MASTER'S THESIS | LUND UNIVERSITY 2018

Implementing and Evaluating a Breadth-First Search in Cypher

Alexander Olsson, Therese Magnusson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-16

Implementing and Evaluating a Breadth-First Search in Cypher

Alexander Olsson
dat13aol@student.lu.se

Therese Magnusson
dat13tma@student.lu.se

September 25, 2018

Master's thesis work carried out at Neo4j Sweden AB.

Supervisors: Tobias Lindaaker, tobias@neo4j.com
Krzysztof Kuchcinski, krzysztof.kuchcinski@cs.lth.se

Examiner: Flavius Gruian, Flavius.Gruian@cs.lth.se

Abstract

This report covers the implementation and evaluation of a Breadth-First Search operand for the Neo4j graph database and the Cypher query language. The evaluation compared the pre-existing Depth-First Search operand with both a non-optimized and two different optimized Breadth-First Searches. The focus of this evaluation was the runtime and memory usage for the different operands and to find out for which kind of graph and query that each operand is the better choice. The evaluation was performed through different benchmarks and self-constructed test cases.

We concluded that using Breadth-First Search on graph databases is beneficial for smaller graphs where we got improvements up to 90% faster and that the Breadth-First Search uses more memory than Depth-First Search. The optimizations gave huge improvements, however they are harder to fit into real world usage.

The Breadth-First Search operand should in theory be used on smaller graphs with smaller queries with more restrictions on the relationships. The better the machine the better results. Breadth-First Search should not be applied to the whole path, instead each relationship shall be evaluated case by case.

Keywords: Breadth-First Search, Graph Database, Neo4j, Cypher, Master's Thesis

Acknowledgements

We would like to thank our supervisors Tobias Lindaaker and Krzysztof Kuchcinski. Thanks for all your help along the way and for answering all our questions.

Thanks also to Alex Averbuch and Andrés Taylor for all the help getting set-up and started. Many thanks to Simon Svensson and Alex Averbuch for all their help with the benchmarks and getting them to run, when we ran into problems.

We would also like to thank Neo4j for providing us with equipment, a place to work and access to the benchmarks used for the evaluation.

Contents

Chapter 1
Introduction

When it comes to searching in trees and graphs there are two well known methods, Breadth-First Search (BFS) and Depth-First Search (DFS). In the Neo4j implementation of the Cypher graph query language DFS is used almost exclusively. This is because of the memory handling for DFS is simpler than for BFS, it could even be handled by the JVM if done recursively. BFS instead requires explicit data structures and uses more memory.

But DFS is not always better and there are many real world queries that would benefit from using BFS instead. To be able to choose between BFS and DFS search for different parts of a query would greatly benefit the Neo4j implementation.

1.1 Problem statement

There are many different classes of queries (see section 2.1) in Cypher and the efficiency of the search strategies differs for each class. This thesis strives to evaluate when BFS is more efficient than DFS as well as the opposite. An empirical evaluation of actual queries will be done and from this a theoretical model will be formulated. This theoretical model can then be used by a query planner to choose search strategy for the different patterns in a query.

1.1.1 Purpose

This thesis work is done to fill the gap in research on BFS and DFS on queries on graph databases, for this thesis it will be for Neo4j. When evaluating for which queries BFS is more efficient and when the current DFS implementation is preferable, the efficiency will be measured in both runtime and memory usage. The work shall give an understanding as an empirical evaluation of actual queries on actual data, and as a theoretical model. The theoretical model can then be applied by a query planner in order to make the planner automatically choose the theoretically optimal search strategy for patterns in the given query. For the evaluation and theoretical model the following questions will be answered:

- For which queries are BFS faster than DFS for Cypher?
- When is BFS more cost efficient than DFS?
- Can the previous points be used to develop a theoretical model in query planning?

1.1.2 Scope

To know what to do for the thesis work, a scope was set. Our scopes says what we will limit our work to and this applies to the implementation and the evaluation.

For our thesis, a prototype version of a BFS operand was implemented in Neo4j and evaluations was done on this prototype. The prototype is limited to work on variable pattern length and it does not support fixed pattern length. This is because it was deemed unnecessary for the thesis, since variable length is more versatile. However, as long as the path contains at least one variable pattern length relationship, the prototype can handle the search. The fixed pattern length relationships are then searched by the pre-existing search algorithm and not by the prototype, which is only used for the variable pattern length relationships in the path.

The prototype was designed to explore all nodes in the search area and does not contain a BFS for calculating shortest path. This was because using BFS for shortest path already exists in Neo4j. This meant that many of the more common optimizations to BFS cannot be applied to the prototype since they are often based on skipping nodes in the search, more unusual optimizations were therefore used for the prototype.

The evaluation was done on graphs that vary in size. Some were from real world datasets while others came from generated datasets, an example of the former was the Pokec benchmark while examples of the latter were the LDBC social network benchmarks. Both types of benchmarks were used to get an accurate evaluation. The queries from the benchmarks are limited by only using queries that contain variable length and queries for shortest path will not be used. There are also queries that are used for other applications in Neo4j and these will not be used. A disadvantage of these benchmarks were that they only measured the runtime and not the memory usage. Therefore, measuring the memory usage is limited to test cases. The runtime will also be measured in the test cases. These test cases contained several different graphs and queries that could be controlled easily.

1.2 Contribution

This thesis developed and evaluated a BFS algorithm for Cypher queries. This evaluation gave an estimation of the cost of BFS for different query patterns that could be used in comparison to other search strategies. The thesis compared BFS and DFS and formulated a theoretical model for which query patterns each search strategy is preferred.

Most of the work on the thesis, both report and implementation wise, was done together since both authors have similar experience and skill sets. Exceptions to the pair work include:

- Therese implemented a majority of the tests.
- Alexander implemented most of the Cartesian BFS optimizations.
- Alexander created the diagrams and tables in chapter 4.
- Alexander drew a majority of the figures.
- Therese wrote appendix B, adding information about the benchmarks and the used queries.

1.3 Related Works

Through out the years there have been many studies evaluating the performance of Breadth-First Search (BFS) and Depth-First Search (DFS). The studies have concluded that both of the algorithms have their advantages and disadvantages. Which meant that there is no clear way to determine which algorithm is better before evaluating them on specific scenarios [1, 2].

A paper about making an extension for computing shortest path in SQL is presented in section 1.3.1. Since we implemented BFS for graph databases could a paper on BFS in relational databases be of interest. It was of interest even though it handled shortest path, which we did not.

In section 1.3.2 an evaluation of multiple different BFS and DFS algorithms in data flow analysis by J. Cobleigh, L. Clarke and L. Osterweil is presented. The paper was chosen since it shows the complexity of evaluating search algorithms and that there is no algorithm that is the best for every search problem.

Presented in section 1.3.3 is an evaluation of queries with a variable length. Since we only handle variable length relationships in the queries, it was an interesting article.

Improving search algorithms can also be done by running them concurrently and this is presented in sections 1.3.4 and 1.3.5. These two articles were used as inspiration for the optimizations we implemented for our BFS operand.

Presented in section 1.3.6 is a study that analysed and evaluated search algorithms for heuristics. This paper was interesting since it analysed search algorithms based on different graph structures and distributions of goal nodes.

1.3.1 Extending SQL for Computing Shortest Paths

One of the algorithms that BFS preferably is used for is the shortest path algorithm, which has been studied for relational databases by D. Leo and P. Broncz [3]. In that study two algorithms were used to implement an SQL extension for solving shortest path. These were a BFS for unweighed graphs and a Dijkstra algorithm with Radix Queue for weighted graphs, since these algorithms are two of the preferred search algorithms for shortest path. The representation of paths was changed for their extension and nested tables were used. However instead of letting the tables be represented by collections of rows as in SQL, they represented a custom type. This means that limitations exists for their implementation. The custom type used as nested tables for the extension was represented as a list of references to the nodes of the table expressions. The extension was tested on the LDBC benchmark and the evaluation showed that the Dijkstra implementation was faster than the BFS, but the larger the dataset the smaller the difference in execution speed. They also concluded that given more time, the BFS implementation could be improved upon and even become faster than Dijkstra.

1.3.2 Comparing data flow analysis algorithms for finite state verification

There are many different versions of both BFS and DFS and they are better suited for different kinds of work. Studies on this have been made and one study that is relevant for this thesis work was done by J. Cobleigh, L. Clarke and L. Osterweil [1]. They conducted a study which evaluated nine algorithms, both BFS and DFS, on three different data flow analysis problems for finite state verification. Finite state verification is used for proving software systems properties and for this paper algorithms specific to the Flow Analysis for Verification of Systems prototype were studied. This prototype approaches the users' specified properties by automatically verifying them with static analysis for both sequential and concurrent systems. The model used by the prototype was based on annotated Control Flow Graphs and for concurrent systems a collection of these graphs were used, which is called a Trace Flow Graph. The results of these evaluations concluded that one of the DFS algorithms was best for exploration, which is to get fast results on an inconclusive problem. However, for shortest path and conclusive problems two different BFS algorithms gave the best results. Even though this experiment was not done on a database it is relevant since it shows that neither algorithm is obviously better than the other.

1.3.3 K-Reach: Who is in Your Small World

A problem in focus for this thesis is variable length queries, where the queries handle paths of a length within a given interval and these are called reachability queries. This is related to the k-hop reachability problem [4], which is to find if there is a path of at most length k between two nodes. This can be done with a regular BFS, however for example in a social network a user can be a celebrity, whom is connected with millions of other users leading to the search getting out of hand. The k-reach algorithm is based on vertex cover which states that given a graph there is a subset of nodes such that each edge has a minimum of one of its endpoints in the subset [5]. An index, called *k*-reach, then can be designed based on the fact that from a vertex cover all nodes can be reached within one edge [4]. The index *k* is the maximum number of edges that can connect two nodes, and is used to calculate weights for the algorithm. From this the algorithm can calculate, by using the small vertex cover, if the two nodes are reachable within *k* edges. The evaluation of *k*-reach showed that it was on par with other reachability algorithms for index construction in speed and memory consumption, but it was never the best. For the query performance *k*-reach was the fastest in many of the evaluations.

1.3.4 Efficient Multi-Source Graph Traversal

A common technique to enhance the performance of search algorithms for graphs is to run the code in parallel. One algorithm that takes advantage of this technique is Multi-Source BFS (MS-BFS) [6]. This algorithm is designed so that it works with a single core but is scaled up when more cores are used. The idea is that instead of parallelizing one BFS, as most algorithms do, it runs numerous BFS simultaneously and when more cores are used more BFS can be run. By running multiple BFS instead of one with many threads workload problems are avoided since the workload depends on which source node is used. Each thread is given its own node to work from, therefore it is challenging to implement so that the workload is divided equally. To start in one node each is done because each iteration of the BFS discovers increasingly more nodes. That means the search works best in small-world networks, when the distance between nodes are relatively small, even though the graph is large such as for social networks. The MS-BFS was ran on the LDBC benchmark with 50 thousand vertices and 1.5 million edges and both a normal and an optimized BFS were run as well. The results from the benchmark showed that MS-BFS was considerably better for both runtime and edges traversed per second per core and it scaled better as well. The idea of splitting the searches into two smaller ones was used to optimize this thesis' BFS prototype.

1.3.5 iBFS: Concurrent BFS on GPUs

To improve the execution time of parallel algorithms, GPUs can be used and one such BFS algorithm is the iBFS algorithm by H. Liu, H. Huang and Y. Hu [7]. This is because on GPUs the algorithm can run on many more threads than if CPUs are used. The iBFS algorithm is related to the MS-BFS algorithm mentioned above as it runs *i* different BFS on *i* different source nodes, but with some improvements. Two techniques were used to improve the speed, *Joint Traversal* and *Group By*. *Joint Traversal* is used to force the concurrent BFS instance to share their outer layer of nodes amongst each other and *Group By* creates optimized batches by selectively combining instances of BFS. The iBFS method can terminate early for a node if another thread has already calculated the node, this affected the execution speed immensely. Since the work of iBFS is memory intensive but the computations are similar, using thousands of light GPU threads is perfect. However, iBFS can be run on a CPU as well. The results of H. Lius, H. Huangs and Y. Hus study showed that iBFS run on GPUs always gave the best result in traversed edges per seconds and the CPU based iBFS was also significantly better than MS-BFS when they were ran on the same amount of threads. That paper was used as additional inspiration for splitting searches.

1.3.6 Analytical Results on the BFS vs. DFS Algorithm Selection Problem

Evaluating BFS and DFS is also interesting within the field of Artificial Intelligence, since a wide range of problems in the field can be formulated as search problems. T. Everitt and M. Hutter did a study on meta-heuristics [8], which are general search methods that are not aimed at a specific type of problem but instead use a neighbourhood relation on the solution space as well as a heuristic function. In these meta-heuristics both BFS and DFS can be used. Their study consisted of a theoretical analysis of average runtime and an experimental verification of the analysis result, for both a BFS and a DFS implementation. A side note is that DFS is substantially more effective with its memory usage, however if the BFS is implemented as an emulated iterative deepening DFS the memory usage can be on par with DFS and the only cost is a penalty for the runtime [8].

The study used goal level to refer to a level in the graph that contains at least one goal node. For a single goal level the study concluded that if the goal level was in the higher regions of the graph then BFS had an advantage, whilst if it was in the lower regions then DFS had the advantage. For multiple goal levels, when there are several levels containing goals, the analysis worked with the first found goal and the conclusions were about the same as for single goal level [8].

Chapter 2
Theoretical background

In this section the relevant theoretical background is presented, this is to give an understanding to this study. Graph databases and Cypher (section 2.1), Search algorithms (section 2.2) as well as background for the chosen benchmarks (section 2.3) will be covered.

2.1 Graph databases

Different databases store information in different ways. Relational databases store information in tables as entities and attributes, with the entities in the rows and the attributes in the columns. Graph databases instead store the entities as nodes and the relationships between entities as edges in a graph. This keeps the data connected and makes it easier to traverse the connections, for example finding information such as ”friend-of-a-friend-of-a-friend” in a social network [9]. Other examples of typically connected data is purchase history, where a person is linked to its purchases, or maps with streets and intersections.

This kind of storage is particularly useful when modelling data with focus on the relationships between entities in the design since they are easily found and traversed, in difference to how they are stored in a relational database. Retrieving information from a graph is called a traversal and involves ”walking” along the elements of the graph. This is most often localized in difference to SQL queries [10]. An attributed, labelled, directed multi-graph is known as a property graph and most other graphs are a subset of property graphs. It also means that almost anything may be represented as a graph [10].

We have worked with the graph database Neo4j, which is open source and can be found on GitHub [11]. There is a planner in the background workings of this database that builds plans for how searches should be executed. These plans have different steps such as finding the nodes in the graph matching a restricted node in the query, running the search algorithm out from the found nodes and filtering these searches to go from the source node to the target node. The planner chooses the most cost effective plan it finds to be executed, this is decided from a cost model that exists in the database. The planner also chooses which part of a relationship to start a search from and it is in the node with the least amount of edges. The cost model contains different methods for the steps and the cost of using them. When we implemented our BFS operand, we circumvented the cost model and specified to the planner, using a hint, that it should use our operand instead.

Cypher is a query language for the Neo4j graph database, similar to the SQL language used for relational databases. It is used to find data matching specific patterns. The format looks like a drawn graph with `()` for the nodes and `-[]-` for (undirected) relationships [9]. Types, length or other properties may be constricted in the nodes and relationships, an example for type constraints can be found in the query for finding "a-friend-of-a-friend" below. A Cypher query contains different clauses that builds the query. Some of these are `MATCH`, `WHERE`, `RETURN` and `CREATE` which do as expected from the names: finding paths that match the given pattern, providing filtering criteria for the matched results, returns the specified values or creates a new node or relationship [9]. The paths returned from queries can contain repeated nodes, but edges cannot be repeated. This is because Neo4j uses edge-isomorphism [12].

An example query with a `MATCH` clause for finding a "friend-of-a-friend", where both the nodes and the (directed) relationships have specified types, is featured below. In this example the nodes are given names so that they can be referred to in the subsequent `WHERE` and `RETURN` clauses. In the `WHERE` clause, the source node is specified to have the name "Alice" and the found friend-of-a-friend cannot be Alice. The `RETURN` clause returns the name of both the friend and the friend-of-a-friend.

```
MATCH (a:Person)-[:FRIEND]->(b:Person)-[:FRIEND]->(c:Person)
WHERE a.name = 'Alice' AND a<>c
RETURN b.name, c.name
```

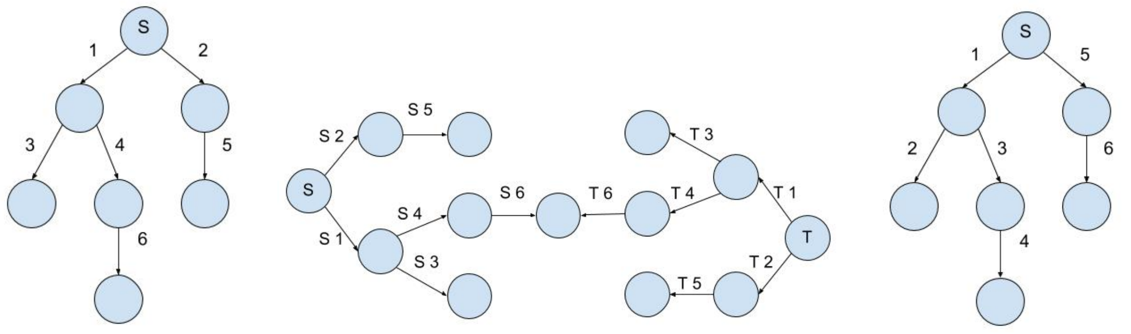
Cypher queries can be divided into several classes depending on their properties, where different aspects of the various properties are combined into the classes. Properties like length divide queries into variable and fixed length. The variable length can then be further divided depending on the restrictions put on the length. These restrictions can be upper, lower or both and restricts the possible path lengths. Another property to divide upon is the return values, such as paths, nodes or distinct nodes.

2.2 Search Algorithms

Breadth-First Search (BFS) is a graph algorithm that starts in one node and searches one depth, or layer, at the time [13, p.79-83]. This means that the first layer, L_1, is composed of all the neighbours to the source node. The next layers, L_i, are constructed of all neighbours to the previous layer, L_{i-1}, that do not exist in any of the previous layers. The algorithm ends when a path to the target node is found, no new nodes are encountered or a given maximum depth is reached. The graph is traversed as wide as possible first, and only goes deeper when all nodes in the current depth have been searched. This pattern can be seen in figure 2.1a, here the search starts in the node S and visits the nodes in the shown order. The implementation of our BFS algorithm is a little bit different as it allows multiple paths to the same node, because of the need to find all paths, but otherwise it is the same. Nodes can be repeated in the paths because of edge-isomorphism.

Another version of the BFS algorithm is the Top down Bottom up BFS. The Top down part is the same as for normal BFS, it starts at one node and builds up new layers, or frontiers, but the Bottom up part is new. The Bottom up starts at the unvisited nodes and searches towards the frontier instead. The performance of the Top down part is best when the frontier is small while the Bottom up part is best when the frontier is big, therefore they complement each other well [14]. This is *not* what our implementation does since there is no real frontier and switches between search directions mid-search is not supported. Instead one search, of half the maximum depth, is started from each of the source and target nodes and they are joined together on mutual nodes, using a pre-existing join operand in Neo4j. The pattern for our search is shown in figure 2.1b. Here it can be seen how the searches starts in both the S and T node and both finds the node in the middle. This is the node the join function uses to connect the two searches, since they both find it. This is inspired by the MS-BFS [6] and the iBFS [7], but without running the multiple searches concurrently.

Another search algorithm is Depth-First Search [13, p.83-86] which, instead of first exploring all neighbours of the source node s, takes the first edge out and follows it to the neighbour v. It then follows the first edge from v and continues in this manner until a node with no new edges to explore is reached. Then the algorithm backtracks until it reaches a node with unexplored edges, and goes down them in the same manner. Here is the graph first traversed as deeply as possible, before retreating only as much as needed. In figure 2.1c this pattern is shown as a search from the S node. This is the current search algorithm of the Neo4j implementation and several optimizations and variations exists.



(a) The search pattern for a BFS from the S node.

(b) The search pattern for a TB BFS from the S node to the T node.

(c) The search pattern for a DFS from the S node.

Figure 2.1: Figures showing the search patterns for the different algorithms. Numbers shows the order the algorithm searches the nodes.

2.3 Benchmarks

To be able to make a reliable evaluation, benchmarks will be used. Benchmarks are used since being able to make the same search for both BFS and DFS without anything being changed is necessary, and being able to redo the tests with the same results. A wide variety of benchmarks will be used to make the planner choose the optimal search algorithm. So the benchmarks will have small, medium and large graphs, from both generated datasets and real ones, and use both complicated and simpler queries on these graphs. The generated datasets can be formed so one can test specific cases, such as a path in the graph having bottlenecks. Real datasets are used since it is difficult to make a generated dataset be exactly like a real one. We only used some of the queries from each benchmark and not all. This was because our BFS operand needed the queries to have variable length relationships in at least one of the `MATCH` clauses. The queries which did not have any relationships of this kind were therefore discarded. The queries that were used for all benchmarks are all shown in table 2.1, the amount of nodes and relationships are also presented as well as memory size and our size classification. We classify benchmarks with graphs containing under 1 million nodes and under 5 million edges as small. Benchmarks with graphs containing under 5 million nodes and under 50 million edges are classified as medium and benchmarks with graphs larger than this are considered as large. All queries used for the evaluation are given in detail in appendix B.

When running a benchmark, the queries will be run multiple times to get an accurate estimation, since the times will not be exactly the same every time. To get the results even more accurate the queries will be run as a warm-up and the times for this will not be included in the results. This is done so that the machine does not have empty caches and operate as they would for real world use.

Some queries and datasets are from the LDBC social network benchmark [15]. This is a reliable and trusted benchmark application for graph databases. For end users it provides scenarios that they can recognize; for vendors there are checklists for performance characteristics and features; and for researchers it can provide interesting scenarios that will challenge most graph database usage [16, p.8]. The LDBC data generator can generate differently sized models of a real social network using different scale factors, we used scale factor 1 (SF01) and scale factor 10 (SF10). These datasets are deterministic, meaning that regardless of configuration parameters the graph will always be the same. The benchmark builds on bottlenecks, which are aspects of query execution known to be problematical. The queries in the benchmark are both of complex and short versions where the complex ones corresponds to the bottlenecks [15, p.25-48]. The LDBC benchmark queries used for the evaluation are shown in appendix B.1.

Another graph is from the Pokec dataset from a social network in Slovakia with 1.6 million profiles [17]. Since this is a real dataset, it is a good benchmark since it tests how the algorithm works with real data. The data in the dataset is already anonymized, but all the profile data is in Slovakian [17, 18]. The used benchmark queries are shown in appendix B.2.

Social Network is a benchmark that is built on use cases from real world usage [19]. The used queries from this benchmark, see appendix B.3, are smaller. They were used because they are from the real world and they are relatively simple queries on a larger graph.

A benchmark that is built on the same use cases as Social Networks is Logistic [19], however they do not use the same queries and the Logistics graph contains fewer nodes and edges. This benchmark was used because it was believed to not be optimal for our BFS, and benchmarks such as this is useful to check to validate a theoreticized downside with the algorithm. We only used one query from this benchmark, which can be seen in appendix B.4.

Another benchmark building on the Social Network and Logistics benchmarks' use cases [19] was Access Control. This is a third set of queries and its graph is slightly smaller than the Social Network graph but much bigger than Logistics. The queries of interest can be found in appendix B.5. This benchmark was chosen because it contains seven different queries that are similar and since they are run against the same data it will be clear in what situation our BFS is useful.

MusicBrainz is yet another real dataset, which contains an encyclopaedia of music information [20]. It is open sourced and community maintained, and a good benchmark to use because of the amount of metadata for the stored music. Some examples of metadata are title, track title, track duration, release date and country [21]. This means that there are many different things to search for and the used queries are shown in appendix B.6.

Another music related dataset, provided by Neo4j, was small and synthetically generated. This Generated Music benchmark can be found in appendix B.7. From this benchmark two similar read queries were used. One write query was also used and was the same as one of the read queries. Instead of returning a result, the write query updates the data and this is interesting to have in the evaluation.

Another small dataset, LevelStory, was provided by Neo4j that contains artificial company data. This benchmark was chosen mostly because of its small size and that it contains a huge and somewhat complicated query that is interesting to evaluate, see appendix B.8.

Benchmark	**Queries**	**#nodes**	**#relationships**	**#bytes**	**Size**
LDBC SF01	1, 3, 5, 6, 9, 10, 11, 12, SQ2, SQ6	3,159,000	16,800,000	1.4 GB	Medium
LDBC SF10	1, 3, 5, 6, 9, 10, 11, 12, SQ2, SQ6	31,000,000	168,000,000	14 GB	Large
Pokec	1, 5, 31, 32, 33, WQ17	1,633,000	30,623,000	3.5 GB	Medium
Social Network	7, 8, 9	1,251,000	56,358,000	1.85 GB	Medium
Logistics	1	807,000	4,842,000	403 MB	Small
Access Control	8, 9, 10, 11, 12, 13, 14	6,054,000	6,211,000	1.54 GB	Medium
MusicBrainz	14, 27	35,779,000	73,433,000	18.9 GB	Large
Generated Music	25, 30, WQ8	114,000	137,000	45.7 MB	Small
LevelStory	1, 2, 3, 9	19,000	72,000	89 MB	Small

Table 2.1: Table of the benchmarks and queries use for the evaluation as well as the size of the benchmarks. If a query is presented as a number it is a read query, SQ means that is a short read query and WQ stands for write query.

Chapter 3
Approach

To start the thesis work, the current Depth-First Search (DFS) implementation as well as the few uses of Breadth-First Search (BFS) in the Neo4j implementation were studied. Then our BFS prototype query processing operand, both a regular non-optimized and several optimized versions, were implemented using the Scala language. These operands were then integrated into Neo4j and an evaluation of their performance characteristics was done.

The result of these evaluations was compared to how the DFS implementation performed on the same queries and databases. A theoretical model, for when each algorithm performs best, was made based on the comparison. This model may then be used by the query planner to choose the search strategy.

3.1 Method

We implemented a BFS operand that uses a non-optimized Breadth-First Search from one source node in the normal case. This search finds all paths, not just the shortest, from the source node to every other reachable node. Afterwards it is filtered so that the paths end in one of the target nodes. It does not find just the shortest path since that was deemed the purpose of the operand, this is also what the current DFS operand does because of the edge-isomorphism. Because of the restriction that all paths should be found, optimizations regarding shortest path cannot be implemented for a general case.

The implementation part was conducted with test-driven development to make sure the code was correct from the beginning. This also made sure the behaviour of the code was not altered when implementing other parts. This was also proof that our solution worked as intended.

At the start of the implementation we made a hint for the planner, forcing the use of our operand. This was done to make sure the planner was not altered for other executions and it was easier to make sure that the operand was implemented correctly. The hint was picked up by the parser and later an option was added to be able to specify which node to start the search from. This was done for testing purposes.

Afterwards, all the piping stages between the operand and the planner were implemented. When this was done our operand was implemented. When our regular BFS was finished, optimizations were implemented as well. For all new implementations test cases were constructed to make sure that the functionality was correct.

When our operand was implemented, we conducted an evaluation. The evaluation was between the regular non-optimized, the different optimized BFS as well as the current search algorithm in the Neo4j implementation (Depth-First Search). It used the results from the benchmarks, found in appendix B, and the self-constructed test cases based on the graphs in section 3.2.5. The evaluation was based on the results of both runtime and memory usage. From this evaluation we constructed a theoretical model that determines which search method should be used for the theoretically fastest response. The plan is that this theoretical model can at some point be implemented into Neo4j to, hopefully, improve the speed of querying.

3.2 Implementation

The implementation was done by implementing an algorithm that uses a Breadth-First Search (BFS) to find all paths matching a pattern. Since all paths were wanted, no shortest path optimizations could be used but we implemented two Top down Bottom up inspired optimizations. These were based on the idea of splitting the searches. This is useful when the target and source node in the path are restricted and it uses multiple BFS to decrease runtime and memory usage. These are decreased since a big search from one node will contain more irrelevant nodes than multiple smaller searches.

3.2.1 The regular BFS operand

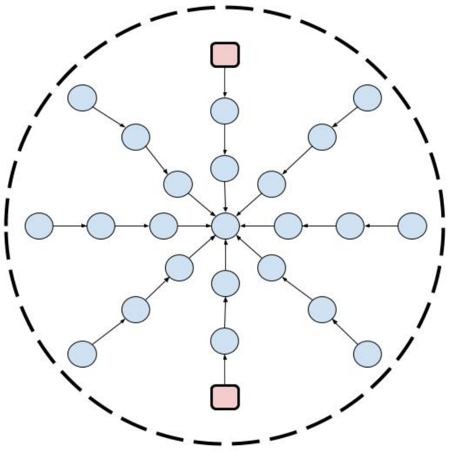
To start the implementation of the prototype, the most effective starting point was to implement a hint for the queries. The hint was ”`USING BFS ON name`” and it forces the planner to use our BFS operand on the path or relationship that was specified. By doing this we did not need to modify the plan or the cost model. By leaving the plan and cost model unmodified we knew that the operands would work correctly. When the hint was in place, the algorithm could be implemented. It was implemented as a regular BFS but without saving which nodes had been visited. This was because of, for all queries, all paths had to be returned and therefore nodes can be visited many times from different nodes. If a query specify that not all paths should be returned the results are filtered after the search is done, since this is the way the pre-existing code and planner were designed.

Our algorithm takes every node that is given as input by the planner and starts a BFS from them. Our BFS operand uses a queue that contains a node and the relationships that leads to it from the source node. A small example of a query and a graph can be found in figure 3.1. The query says to find paths of length zero to six and returning all paths that fit the specifications. It then specifies one node to be called ”source” and the other ”target”. The graph shows the nodes ”source” and ”target” as the squares and the search finds all nodes inside the dashed marking. For this query and graph all nodes will be found.

When our algorithm was working we decided, for testing purposes, that it would be useful to be able to specify from which node in the relationship the BFS should start from so the hint was modified to optionally add ”`FROM node`”. Before the benchmarks could be run the operand had to be able to handle paths, which can contain single or multiple relationships, so this functionality was added and tested.

```
MATCH p=(n1)-[*0..6]-(n2)
USING BFS ON p
WHERE n1.name = "source"
  AND n2.name = "target"
RETURN p
```

(a) A query for finding paths between two specified nodes.



(b) An example graph where the squares are the source and target nodes. The dashed marking shows the search area.

Figure 3.1: A query and graph example for our regular BFS operand.

3.2.2 Optimization on divided paths

When our BFS operand was working it had to be optimized and this was not an easy task to do for a general query. This was the case because we could not use shortest path optimizations, because we had to allow repeated nodes and needed to find all paths between the nodes. The first optimization that we implemented *only* worked on divided paths, such as `p=()-[*0..5]-()-[*0..5]-()`, while our regular BFS could also work for united paths like `p=()-[*0..10]-()`. It built on our Top down Bottom up design and uses multiple Breadth-First Searches. These searches each take one input node, either a specified source or target node, and are then joined on mutual nodes by a pre-existing join operand. This means that for a query with specified source and target nodes separate searches are started in each of these nodes, which can be seen in figure 3.2b.

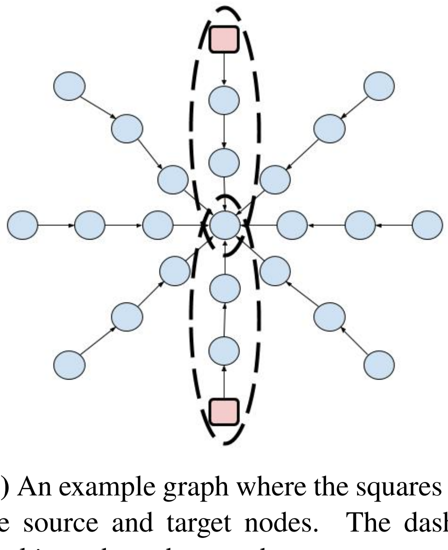
To use this divided Top down Bottom up BFS (TB BFS) we modified the `FROM` hint to handle two nodes instead of one and reduced the cost for a plan with the pre-existing join function in the presence of the modified hint. The idea of this solution is that conducting two BFS searches with half of the depth will dramatically decrease the number of nodes that will be unnecessary visited in the search and therefore reducing the runtime and used memory. This can be seen in figure 3.2 where an example query can be found together with a graph that has its search space marked. The query says to find two paths of length zero to three and these two paths should be connected by an anonymous node. It then specifies one node to be called ”source” and the other ”target”. Then it should return all full paths that fit these specifications. This query is the divided version of the query in figure 3.1 and it is manually split in the query. The graph in figure 3.2 shows the nodes ”source” and ”target” as the squares and the search finds all nodes inside the two dashed markings. For this query and graph only the nodes in the path through the middle will be found, so the search uses the middle node as the anonymous node between the two paths. Comparing the search space for the graph in figure 3.2 and 3.1 shows a great reduction in searched nodes for our TB BFS operand. For this graph and query our TB BFS will find a quarter of the nodes that our BFS finds.

This optimization is not always useful since for small searches it might do more work than the regular non-optimized version and to use this optimization the relationship has to be divided in the query. This optimization is also worse than our regular BFS operand when the source and target nodes are not specified, since this leads to computing a search from every node instead of just from one node. However our TB BFS operand is very useful and scalable when some restrictions are given for the source and target nodes. As an example, if the query handles a path where the types of the source and target nodes are specified, a BFS is started in each node in the graph that fits the given types. This can greatly reduce the amount of searched nodes for larger graphs.

When implementing our TB BFS we started with two specific nodes, using a `WHERE` clause, and later modified it to also handle the source and target nodes being specified by type. This enables an easier way to search with multiple source and target nodes, which is more versatile and therefore useful. Because real world queries almost never contain two specific nodes but searching by types or mixing a specific node and types are quite common, as we could see in the benchmarks. Executing our TB BFS on types increases the speed and decreases the memory usage compared with our regular BFS when the graph is large and highly connected. This is since a high number of nodes can be discarded from the search and doing joins on multiple nodes cost as much as using it for our TB BFS with two specified nodes.

```
MATCH p=(n1)-[*0..3]-()
     -[*0..3]-(n2)
USING BFS ON p FROM n1,n2
WHERE n1.name = "source"
  AND n2.name = "target"
RETURN p
```

(a) A query for finding paths between two specified nodes. This is a divided version of the relationship found in figure 3.1a.



(b) An example graph where the squares are the source and target nodes. The dashed markings show the search areas.

Figure 3.2: A query and graph example for our divided Top down Bottom up BFS operand.

3.2.3 Optimization on united paths

When we ran our regular BFS on types a bug appeared, that caused us to get almost 110,000 results instead of the correct single result. The origin of this bug was that instead of one input node, the search got both the source and target node as input at the same time. The two input nodes came from the use of a pre-existing operand, called Cartesian product, in the plan. A Cartesian product of two sets is a set of all the combinations of the elements in the input sets [22], for example $A = \{1, 2\}$ and $B = \{3, 4\}$ gives the Cartesian product $A \times B = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$. For our BFS operand this means that the input to the search is all combinations for the source and target nodes. When the Cartesian product was used, the normal filtering step was not planed and therefore not used. This meant that the result from our regular BFS contained all paths that could be reached from the source nodes and therefore thousands of unwanted paths. This lead to us implementing two optimizations, one that does a single search and one that does two searches for each input, which both filters their own result.

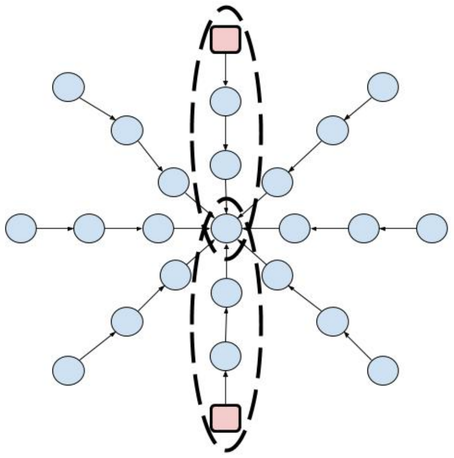
The first optimization was a single search algorithm which was just a modified version of our regular BFS. It only returned the paths leading from the source to the target node, instead of just returning every path from the source node. This is because of the missed filtering step. This method was used if the maximum depth of the search was unspecified or too small, since it would not benefit from a Top down Bottom up search in those cases.

The second optimization was an implementation for two searches built on our Top down Bottom up design which no longer required a divided path, instead having united paths like `p=()-[*0..10]-()`. This Cartesian Top down Bottom up BFS optimization (CTB BFS) also required the use of the `FROM` hint with two nodes, in addition to a maximum depth. Our CTB BFS starts two BFS of half the depth of the query, one from the source node and one from the target node, since otherwise more nodes would be visited than for our regular BFS. These two searches were then manually joined on the mutual nodes and only the whole paths were returned. An example query and graph, with marked search space, can be found in figure 3.3. The query says to find paths of length zero to six between two nodes, where one node has type ”S” and the other has type ”T”. It then returns all paths that fit these specifications. The graph shows the nodes of types ”S” and ”T” as the squares and the search finds all nodes inside the two dashed markings. For this query and graph the same amount of nodes will be found as in the example in 3.2. However, the query in this example is much simpler since it uses a united path. It also looks more like the query from the example in figure 3.1.

An optimization that we made for CTB BFS is that the result from the searches were stored, together with their input node and maximum depth, since they are most likely to be reused at a later search and then does not have to be recalculated which leads to a faster runtime. The results are stored together with their input node and maximum depth because if a node is valid as both a source and a target node and the depth is uneven, the same results cannot be used in both cases and therefore the same input node has to be stored twice. The reuse of results is likely because of the Cartesian product, in the example above is all the original input featured twice in the combined input and for bigger sets of A and B will each input feature even more times. The biggest differences between our CTB BFS and TB BFS is that the search itself has two input nodes instead of one, return the already filtered paths from the source to the target node and the relationship does not have to be split in the query.

```
MATCH p=(n1:S)-[*0..6]-(n2:T)
USING BFS ON p FROM n1,n2
RETURN p
```

(a) A query for finding a path between two nodes with specified types.



(b) An example graph where the squares are the source and target nodes. The dashed markings show the search areas.

Figure 3.3: A query and graph example for our Cartesian Top down Bottom up BFS operand.

3.2.4 Abandoned optimizations

Splitting relationship

Before we thought of the Cartesian product, we considered another optimization for our divided Top down Bottom up BFS optimization (TB BFS). This optimization was built on the thought that the path should not have to be divided manually but automatically for the queries that would benefit from this. This method would split the concerned relationship in half and add an anonymous node in the middle, keeping all information about the old relationship in both the new relationships. This would not be much different than the TB BFS in itself, except that the query would not need to be manually split. The automatic splitting of relationships sadly did not work, since the amount of variations in queries and special cases were too great. In addition to this the execution plan had to be verified before the execution and in this verification step the split query would be rejected based on it containing the new and unknown node and relationships and thus the functionality had to be abandoned.

Building several paths at once

Another abandoned optimization was built on the thought of reducing the amount of times a node would be handled in the search, since the paths going from the node would be re-calculated every time. To do this the queue of nodes waiting to be visited was changed to hold every found path to the nodes instead of having the same node multiple times with different paths. This lead to the queue being searched every time a node was found, if the node already existed in the queue then the list of paths leading to it was updated with the new path and if it did not exist then the node and path were added at the end of the queue.

Like our regular BFS this method was supposed to return one path at the time but when handling each node several paths would be found. These paths were stored and instead of continuing the search with the next node in the queue, they were returned one at the time until all had been returned and only then did the search continue from the queue.

This was abandoned for several reasons, one being the need for a lot more storage and the time spent searching the queue for each node that was found. A second reason was that the order of the returned paths no longer followed the BFS pattern, growing by length, but instead came in a random order. This random order occurred since the longer paths were inserted into the queue, to where their nodes were first discovered, instead of added to the end. We fully implemented this optimization, in difference to the splitting relationship optimization, finding it to be slower than our completely non-optimized BFS. This also contributed to the abandonment.

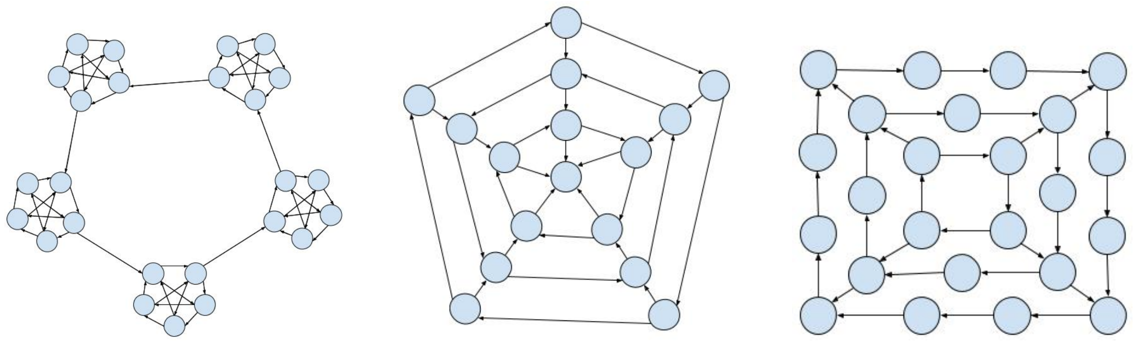
3.2.5 Graphs for memory evaluation

We manually constructed several graphs to be used in the test cases for monitoring memory usage. Regular test cases were used instead of benchmarks because memory usage could not be monitored for the benchmarks in Neo4j's implementation. It was also easier to manage what the searches were and the test cases could be run more frequently whilst making sure that the right plans were used. The graphs we used in these test cases were the star graph (figure 3.4a), the cobweb graph (figure 3.4b) and the cube graph (figure 3.4c), described below.

Star graph This graph contains clusters of five nodes where all the nodes in a cluster are connected to each other. Two of the nodes in a cluster are connected with one extra node each, both from a different cluster. This forms a ring of clusters.

Cobweb graph This graph is built by one node in the centre of the cobweb with tendrils containing three nodes that are connected to it, where the nodes in the tendrils are related towards the centre of the cobweb. The nodes in the tendrils are related to the corresponding nodes in the two neighbouring tendrils. This goes on for all tendrils thus forming circles between each layer in the tendrils, just like a cobweb.

Cube graph This graph has an inner layer that contains four nodes and each new layer of nodes have four more nodes than the previous. All nodes in each layer are connected with two other which forms a square. The corners of each square are connected to the corresponding corners in the two neighbouring squares.



(a) A star graph with a ring of five connected star clusters, the clusters always contain five nodes each.

(b) A cobweb graph with five tendrils, each tendril is always three nodes long.

(c) A cube graph with three layers, each layer always contains four nodes more than the previous layer.

Figure 3.4: Smaller versions of the graphs that were used to test the runtime and memory usage for our different BFS operands, as well as the pre-existing DFS operand. The graph in (b) was used with both directed and undirected crossings between the tendrils.

We designed the graphs to emulate social networks with the star graph representing cliques of five friends where two of the friends are friends with people from two different cliques. The cobweb graph mimics a network where a celebrity connects all of the users with a shortest path, and comes in both a directed and an undirected version. Lastly the cube graph was designed so that our TB BFS would not be optimal in all cases. This is because when one of the source nodes and one of the target nodes are in the outermost layers our TB BFS cannot discard any nodes. The reason we only used our designed graphs was because there did not exist any pre-made graphs for testing, since benchmarks are used for this for Neo4j. Since we had to build the graphs ourselves, the only manageable option was to use for-loops and therefore the graphs are symmetrical. We also wanted to be able to have the same types of graphs in different sizes which could not be done by randomizing the graph each time. Randomizing the graph would also make it impossible for us to manage the searches.

Chapter 4
Evaluation

Here follows the results from the presented benchmarks and self-constructed test cases. They are compared and discussed further, before the theoretical model is developed and presented.

4.1 Experimental Setup

The evaluation was done by using our different versions of the Breadth-First Search (BFS) and the pre-existing Depth-First Search (DFS) on the queries shown in appendix B. This gave a good comparison of the speed of the searches but not the memory usage. To check the memory usage some test cases were constructed that measured the memory usage and the runtime. These test cases were built on the graphs found in section 3.2.5. The runtimes for the benchmarks were measured by Neo4j's benchmark tools and then delivered to us. These times were mean times of the runs and some other metrics that we did not really use. For the test cases however, we had to measure the times and memory consumption ourselves. The runtime was done by checking the system time before the run and then afterwards and keeping the difference. For the memory usage, a method existed that gave the allocated bytes. We used this method in the same way as the time measurement. Both comparisons were weighted together, by manually comparing the results between operands, to get a better evaluation. This was done since the cost efficiency cannot be determined by only using the speed, the memory usage is just as important for it. Testing all algorithms on the same queries and benchmarks gave a good evaluation and gave results that was good enough to construct a heuristic model, since some patterns could be seen.

4.2 Results

The results from running the benchmarks and test cases will be presented in diagrams and tables in this section. For some benchmarks the diagrams and tables are divided in multiple figures, this is because of readability since the scales are different and not because some queries are more interesting. The results from the benchmarks are the mean times since this was the information returned from the benchmarks. Before each result was taken we checked that the variance was acceptable and since we ran each query multiple times they almost always were. The only times they were not was when something went wrong, such as measuring the time in milliseconds when the query executed in microseconds. For the test cases we only measured the time and added to our tables, not calculating the mean time nor any variance. The results will be divided into sections based on the benchmarks in question (see appendix B), with one section for each benchmark and one for the test cases. In the next section (4.3) will the results be discussed and compared.

4.2.1 LDBC SF01

This section contains the results of running the LDBC benchmark (found in appendix B.1) with scale factor 1 for two versions of our regular non-optimized BFS, our divided Top down Bottom up BFS optimization (TB BFS) and the original DFS. The two regular BFS versions are: one where the node to start the search from is specified and one where the planner choose start node by itself. The algorithms were ran on two different computers, both the regular BFS versions on one and TB BFS and DFS on the other, however both programs contained the same code.

The results for running our two regular BFS and the DFS locally are found in figure 4.1 and indicated that for all but two of the queries our BFS solution without any optimization was slightly slower than the DFS algorithm. For Query 5 and 12 our BFS was substantially slower, however using the `FROM` hint and specifying which node to start searching from improved the runtime. For Query 12, our BFS operand was 6 times slower than DFS and when using the `FROM` hint it was only 4 times slower than DFS, seen in figure 4.1a. Using the `FROM` hint was always faster than regular BFS and it was also faster than the DFS for Query 5 and 10 and for Short Query 2. In figure 4.1c, we can see that for Short Query 2 that using the `FROM` hint for our BFS was 41% faster than our BFS without the hint and 21% faster than the DFS.

For our TB BFS we had to manually divide queries to run the optimization. This could only be done for two of the LDBC queries, Query 1 and 10, since the rest had too short length in the relationship to be divided. For these two queries we also tried increasing the length of the relationship by one so that we could see how big of a difference it made. One of the queries can be found in table 4.1, both the original version and the divided and increased maximum length versions.

In table 4.2 the results of the locally run TB BFS are shown for these queries. It shows that the DFS was faster than our TB BFS for the unchanged maximum length of the relationships in the paths. Increasing the maximum length of the relationships by one showed that TB BFS was a little faster for Query 10 and much faster for Query 1. The difference for Query 1 was great enough that our TB BFS was run little more than 60 times while the DFS was only run less than 10 times, this was because running it took much time.

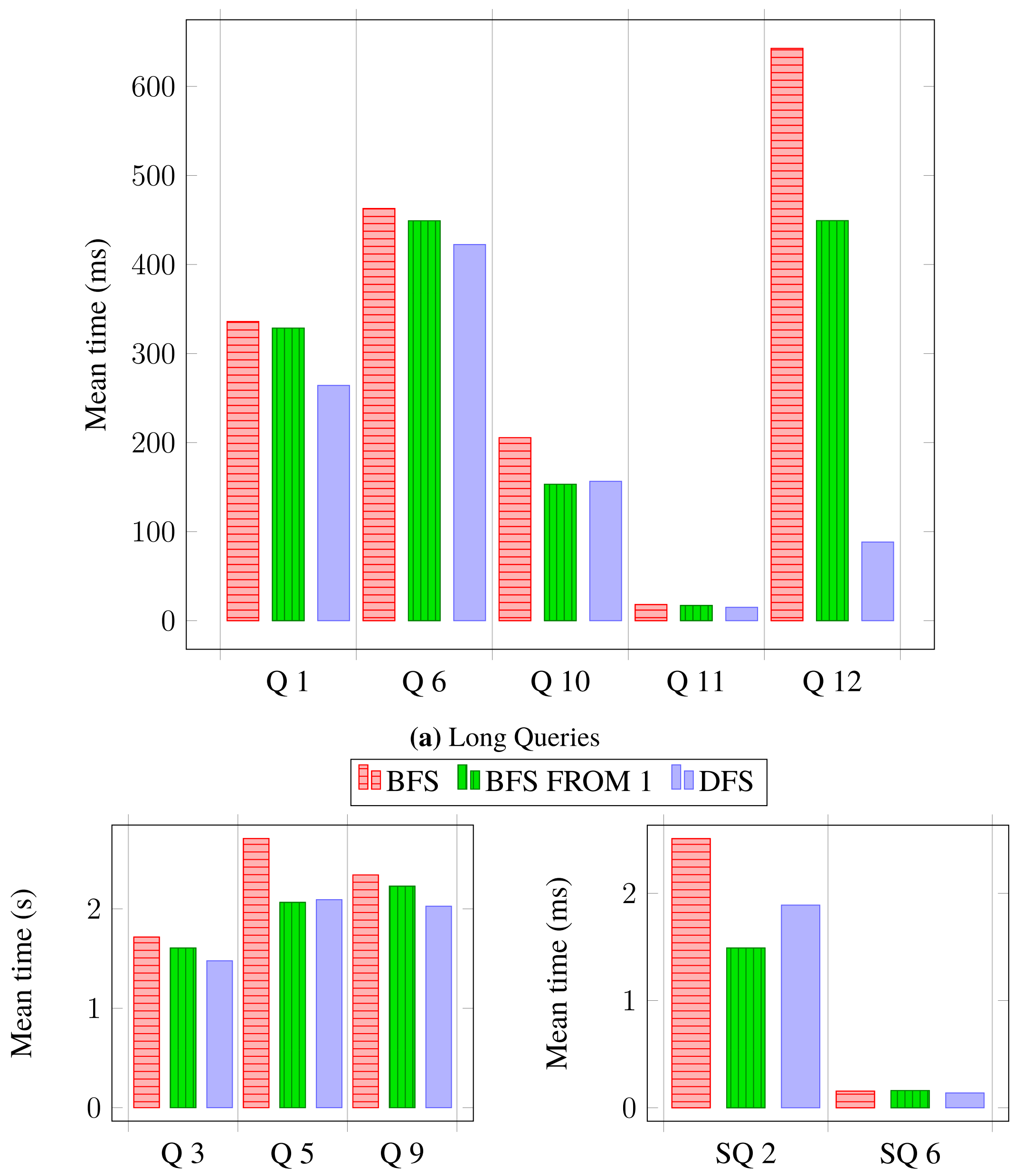


Figure 4.1: The mean times, in ms and s, for the local runs of the LDBC benchmark with scale factor 1. In (a) and (b) the results from the longer and more complex queries can be found while (c) contains the result from the two shorter queries. BFS FROM 1 is our regular BFS with a manually specified node to start the search from.

	United path	**Divided path**
Original length	(person:Person)-[:KNOWS*1..3]-(friend)	(person:Person)-[:KNOWS*0..1]-()-[:KNOWS*1..2]-(friend)
Increased length	(person:Person)-[:KNOWS*1..4]-(friend)	(person:Person)-[:KNOWS*0..2]-()-[:KNOWS*1..2]-(friend)

Table 4.1: The relationship that was divided for our TB BFS use from Query 1 of the LDBC benchmark. Both with and without the increased length.

Benchmark	**TB BFS**	**DFS**
LDBC Query 1	378.2 ms	298.1 ms
LDBC Query 10	180.6 ms	152.7 ms

(a) Queries with unchanged maximum length.

Benchmark	**TB BFS**	**DFS**
LDBC Query 1	1,780.6 ms	38,434.0 ms
LDBC Query 10	1,847.5 ms	1,955.5 ms

(b) Queries with increased maximum length by 1.

Table 4.2: Tables over the mean time, in ms, for the local runtimes of the LDBC benchmark with scale factor 1. TB BFS is our divided Top Down Bottom Up BFS optimization. These are the benchmarks where our TB BFS could be applied.

To get more reliable evaluation of the results the same benchmarks were also run on remote machines. These machines are the ones Neo4j use for their own benchmarking. The results from remote runs could then be compared to the local results. The results of these runs can be seen in figure 4.2 for the DFS and our two regular BFS versions. The results of our BFS and the DFS runs showed the same patterns as when run locally and therefore the locally run benchmarks could be trusted. Running more benchmarks both locally and remotely would have made the results more reliable, but this was only applicable for this benchmark. However our BFS was faster than the DFS for Query 5, 6, 9 and 10 as well as for Short Query 2, when using the `FROM` hint it was also faster on Query 3 and 11. Just as for the local runs using the `FROM` hint was always faster than not using it. When it comes to Query 12 we were still slower than the DFS, but now we were only 4 times slower without the `FROM` hint and 3 times slower with it.

The results of splitting Query 1 and 10 and running TB BFS is shown in table 4.3, together with the result of the DFS runs. Without the increased maximum length is the DFS faster than our TB BFS for Query 1, but for Query 10 is our TB BFS much faster than the DFS. For the increased length is TB BFS much faster than DFS for both queries. In difference to the local runs, the number of runs were equal between the different algorithms. However for the increased length, DFS had only one repetition while the rest had five.

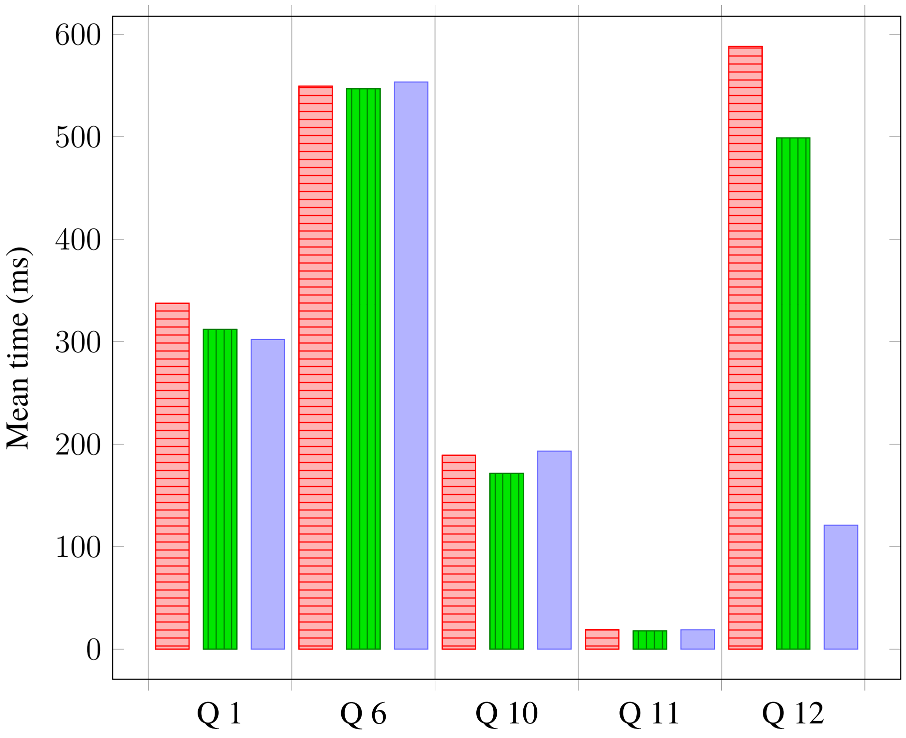
Benchmark	**TB BFS**	**DFS**
LDBC Query 1	1,650 ms	302.2 ms
LDBC Query 10	4.26×10^{-3} ms	193.9 ms

(a) Queries with unchanged maximum length.

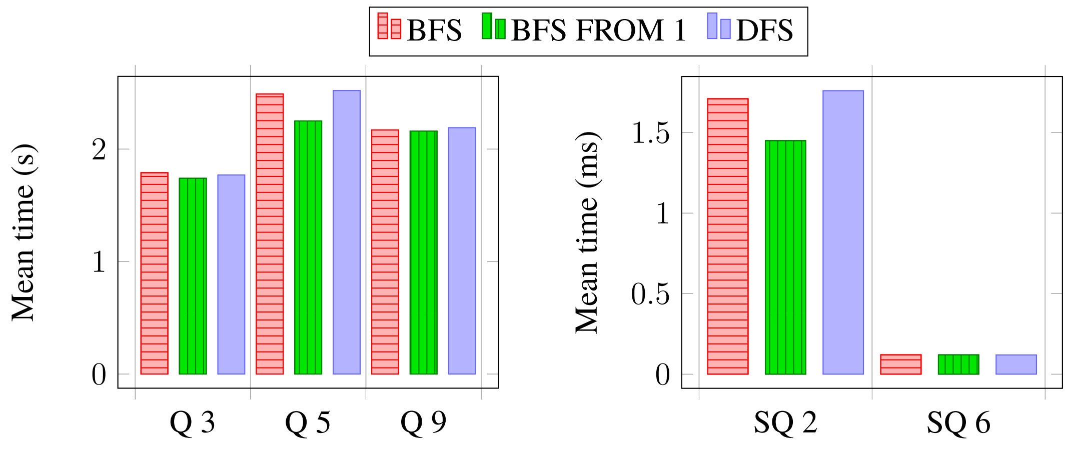
Benchmark	**TB BFS**	**DFS**
LDBC Query 1	1,780 ms	32,504 ms
LDBC Query 10	2.88 ms	1,080 ms

(b) Queries with maximum length increased by 1.

Table 4.3: Tables over the mean time, in ms, for the remotely run LDBC benchmark with scale factor 1. TB BFS is our divided Top Down Bottom Up BFS optimization. These are the benchmarks where our TB BFS could be applied.



(a) Long Queries



(b) Long Queries **(c)** Short Queries

Figure 4.2: The mean times, in ms and s, for the LDBC benchmark with scale factor 1, run remotely. In (a) and (b) the results from the longer and more complex queries can be found while (c) contains the result from the two shorter queries. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

4.2.2 LDBC SF10

The LDBC benchmarks (found in appendix B.1) were also run on scale factor 10, this means that the queries are the same but the size of the graph is bigger than for scale factor 1. The benchmarks were only run remotely because when they were run locally, our machines could not allocate enough memory. The result from running our two regular BFS and the DFS queries can be found in figure 4.3. The results in this figure are in a different order than for SF01, this is because of readability. The figure still contains the same queries. From these results it could be seen that it was not always faster to use the `FROM` hint, in fact it only gave a faster result for Query 5 and 6 where it was the fastest. However, now our regular BFS was faster than the DFS for Query 3, 5, 9, 10 and 11 and for Short Query 2. For some of these queries, the runtime difference between our BFS and the DFS was very small. An example is for Query 9 where the difference is 71 μs.

In table 4.4 the result of running TB BFS and DFS can be seen for the queries that TB BFS could be applied to, the same queries as for scale factor 1. For the unchanged maximum length can the same observation be made as for the remotely run scale factor 1, Query 1 is much slower but Query 10 is much faster for TB BFS than for DFS. For the increased length is both queries much faster for TB BFS than for DFS.

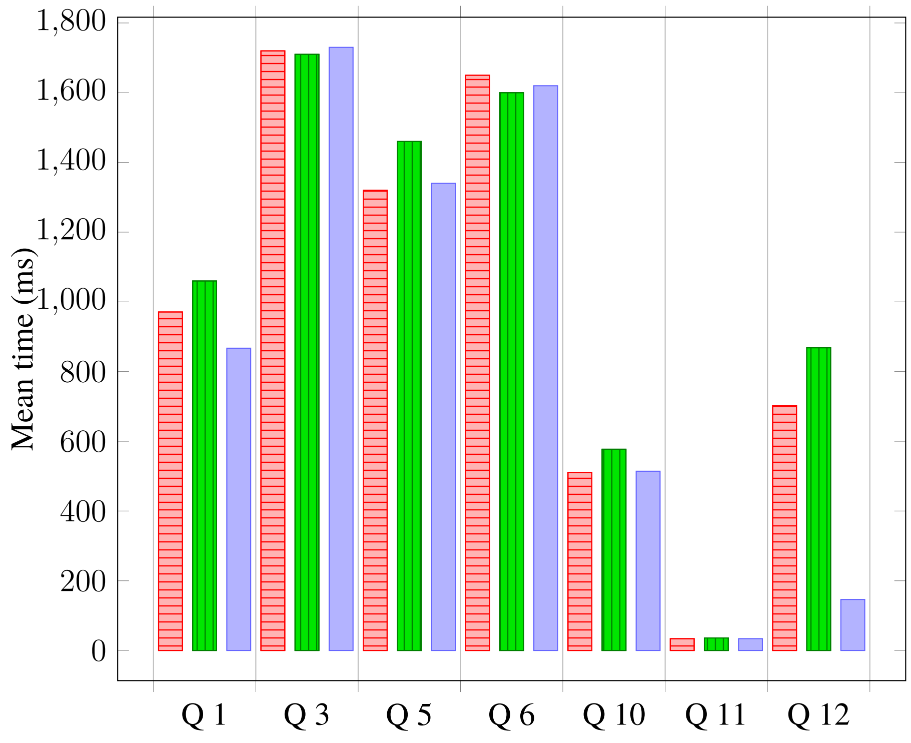
Benchmark	TB BFS	DFS
LDBC Query 1	18,650 ms	867.1 ms
LDBC Query 10	4.71×10^{-3} ms	514.1 ms

(a) Queries with unchanged maximum length.

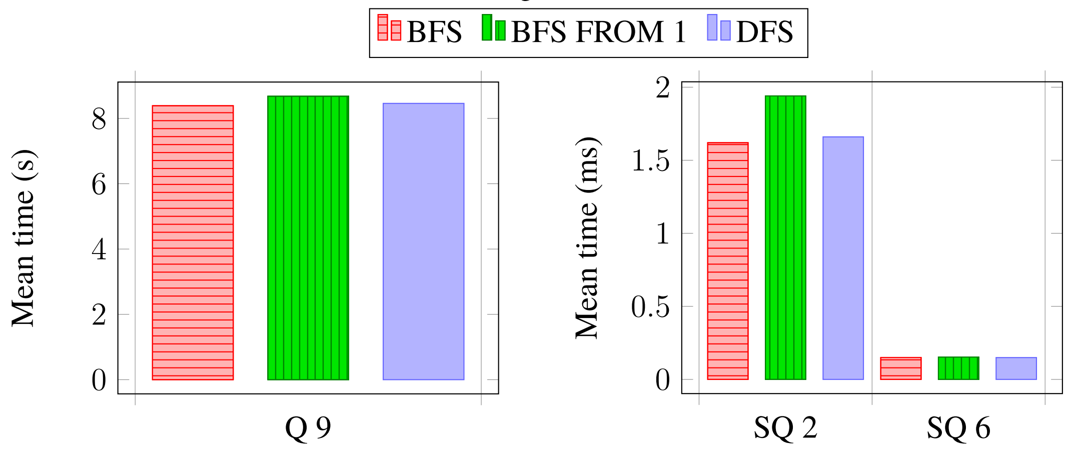
Benchmark	TB BFS	DFS
LDBC Query 1	72,630 ms	229,480 ms
LDBC Query 10	10.95 ms	8,290 ms

(b) Queries with maximum length increased by 1.

Table 4.4: Tables over the mean time, in ms, for the remotely run LDBC benchmark with scale factor 10. TB BFS is our divided Top Down Bottom Up BFS optimization. These are the benchmarks where the TB BFS could be applied.



(a) Long Queries



(b) Long Query 9

(c) Short Queries

Figure 4.3: The mean times, in ms and s, for the LDBC benchmark with scale factor 10, ran remotely. In (a) and (b) the results from the longer and more complex queries can be found while (c) contains the result from the two shorter queries. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

4.2.3 Pokec

For the Pokec benchmarks (found in appendix B.2) our BFS operand was slower for most read queries (see figure 4.4). However, our BFS operand was 10% faster than the DFS operand for read Query 1, which is the smallest Pokec query. Query 31, 32 and 33 from this benchmark used an optimization for the DFS operand, which were not implemented for our BFS operand. The results for these queries can be found in figure 4.4) and table 4.5. The optimized DFS operand were 3 times faster than our BFS operand for Query 31. For Query 32 and 33 the optimized DFS operand took around 3 and 10 seconds respectively, while our BFS operand ran out of memory. When running the DFS operand without its optimizations it showed that it was slower than our BFS operand for Query 31. However, the DFS operand without optimizations could execute Query 32, even though it took 4 times longer than optimized DFS operand, whereas our BFS operand could not.

For Write Query 17 our BFS operand was 15% faster than the DFS operand, again see figure 4.4). However, the query is very similar to Query 5 and for this query the DFS operand was slightly faster than our BFS operand. The reason we are faster than DFS operand for Write Query 17 is how we handle the transactions. Our BFS operand finds all paths and then performs the write aspect on all found paths at once, whereas the DFS operand returns the found paths one by one as they are found. Because of this, the search transactions fight for resources with the write transaction. This is due to the fact that search transactions wants to read from the database while the write transaction locks the database to be able to modify data uninterrupted. This can slow down the operand.

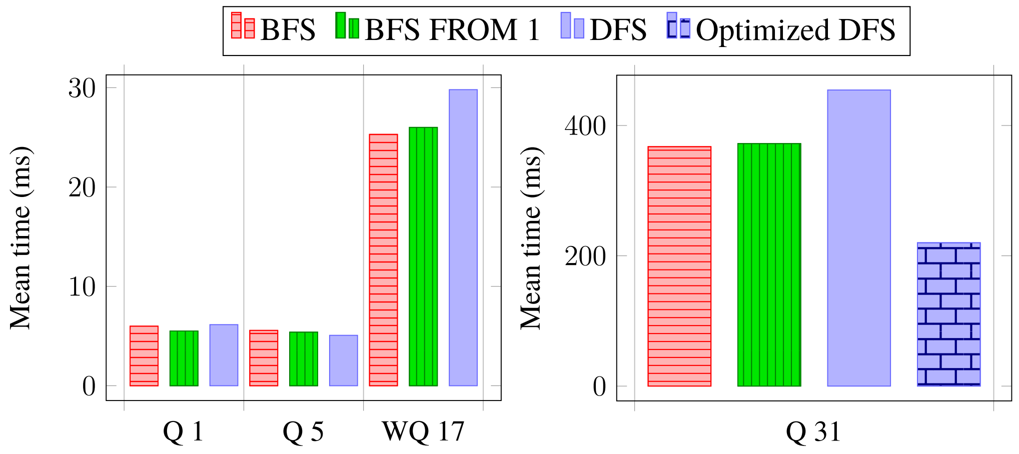


Figure 4.4: The mean times, in ms, for the Pokec benchmark. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

Query	DFS	Optimized DFS
Q 32	9.2 s	2.6 s
Q 33	Out of memory	10.3 s

Table 4.5: Results, in seconds, for Query 32 and 33 for DFS and its optimization.

4.2.4 Social Network

From the results of the Social Network benchmark (found in appendix B.3) it could be seen how much difference it makes when our BFS operand is only applied to one relationship in the query, see figure 4.5. We ran the queries on both the whole path and on specific relationships in the path, since that could lead to better results. For Query 7, our BFS operand used on the whole path was 17% faster than the DFS operand and when used on one relationship we were 22% faster. For Query 8 our BFS operand used on one relationship was 14% slower than the DFS operand, however the DFS operand was 17% faster than our BFS operand used on the whole path. For Query 9 our BFS operand used on the whole path was 13% faster than the DFS operand but our BFS operand used on one relationship was slower than the DFS operand. To summarize the results, our BFS operand used on one relationship was the fastest for Query 7 but worst for Query 9. When used on the whole path, we had the best performance for Query 9 but the worst for Query 8. To compare, the DFS operand gave the slowest result for Query 7 but was the best choice for Query 8.

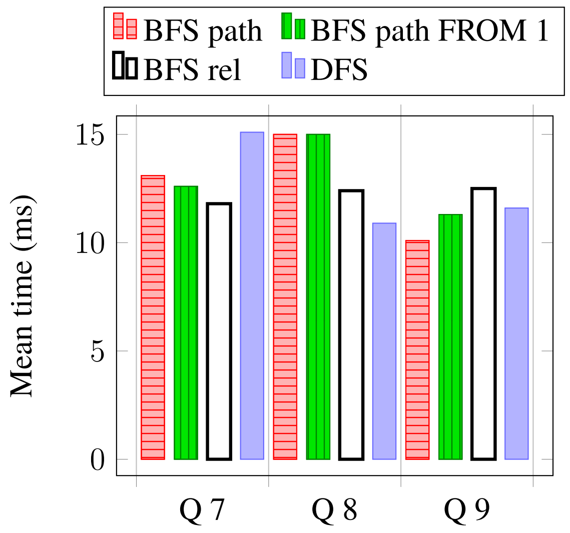


Figure 4.5: The mean times, in ms, of the runs on the Social Network benchmark. BFS path FROM 1 is our regular BFS on the whole path where the nodes to start the search from are manually specified. BFS rel is our regular BFS specified to only one relationship in the path, with the rest of the relationships running DFS.

4.2.5 Logistics

The Logistics benchmark (found in appendix B.4) was thought to be suboptimal for BFS because of the small size. However, as can be seen in figure 4.6 that was not the case. The DFS operand took about 125 ms while our BFS operand took just below 12 ms, making our BFS operand 91% faster than the DFS operand. The logistics graph is not only small but also well connected, averaging 6 relationships per node, which might be the reasons for the big improvement.

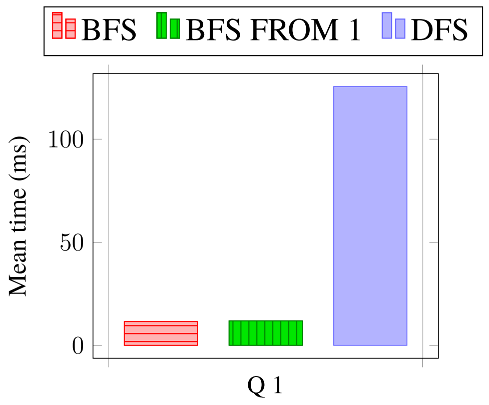


Figure 4.6: The mean times, in ms, for the Logistics benchmark. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

4.2.6 Access Control

The results, for the Access Control benchmark (found in appendix B.5), in figure 4.7 shows that using our BFS operand on one relationship in the path instead of on the whole path can be very powerful. The runtime is also affected by the usage of the `FROM` hint, which can be seen especially for Query 9. For this query our BFS operand used on the whole path with the `FROM` hint is faster than using it on one relationship without the hint, while our BFS operand on the whole path without the hint is slower. When comparing our BFS and the DFS operands for Query 9, we find that our BFS operand used on the whole path is slower than the DFS operand. However, our BFS operand used on one relationship, both with and without the `FROM` hint, and on the whole path with the `FROM` hint are about 10% faster.

For Query 10, 11 and 13 our BFS operand used on the whole path, both with and without the `FROM` hint, is faster than using it on one relationship without the hint. For Query 10 our BFS operand on one relationship with the `FROM` hint is faster than both the whole path versions. When it comes to comparisons with DFS, all our BFS versions are slower for Query 13. However for Query 10, our BFS operand used on one relationship with the `FROM` hint is slightly faster than the DFS operand. We are also slightly faster then the DFS operand on Query 11 for both versions of our BFS operand used on the whole path.

All our BFS versions are slower than the DFS operand for Query 12, but there are differences between our BFS operand being used on the whole path and on one relationship. Comparing using our BFS operand on the whole path with the `FROM` hint and on one relationship gives $10 - 20\%$ faster runtimes in favour of relationship, depending on the use of the `FROM` hint for the relationship version. The results from all the Access Control queries can be found in appendix A.1.

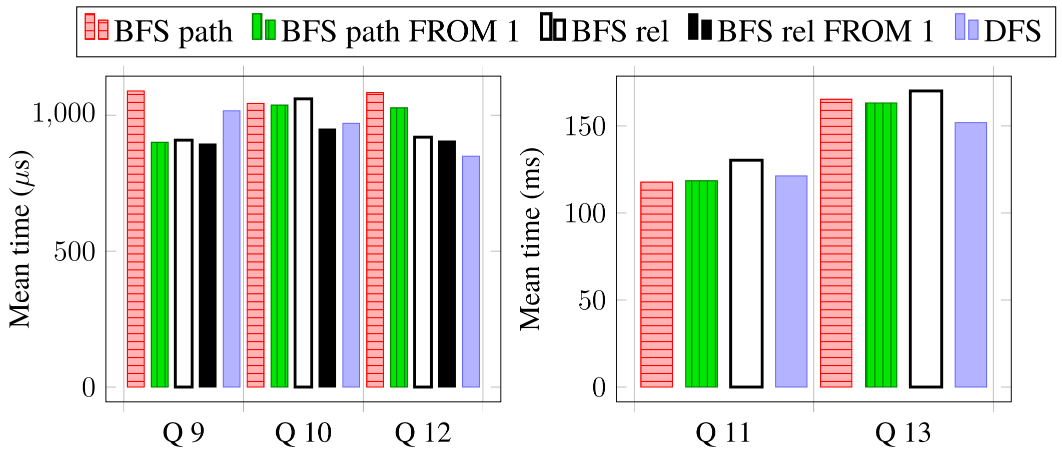


Figure 4.7: The mean times, in μs and ms, for some of the runs on the Access Control benchmark. BFS path FROM 1 is our regular BFS on the whole path where the nodes to start the search from are manually specified. BFS rel is our regular BFS specified to only one relationship in the path, with the rest of the relationships running DFS.

4.2.7 Music Brainz

In the Music Brainz benchmark (found in appendix B.6) there was an opportunity to try to use our CTB BFS on Query 27. Unfortunately, this attempt ran out of memory but our other BFS versions were on par with the DFS runtime, see table 4.6. However, the mean times were very fast and the small differences could depend on several different factors. The results from all the Music Brainz queries can be found in appendix A.2.

Operand	Time in μs
BFS	16
BFS FROM 1	17
DFS	16

Table 4.6: Results for Query 27 for the Music Brainz benchmark. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

4.2.8 Generated Music

For Query 25 for the Generated Music benchmark our CTB BFS could be used. However, it did not perform well as it took 60 times longer than using our regular BFS, as seen in table 4.7a. In tables 4.7b and 4.7c the results for Query 30 and Write Query 8 are presented. Query 30 has less restrictions on the nodes than Query 25 however the relationship has restrictions on it. For this query our BFS operand is 15% faster than the DFS. Write Query 8 is similar to Query 30, but has an additional write part. Our BFS operand was only slightly faster than the DFS operand for Write Query 8 which can be attributed to the extra part of the query. The Generated Music benchmark can be found in appendix B.7 and the results from all the queries, in diagrams, can be found in appendix A.3.

operand	Time in μs
CTB BFS	16,293
BFS	269
BFS FROM 1	261
DFS	231

(a) Query 25

operand	Time in ms
BFS	3.057
BFS FROM 1	2.963
DFS	3.480

(b) Query 30

operand	Time in ms
BFS	6.069
BFS FROM 1	5.987
DFS	6.079

(c) Write Query 8

Table 4.7: Results for the Generated Music benchmark. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product.

4.2.9 LevelStory

For the LevelStory benchmarks (found in appendix B.8) our BFS was better for all queries except for Query 1 where it was on par with the DFS, see figure 4.8. Query 1 has an unbound maximum length relationship, which affects our BFS negatively. For the small Query 3, our BFS operand was 30% faster than the DFS. Query 2 contained a relationship with a bigger length and for this our BFS was 6% faster than the DFS.

For Query 9 there was an opportunity to use our BFS on one relationship in the path and not on the whole path. It was a bigger query and when using our BFS operand on the whole path it was slower than the DFS operand. However, when applying our BFS on one relationship we were 9% faster than the DFS instead. We were also 20% faster than using our BFS on the whole path. Using the `FROM` hint on the relationship version gave even faster results, while using it on our BFS on the whole path gave a slower runtime than without the hint.

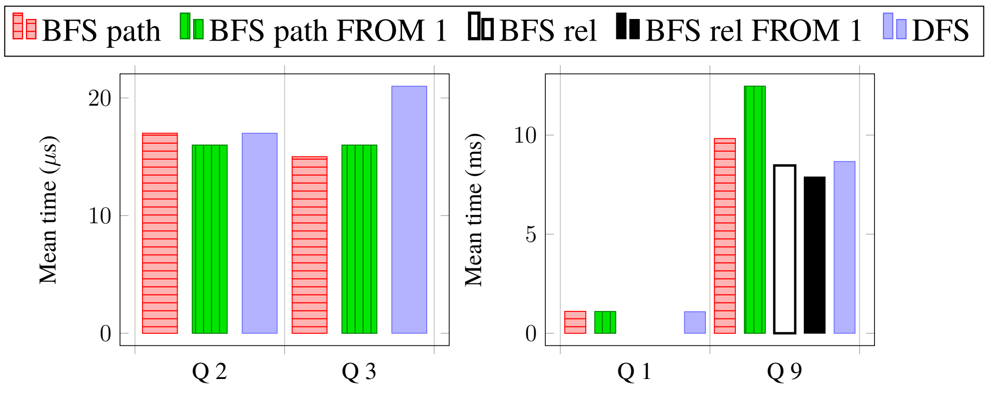


Figure 4.8: The mean times, in μs and ms, of the runs on the LevelStory benchmark. BFS path FROM 1 is our regular BFS on the whole path where the nodes to start the search from are manually specified. BFS rel is our regular BFS specified to only one relationship in the path, with the rest of the relationships running DFS.

4.2.10 Test Cases

The performance of our BFS was also tested with regular test cases using the graphs presented in section 3.2.5. For these test cases our divided Top down Bottom up BFS optimization (TB BFS), our Cartesian Top down Bottom up BFS optimization (CTB BFS), our regular BFS and the DFS tested at the same time with the same settings. It also gave the option to run the same query multiple times. Running the same configuration multiple times gives better results since this will act as a sort of warm-up, like for the benchmarks. The simple queries for the test cases, their runtimes and memory usages can be found in tables 4.8 and 4.9 for the star graph, tables 4.10 and 4.11 for the cobweb graph with undirected crossings and table 4.12 for the cube graph. Full results can be found in appendix A.4 and only the most significant will be presented here. The results in the result tables are from running the queries three times and keeping the last two results, since after one run the caches can be used so the first run is a warm-up.

For queries with two specified nodes our CTB BFS cannot be used since it will never be scheduled without the use of types, therefore there are no results of that. For most of the test cases which uses types in the query our CTB BFS were scheduled. Our regular BFS execution plans were not checked to see if they used the Cartesian product or not, therefore is it possible that they use our single search Cartesian BFS optimization. This was not seen as a problem since the only difference between regular BFS and single search Cartesian BFS was the filtering. This was not expected to make much difference in runtime and memory usage. Since the `FROM` hint is not used for the regular BFS, the planner decides where to start the search instead.

We could see that our TB BFS gets worse runtime than our regular BFS on short queries but on longer queries it gives significant improvement to the runtime, as long as the source and target nodes have hard enough restrictions. As an example table 4.11 shows that by just increasing the length with one, our TB BFS goes from having a runtime 10 times faster to a runtime 90 times faster than the DFS. Table 4.10 shows that our TB BFS is slower than our regular BFS for small maximum depth. It also shows that increasing the graph size from 10 to 100 tendrils does not affect the TB BFS runtime a lot. However, the runtime of our BFS was increased by a factor of 10 and for the DFS the runtime was 5 times higher. The TB BFS optimization was faster for many of the test queries however our regular BFS was as good as the DFS or much worse, both for runtime and memory usage.

Our CTB BFS results shows that, for the queries that it is scheduled for, it is faster than our regular BFS and the DFS. The results between the two optimizations show that none of them are clearly better. For example, in table 4.11 it can be seen that our TB BFS had a runtime three times shorter than our CTB BFS but used fourteen times more memory. However, the results for the cube graph in table 4.12 show that our CTB BFS runs five times faster but uses twice the amount of memory compared to our TB BFS.

Query		TB BFS	Regular BFS	DFS

p = (n0)-[*0..5]-()-[*0..5]-(n1)
p = (n0)-[*0..10]-(n1)
n0 is ”0s” and n1 is ”4t”
With 30 star clusters
	Memory
	(bytes)
		6,674,584
		6,573,592
			459,690,096
			459,690,096
				457,573,304
				457,573,304

	Time (ms)	40 34	545 634	383 421

p = (n0)-[*0..7]-()-[*0..7]-(n1)
p = (n0)-[*0..14]-(n1)
n0 is ”0s” and n1 is ”6t”
With 30 star clusters
	Memory
	(bytes)
		45,689,744
		47,842,136
			18,142,425,568
			18,142,110,960
				21,265,006,768
				21,265,006,768

	Time (ms)	288 239	38,400 39,087	21,804 23,991

p = (:S)-[*0..4]-()-[*0..4]-(:T)
p = (:S)-[*0..8]-(:T)
All S and T nodes
With 40 star clusters
	Memory
	(bytes)
		4,987,167,232
		4,987,435,936
			2,219,158,848
			2,219,158,848
				2,209,635,312
				2,209,635,312

	Time (ms)	4,941 3,971	2,096 2,873	2,178 2,422

p = (:S)-[*0..4]-()-[*0..4]-(:T)
p = (:S)-[*0..8]-(:T)
All S and T nodes
With 400 star clusters
	Memory
	(bytes)
		49,970,956,792
		49,970,653,488
			22,661,302,328
			22,661,302,328
				22,324,035,768
				22,324,035,768

	Time (ms)	40,727 40,211	25,295 24,991	22,609 22,508

Table 4.8: Table over the memory usage in bytes and runtime in ms for the star graph in figure 3.4a. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for our regular BFS and the DFS. TB BFS is our divided Top Down Bottom Up BFS optimization. The specified `S` nodes are the top of the star furthest from the connecting ring and the `T` nodes are one of the nodes connecting the star to the ring, with the numbers specifying which cluster they belong to.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p = (:S)-[*0..5]-()-[*0..4]-(:T) p = (:S)-[*0..9]-(:T) *S = {5}, T = {1,11}* With 16 star clusters	Memory (bytes)	5,056,552 5,094,536	3,242,832 3,242,832	195,164,800 196,064,128	191,996,856 192,823,264
	Time (ms)	48 32	35 24	369 218	156 173
p = (:S)-[*0..5]-()-[*0..5]-(:T) p = (:S)-[*0..10]-(:T) *S = {5,15,25}, T = {1,11,21}* With 26 star clusters	Memory (bytes)	154,605,960 154,197,152	22,892,872 22,643,584	2,794,004,304 2,794,004,304	1,213,247,792 1,212,892,832
	Time (ms)	397 358	104 90	2,075 2,034	997 1,083
p = (:S)-[*0..7]-()-[*0..7]-(:T) p = (:S)-[*0..14]-(:T) *S = {5,21,34,200},* *T = {0,13,29,206}* With 400 star clusters	Memory (bytes)	690,973,992 690,018,480	182,509,120 182,194,480	75,864,000,064 75,864,000,064	75,401,728,632 75,401,149,256
	Time (ms)	1,386 1,366	12,988 12,071	120,958 120,760	54,526 54,181
p = (:S)-[*0..7]-()-[*0..7]-(:T) p = (:S)-[*0..14]-(:T) *S = {5,21,34,200,315},* *T = {0,13,29,206,321}* With 400 star clusters	Memory (bytes)	704,554,096 702,396,168	229,820,888 229,506,456	94,709,360,280 94,709,360,280	93,360,010,288 93,360,010,288
	Time (ms)	732 511	18,678 20,109	161,438 161,020	74,331 69,086

Table 4.9: Table over the memory usage in bytes and runtime in ms for the star graph in figure 3.4a. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for the rest. TB BFS is our divided Top Down Bottom Up BFS optimization and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product. The `S` nodes are the top of the star furthest from the connecting ring and the `T` nodes are one of the nodes connecting the star to the ring, with the numbers specifying which cluster they belong to.

Query		TB BFS	Regular BFS	DFS
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory	1,388,176	3,337,984	3,014,512
p = (n0)-[*0..6]-(n1)	(bytes)	1,388,176	3,389,048	3,014,512
n0 is "2s" and n1 is "7t"	Time	21	23	12
With 10 tendrils	(ms)	37	21	12
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory	2,126,680	87,837,000	85,059,576
p = (n0)-[*0..6]-(n1)	(bytes)	2,125,616	87,342,432	84,329,712
n0 is "25s" and n1 is "75t"	Time	23	269	103
With 100 tendrils	(ms)	22	158	76
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory	10,113,104	656,077,800	651,211,656
p = (n0)-[*0..6]-(n1)	(bytes)	10,114,152	656,077,560	651,211,656
n0 is "25s" and n1 is "75t"	Time	57	1,132	489
With 1,000 tendrils	(ms)	43	512	474
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory	69,999,480	Out of memory	1,109,893,006,928
p = (n0)-[*0..6]-(n1)	(bytes)	69,723,592		1,109,892 692,496
n0 is "25s" and n1 is "75t"	Time	224	–	782,257
With 2,000 tendrils	(ms)	179		832,103

Table 4.10: Table over the memory usage in bytes and runtime in ms for the cobweb graph in figure 3.4b where the crossings are undirected, with 10, 100, 1,000 and 2,000 tendrils. The source and target nodes were specified to be end nodes in the given tendrils. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for our regular BFS and the DFS. TB BFS is our divided Top Down Bottom Up BFS optimization.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	7,037,216	3,211,584	8,672,910,288	8,482,097,440
p=(:S)-[*0..6]-(:T)	(bytes)	7,037,216	3,211,656	8,672,681,536	8,482,096,880
S = {10}	Time	45	12	9,869	4,751
T = {5}	(ms)	33	10	9,437	4,690
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	27,859,384	13,918,864	134,987,937,144	2,173,144,848
p=(:S)-[*0..6]-(:T)	(bytes)	27,734,536	13,814,240	134,987,584,360	2,172,763,784
S = {10,19,32,54}	Time	94	87	159,288	1,495
T = {5,14,24,63}	(ms)	68	54	163,301	1,510
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	41,513,496	22,236,760	3,454,503,832	3,408,665,432
p=(:S)-[*0..6]-(:T)	(bytes)	41,328,640	22,043,896	3,454,442,712	3,408,665,432
S = {10,19,32,54,166,256}	Time	220	118	3,463	2,644
T = {5,14,24,63,875,932}	(ms)	104	88	2,964	2,228
p=(:S)-[*0..4]-()-[*0..4]-(:T)	Memory	9,052,905,384	681,234,904	Out of memory	1,571,813,884,928
p=(:S)-[*0..8]-(:T)	(bytes)	9,051,705,872	681,234,904		1,571,813,884,928
S = {10,19,32,54,166,256}	Time	10,360	28,890	–	934,258
T = {5,14,24,63,875,932}	(ms)	10,500	27,622		941,732

Table 4.11: Table over the memory usage in bytes and runtime in ms for the cobweb graph in figure 3.4b where the crossings are undirected, using 1,000 tendrils. The source nodes were specified to be in the set `S` which are the middle nodes in the specified tendrils and target nodes to be in the set `T` which are the end nodes in the specified tendrils. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one is for the rest. TB BFS is our divided Top Down Bottom Up BFS optimization and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p=(n0)-[*0..15]-()-[*0..15]-(n1) p=(n0)-[*0..30]-(n1) *n0 is "13s" and n1 is "13t"* With 25 Layers	Memory (bytes)	52,299,440 52,295,344	Was not scheduled	912,450,656 912,450,416	1,042,277,920 1,042,277 920
	Time (ms)	119 85	–	1,082 1,111	904 908
p=(:S)-[*0..6]-()-[*0..6]-(:T) p=(:S)-[*0..12]-(:T) *S = {1,7,9}* *T = {4,8,12}* With 13 Layers	Memory (bytes)	8,920,728 8,954,072	2,668,232 2,668,232	63,543,464 68,782,392	11,703,144 11,702,120
	Time (ms)	36 28	21 19	156 106	39 39
p=(:S)-[*0..12]-()-[*0..12]-(:T) p=(:S)-[*0..24]-(:T) *S = {1,7,9,15,18,20,23}* *T = {4,8,12,13,18,21,24}* With 25 Layers	Memory (bytes)	114,637,864 114,433,080	225,345,992 225,345,808	3,377,192,848 3,377,192,848	3,348,277,568 3,348,226,568
	Time (ms)	7,993 8,806	1,653 1,552	3,946 3,892	2,373 2,363

Table 4.12: Table over the memory usage in bytes and runtime in ms for the cube graph in figure 3.4c. The source and target nodes were specified to be in opposite corners either by specific nodes or the types `S` and `T`, where the numbers represents which layer the nodes are in. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one is for the rest. TB BFS is our divided Top Down Bottom Up BFS optimization and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product.

4.3 Discussion

In our results, the same pattern could be seen as in the study by J. Cobleigh, L. Clarke and L. Osterweil [1], the DFS was often faster than the BFS for exploration. This could, in part, be because of the fact that the strength of BFS is shortest path, the first path it finds will always be the shortest. In comparison, the DFS has to check all paths between two nodes to see if they are shorter than the already found shortest path between them [13, p.70-86]. This means that, for the shortest path problem, the DFS can search in the opposite direction of the shortest path and waste runtime. When all paths have to be visited this strength is removed from our BFS, thus slowing it down. This might lead to our BFS operand being slower than the DFS.

Another reason that the DFS might often be faster than our BFS for finding all paths might lie in how they search the graph. The DFS explores close parts of the graph at the time and neighbouring nodes are more likely to be encountered with fewer calculations between them compared to the BFS. The BFS will instead be more likely encounter the neighbouring nodes at a later time, especially at the widest point of the graph. Due to this, it is more likely for the DFS to use the cache for the neighbouring nodes than it is for our BFS. Of course, this assumes that the neighbouring nodes are stored close by in the memory so the whole neighbourhood is fetched to the cache. So as a summation of our findings, DFS is better for finding all paths while BFS is preferable for finding the shortest path.

Our results also had the same pattern as T. Everitt and M. Hutter's study, they found that a greater distance between the source node and the target node gave better performance for the DFS [8]. Their study found that BFS was better than DFS when the source and target nodes were close. The findings in this thesis showed that our BFS performance was more on par with DFS under these circumstances. This can be explained by the points above and that the DFS operand has more optimizations than our BFS operand and that the planner can make better plans for DFS.

4.3.1 Comparison of the Benchmark results

LDBC SF01

From the results of the LDBC benchmarks that were run locally, shown in figure 4.1, it could be concluded that our regular BFS operand was on par with the DFS operand on most queries. This is because for the searches all of the nodes in the range have to be searched and therefore there is not much difference in how the search is done. The existing DFS operand also have more optimizations that were not implemented for our BFS. The planner can also handle DFS much better since there is an actual cost model for DFS instead of forcing the use as for our BFS. These are some reasons that the DFS is a little faster for most benchmark queries.

It could also be seen that manually telling which node to start the search from in the hint drastically improved the runtime for some of the queries, as in figure 4.1c. By using the `FROM` hint for our BFS and telling the planner to start the search in the nodes that are the most specified, we saw that our BFS became faster. This shows that the planner does not always plan an optimal plan, as specifying where to start made the search faster than the DFS in some cases.

When running the same queries remotely the results had the same pattern as the locally run ones. However, our BFS was faster than DFS for more queries and for the long queries where our BFS was faster than DFS locally the remotely run ones were even faster, as seen in figure 4.2. This means that when running our BFS on a faster machine with more memory our BFS can get better performance. Therefore, which operand to use can depend on how powerful of a machine is used.

Using our BFS remotely on Query 12 was still much slower than DFS but faster than the locally run, see figure 4.2a. However, from this run we were able to see the plans for the queries and it turned out that the plans for DFS and BFS was very different. The DFS was planned much better since the planner can handle the DFS much better than our BFS and this lead to the big difference in runtime. The comparisons of these results are therefore not accurate since if the planner could handle our BFS without the hint the plan would be more like the DFS plan. Therefore we will not consider Query 12 for the theoretical model.

For the few queries where our divided Top down Bottom up BFS optimization (TB BFS) could be used it was not faster than the DFS locally. This is because the range of the searches was not big enough to make a difference since not many nodes can be avoided in the search, see table 4.2a. When the maximum depth of the search was increased by one a big difference in time could be seen, see table 4.2b. Now the optimization was much more efficient since many nodes could be avoided in the big graph.

This was evident for the remote runs as well, our TB BFS became even faster while the DFS were about the same, see table 4.3. For Query 1, the unchanged depth was slower than the locally run Query 1 and when the depth was increased the results were very similar to locally. For Query 10 our TB BFS was much faster than the local, where as the performance of the DFS was not affected as much. This is further evidence that the performance of the machine is a factor of the performance of the operand.

To compare our TB BFS with our regular BFS we tried to run our BFS remotely with increased maximum depth, as for the DFS, with scale factor 1. This BFS run did not finish but got an out-of-memory error and it was not tried neither locally nor remotely with scale factor 10. This is a big sign that our BFS uses more memory than the DFS and our TB BFS, as well as runtime.

LDBC SF10

For LDBC scale factor 10, our BFS was faster than DFS on Query 11 in addition to the queries where it was faster for scale factor 1. However, the runtime difference was much smaller between BFS and DFS on scale factor 10 compared to scale factor 1 and this indicates that our BFS works better for smaller graphs. For scale factor 10 using the `FROM` hint was almost always worse than not using it which means that on bigger graphs the way the planner chooses which node to start the search from is more efficient. The planner chooses to start on the side of the relationship where the nodes have the least amount of edges. For smaller graphs this might not be the most efficient place since this will start more searches and will therefore encounter more nodes but for larger graphs this is not the case. Query 12 was still much slower for scale factor 10 and it was because of the same reason as for scale factor 1, the plan for our BFS was not as good as for the DFS.

The results from our normal length TB BFS, table 4.4a, show that Query 1 is even slower compared to the DFS than for scale factor 1. The runtime for our TB BFS for Query 10 is about the same as for scale factor 1 but the DFS is slower than for scale factor 1, making the difference greater in favour of our TB BFS. When increasing the maximum depth, see table 4.4b, Query 1 is a lot faster for our TB BFS than for DFS, just as for scale factor 1. For Query 10 the same can be observed, it continues to be much faster for our TB BFS than the DFS.

Pokec

The results of the queries from the Pokec benchmark showed that our BFS operand works better for smaller queries since it was faster than the DFS operand for Query 1, which is the smallest from this benchmark, but slower for the larger queries. We could also see that our BFS operand requires more memory. This could be seen for Query 32 where we ran out of memory while the DFS operand could run properly. For Query 31, we could see the difference between the optimized DFS and the unoptimized DFS operands. Here, the optimized DFS operand worked much better than our BFS operand however our BFS operand was faster than the unoptimized DFS operand. When it comes to the write query, we could see that our BFS operand handles write queries better than the DFS. We found it to be faster for the write version while slower for the read version of the same query. This can be explained by how the operators handles the transaction order. Our BFS writes after all the searches have finished, whilst DFS writes during searches and therefore locks the database when it writes.

Social Network

The results from the Social Network benchmark, shown in figure 4.5, shows how much difference it makes when our BFS operand is only applied to one relationship in the query. This is because each relationship in the query gets its own search and every relationship has its own optimal operand. Therefore, it is not optimal to force the same operand on the whole path in the query. We could also see that our BFS operand works better for smaller queries on the Social Network graph. This shows for Query 8 and Query 9, since Query 9 was a smaller version of Query 8. Using our BFS operand on Query 9 was faster than the DFS operand and for Query 8 DFS was faster. Another observation is that more restrictions makes our BFS operand faster than the DFS operand. This can be seen when comparing the results for Query 7 and Query 8. These two queries are the same except that Query 7 has more restrictions on one of the end nodes.

Logistics

We believed that the Logistics benchmark would be better for the DFS operand than for our BFS operand but the result for running the query was, surprisingly, the exact opposite. Instead, it is the best improvement for our BFS operand compared to the pre-existing DFS operand. The reason we believed this benchmark to be suboptimal for our BFS was because of its small size. We expected to see improvements over the DFS operand for bigger graphs. However, as can be seen for most of the benchmarks are the opposite true, our BFS operand is better for smaller graphs.

Access Control

From the Access Control benchmark results in figure 4.7 we can see the differences between running our regular BFS over the whole path or just the relationships of interest from Access Control. It differs greatly whether using our BFS on the whole path or on one relationship is the fastest, for some queries it is the relationship and for others it is the path. If it is faster to use our BFS operand on the whole path than on the relationship it means that another relationship in the query is improved by BFS. We could also see that the usage of the `FROM` hint affects which is faster. For some queries it is better to use the `FROM` hint and for others it is not. This means that the best node to start the search from differs and that depends on how the graph looks.

Music Brainz

We had an unsuccessful attempt at our CTB BFS, it ran out of memory, for the Music Brainz benchmark. Since our intended CTB BFS search did not finish, there was no way to verify if the search used CTB BFS. One reason that it ran out of memory could be due to the fact that our CTB BFS starts a search from each node that was of the type person in the graph. Since persons are a big part of the graph, this leads to many searches being started. From this we can conclude that if the nodes in the query are not restricted our CTB BFS should not be used. Another possible explanation for running out of memory is if the hint forced a bad plan to be made, but it is less likely.

Generated Music

From Generated Music we could see further proof that our CTB BFS should not be used on queries with too few restrictions on the nodes. This search did not run out of memory since Generated Music is significantly smaller than Music Brainz. We could also see that having restrictions on the relationships in the query improves the performance of our BFS operand. This is because restricting the relationships will have the same effect as restricting the nodes. Both will reduce the number of visited nodes in the search.

LevelStory

The LevelStory benchmark was the smallest graph and just as most of the smaller graphs, using our BFS operand lead to major improvements. We also got better results on the smaller queries, shorter length on the relationships, from this benchmark. When our BFS operand was used on a relationship in the query and not the whole path we got significant improvements which further proves that using our BFS directly on relationships can be very powerful. From the biggest query in the benchmark we could see that using the `FROM` hint on this query made a big difference. When the hint was used together with our BFS on the whole path we were slower then without the hint. However, when we ran our BFS on one relationship did the hint speed up the search. This shows that it is not only the graph that affects the `FROM` hint, it is also the search itself.

Correlations between the Benchmarks

By evaluating all results from the benchmarks, we could draw some correlations between the benchmarks. In general, we could see that our BFS worked better for smaller datasets and graphs. We got better result for the LDBC scale factor 1 than for scale factor 10. The smaller benchmarks Pokec, Logistics and LevelStory also gave better results compared to the larger benchmarks. From the Pokec, Social Network and LevelStory benchmark, we could also see that a shorter relationship in the query is preferred. These benchmarks contained queries with both longer and shorter relationships and we were faster for the shorter relationship queries in all cases.

When it comes to the usage of the `FROM` hint, we could see for the LDBC benchmark using the hint was faster for scale factor 1 but slower for scale factor 10. This means that for a smaller graph it is better to start the search in the node which is most specified. This is what the hint forces the planner to do, which leads to fewer searches being started. However, when filtering the results later it is harder to find the wanted nodes since they are less specified. For larger graphs this means that there are many nodes that has to be filtered. By not using the `FROM` hint the search is started in the nodes with the least amount of edges. Even though this starts more searches, the searches find fewer nodes and in the filtering it is easier to find the wanted nodes since they are more specified.

For some queries from three of the benchmarks we used our hint to specify that BFS should be used directly on one relationship, instead of on the whole path. This gave mixed results, sometimes it gave better runtime and other times worse. For LevelStory we only tried it for one query and there it was significantly better. The Access Control benchmark gave mixed results among its queries and for the Social Network benchmark, it was better a majority of the time.

Most benchmarks were quite connected, averaging around 5 relationships per node. The benchmark Logistics gave the best results and it had a average of 6 relationships per nod. This lead us to believe that it is the optimal range for the connectivity. Though, since the benchmarks have similar average connectivity we cannot really draw any conclusions about it. However, we can see that the graph structure is more important than the connectivity.

Even though we did not measure memory for the benchmarks, we could see that our BFS operand used more memory than the DFS. For example, when we tried to use our BFS for the increased maximum in Query 1 and 10 from the LDBC benchmark it ran out of memory, even for scale factor 1. The DFS however, could finish the searches without any problems, which indicates that our BFS requires more memory. For Query 32 of the Pokec benchmark the same results could be seen, the DFS finished whilst our BFS ran out of memory.

Since our BFS requires more memory than the DFS, the choice of machine matters. This could be seen for the two scale factors of LDBC. LDBC scale factor 1 could be run locally whereas for scale factor 10 our machines ran out of memory. Running scale factor 10 remotely on much better machines with more memory worked without problems. Since scale factor 10 is ten times larger than scale factor 1 we can conclude that for larger graphs, machines with more memory are required. However, we could run the Music Brainz benchmark locally even though it has a bigger dataset than LDBC scale factor 10. This was because the graph that the Music Brainz dataset builds contains a smaller amount of nodes and relationships than the LDBC scale factor 10 graph. This means that the graph size is more significant than just the memory size of the datasets.

Our two Top down Bottom up optimizations could only be used for a handful of queries. We could see that for the two LDBC benchmarks, the TB BFS optimization was much faster than the regular operands. However, for Music Brainz and Generated Music using the CTB BFS optimization was much slower than both regular operands. They also used more memory than the regular operands, which could be seen for Music Brainz since it ran out of memory. That the optimizations was better for LDBC is due to the fact that LDBC has more restrictions than Music Brainz and Generated Music. When there are not enough restrictions, too many searches will be started and this means that too many nodes that not relevant for the search are visited. Therefore, the optimizations should only be used for queries with harder restrictions.

4.3.2 Comparison of the Test results

When writing the test cases the runtime and memory usage of our BFS, both our Top down Bottom up BFS optimizations and the DFS were monitored for the same queries and graphs. The graphs used for these comparisons were highly connected and dense. In addition, they often contained at least one bottleneck. Two of them were designed to give advantage to our different BFS, but one was designed to give a disadvantage. The graphs are explained and motivated in section 3.2.5. Most of our focus in this discussion will be on the optimizations since we barely have any results about them from the benchmarks.

Comparing the test case results of our regular BFS operand and the DFS operand showed that, most often, our BFS was either slightly slower and used a little more memory than DFS or it used much more memory and was much slower than DFS. From this, conclusions can be drawn that our BFS uses more memory than the DFS. This is because during the search our BFS queue becomes bigger for each new depth, the queue contains the whole of that depth, and for the maximum depth this is a great deal of nodes. The DFS has a more constant number of nodes in its list so it uses less memory.

In the results from the test cases it can be seen that our BFS operand becomes slower compared to the DFS operand when the search becomes longer, seen in table 4.8. When the graph becomes bigger, our BFS takes longer time compared to the DFS as shown in table 4.10. Our BFS also becomes slower than the DFS when the query can be applied to more nodes which is what happens in 4.9 when the sets of nodes becomes larger.

Since our BFS operand used as much as or more memory, when it was on par with the DFS operand in runtime, comparisons can be drawn with the benchmarks results. We could see from both the increased length LDBC and Pokec benchmarks that we used more memory even though it was not measured, because we ran out of memory while the DFS could run the queries. To get an estimate on the memory usage however, we needed the test cases. The cost efficiency of our BFS operand was therefore estimated from both the benchmarks and the test cases. From the test cases we could see that our BFS uses more memory than the DFS, even when the BFS is faster. From this we can conclude that when our BFS is slightly faster than the DFS in the benchmarks it uses more memory and is thus less cost efficient. For example for Query 9 in LDBC scale factor 10, see figure 4.3b, the improvement in runtime is very insignificant but none the less it is an improvement. However if this search used more memory, and based on the memory testing from the test cases it did, it is not cost efficient. This is because no user would ever notice the time difference but with an older machine, where memory might be a problem, the increased memory usage could be noticed. The cost efficiency depends on multiple variables. For us runtime is a little more important, however if the difference in runtime is small memory is more prioritized. Since the runtime and memory usage of our BFS depends on the graph, the cost efficiency of the operand also depends on the graph.

Optimizations

When it comes to our two Top down Bottom up BFS optimizations, the test cases expose their advantages. The optimizations were faster, could handle much longer search paths without loosing speed and used less memory than the DFS and especially our regular BFS. Though, a disadvantage of the optimizations was that for some searches, multiple occurrences of the same path could exist in the result. This is because they are joined together on different nodes along the path and therefore not identical at the inner filtering step. Since the paths differ in the search they cannot be filtered away without traversing each list twice and comparing each path. This would slow down the runtime immensely for queries with many results.

From the results of the test cases, it could be seen that the more nodes and relationships the graph contains, the bigger the difference is between our regular BFS and our divided Top down Bottom up BFS optimization (TB BFS). This also goes hand in hand with the benchmarks, which showed that our regular BFS was better for smaller graphs. This is because when there are many high degree nodes, our regular BFS finds so many more unnecessary nodes that are not in the desired paths. This also means that more nodes are stored in the search, which leads to wasted memory. This was seen in the test cases when our TB BFS executed the query in less than one second but the regular ran out of memory after a couple of minutes. Our TB BFS version also scaled better when more nodes were added or the length of the desired paths were increased, because the increase in searched nodes are greater for our regular BFS than for our TB BFS. This could also be seen for the LDBC benchmark when we tried to increase the length of the query and our regular BFS ran out of memory while our TB BFS got fast results. However, our test cases were designed to give an advantage to our TB BFS so it did not come as a surprise that it excelled over our BFS.

A downside with our TB BFS is that if the source and/or target node does not have enough restrictions on them the operand will not be as effective. This is because there are too many nodes that fit the loose restrictions and therefore are eligible to start a search from. The point of our TB BFS is to reduce the number of visited nodes, but when too many searches are started the number of visited nodes will increase, as seen in table 4.8. This will lead to our TB BFS being much slower and use more memory than both the DFS and our regular BFS. This could also be seen in the benchmarks, if we did not have enough restrictions on the search then we got slower results or ran out of memory. The LDBC, Generated Music and Music Brainz benchmarks showed this clearly since for the more restricted LDBC queries our TB BFS optimization worked great, while our CTB BFS optimization were much slower or ran out of memory for the Generated Music and Music Brainz benchmarks. However, for many searches it is not applicable to put heavy restrictions on both the source and target node, so our TB BFS is not useful for all searches.

For a small maximum depth, our TB BFS is slower than our regular BFS, as seen for the cobweb graph in table 4.10. This is in part because the join is too costly and also that starting multiple searches can cause more nodes to be visited. More visited nodes can cause a big overlap between the searches, which also slows down our TB BFS. Therefore, it is more effective to use our regular BFS on smaller queries. In the same table it can be seen that the runtime for our TB BFS using 10 tendrils and 100 tendrils is almost the same. However, for our BFS and the DFS the difference is much higher. This shows that the join operand takes the majority of the TB BFS runtime for smaller graphs and it is a bottleneck for speed.

The length of the relationship in the query also affects the efficiency of the operands, as seen in table 4.11. These results show that by just increasing the length with one, the difference in runtime between our TB BFS and the DFS is increased a lot in favour of the TB BFS. When the length is too small the DFS is much faster and uses less memory. This is because the greater the length the more nodes can be avoided in the searches.

The optimization done after a Cartesian product operand has both advantages and disadvantages compared to our TB BFS. Our Cartesian Top down Bottom up BFS optimization (CTB BFS) does not require manually inserting a node in the middle of the query since the input to our BFS is changed. This means that our CTB BFS is more user friendly. However, there is no guarantee that it will be planned, since forcing the Cartesian product to be used is not practical. This is because forcing a Cartesian product will lead to plans that are not optimal and some of these plans can ruin our BFS operand. Another downside to our CTB BFS is that the plan will not join or filter the results after the searches. Therefore, this had to be done in our BFS and cannot be done with the same efficiency, since the pre-existing join and filtering operands are more optimized.

An implemented optimization of our CTB BFS operand was storing the result of the two smaller searches with their input node and maximum depth. This lead to not re-running the searches if a node with the same maximum depth was re-encountered, but instead fetching it from the stored values which conserves both time and memory. However, if there was a new depth then the search would be re-run. This optimization worked since the input nodes can exist in multiple input pairs, due to the Cartesian product.

In table 4.11 it can be seen that sometimes our TB BFS is faster but uses more memory than our CTB BFS. Table 4.12 however shows that the opposite is true, the CTB BFS is faster but uses more memory. It can also be seen that our TB BFS gave worse results compared to the other three operands and our CTB BFS gave the best results for the queries it was scheduled for. The reason for the TB BFS results is that when one of the source nodes and one of the target nodes are in the outer most layers on opposite sides our TB BFS cannot avoid any unnecessary nodes in the search. And since there were many nodes that fit the specifications the join had many nodes to filter through, thus slowing down the runtime. Our CTB BFS however is faster since it saves the already calculated searches and can therefore execute faster than our regular search. When the search on a cube graph is between two nodes that are on opposite sides in the middle layers our TB BFS is much faster than our regular BFS and the DFS since now nodes can be avoided in the search. From this it can be concluded that the choice of optimization entirely depends on how the graph is constructed and more importantly how the searched part of the graph is constructed.

When comparing our TB BFS and CTB BFS on smaller graphs it can be seen that our CTB BFS runs much faster because it lacks the costly join, see tables 4.9, 4.11 and 4.12. However, our CTB BFS was faster than both our regular BFS and the DFS on smaller graphs because it still benefited from performing the split search. This is because the joining and filtering is done internally in the search and this can be seen in table 4.12.

From all this, it can be seen that the performance of both optimizations is affected by the restrictions of the source and target nodes and the length of the relationship in the query. The performance also depends on the size of the graph and the amount of relationships per node in the graph. Based on this, implementing a theoretical model of our BFS is no easy task.

4.3.3 Abandoned optimizations

Splitting relationship

The ability to have the planner split relationships automatically for our TB BFS could have improved the usefulness and the performance of the operand. The operand would have become easier to use since it would no longer be mandatory to change the query to use our TB BFS. The planner could also use our TB BFS more frequently and make better plans for it. The performance could have been improved since if the planner could choose where to split the relationship it could split it at an optimal way. However as seen with the benchmarks the planner is not always optimal, since adding the `FROM` hint improved the performance. If this happened with the splitting, it could split the relationship in a way such that it made a longer relationship for the side of the query that would have more eligible nodes. This could increase the runtime and make our TB BFS less useful.

The reason this functionality was abandoned was that it would require too many special cases to even make it useful for a handful of queries. Even if these special cases were implemented the validation plan would also have to be altered since they are not made in the same way. All of this was deemed to be outside of the scope of the thesis.

Building several paths at once

Our current operand starts a new search from a node every time it is re-encountered. We thought that we might be able to reduce the runtime if we handled and searched from each node fewer times. At first we considered keeping all the paths leading from a node in the memory and when re-encountering the node just use the saved paths instead of redoing the search. However, that is a DFS optimization. It works for DFS since the found paths can be returned back to the node when the algorithm backtracks. For BFS, this is not an option since BFS does not return to the current node after its neighbours have been handled.

What we instead could do for BFS was to keep all found paths leading into a node in the memory. Then when a node was visited three things would happen. First, all paths in the memory leading to the node would be updated to include the node. Secondly, all neighbouring nodes already in the queue would get their memory updated to include these paths. Thirdly, the neighbouring nodes that did not exist in the queue already would be added with all the updated paths. For this optimization multiple lists had to be introduced. When new relationships were found, all paths in these lists had to be updated. These lists had to be traversed every time a node was visited and since we could re-encounter nodes, this was done more times than there were nodes in the search. Doing this used a large amount of memory and a lot of time was spent searching the memory to check for the nodes, leading to it being much slower than just redoing the searches as for our regular BFS. For this reason, the functionality was abandoned since there was no apparent way to improve on it. The focus was then shifted onto the Top down Bottom up optimizations, as these were deemed more important and could give better results.

Another downside to this functionality was that storing the paths lead to the results not returning in a traditional BFS order. The traditional order would be shortest paths first but since we updates the memory for the nodes in the queue with new paths, these paths jump the line and will be processed earlier than expected. This will produce a order where longer and shorter paths are mixed together which is not what we expects from BFS. It did not affect the performance or the amount of results and if wanted the result could be reordered by a clause in the query.

We reused part of this idea for our Cartesian Top down Bottom up optimization (CTB BFS), where we took advantage of storing calculations and using them later. However, instead of storing calculations in the middle of the search as for the multiple path optimization above, we stored the finished searches. This follows the DFS version of the optimization instead, which is more effective. Storing the finished searches means that they are not modified during the continued search, which leads to them not needing to be traversed multiple times per search. Therefore, the performance is not affected negatively as it is for the multiple path BFS.

4.3.4 Further development

The first step to develop our work further would be to modify our heuristic model into a real theoretical model with mathematical formulas. From this a cost model can be formulated and this could then be implemented into Neo4j. By doing this the planner can use our BFS in plans without using the hint. This would be a direct continuation of our work. Developing a real theoretical model will most likely require more testing since our results did not allow us to formulate mathematical formulas.

One way to improve our BFS operand is to introduce threads, since the operand starts a BFS on each eligible node. Therefore, running them concurrently might improve the performance immensely. This would work since the results are filtered after every search is completed and as long as the query is a search, the data should not be modified and corrupting data should not be an issue. Locking and unlocking would still have to be used for the retrieving of the nodes. However, no data is stored in the nodes as it is saved in the search, so not much time would be spent inside locks. This development would also improve the runtime of our divided Top down Bottom up BFS optimization, since running those BFS concurrently would give the same results. This development would however not improve the memory usage since running one BFS per thread does not affect the number of nodes that are visited per search.

Another improvement would be to implement support for shortest path instead of all paths, since this is where BFS excels. This would give a faster runtime since a lot of searching could be avoided by keeping track of the already visited nodes and not revisit them as the current algorithm does. Though this might increase the memory usage somewhat, since all visited nodes need to be kept in memory and then be checked against in every iteration of the search.

The pre-existing optimizations that are implemented for the DFS operand could be adapted for our BFS operand. This could make it so that it would be more on par with the DFS on the queries where DFS can use its optimizations and neither our TB BFS or CTB BFS can be applied.

There were discussions of the possibility of combining a BFS with a DFS. The idea is to start with a BFS and executing this in the beginning. Then after a certain depth switching over to DFS and finishing the search. The order of this is based on our findings, which showed that BFS was faster than DFS for smaller searches while DFS was faster for larger searches. Combining the searches in one operand could lead to interesting results. Unfortunately, we did not try to implement this because of time restraints and we did not prioritize this since it was not part of our scope. For this operand one of the most time consuming parts will be evaluating when to switch between BFS and DFS since this will be the deciding factor for the usability.

4.4 Theoretical Model

From the results of the benchmarks and test cases a theoretical model could be constructed. Our theoretical model is technically a heuristic model, since we do not define any mathematical rules or formulas, since our results or evaluations could not be applied for that. The efficiency of our BFS operand depends on many circumstances: the size of the graph, how the graph is constructed, the length of the relationships in the query, how many restrictions are put on the nodes and relationship in the query and the performance of the machine. The model will be presented here and have a summation at the end.

Our BFS operand performed best on graphs that were small or medium in size, this means graphs containing under 5 million nodes and under 50 million edges. When the graphs were larger, our BFS performed worse compared to the DFS. The amount of relationships per node also made a difference for the performance and from our experiments our BFS performed best when the graph contained an average of around 5 relationships per node. Our BFS is recommended on queries with relationships of a length between 2 and 5. For longer relationships and path more restrictions are needed for our BFS to work at its best.

When it comes to the node to start the search in, it differs which is the best. For smaller graphs it is most often optimal to start in the node with the most restrictions in the search, limiting the number of started searches. But when the graph gets bigger this changes and instead the optimal start node is the node with the least number of relationships. Our BFS is not the best for all relationships in a query and therefore it cannot be applied to the whole path. Thus, all parts of the query have to be evaluated if our BFS should be applied.

The result of the test cases showed that our BFS used more memory than DFS and because of this the performance of the operand depends on the machine that is used. For the best performance of our BFS operand, more memory has to be available than for DFS. Even though our BFS might be faster on a query if the machine does not have enough memory the search will not be executed. We could see this when LDBC scale factor 10 could not be run locally, because of the extensive memory usage, while there where improvements in runtime when it was run remotely on better machines.

The optimal graph and query can be seen in the results from the Logistics benchmark where our BFS was 90% faster than DFS. The Logistics graph contains $800,000$ nodes and 4.8 million edges, which is the third smallest. It also has an average of 6 relationships per node, which is a fair amount. The query was smaller and the three relationships in this query, where BFS was used, had a maximum length of 2 which is on the smaller side and for all relationships at least one node was specified. Since no other benchmark had a runtime improvement that was even close to this one it can be concluded that this is the optimal graph to use our BFS on.

Our optimizations for BFS should only be used for queries where the nodes have strong restrictions, this is because otherwise to many searches will be started. The strong restrictions on nodes limit the number of searches substantially while stronger restrictions on the relationships limit the scope of the searches. The optimizations works best on larger highly connected graphs. This is where the largest amount of nodes can be avoided in the search and therefore memory and runtime saved. This is an area where neither the DFS nor our non-optimized BFS is all that great, which means that the optimal operands for these graphs are our Top down Bottom up optimizations. Our divided Top down Bottom up BFS optimization (TB BFS) is easier to schedule since it can be forced whereas our Cartesian Top down Bottom up BFS optimization (CTB BFS) is harder to schedule, but more versatile. This is mostly because of the restriction to the minimum length of the relationship where our CTB BFS only needs a minimum length of 2 to be useful while our TB BFS needs at least the minimum length 5, owing to the costly join.

Here follows guidelines for the theoretical model.

- Our BFS should be used when the graph is of a small or medium size and preferably quite connected.
- The query should preferably have relationships in the range 1-9, where there are more restrictions on the nodes and/or the relationships.
- The optimal start node for the search is the one with the most restrictions for small graphs but the node with the least number of relationships for large graphs.
- The optimal execution is a restricted query with relationship length between 1 and 4 on a small graph with on average 5 relationships per node.
- Our BFS should not be applied on the whole paths, since not all relationships are optimal for our BFS, but rather on the individual relationships.
- Our BFS should be used on machines where available memory is not an issue, because our BFS has a higher chance to run out of memory than DFS.
- Our BFS optimizations should only be used when the sides of the relationships have strong restrictions. Even better if the relationships are restricted as well.
- Our BFS optimizations gives the best efficiency when the graph is large and well connected, around an average of 7 relationships per node. This is because then more nodes can be avoided in the search.
- Our TB BFS should not be used for queries with relationships with length under 5 since it is not faster in these cases.

To be able to implement our model into the Neo4j planner, the model would need to be improved. This could be done by running more benchmarks and evaluating those results, since for every graph and query the operand will behave differently and more information will give a more accurate model. Adding improvements to the operand could also give a different theoretical model.

Chapter 5
Conclusions

We have implemented and evaluated several optimized and non-optimized versions of Breadth-First Search. The evaluation was done through several benchmarks and self-constructed test cases. From the benchmarks and test cases we can draw the conclusions that our non-optimized BFS is better for smaller graphs and shorter queries. In these cases our BFS is faster than the DFS while not using too much memory. Another aspect to the graphs benefiting our BFS is higher connectivity, since the search is more likely to find the target nodes faster in those cases. Our BFS operand also works better when restrictions are put upon the wanted paths to limit the search scope. This is especially relevant for the implemented optimizations, which need the restrictions to work properly and not start excessive searches.

With restrictions in place, our different Top down Bottom up BFS optimizations can handle much longer paths and larger graphs than our non-optimized BFS, as well as the DFS. In fact, for our divided Top down Bottom up BFS optimization (TB BFS) the path needs to be longer, since for short paths the overlap is too great between the searches and the join too costly for it to be effective. While the overlap is a problem for our Cartesian Top down Bottom up BFS optimization (CTB BFS), the join is not and it can therefore be used for smaller queries than our TB BFS, though it still works better for longer paths.

When it comes to cost efficiency it can be seen that our BFS sometimes is less cost efficient than the DFS even when it is faster, because of the memory consumption. This is because the runtime improvement over DFS is often on the smaller side and the increased memory consumption might overwhelms the improved runtime. In contrast, both our optimized BFS are often more cost effective, in both runtime and memory usage, than the DFS and our non-optimized BFS when they are optimally used. When not optimally used, like too short paths and too few restrictions, the optimizations are not very cost effective compared to the DFS nor our non-optimized BFS.

For our BFS to be useful in Cypher and Neo4j it needs continued work. Currently it still uses a hint to force its use and we have no proper way of knowing in which scenario its better than the DFS. It could currently be useful if you know that your graph structure and size is better for BFS but probably not otherwise. If the usage of the hint could be removed and a proper cost model developed, then it will be more useful. To incorporate a BFS and DFS combined search would also be useful, but was not covered by this thesis.

Bibliography

[1] J.M. Cobleigh, L.A. Clarke, and L.J. Ostenveil. The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite State Verification. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 37–46, May 2001.

[2] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for Association Rule Mining – a General Survey and Comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, June 2000.

[3] D. De Leo and P. Boncz. Extending SQL for Computing Shortest Paths. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 10:1–10:8. ACM, 2017.

[4] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. Xu Yu. K-Reach: Who is in Your Small World. 5, July 2012.

[5] R. Bar-Yehuda and S. Even. A Linear-Time Approximation Algorithm for the Weighted Vertex Cover Problem. *Journal of Algorithms*, 2:198 – 203, 1981.

[6] M. Then, M. Kaufmann, F. Chirigati, T.A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H.T. Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.*, 8(4):449–460, 2014.

[7] H. Liu, H.H. Huang, and Y. Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *SIGMOD Conference*, pages 403–416, June 2016.

[8] T. Everitt and M. Hutter. A Topological Approach to Meta-heuristics: Analytical Results on the BFS vs. DFS Algorithm Selection Problem. *CoRR*, abs/1509.02709, 2015.

[9] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

[10] J.J. Miller. Graph Database Applications and Concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference*, pages 141–147, Atlanta, GA, USA, March 2013.

[11] Neo4j. `https://github.com/neo4j`. Last accessed 14 September 2018.

[12] A. Green. Querying Graphs: A More Natural Data Model, October 2017. From 17th International Workshop on High Performance Transaction Systems (HPTS).

[13] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson, 2014.

[14] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. *Scientific Programming*, 21(4):137 – 148, 2013.

[15] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD '15: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630, 2015.

[16] LDBC, The Graph & Benchmark Reference. `https://github.com/ldbc/ldbc_snb_docs`. Last retrieved 13 March 2018.

[17] Pokec social network. `https://snap.stanford.edu/data/soc-pokec.html`. Last accessed 14 March 2018.

[18] L. Takac and M. Zabovsky. Data Analysis in Public Social Networks. In *International Scientific Conference & International Workshop Present Day Trends of Innovations*, May 2012.

[19] Use Cases for the Logistics, Social Network and Access Control Benchmarks. `https://github.com/iansrobinson/graph-databases-use-cases`. Last accessed 23 March 2018.

[20] MusicBrainz. `https://musicbrainz.org/`. Last accessed 23 March 2018.

[21] Documentation for the MusicBrainz Database. `https://musicbrainz.org/doc/MusicBrainz_Database`. Last accessed 23 March 2018.

[22] O. Agesen. The Cartesian Product Algorithm. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Aarhus, Denmark, August 7–11, 1995*, pages 12–14. Springer Berlin Heidelberg, 1995.

Appendices

Appendix A
Additional Evaluation Results

This appendix contains diagrams and tables with the results from some of the benchmarks and all of the test cases. For the benchmarks and test cases presented here some results are presented in section 4.2 in addition to here, but the appendix has additional results. For the benchmarks not presented here all of the results are presented in section 4.2.

A.1 Access Control

Here is the full result of the Access Control benchmark, from appendix B.5. Query 9 to 13 are presented in detail in section 4.2.

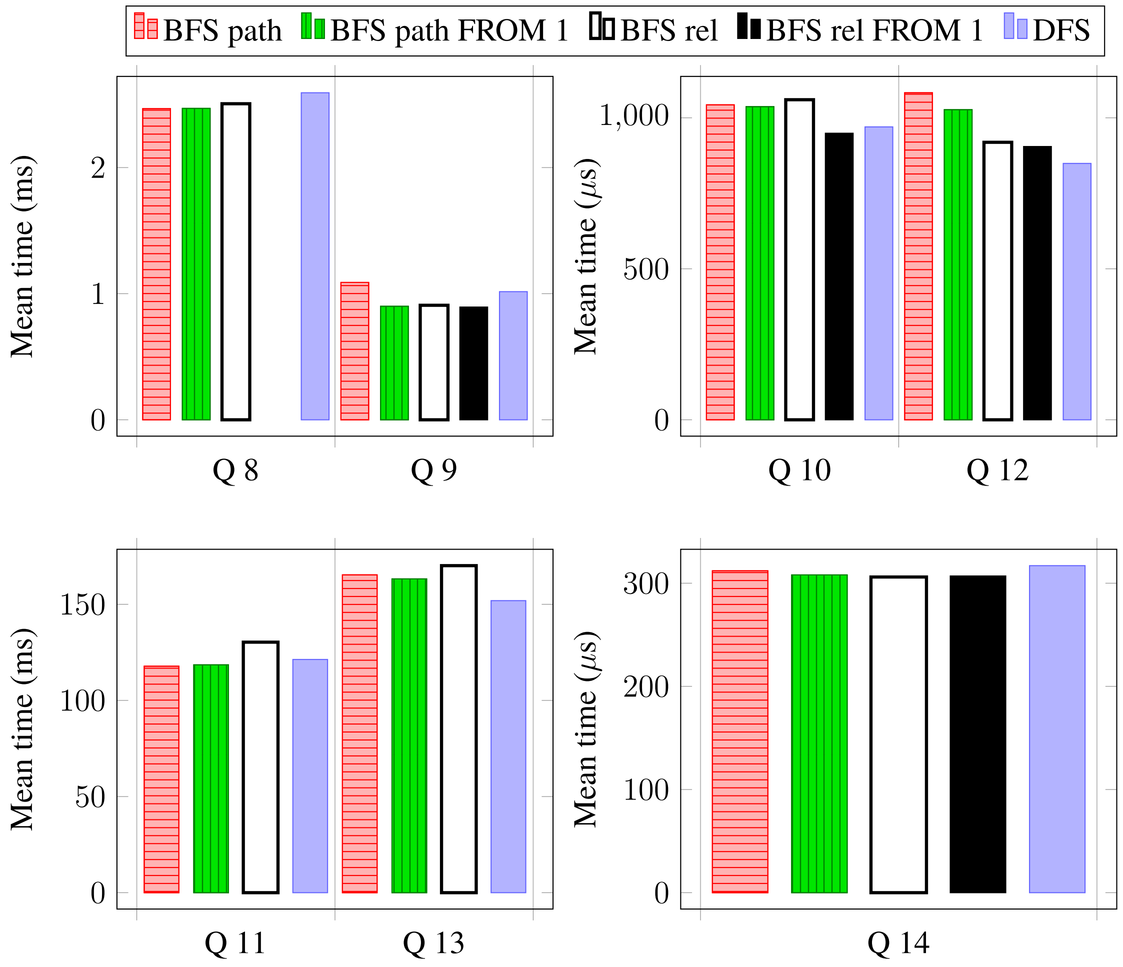


Figure A.1: The mean times, in μs and ms, of the runs on the Access Control benchmark. BFS path FROM 1 is our regular BFS on the whole path where the nodes to start the search from are manually specified. BFS rel is our regular BFS specified to only one relationship in the path, with the rest of the relationships running DFS.

A.2 Music Brainz

Here is the full results of the Music Brainz benchmark, from appendix B.6. Our Cartesian Top down Bottom up BFS optimization (CTB BFS) was tried for Query 27 but could not run because of memory limits, because of this could it not be confirmed that it actually used the CTB BFS. More details about the results are found in section 4.2.

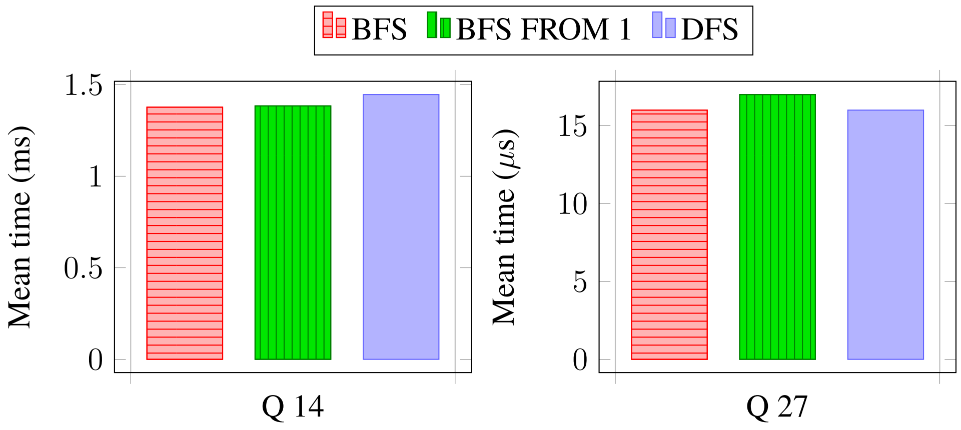


Figure A.2: The mean times, in μs and ms, of the runs on the Music Brainz benchmark. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

A.3 Generated Music

The full results of the Generated Music benchmark, from appendix B.7, is presented here. In section 4.2 are the results presented in more detail. For Query 25, our Cartesian Top down Bottom up BFS optimization was tried and it took 16,293 μs while the other operands took less than 300 μs.

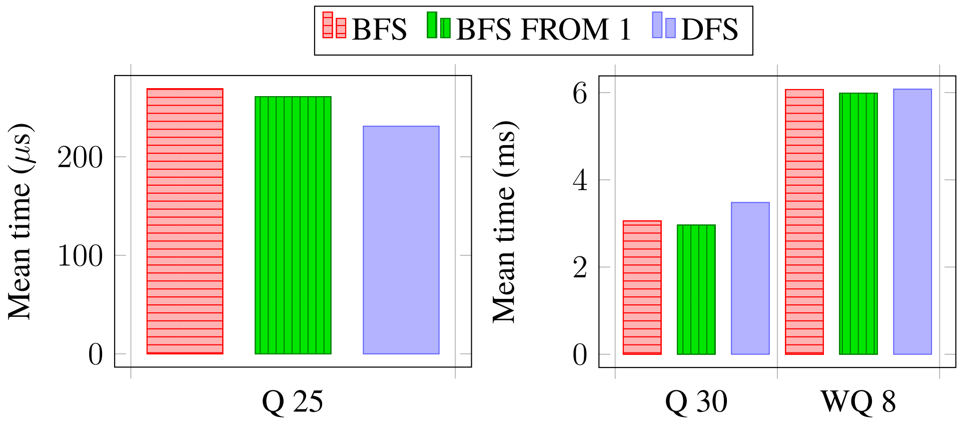


Figure A.3: The mean times, in μs and ms, of the runs on the Generated Music benchmark. BFS FROM 1 is our regular BFS where the nodes to start the search from are manually specified.

A.4 Test Cases

Here follows the full results of our test cases, for the memory testing, built on the graphs presented in section 3.2.5. Partial result tables can be found in section 4.2, where they are properly presented.

Query		TB BFS	Regular BFS	DFS
p = (n0)-[*0..5]-()-[*0..5]-(n1)	Memory (bytes)	6,674,584	459,690,096	457,573,304
p = (n0)-[*0..10]-(n1)		6,573,592	459,690,096	457,573,304
n0 is ”0s” and n1 is ”4t”	Time (ms)	40	545	383
With 30 star clusters		34	634	421
p = (n0)-[*0..7]-()-[*0..7]-(n1)	Memory (bytes)	45,689,744	18,142,425,568	21,265,006,768
p = (n0)-[*0..14]-(n1)		47,842,136	18,142,110,960	21,265,006,768
n0 is ”0s” and n1 is ”6t”	Time (ms)	288	38,400	21,804
With 30 star clusters		239	39,087	23,991
p = (:S)-[*0..4]-()-[*0..4]-(:T)	Memory (bytes)	4,987,167,232	2,219,158,848	2,209,635,312
p = (:S)-[*0..8]-(:T)		4,987,435,936	2,219,158,848	2,209,635,312
All S and T nodes	Time (ms)	4,941	2,096	2,178
With 40 star clusters		3,971	2,873	2,422
p = (:S)-[*0..4]-()-[*0..4]-(:T)	Memory (bytes)	49,970,956,792	22,661,302,328	22,324,035,768
p = (:S)-[*0..8]-(:T)		49,970,653,488	22,661,302,328	22,324,035,768
All S and T nodes	Time (ms)	40,727	25,295	22,609
With 400 star clusters		40,211	24,991	22,508

Table A.1: Table over the memory usage in bytes and runtime in ms for the star graph in figure 3.4a. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for our regular BFS and the DFS. TB BFS is our divided Top Down Bottom Up BFS optimization. The specified `S` nodes are the top of the star furthest from the connecting ring and the `T` nodes are one of the nodes connecting the star to the ring, with the numbers specifying which cluster they belong to.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p = (:S)-[*0..5]-()-[*0..4]-(:T) p = (:S)-[*0..9]-(:T) *S = {5}, T = {1,11}* With 16 star clusters	Memory (bytes)	5,056,552 5,094,536	3,242,832 3,242,832	195,164,800 196,064,128	191,996,856 192,823,264
	Time (ms)	48 32	35 24	369 218	156 173
p = (:S)-[*0..5]-()-[*0..4]-(:T) p = (:S)-[*0..9]-(:T) *S = {5,15}, T = {1,11}* With 16 star clusters	Memory (bytes)	35,214,064 35,163,400	7,232,176 7,232,176	383,372,416 382,585,360	376,791,008 375,833,904
	Time (ms)	99 137	49 67	353 383	253 233
p = (:S)-[*0..5]-()-[*0..4]-(:T) p = (:S)-[*0..9]-(:T) *S = {5,15,25}, T = {1,11,21}* With 26 star clusters	Memory (bytes)	38,601,992 37,845,784	10,761,512 10,631,568	990,255,568 990,203,632	422,369,728 422,077,928
	Time (ms)	87 116	74 39	954 899	371 336
p = (:S)-[*0..5]-()-[*0..5]-(:T) p = (:S)-[*0..10]-(:T) *S = {5,15,25}, T = {1,11,21}* With 26 star clusters	Memory (bytes)	154,605,960 154,197,152	22,892,872 22,643,584	2,794,004,304 2,794,004,304	1,213,247,792 1,212,892,832
	Time (ms)	397 358	104 90	2,075 2,034	997 1,083
p = (:S)-[*0..7]-()-[*0..7]-(:T) p = (:S)-[*0..14]-(:T) *S = {5,21,34}, T = {0,13,29}* With 40 star clusters	Memory (bytes)	641,537,208 639,960,944	132,565,352 132,565,504	126,700,821,936 126,700,507,328	52,165,362,632 52,165,157,512
	Time (ms)	1,391 1,217	7,210 5,557	294,983 285,883	55,403 60,529
p = (:S)-[*0..7]-()-[*0..7]-(:T) p = (:S)-[*0..14]-(:T) *S = {5,21,34},* *T = {0,13,29}* With 400 star clusters	Memory (bytes)	614,667,064 613,303,704	136,183,064 136,497,368	132,341,751,672 132,341,751,672	56,127,352,112 56,127,143,008
	Time (ms)	659 508	7,570 7,082	281,756 283,417	41,938 41,719
p = (:S)-[*0..7]-()-[*0..7]-(:T) p = (:S)-[*0..14]-(:T) *S = {5,21,34,200},* *T = {0,13,29,206}* With 400 star clusters	Memory (bytes)	690,973,992 690,018,480	182,509,120 182,194,480	75,864,000,064 75,864,000,064	75,401,728,632 75,401,149,256
	Time (ms)	1,386 1,366	12,988 12,071	120,958 120,760	54,526 54,181
p = (:S)-[*0..7]-()-[*0..7]-(:T) p = (:S)-[*0..14]-(:T) *S = {5,21,34,200,315},* *T = {0,13,29,206,321}* With 400 star clusters	Memory (bytes)	704,554,096 702,396,168	229,820,888 229,506,456	94,709,360,280 94,709,360,280	93,360,010,288 93,360,010,288
	Time (ms)	732 511	18,678 20,109	161,438 161,020	74,331 69,086

Table A.2: Table over the memory usage in bytes and runtime in ms for the star graph in figure 3.4a. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for the rest. TB BFS is our divided Top Down Bottom Up BFS optimization and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product. The `S` nodes are the top of the star furthest from the connecting ring and the `T` nodes are one of the nodes connecting the star to the ring, with the numbers specifying which cluster they belong to.

Query		TB BFS	Regular BFS	DFS
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	1,081,088	3,243,144	2,995,144
p = (n0)-[*0..6]-(n1)		1,064,848	3,243,064	2,995,944
n0 is ”2s” and n1 is ”7t”	Time (ms)	21	18	9
With 10 tendrils		16	16	8
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	1,388,176	3,337,984	3,014,512
p = (n0)-[*0..6]-(n1)		1,388,176	3,389,048	3,014,512
n0 is ”2s” and n1 is ”4t”	Time (ms)	21	23	12
With 10 tendrils		37	21	12
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	1,821,456	21,727,616	21,172,008
p = (n0)-[*0..6]-(n1)		1,821,360	22,008,848	21,123,840
n0 is ”25s” and n1 is ”75t”	Time (ms)	21	82	28
With 100 tendrils		18	41	43
p = (n0)-[*0..4]-()-[*0..5]-(n1)	Memory (bytes)	29,831,336	5,396,855,216	5,310,326,400
p = (n0)-[*0..9]-(n1)		30,021,432	5,397,166,576	5,310,323,808
n0 is ”25s” and n1 is ”75t”	Time (ms)	57	5,585	3,057
With 100 tendrils		48	6,270	3,039
p = (n0)-[*0..5]-()-[*0..5]-(n1)	Memory (bytes)	205,392,368	Out of memory	34,465,984,352
p = (n0)-[*0..10]-(n1)		204,935,136		34,465,669,920
n0 is ”25s” and n1 is ”75t”	Time (ms)	255	–	20,956
With 100 tendrils		217		20,379
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	9,511,456	195,731,136	192,626,624
p = (n0)-[*0..6]-(n1)		9,421,160	194,483,240	192,626,624
n0 is ”25s” and n1 is ”75t”	Time (ms)	63	352	207
With 1,000 tendrils		40	169	211
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	8,866,912	Out of memory	302,441,871,216
p = (n0)-[*0..6]-(n1)		8,866,672		302,441,556,784
n0 is ”25s” and n1 is ”75t”	Time (ms)	17	–	219,181
With 2,000 tendrils		18		202,391

Table A.3: Table over the memory usage in bytes and runtime in ms for the directed cobweb graph in figure 3.4b with 10, 100, 1,000 and 2,000 tendrils. The source and target nodes were specified to be end nodes in the given tendrils. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for our regular BFS and the DFS. TB BFS is our divided Top Down Bottom Up BFS optimization.

Query		TB BFS	Regular BFS	DFS
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	1,388,176	3,337,984	3,014,512
p = (n0)-[*0..6]-(n1)		1,388,176	3,389,048	3,014,512
n0 is "2s" and n1 is "7t"	Time (ms)	21	23	12
With 10 tendrils		37	21	12
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	9,442,784	32,313,648	36,826,968
p = (n0)-[*0..6]-(n1)		9,431,208	32,312,104	36,811,848
n0 is "2s" and n1 is "4t"	Time (ms)	53	44	46
With 10 tendrils		38	41	42
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	2,126,680	87,837,000	85,059,576
p = (n0)-[*0..6]-(n1)		2,125,616	87,342,432	84,329,712
n0 is "25s" and n1 is "75t"	Time (ms)	23	269	103
With 100 tendrils		22	158	76
p = (n0)-[*0..4]-()-[*0..5]-(n1)	Memory (bytes)	114,948,672	Out of memory	46,225,707,360
p = (n0)-[*0..9]-(n1)		114,545,984		46,225,392,928
n0 is "25s" and n1 is "75t"	Time (ms)	219	–	41,294
With 100 tendrils		175		35,155
p = (n0)-[*0..5]-()-[*0..5]-(n1)	Memory (bytes)	1,303,432,096	Out of memory	385,518,770,192
p = (n0)-[*0..10]-(n1)		1,303,604,704		385,518,455,760
n0 is "25s" and n1 is "75t"	Time (ms)	893	–	297,847
With 100 tendrils		939		277,269
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	10,113,104	656,077,800	651,211,656
p = (n0)-[*0..6]-(n1)		10,114,152	656,077,560	651,211,656
n0 is "25s" and n1 is "75t"	Time (ms)	57	1,132	489
With 1,000 tendrils		43	512	474
p = (n0)-[*0..3]-()-[*0..3]-(n1)	Memory (bytes)	69,999,480	Out of memory	1,109,893,006,928
p = (n0)-[*0..6]-(n1)		69,723,592		1,109,892 692,496
n0 is "25s" and n1 is "75t"	Time (ms)	224	–	782,257
With 2,000 tendrils		179		832,103

Table A.4: Table over the memory usage in bytes and runtime in ms for the cobweb graph in figure 3.4b where the crossings are undirected, with 10, 100, 1,000 and 2,000 tendrils. The source and target nodes were specified to be end nodes in the given tendrils. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one for our regular BFS and the DFS. TB BFS is our divided Top Down Bottom Up BFS optimization.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	4,276,536	2,459,200	3,931,659,928	3,846,384,896
p=(:S)-[*0..6]-(:T)	(bytes)	4,275,216	2,459,200	3,931,652,952	3,846,384,896
S = {10}	Time	30	10	3,816	1,880
T = {5}	(ms)	23	9	4,023	1,904
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	16,326,768	10,489,408	687,716,968	683,691,984
p=(:S)-[*0..6]-(:T)	(bytes)	16,267,680	10,489,408	687,553,928	682,864,696
S = {10,19,32,54}	Time	74	31	518	451
T = {5,14,24,63}	(ms)	62	32	480	428
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	25,355,120	17,555,560	1,053,770,080	1,033,182,560
p=(:S)-[*0..6]-(:T)	(bytes)	25,076,672	17,560,512	1,042,973,568	1,033,182,512
S = {10,19,32,54,166,256}	Time	81	41	730	663
T = {5,14,24,63,875,932}	(ms)	63	38	697	632
p=(:S)-[*0..4]-()-[*0..4]-(:T)	Memory	3,687,943,160	303,902,776	Out of memory	381,341,040,344
p=(:S)-[*0..8]-(:T)	(bytes)	3,686,476,608	304,216,264		381,341,040,344
S = {10,19,32,54,166,256}	Time	2,111	7,665	–	264,225
T = {5,14,24,63,875,932}	(ms)	2,096	10,469		260,276

(a) Results for the directed cobweb graph.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	7,037,216	3,211,584	8,672,910,288	8,482,097,440
p=(:S)-[*0..6]-(:T)	(bytes)	7,037,216	3,211,656	8,672,681,536	8,482,096,880
S = {10}	Time	45	12	9,869	4,751
T = {5}	(ms)	33	10	9,437	4,690
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	27,859,384	13,918,864	134,987,937,144	2,173,144,848
p=(:S)-[*0..6]-(:T)	(bytes)	27,734,536	13,814,240	134,987,584,360	2,172,763,784
S = {10,19,32,54}	Time	94	87	159,288	1,495
T = {5,14,24,63}	(ms)	68	54	163,301	1,510
p=(:S)-[*0..3]-()-[*0..3]-(:T)	Memory	41,513,496	22,236,760	3,454,503,832	3,408,665,432
p=(:S)-[*0..6]-(:T)	(bytes)	41,328,640	22,043,896	3,454,442,712	3,408,665,432
S = {10,19,32,54,166,256}	Time	220	118	3,463	2,644
T = {5,14,24,63,875,932}	(ms)	104	88	2,964	2,228
p=(:S)-[*0..4]-()-[*0..4]-(:T)	Memory	9,052,905,384	681,234,904	Out of memory	1,571,813,884,928
p=(:S)-[*0..8]-(:T)	(bytes)	9,051,705,872	681,234,904		1,571,813,884,928
S = {10,19,32,54,166,256}	Time	10,360	28,890	–	934,258
T = {5,14,24,63,875,932}	(ms)	10,500	27,622		941,732

(b) Results for the cobweb graph where the crossings are undirected.

Table A.5: Table over the memory usage in bytes and runtime in ms for the cobweb graph in figure 3.4b using 1,000 tendrils. The source nodes were specified to be in the set `S` which are the middle nodes in the specified tendrils and target nodes to be in the set `T` which are the end nodes in the specified tendrils. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one is for the rest. TB BFS is our divided Top Down Bottom Up BFS optimization and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product.

Query		TB BFS	CTB BFS	Regular BFS	DFS
p=(n0)-[*0..10]-()-[*0..10]-(n1) p=(n0)-[*0..20]-(n1) *n0 is "13s" and n1 is "13t"* With 25 Layers	Memory (bytes)	5,122,120 5,121,304	Was not scheduled	15,604,112 15,604,056	15,102,888 15,102,888
	Time (ms)	40 28	–	48 45	36 37
p=(n0)-[*0..15]-()-[*0..15]-(n1) p=(n0)-[*0..30]-(n1) *n0 is "13s" and n1 is "13t"* With 25 Layers	Memory (bytes)	52,299,440 52,295,344	Was not scheduled	912,450,656 912,450,416	1,042,277,920 1,042,277 920
	Time (ms)	119 85	–	1,082 1,111	904 908
p=(:S)-[*0..6]-()-[*0..6]-(:T) p=(:S)-[*0..12]-(:T) *S = {1,7,9}* *T = {4,8,12}* With 13 Layers	Memory (bytes)	8,920,728 8,954,072	2,668,232 2,668,232	63,543,464 68,782,392	11,703,144 11,702,120
	Time (ms)	36 28	21 19	156 106	39 39
p=(:S)-[*0..8]-()-[*0..8]-(:T) p=(:S)-[*0..16]-(:T) *S = {1,7,9}* *T = {4,8,12}* With 13 Layers	Memory (bytes)	118,578,104 121,798,536	8,727,008 8,726,896	477,128,000 476,876,720	86,402,872 86,441,600
	Time (ms)	150 177	76 80	467 377	120 139
p=(:S)-[*0..8]-()-[*0..8]-(:T) p=(:S)-[*0..16]-(:T) *S = {1,7,9}* *T = {4,8,12}* With 25 Layers	Memory (bytes)	12,767,871,304 12,767,701,632	9,003,288 9,054,504	416,788,792 416,381,504	84,313,672 84,206 808
	Time (ms)	221 155	41 41	361 364	102 85
p=(:S)-[*0..10]-()-[*0..10]-(:T) p=(:S)-[*0..20]-(:T) *S = {1,7,9,15,18,20,23}* *T = {4,8,12,13,18,21,24}* With 25 Layers	Memory (bytes)	1,232,826,376 1,233,059,808	Was not scheduled	598,931,344 598,809,200	590,916,624 590,852,384
	Time (ms)	823 652	–	586 604	589 669
p=(:S)-[*0..12]-()-[*0..12]-(:T) p=(:S)-[*0..24]-(:T) *S = {1,7,9,15,18,20,23}* *T = {4,8,12,13,18,21,24}* With 25 Layers	Memory (bytes)	114,637,864 114,433,080	225,345,992 225,345,808	3,377,192,848 3,377,192,848	3,348,277,568 3,348,226,568
	Time (ms)	7,993 8,806	1,653 1,552	3,946 3,892	2,373 2,363

Table A.6: Table over the memory usage in bytes and runtime in ms for the cube graph in figure 3.4c. The source and target nodes were specified to be in opposite corners either by specific nodes or the types `S` and `T`, where the numbers represents which layer the nodes are in. Every query has two `MATCH p` clauses, the top one is for our TB BFS and bottom one is for the rest. TB BFS is our divided Top Down Bottom Up BFS optimization and CTB BFS is our Top down Bottom up optimization used together with a Cartesian product.

Appendix B

Benchmarks and Queries for Evaluation

Here follows descriptions of the benchmarks used in the evaluation. This includes size, some general information and the used queries from each benchmark. These queries were chosen because they had relationships with variable length in at least one `MATCH` clause, which is a criterion for our BFS algorithm. These relationships are **marked** in the queries for clarification. The results of the benchmark evaluation are not presented in this appendix.

B.1 LDBC Social Network Benchmark

With the LDBC social network benchmark [16] two different datasets were used, one small and one medium sized. The small dataset has an approximate size of 1.4 GB, the graph containing 3,158,994 nodes and 16,800,936 relationships. With an approximate size of 14 GB, 31,000,000 nodes and 168,000,000 relationships is the medium dataset about 10 times bigger than the smaller. There are ten queries of interest from this benchmark and they will be shown below.

Query 1

Finding the descriptions of friends with a given first name. The person whose friends are to be found is determined by a given id and the amount of returned friends are limited by a given resultLimit.

```
MATCH path=(:Person {id:{Person}})-[:KNOWS*1..3]-(friend)
WHERE friend.firstName={Name}
WITH friend, min(length(path)) AS distance
ORDER BY distance ASC, friend.lastName ASC, friend.id ASC
LIMIT {resultLimit}
MATCH (friend)-[:PERSON_IS_LOCATED_IN]->(friendCity:City)
OPTIONAL MATCH (friend)-[studyAt:STUDY_AT]->(uni:University)
```

```
   -[:ORGANISATION_IS_LOCATED_IN]->(uniCity:City)
WITH friend, collect(CASE WHEN uni IS NULL THEN null
   ELSE [uni.name, studyAt.classYear, uniCity.name] END)
   AS unis, friendCity, distance
OPTIONAL MATCH (friend)-[worksAt:WORKS_AT]->(company:Company)
   -[:ORGANISATION_IS_LOCATED_IN]->(country:Country)
WITH friend, collect(CASE WHEN company IS NULL THEN null
   ELSE [company.name, worksAt.workFrom, country.name] END)
   AS companies, unis, friendCity, distance
RETURN friend.id AS id, friend.lastName AS lastName, distance,
   friend.birthday AS birthday, friend.creationDate AS creationDate,
   friend.gender AS gender, friend.browserUsed AS browser,
   friend.locationIP AS locationIp, friend.email AS emails,
   friend.languages AS languages, friendCity.name AS cityName,
   unis, companies
ORDER BY distance ASC, friend.lastName ASC, friend.id ASC
```

Query 3

Finding friends within two steps that have recently (bounded by two given dates) travelled to two given countries. The person whose friends are to be found is determined by a given id and the amount of returned friends are limited by a given resultLimit.

```
MATCH (countryX:Country {name:{Country1}}),
   (countryY:Country {name:{Country2}}),
   (person:Person {id:{Person}})
WITH person, countryX, countryY
LIMIT 1
MATCH (city:City)-[:IS_PART_OF]->(country:Country)
WHERE country IN [countryX, countryY]
WITH person, countryX, countryY, collect(city) AS cities
MATCH (person)-[:KNOWS*1..2]-(friend)-[:PERSON_IS_LOCATED_IN]->(city)
WHERE NOT person=friend AND NOT city IN cities
WITH DISTINCT friend, countryX, countryY
MATCH (friend)<-[:POST_HAS_CREATOR|COMMENT_HAS_CREATOR]-(message),
   (message)-[:POST_IS_LOCATED_IN|COMMENT_IS_LOCATED_IN]->(country)
WHERE {Date1}>message.creationDate>={Date0}
   AND country IN [countryX, countryY]
WITH friend, CASE WHEN country=countryX THEN 1
   ELSE 0 END AS messageX, CASE WHEN country=countryY THEN 1
   ELSE 0 END AS messageY
WITH friend, sum(messageX) AS xCount, sum(messageY) AS yCount
WHERE xCount>0 AND yCount>0
RETURN friend.id AS friendId, friend.firstName AS friendFirstName,
   friend.lastName AS friendLastName, xCount, yCount,
   xCount + yCount AS xyCount
ORDER BY xyCount DESC, friendId ASC
LIMIT {resultLimit}
```

Query 5

Finding new groups that friends and friends-of-friends (of a given person) have joined recently (after a given date). The amount of returned groups are limited by a given resultLimit.

```
MATCH (person:Person {id:{Person}})-[:KNOWS*1..2]-(friend)
WHERE NOT person=friend
WITH DISTINCT friend
MATCH (friend)<-[membership:HAS_MEMBER]-(forum)
WHERE membership.joinDate>{Date0}
WITH forum, collect(friend) AS friends
OPTIONAL MATCH (friend)<-[:POST_HAS_CREATOR]-(post)<-[:CONTAINER_OF]-(forum)
WHERE friend IN friends WITH forum, count(post) AS postCount
RETURN forum.title AS forumName, postCount
ORDER BY postCount DESC, forum.id ASC
LIMIT {resultLimit}
```

Query 6

Find tags that occur together with a given tag on posts made by a given persons friends and friends-of-friends. The amount of returned tags are limited by a given resultLimit.

```
MATCH (knownTag:Tag {name:{Tag}})
MATCH (person:Person {id:{Person}})-[:KNOWS*1..2]-(friend)
WHERE NOT person=friend
WITH DISTINCT friend, knownTag
MATCH (friend)<-[:POST_HAS_CREATOR]-(post)
WHERE (post)-[:POST_HAS_TAG]->(knownTag)
WITH post, knownTag
MATCH (post)-[:POST_HAS_TAG]->(commonTag)
WHERE NOT commonTag=knownTag
WITH commonTag, count(post) AS postCount
RETURN commonTag.name AS tagName, postCount
ORDER BY postCount DESC, tagName ASC
LIMIT {resultLimit}
```

Query 9

Find the latest (before a given date) posts made by the friends and friends-of-friends of a given person. The amount of returned posts are limited by a given resultLimit.

```
MATCH (person:Person {id:{Person}})-[:KNOWS*1..2]-(friend)
WHERE NOT person=friend
WITH DISTINCT friend
MATCH friend)<-[:POST_HAS_CREATOR|COMMENT_HAS_CREATOR]-(message)
WHERE message.creationDate < {Date0}
WITH friend, message
ORDER BY message.creationDate DESC, message.id ASC
LIMIT {resultLimit}
RETURN message.id AS messageId,
   coalesce(message.content,message.imageFile) AS messageContent,
   message.creationDate AS messageCreationDate,
   friend.id AS personId, friend.firstName AS personFirstName,
   friend.lastName AS personLastName
```

Query 10

Find friends recommendation, that is friends of a friend that posts much about the interests and little of not interesting topics for a given person. The recommendation is restricted by horoscope sign (by giving the month). The amount of returned recommendations are limited by a given resultLimit.

```
MATCH (person:Person {id:{Person}})-[:KNOWS*2..2]-(friend),
   (friend)-[:PERSON_IS_LOCATED_IN]->(city)
WHERE NOT friend=person AND NOT (friend)-[:KNOWS]-(person)
   AND (
   (friend.birthday_month={HS0} AND friend.birthday_day>=21)
   OR (friend.birthday_month=({HS0}%12)+1 AND friend.birthday_day<22)
   )
WITH DISTINCT friend, city, person
OPTIONAL MATCH (friend)<-[:POST_HAS_CREATOR]-(post)
WITH friend, city, collect(post) AS posts, person
WITH friend, city, length(posts) AS postCount,
   length([p IN posts WHERE (p)-[:POST_HAS_TAG]->()
   <-[:HAS_INTEREST]-(person)]) AS commonPostCount
RETURN friend.id AS personId, friend.firstName AS personFirstName,
   friend.lastName AS personLastName, friend.gender AS personGender,
   commonPostCount-(postCount-commonPostCount) AS commonInterestScore,
   city.name AS personCityName
ORDER BY commonInterestScore DESC, personId ASC
LIMIT {resultLimit}
```

Query 11

Find job referrals by finding friends or friends-of-friends of a given person that have worked for a long time (since before a given year) at a company in a given country. The amount of returned job referrals are limited by a given resultLimit.

```
MATCH (country:Country {name:{Country}})
MATCH (person:Person {id:{Person}})-[:KNOWS*1..2]-(friend)
WHERE NOT person=friend
WITH DISTINCT friend, country
MATCH (friend)-[worksAt:WORKS_AT]->(company)
   -[:ORGANISATION_IS_LOCATED_IN]->(country)
WHERE worksAt.workFrom<{Year}
RETURN friend.id AS friendId, friend.firstName AS friendFirstName,
   friend.lastName AS friendLastName, worksAt.workFrom AS workFromYear,
   company.name AS companyName
ORDER BY workFromYear ASC, friendId ASC, companyName DESC
LIMIT {resultLimit}
```

Query 12

Find friends of a given person who have replied to the most posts with a tag from a given tag category. The amount of returned friends are limited by a given resultLimit.

```
MATCH (:Person {id:{Person}})-[:KNOWS]-(friend:Person),
   (friend)<-[:COMMENT_HAS_CREATOR]-(comment:Comment),
   (comment)-[:REPLY_OF_POST]->(post:Post),
   (post)-[:POST_HAS_TAG]->(tag:Tag),
   (tag)-[:HAS_TYPE|IS_SUBCLASS_OF*0..]->(:TagClass{name:{TagType}})
RETURN friend.id AS friendId, friend.firstName AS friendFirstName,
   friend.lastName AS friendLastName,
   collect(DISTINCT tag.name) AS tagNames,
   count(DISTINCT comment) AS count
ORDER BY count DESC, friendId ASC
LIMIT {resultLimit}
```

Short Query 2

Find the recent messages of a given person. The amount of returned messages are limited by a given resultLimit.

```
MATCH (:Person {id:{1}})<-[:POST_HAS_CREATOR|COMMENT_HAS_CREATOR]-(message)
WITH message, message.id AS messageId,
   message.creationDate AS messageCreationDate
ORDER BY messageCreationDate DESC, messageId ASC
LIMIT {resultLimit}
MATCH (message)-[:REPLY_OF_COMMENT|REPLY_OF_POST*0..]->(post:Post),
   (post)-[:POST_HAS_CREATOR]->(person)
RETURN messageId, messageCreationDate,
   coalesce(message.imageFile,message.content) AS messageContent,
   post.id AS postId, person.id AS personId,
   person.firstName AS personFirstName,
   person.lastName AS personLastName
ORDER BY messageCreationDate DESC, messageId ASC
```

Short Query 6

Find a forum and and the person that moderates it from a given message.

```
MATCH (post:Post)<-[:REPLY_OF_POST|REPLY_OF_COMMENT*0..]-(:Message {id:{1}}),
   (moderator)<-[:HAS_MODERATOR]-(forum)-[:CONTAINER_OF]->(post)
RETURN forum.id AS forumId, forum.title AS forumTitle,
   moderator.id AS moderatorId, moderator.firstName AS moderatorFirstName,
   moderator.lastName AS moderatorLastName
LIMIT 1
```

B.2 Pokec Benchmark

The Pokec benchmark is built upon a dataset from the online social network Pokec in Slovakia. It is anonymized and the user profiles contains among other things gender, age, hobbies, interest and education. Its graph contains 1,632,803 nodes and 30,622,564 relationships, with a size of 3.5 GB [17]. Six queries were looked at in particular from this benchmark, Q1, Q5, Q31, Q32, Q33 and WQ17, which are shown below.

Query Q1

Match profiles with specified variable distance between them, unspecified relationships. One of the profiles are given by its key.

```
MATCH (s:PROFILES {_key:{key}})-[*1..2]->(n:PROFILES)
RETURN DISTINCT n._key
```

Query Q5

Match profiles with specified distance between them, relationships specified to `RELATION`. One of the profiles are given by its key.

```
MATCH (s:PROFILES {_key:{key}})-[:RELATION*2..2]->(n:PROFILES)
RETURN DISTINCT n._key
```

Query Q31, Q32, Q33

Match profiles with specified distance between them, unspecified relationships. One of the profiles are given by its key. The difference between the three queries is the length of the relationship, for Q31 it is `3..3`, Q32 has `4..4` and Q33 has `5..5`.

```
MATCH (s:PROFILES {_key:{key}})-[*3..3]->(n:PROFILES)
RETURN DISTINCT n._key
```

Write Query WQ17

Match profiles with specified distance between them, relationships specified to `RELATION`. One of the profiles are given by its key. Creates a new direct relationship between the found profiles, this relationship is given an attribute holding the found paths length.

```
MATCH (a:PROFILES {_key: {key}})-[pathRels:RELATION*2..2]->(b)
FOREACH (r IN pathRels | DELETE r)
CREATE (a)-[:KNOWS {pathlen: size(pathRels)}]->(b)
```

B.3 Social Network Benchmark

The social network benchmark builds on real-world use cases from the O'Reilly book Graph Databases [9], and contains sub-domains such as HR, recruitment, skills and projects [19]. It contains 1,251,061 nodes and 56,357,466 relationships, with a size of 1.85 GB. Three queries were looked at in particular from this benchmark, Q7, Q8 and Q9, which are shown below.

Query Q7

Finding a friend of a friend to a given person with a specified interest, where the interest and resultLimit are given.

```
MATCH (subject:User {name:{name}})
MATCH p=(subject)-[:WORKED_ON]->()-[:WORKED_ON*0..2]-()
   <-[:WORKED_ON]-(person)-[:INTERESTED_IN]->(interest)
WHERE person<>subject AND interest.name={topic}
WITH DISTINCT person.name AS name, min(length(p)) AS pathLength
ORDER BY pathLength ASC
LIMIT {resultLimit}
RETURN name, pathLength
```

Query Q8

Finding a friend of a friend to a given person with an interest that belongs to a given set of interests. The amount of returned friends are given as resultLimit.

```
MATCH (subject:User {name:{name}})
MATCH p=(subject)-[:WORKED_ON]->()-[:WORKED_ON*0..2]-()
   <-[:WORKED_ON]-(person)-[:INTERESTED_IN]->(interest)
WHERE person<>subject AND interest.name IN {interests}
WITH person, interest, min(length(p)) as pathLength
ORDER BY interest.name
RETURN person.name AS name, count(interest) AS score,
   collect(interest.name) AS interests, ((pathLength-1)/2) AS distance
ORDER BY score DESC
LIMIT {resultLimit}
```

Query Q9

Finding a colleague of a given person who have worked with a person who have an interest that belongs to a given set of interests. The amount of returned persons are given as resultLimit.

```
MATCH (subject:User {name:{name}})
MATCH p=(subject)-[:WORKED_WITH*0..1]-()-[:WORKED_WITH]
   -(person)-[:INTERESTED_IN]->(interest)
WHERE person<>subject AND interest.name IN {interests}
WITH person, interest, min(length(p)) as pathLength
RETURN person.name AS name, count(interest) AS score,
   collect(interest.name) AS interests, (pathLength-1) AS distance
ORDER BY score DESC
LIMIT {resultLimit}
```

B.4 Logistics Benchmark

The Logistics benchmark builds on the same kind of use cases as the Social Network benchmark in section B.3. It is a small dataset, the graph only containing 806,799 nodes and 4,841,738 relationships, with a size of 402.7 MB. One query was looked at in particular from this benchmark, Q1, which is shown below.

Query Q1

Finding the shortest path between a given start and end location using cypher reduce. The start and end dates of the delivery route relationship are also given.

```
MATCH (s:Location {name:{startLocation}}),
   (e:Location {name:{endLocation}})
MATCH upLeg = (s)<-[:DELIVERY_ROUTE*1..2]-(db1)
WHERE all(r in relationships(upLeg)
   WHERE r.start_date <= {intervalStart} AND r.end_date >= {intervalEnd})
WITH  e, upLeg, db1
MATCH downLeg = (db2)-[:DELIVERY_ROUTE*1..2]->(e)
WHERE all(r in relationships(downLeg)
   WHERE r.start_date <= {intervalStart} AND r.end_date >= {intervalEnd})
WITH  db1, db2, upLeg, downLeg
MATCH topRoute = (db1)<-[:CONNECTED_TO]-()-[:CONNECTED_TO*1..3]-(db2)
WHERE all(r in relationships(topRoute)
   WHERE r.start_date <= {intervalStart} AND r.end_date >= {intervalEnd})
WITH  upLeg, downLeg, topRoute,
   reduce(weight=0, r in relationships(topRoute) | weight+r.cost) AS score
ORDER BY score ASC
LIMIT 1
RETURN (nodes(upLeg)+tail(nodes(topRoute))+tail(nodes(downLeg))) AS n
```

B.5 Access Control Benchmark

Another benchmarks that builds on the same kind of use cases as the Social Network benchmark, section B.3, is the Access Control benchmark. It is fairly big, containing 6,053,922 nodes and 6,211,289 relationships with a size of 1.54 GB. We looked at seven queries in particular from this benchmark, Q8 to Q14, which are shown below.

Query Q8

Find the Companies accessible for a given admin.

```
MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()
   <-[:CHILD_OF*0..3]-(company)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(company))
RETURN company.name AS company
UNION
MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->(company)
RETURN company.name AS company
```

Query Q9

Find admin with allowed inherit as well as not allowed inherit for a given account resource.

```
MATCH (resource:Resource {name:{resourceName}})
MATCH p=(resource)-[:WORKS_FOR|HAS_ACCOUNT*1..2]-(company)
   -[:CHILD_OF*0..3]->()<-[:ALLOWED_INHERIT]-()<-[:MEMBER_OF]-(admin)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(company))
RETURN admin.name AS admin
UNION
MATCH (resource:Resource {name:{resourceName}})
MATCH p=(resource)-[:WORKS_FOR|HAS_ACCOUNT*1..2]-(company)
   <-[:ALLOWED_DO_NOT_INHERIT]-()<-[:MEMBER_OF]-(admin)
RETURN admin.name AS admin
```

Query Q10

Find accessible accounts for given admin and company.

```
MATCH (admin:Administrator {name:{adminName}}),
   (company:Company {name:{companyName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_INHERIT]->(company)
   <-[:CHILD_OF*0..3]-(subcompany)<-[:WORKS_FOR]-(employee)
   -[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(subcompany))
RETURN account.name AS account
UNION
MATCH (admin:Administrator {name:{adminName}}),
   (company:Company {name:{companyName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_DO_NOT_INHERIT]->(company)
   <-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN account.name AS account
```

Query Q11

Find all accessible accounts for a given admin and any matching company.

```
MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_INHERIT]->(company:Company)
   <-[:CHILD_OF*0..3]-(subcompany)<-[:WORKS_FOR]-(employee)
   -[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(subcompany))
RETURN account.name AS account
UNION
MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)
   -[:ALLOWED_DO_NOT_INHERIT]->(company:Company)
   <-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN account.name AS account
```

Query Q12

Find admin, for a given company, with allowed inherit.

```
MATCH (company:Company {name:{companyName}})
MATCH (company)-[:CHILD_OF*0..3]->()<-[:ALLOWED_INHERIT]-()
   <-[:MEMBER_OF]-(admin)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(company))
RETURN admin.name AS admin
UNION
MATCH (company:Company {name:{companyName}})
MATCH (company)<-[:ALLOWED_DO_NOT_INHERIT]-()<-[:MEMBER_OF]-(admin)
RETURN admin.name AS admin
```

Query Q13

For a given admin with allowed inherit find sub-companies, their employees and accounts.

```
MATCH (admin:Administrator {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()
   <-[:CHILD_OF*0..3]-(company)<-[:WORKS_FOR]-(employee)
   -[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(company))
RETURN employee.name AS employee, account.name AS account
UNION
MATCH (admin:Administrator {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->()
   <-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN employee.name AS employee, account.name AS account
```

Query Q14

Find if a given admin has access to a given resource.

```
MATCH (admin:Administrator {name:{adminName}}),
   (resource:Resource {name:{resourceName}})
MATCH p=(admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()
   <-[:CHILD_OF*0..3]-(company)-[:WORKS_FOR|HAS_ACCOUNT*1..2]-(resource)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
   <-[:CHILD_OF*0..3]-(company))
RETURN count(p) AS accessCount
UNION
MATCH (admin:Administrator {name:{adminName}}),
   (resource:Resource {name:{resourceName}})
MATCH p=(admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->(company)
   -[:WORKS_FOR|HAS_ACCOUNT*1..2]-(resource)
RETURN count(p) AS accessCount
```

B.6 MusicBrainz Benchmark

MusicBrainz is a large music dataset with information about artists, their recorded works and the relationships between them [21]. The graph contains 35,778,711 nodes and 73,433,368 relationships, with a dataset size of 18.85 GB. Two queries were looked at in particular from the Neo4j benchmark on this graph, Q14 and Q27, which are shown below.

Query Q14

Find the 3rd degree network (via bands) for John Lennon.

```
MATCH (a:Artist {name:'John Lennon'})-[r:MEMBER_OF_BAND*3..6]-(o:Person)
RETURN o.name,count(*)
ORDER BY count(*) DESC
LIMIT 10
```

Query Q27

Find the 3rd degree network (via bands) for John Lennon, with a restriction placed on the found artists.

```
MATCH p=(a:Artist {name:'John Lennon'})-[r:MEMBER_OF_BAND*..10]-(o:Person)
WHERE all(n in nodes(p) WHERE n.prop = 42)
RETURN p
```

B.7 Generated Music A benchmark provided by Neo4j

This benchmark was provided by Neo4j and contains a small synthetically generated music dataset of size 45.7 MB, with 113,754 nodes and 136,800 relationships in its graph. Three queries from this benchmark were looked at in particular, Q25, Q30 and WQ8, which are shown below.

Query Q25

Find all distinct artists that have worked four steps away from a given artist, that is they have worked with someone who worked with someone who worked with someone who worked with the given artist.

```
MATCH (a1:Artist)-[:WORKED_WITH * 4]->(a2:Artist)
WHERE id(a1) = {id}
RETURN DISTINCT a2
```

Query Q30

Find all artists that have worked together (or within five persons from each other) in a given year.

```
MATCH (a:Artist)-[:WORKED_WITH*..5 { year: {year} }]->(b:Artist)
RETURN *
```

Write Query WQ8

A write query using the same `MATCH` as Q30. Creates a `COLLABORATOR` relationship between the found artists if not already existing, otherwise update the existing relationship.

```
MATCH (a:Artist)-[:WORKED_WITH*..5 { year: {year} }]->(b:Artist)
MERGE (a)-[r:COLLABORATOR {year: {year}}]->(b)
SET r.updated_at = timestamp()
```

B.8 LevelStory
Another benchmark provided by Neo4j

This benchmark was provided by Neo4j and is built on a small artificial dataset, only 89 MB, its graph containing 18,748 nodes and 72,047 relationships. This datasets content emulates a company, with things like employees, contractors, admins and invoices. Four queries were looked at in particular from this benchmark, Q1, Q2, Q3 and Q9, which are shown below.

Query Q1

Find the activity for a given project. The amount of activities is restricted by given parameters.

```
MATCH (:project {id:{p01}})-[:ACTIVITY*]->(a:activity)
WITH DISTINCT a SKIP {p02} LIMIT {p03}
MATCH (c:contact {id: a.contactid})
RETURN COLLECT({id: a.id, object: a.object, values: a.values,
   createdat: a.createdat, primaryaction: a.primaryaction,
   primaryobject: a.primaryobject, primaryno: a.primaryno,
   primaryname: a.primaryname, primaryid: a.primaryid,
   createdby: {id: c.id, object: c.object, name: c.name,
      description: c.imageurl}
   }) AS r
```

Query Q2

Find the history of a given task. The amount of activities is restricted by given parameters.

```
MATCH (i:task {id: {p01}})
WITH i
MATCH (i)-[rel:HISTORY*0..20]->(a:activity)
WHERE ALL(r IN rel WHERE r.id = i.id)
WITH DISTINCT a
SKIP {p02} LIMIT {p03}
MATCH (c:contact {id: a.contactid})
RETURN COLLECT({id: a.id, object: a.object, values: a.values,
   createdat: a.createdat, primaryaction: a.primaryaction,
   primaryobject: a.primaryobject, primaryno: a.primaryno,
   primaryname: a.primaryname, primaryid: a.primaryid,
   createdby: {id: c.id, object: c.object, name: c.name,
      description: c.imageurl}
   }) AS r
```

Query Q3

Find alerts for a given user.

```
MATCH (u:user {id: {p01}})
WITH u
MATCH (u)-[:CONTACT]->(c:contact)-[:CONTACT*1..3]->(r:projectrole)
   -[:CONTACT*1..3]->(p:project {status: 'active'})
WITH u, r
MATCH (r)-[:ALERT]->(:alerts)-[x:ALERT]->(y)
WITH u, {id: x.id, object: x.object, projectid: y.projectid,
   alerttype: x.alerttype, severity: COALESCE(x.severity, 0),
   visibility: r.name, code: x.code, data: x.data, createdat: x.createdat,
   isread: COALESCE(u.lastreadalert, 0) >= x.createdat,
   target: {id: y.id, object: y.object, name: y.name,
      description: TOSTRING(y.no)}
   } AS r
LIMIT 100
RETURN COLLECT(r) as r
```

Query Q9

Find user project tasks by use of optional match. The user and task ids are given.

```
MATCH (u:user {id: {p01}}), (x:task {id: {p02}})
WITH u, x
MATCH (u:user {id: {p01}}), (t:task {id: {p02}})
WITH u, t
MATCH (u)-[:CONTACT]->(c:contact), (xp:project)->(:tasks)->(t)
WHERE c.teamid = xp.teamid
WITH c, t, xp
OPTIONAL MATCH (c)-[:CONTACT]->(r:role)-[:CONTACT*1..10]->(xp)
WITH t, c, COLLECT(r) AS rs
WITH t, {accessedby: c.id, accessedrole: CASE
   WHEN ANY(r IN rs WHERE r.name = 'Owners') THEN 'Owners'
   WHEN ANY(r IN rs WHERE r.name = 'Admins') THEN 'Admins'
   WHEN ANY(r IN rs WHERE r.name = 'ProjectAdmins') THEN 'ProjectAdmins'
   WHEN ANY(r IN rs WHERE r.name = 'ProjectClients') THEN 'ProjectClients'
   WHEN ANY(r IN rs WHERE r.name = 'ProjectContacts')THEN 'ProjectContacts'
   ELSE 'None' END} AS r
OPTIONAL MATCH (t)-[:CREATED]->(a:activity)
WITH t, {accessedrole: r.accessedrole, accessedby: r.accessedby,
   createdat: a.createdat, createdby: {id: a.contactid,
      object: 'contact'}} AS r
OPTIONAL MATCH (t)-[:HISTORY]->(a:activity)
WITH t, {accessedrole: r.accessedrole, accessedby: r.accessedby,
   createdat: r.createdat, createdby: r.createdby, updatedat: a.createdat,
   updatedby: {id: a.contactid, object: 'contact'}} AS r
OPTIONAL MATCH (t)<-[:TASK|LINEITEM]-(x)
WITH t, COLLECT(x) AS xs, r
WITH t, {accessedrole: r.accessedrole, accessedby: r.accessedby,
   createdat: r.createdat, createdby: r.createdby, updatedat: r.updatedat,
   updatedby: r.updatedby, tasksafter: [x IN xs WHERE
   x.object = 'task' | {id: x.id, object: x.object}],
   lineitems: [x IN xs WHERE x.object='lineitem' | {id: x.invoiceid,
   object: 'lineitem', amount: x.amount, rate: x.rate}]} AS r
```

```
OPTIONAL MATCH
   (t)-[:TASKTYPE|AREA|CONTACT|TASK|MATERIAL|FILE|MESSAGE|TAG]->(x)
WITH t, COLLECT(x) AS xs, r
WITH t, {accessedrole: r.accessedrole, accessedby: r.accessedby,
   createdat: r.createdat, createdby: r.createdby, updatedat: r.updatedat,
   updatedby: r.updatedby, tasksafter: r.tasksafter,
   lineitems: r.lineitems, area: HEAD([x IN xs WHERE
   x.object='area' | {id: x.id, object: x.object}]),
   tasktype: HEAD([x IN xs WHERE x.object = 'tasktype' | {id: x.id,
   object: x.object, name: x.name}]), assignedto: [x IN xs WHERE
   x.object = 'contact' | {id: x.id, object: x.object}],
   tasksbefore: [x IN xs WHERE x.object = 'task' | {id: x.id,
   object: x.object}], materials: [x IN xs WHERE
   x.object = 'material' | {id: x.id, object: x.object}],
   messages: [x IN xs WHERE x.object = 'message' | {id: x.id,
   object: x.object}], files: [x IN xs WHERE
   x.object = 'file' | {id: x.id, object: x.object}],
   tags: [x IN xs WHERE x.object = 'tag' | {id: x.id,
   object: x.object, name: x.name}]} AS r
MATCH (t)-[:SCHEDULE]->(x:schedule)
OPTIONAL MATCH (x)<-[:SCHEDULE*1..4]-(c:contact)
OPTIONAL MATCH (c)<-[:CONTACT]-(u:user)
WITH t, x, COLLECT(DISTINCT {contact: c, user: u}) AS cs, r
WITH t, {schedule: {id: x.id, object: x.object, name: x.name,
   description: x.description, teamid: x.teamid,
   scheduletype: x.scheduletype, startat: x.startat,
   endat: x.endat, isautoscheduled: x.isautoscheduled,
   flexscore: x.flexscore, schedules: x.schedules, worklogs: x.worklogs,
   info: x.info, contacts: [c IN cs | {id: c.contact.id,
   object: c.contact.object, name: COALESCE(c.contact.displayname,
   c.contact.name, c.user.displayname, c.user.name)}]},
   accessedrole: r.accessedrole, accessedby: r.accessedby,
   createdat: r.createdat, createdby: r.createdby,
   updatedat: r.updatedat, updatedby: r.updatedby,
   tasksafter: r.tasksafter, lineitems: r.lineitems,
   area: r.area, tasktype: r.tasktype, assignedto: r.assignedto,
   tasksbefore: r.tasksbefore, materials: r.materials,
   messages: r.messages, files: r.files, tags: r.tags} AS r
OPTIONAL MATCH ()-[x:ALERT]->(t)
WITH t, COLLECT(x) AS xs, r
WITH t, {schedule: r.schedule, accessedrole: r.accessedrole,
   accessedby: r.accessedby, createdat: r.createdat,
   createdby: r.createdby, updatedat: r.updatedat,
   updatedby: r.updatedby, tasksafter: r.tasksafter,
   lineitems: r.lineitems, area: r.area,
   tasktype: r.tasktype, assignedto: r.assignedto,
   tasksbefore: r.tasksbefore, materials: r.materials,
   messages: r.messages, files: r.files, tags: r.tags,
   alerts: [x IN xs | {id: x.id, object: x.object,
   alerttype: x.alerttype, severity: COALESCE(x.severity, 0),
   visibility: r.name, code: x.code, data: x.data,
   createdat: x.createdat}]} AS r
RETURN COLLECT({id: t.id, object: t.object, projectid: t.projectid,
   status: t.status, no: t.no, name: t.name, description: t.description,
   workhours: t.workhours, priority: t.priority, fixedcost: t.fixedcost,
   isfixedcost: t.isfixedcost, ischange: t.ischange, sameday: t.sameday,
minutestotal: t.minutestotal, delayamount: t.delayamount,
delayperiod: t.delayperiod, minassigned: t.minassigned,
maxassigned: t.maxassigned, minschedule: t.minschedule,
scheduletype: COALESCE(t.scheduletype, 'none'),
fixedstartat: t.fixedstartat, fixedendat: t.fixedendat,
fixedschedule: t.fixedschedule, accessedrole: r.accessedrole,
accessedby: r.accessedby, createdat: r.createdat,
createdby: r.createdby, updatedat: r.updatedat, updatedby: r.updatedby,
tasksafter: r.tasksafter, lineitems: r.lineitems, area: r.area,
tasktype: r.tasktype, assignedto: r.assignedto,
tasksbefore: r.tasksbefore, materials: r.materials, messages: r.messages,
files: r.files, tags: r.tags, schedule: r.schedule,
alerts: r.alerts}) AS r
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2018-08-30

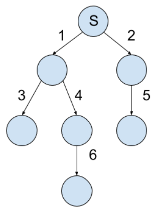
EXAMENSARBETE Implementing and Evaluating a Breadth-First Search in Cypher
STUDENTER Alexander Olsson, Therese Magnusson
HANDLEDARE Krzysztof Kuchcinski (LTH), Tobias Lindaaker (Neo4j)
EXAMINATOR Flavius Gruian (LTH)

Kan vi söka effektivare i grafdatabaser?

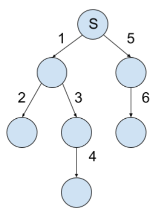
POPULÄRVETENSKAPLIG SAMMANFATTNING **Alexander Olsson, Therese Magnusson**

Grafdatabaser blir mer och mer populära. När populariteten ökar så önskas det att sökningar i grafdatabasen utförs snabbt och korrekt. Därför behövs det snabba och pålitliga sökalgoritmer för att genomföra sökningarna.

Eftersom grafdatabaser blir mer och mer populära behövs bra sökalgoritmer för att snabbt och pålitligt söka efter matchningar till den givna förfrågan. De två vanligaste sökalgoritmerna i grafer är bredden-först sökning (BFS) och djupet-först sökning (DFS). BFS och DFS är två liknande algoritmer, där skillnaden är att BFS söker brett först medans DFS söker djupt först. Exempel på deras sökordningar visas i figurerna nedan.



En BFS sökning



En DFS sökning

Båda algoritmerna har länge studerats men ej i anslutning till grafdatabaser. Grafdatabasen Neo4j och dess språk Cypher kan få mycket nytta med att ha en BFS operator utöver deras nuvarande DFS operator. Detta eftersom BFS och DFS har olika styrkor och vilken av dem som passar bäst för ett visst problem är ej självklart innan de jämförs på, iallafall, liknande problem.

Grafdatabaser är en form av databas som lagrar sin information i grafer, där relationerna mellan datan sparas. Exempel på områden där detta är användbart är bedrägeri detektering, nätverk- och IT-verksamhet, identitets- och åtkomsthantering och motorer för realtidsrekommendationer[1]. Grafdatabaser används bland annat av Google, Facebook, LinkedIn och PayPal.

I vårt examensarbete samarbetade vi med företaget Neo4j och utvecklade en BFS prototyp med flera olika optimeringar till deras grafdatabas. De utvärderades gentemot varandra och den redan existerande sökalgoritmen. De optimeringar vi implementerade rörde att dela upp sökningen i flera mindre sökningar, så kallat Top down Bottom up mönster. Fördelen med dessa optimeringar är att undvika onödiga noder i sökningen.

Resultatet från vår utvärdering visade att våran BFS prototyp vanligen var jämlik med DFS operatorn. Men för mindre grafer var vi upp till 90% snabbare och för större grafer upp till sex gånger långsammare. Överlag var BFS bäst för mindre grafer och mindre sökningar, med strängare restriktioner. Optimeringarna gav signifikanta förbättringar för större grafer. De kräver dock ännu fler restriktioner och är därför svårare att använda i dagliga situationer.

Vår utvärdering visade att använda BFS på grafdatabaser kan ge både stora förbättringar och försämringar, beroende på grafen. Detta är intressant eftersom det visar att detta är ett område som borde utforskas mer.

[1]https://neo4j.com/why-graph-databases/