevOf (PreLabel node) =
    case prevSibling node of
        Just prev -> PostLabel prev
        Nothing -> PreLabel (parentOf node)
prevOf (PostLabel node) =
    case lastChild node of
      Just last -> PostLabel last
        Nothing -> PreLabel node
```

A similar process can be followed for the introduction of any new node, which is then labelled with the average of parent nodes or the average of previous and parent nodes depending on whether the new node is a first child node or not. Once the labelling is completed for every node, adding a new edge or deleting an existing edge will become an easy process as there is no need of frequent relabelling. Since we are using average insert strategy, we will eventually run out of possible interval labels when the gap between *pre* and *post* of a node becomes ≤ 1 where we will not be able to add a new child node. In order to resolve this issue we relabel a fraction of all nodes and preserve the order using the algorithm mentioned in Section 4.2. We need to verify if there is sufficient gap available before inserting every new node. If there is no sufficient gap we need to relabel some nodes.

5
Conclusion

5.1 Conclusion

In summary, the progress of the Masters thesis as of midterm has been reported and the improvised objective has been clarified. Limitations of D-AILabel has been identified and the further timeline of the thesis would revolve around collecting the results of the algorithms implemented and documenting the same.

Bibliography

[1] Krasimir Angelov. Daison.

[2] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, page 152–164, Berlin, Heidelberg, 2002. Springer-Verlag.

[3] 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, page 119–128, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Feng Shuo, Xie Ning, Shen de Rong, Li Nuo, Kou Yue, and Yu Ge. Ailabel: A fast interval labeling approach for reachability query on very large graphs. In Reynold Cheng, Bin Cui, Zhenjie Zhang, Ruichu Cai, and Jia Xu, editors, *Web Technologies and Applications*, pages 560–572, Cham, 2015. Springer International Publishing.

[5] Dan Woods. Improve Your Graph IQ: What Are Graph Queries, Graph Algorithms And Graph Analytics?

[6] Jeffrey Xu Yu and Jiefeng Cheng. *Graph Reachability Queries: A Survey*, pages 181–215. Springer US, Boston, MA, 2010.