TUHH
Hamburg University of Technology

Bachelor Thesis

Hai Nam Tran

# Performance prediction of reachability queries over a large knowledge graph

January 24, 2022

supervised by:
Prof. Dr. Sibylle Schupp
Lars Beckers

Hamburg University of Technology (TUHH)
*Technische Universität Hamburg*
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

# Declaration of Originality[1]

Hereby I confirm, Hai Nam Tran, Mat. Nr. 21652379, that this assignment is my own work and that I have only sought and used mentioned tools.

I have clearly referenced in the text and the bibliography all sources used in the work (printed sources, internet or any other source), including verbatim citations or paraphrases.

I am aware of the fact that plagiarism is an attempt to deceit which, in case of recurrence, can result in a loss of test authorization.

Furthermore, I confirm that neither this work nor parts of it have been previously, or concurrently, used as an exam work – neither for other courses nor within other exam processes.

Hamburg, January 24, 2022

Hai Nam Tran

[1] Text source: University of Tübingen, Department of Sociology

**Abstract**

Graph databases, particularly knowledge graphs, have become widely popular in recent times. Therefore, the need for querying such databases has increased substantially and resulted in a growing demand for supporting effective querying over such extensive graph-structured databases. One major obstacle is that graph analytical queries are more complex and have no well-defined structure, unlike their traditional relational counterparts. New and effective models of prediction are required to achieve satisfactory results. In this thesis, we take a look at a novel performance prediction model for graph queries. We then evaluate the framework with reachability queries over a large knowledge graph using real-world data. Finally, we design experiments to show that the framework is ready to be applied to the resource allocation and optimization problem.

# Contents

# List of Figures

# List of Listings

# List of Algorithms

# List of Tables

# 1 Introduction

The internet is a vast collection of information readily available for everyone to query its knowledge. *Knowledge Graph* (KG) is a way to organize and abstractly portray relationships between entities of interest. KGs are utilized broadly in search engines to display related information and recommendations to the users. Over those KGs, reachability queries are used for data mining, traffic analysis, and knowledge extraction. Reachability queries have been studied extensively in the past, albeit only in small-scale graphs. In real-world KGs with billions of entities and relations, such a query can be costly and unpredictable.

Systems with popular databases that get queried frequently need to optimize and allocate resources to allow a high quality of service. That includes predicting the performance of incoming queries to execute those that bring the highest benefit. Various prediction models are available and effectively utilized in relational databases. However, those models are often not applicable or only deliver poor results for graph databases [7]. Graph queries are often abstract and do not provide enough information to the traditional models. By exploiting unique features that only exist in graphs, new models have been derived to tackle the performance prediction problem for graph databases.

Database services often need to prioritize critical queries that bring a high profit or short queries to serve as many users as possible. An optimizer that chooses a subset of queries to execute that satisfies the requirement of the database is essential. With the runtime provided by the predictive models, the optimizer can easily aid with database query selection. Multiple optimization strategies are compared in different scenarios to show their robustness against multiple variables.

This thesis presents an implementation of a novel performance prediction framework. The following contributions are discussed and implemented in detail:

- A method to parameterize graph queries by exploiting unique features of graph databases.
- An algorithm to solve the reachability problem by using bi-directional breadth-first search.
- Implementation and comparison of multiple predictive models that predict query execution time.
- Application of predictive models on workload optimization problems.
- Experiments to verify the effectiveness of the framework.

We show that the framework can reliably deliver satisfactory results using real-world data. The best predictive model has a high accuracy score without complicated or prolonged training. In a resource-bound scenario, the optimizer can significantly improve the effectiveness of the database. The framework promises a valuable tool for database servers to utilize.

The rest of the thesis is organized as follows. Chapter 2 establishes the background of the thesis. We introduce the original paper and the theoretical connotations for the methods used in the thesis. Chapter 3 showcases our implementation of the framework; this covers both the predictive models and the optimizer. In Chapter 4, experiments are conducted to showcase the framework in action. Furthermore, we also discuss the process of data gathering and share the anomalies that were encountered during the process. We conclude the thesis and give an outlook on possible future works in Chapter 5.

# 2 Background

This chapter presents the theoretical background needed for the framework and summarizes the concepts and topics in the thesis. In particular, we introduce the concept of *Knowledge Graph*, its application in graph databases, and the type of queries performed on it. We also present various types of machine learning utilized in the framework.

In their 2017 paper "Performance prediction for graph queries" [7], Namaki et al. developed and proposed the original framework. They identified the lack of predictive models for graph queries, even though query performance prediction generally plays an integral role in resource optimization. They contributed a general learning framework and tested it on two classes of graph queries: reachability and graph pattern. We propose an implementation of their framework and go in-depth with the reachability queries.

## 2.1 Knowledge graph

A graph is a mathematical model consisting of nodes and edges connecting those nodes. A *Knowledge Graph* is a directed graph that has well-defined labels for nodes and edges. Each node represents a unique entity in the system, and each edge shows the relationship between any two nodes. A node can be, for example, a person, an award, or a place. Figure 2.1 shows an example of a KG. It contains relationships such as "person A - *has* - award C". In this relationship, *has* is the label of the edge that defines the relationship between two nodes, *person A* and *award C*. The graph is directed, which means the edge - thus the relationship - is non-symmetric. However, it allows the existence of an opposite edge. That means "Cows - *eat* - Grass" does not necessarily mean "Grass - *eat* - Cows", however "Grass - *is food of* - Cows" is possible.

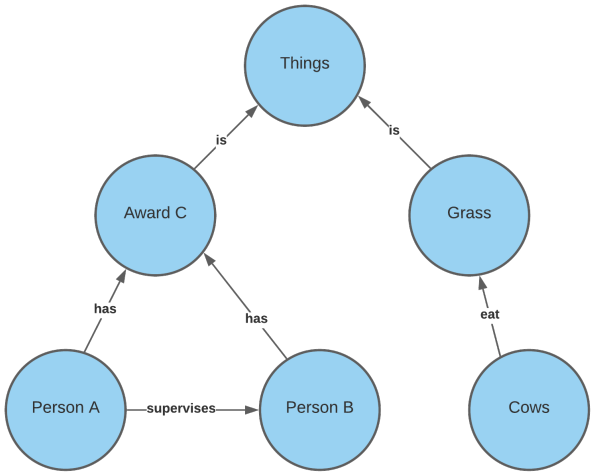


Figure 2.1: An example of a KG

**Definition 1** (Knowledge graph) [7]. A knowledge graph is a labeled and directed graph $G = (V, E, \mathcal{L})$ where V is the node set and E is the edge set. For each node $v \in V$ (and edge $e \in E$), its label is defined by $\mathcal{L}(v)$ (and $\mathcal{L}(e)$).

Nowadays, KGs are ubiquitous and help create an interconnected web of data. Google employs a successor of Freebase to enrich the search results. For example, when a user queries for an author of a book, Google will additionally show other authors that have collaborated or written in the same genre as the original author.

## 2.2 Graph database

**Definition 2** (RDF Triple). A KG is usually stored in the form of RDF triples. RDF triples are the data entities in the Resource Description Framework (RDF) model. Each triple consists of three entities following the *subject-predicate-object* model. With this representation, computers can easily read and perform queries on data and knowledge. Contrasted to traditional relational databases, which use tables for storage, graph databases store their data in text files using triples. The main reason for that choice is the complicated relationships between multiple entities in the system need a multitide of table.

RDF has become a W3C recommendation as a standard for data interchange on the web. Each entity in the RDF model is represented using an *Internationalized Resource Identifier* (IRI), which itself is a generalization of *Uniform Resource Identifiers* (URI). That allows each entity to have a unique identification in the system that reduces the ambiguity when describing those resources.

This thesis focuses on data from DBPedia, a crowd-sourced project that aims to extract structural and factual information from Wikipedia and store them using RDF. Resources from DBPedia are publicly available for querying. They also regularly publish a part of their database that contains basic information from every Wikipedia articles to the public. This information generally includes the abstract infobox that summarizes each article, how each article relates to the other, and links to similar resources from other databases. Their database is already used in multiple projects, such as the version of IBM Watson that won Jeopardy! [2]. Listing 1 shows a subset of triples taken from DBPedia that describe the Wikipedia page *Albert Einstein*.

```
...
dbr:Albert_Einstein    dbo:academicDiscipline    dbr:Physics
dbr:Albert_Einstein    dbo:wikiPageID            736
dbr:Albert_Einstein    dbo:wikiPageWikiLink      dbr:Vladimir_Lenin
...
```

Listing 1: Sample subset of RDF triples describing Albert Einstein's page

We can see from that small subset of triples that Einstein has done his academic works in the field of Physics. The Wikipedia page about him has the id of *736*. The page also has a link that leads to the page about Vladimir Lenin. Additional triples can describe other relationships with different entities in the KG.

## 2.3 Graph query

**Definition 3** (SPARQL). The SPARQL Protocol and RDF Query Language (SPARQL) is a query language that is used to query databases with RDF storage. It is the *de facto* standard as the W3C has officially recommended its usage to query RDF since 2008 [9]. SPARQL queries can be executed over a graph database on a SPARQL endpoint, that is publicly available at popular sites that use KG, such as DBPedia. A SPARQL query is composed of three parts [8]: the *output* part sets the type of query, e.g. SELECT to choose a predetermined set of triple or DESCRIBE to get all triple related to an entity; a *pattern matching* part that describes the result; lastly, a *solution modifier* that allows basic operation on the set of results, e.g. set a limit or sorting. Listing 2 shows an example for a SPARQL query, with its result set depicted in Figure 2.2.

```
1    SELECT ?origin ?linkto                          #output
2    WHERE {                                         #pattern matching
3        ?origin  dbo:wikiPageWikiLink  ?linkto.
4        ?origin  dbo:wikiPageID        736.
5    } LIMIT 10                                      #solution modifier
```

Listing 2: A SPARQL Query

SPARQL | HTML5 table

| origin | linkto |
|---|---|
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Category:Charles_University_faculty |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Category:Swiss_Jews |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/New_Humanist |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Spinozism |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Why_Socialism%3F |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Grading_systems_by_country |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Leiden_University |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/Universal_Studios |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/UNESCO |
| http://dbpedia.org/resource/Albert_Einstein | http://dbpedia.org/resource/University_of_Bern |

Figure 2.2: Set of results returned by DBPedia

The query above, once executed, accomplishes the following. It searches for and selects every *origin* and *linkto* entity in the database, whereas these patterns must match: *origin* entity has the *wikiPageWikiLink* relationship with *linkto* entity; *origin* entity

has *wikiPageID* relation with *736*; i.e., the query searches the database for triples that match the patterns *origin-wikiPageWikiLink-linkto*, and *origin-wikiPageID-736*. Lastly, the query limits the result set to only ten results.

**Definition 4** (Reachability query). This thesis studies a familiar query class: reachability queries. Reachability queries are an elemental procedure that can be applied to a graph. They are extensively and regularly used for data and traffic analysis. Given a directed graph $G$, a source node $s$, a destination node $t$, and a bound $d$, the query $Q(G, s, t, d)$ answers the question: "Does a path in $G$ between $s$ and $t$ exist that is bounded in length by $d$?"

## 2.4 Performance prediction with machine learning

This section gives an overview of machine learning with its definition and usage. Since there are many different types of machine learning, we limit the overview to the types of models used in our implementation. We show metrics that help evaluate the model performance and introduce various machine learning models that we implemented.

### 2.4.1 Machine learning

**Definition 5** (Machine learning). The framework uses machine learning to make predictions about the runtime of the queries. Machine learning is the class of computer algorithms that enable programs to solve problems without explicit instruction. In this thesis, we employ machine learning to learn from a set of predefined queries to predict the outcome of future queries that have not yet been encountered. Two main types of machine learning algorithms exist: supervised and unsupervised learning. In the thesis, we focus on the former.

**Definition 6** (Supervised training). In supervised learning, the machine learns from a set of training examples [10]. These training data are prelabeled and consist of pairs of input and output. Additional preprocessing may be needed such as removing outliers or standardizing the values. Preprocessing of the dataset helps improve the general quality of the dataset and remove noise and outliers that interfere with the learning phase of the model. From the set of training data a predictive model is built and the machine will make prediction on further input according to that model [6].

Figure 2.3 shows the general workflow of the framework proposed by Namaki et al. [7]. The first part is offline learning, where a machine learning model is developed and trained on the database. Random queries are chosen and appropriately labeled for the model to learn from. The model is used for the second stage: online prediction. The model predicts the runtime of given graph queries and these results are in turn used for the workload optimizer. Together with additional input from the user, they form a closed loop that is centered around the graph database.

With numerous machine learning models to choose from, the task of choosing the right one for the job becomes an integral part of the process. The best model is the one that has the highest accuracy according to several metrics.

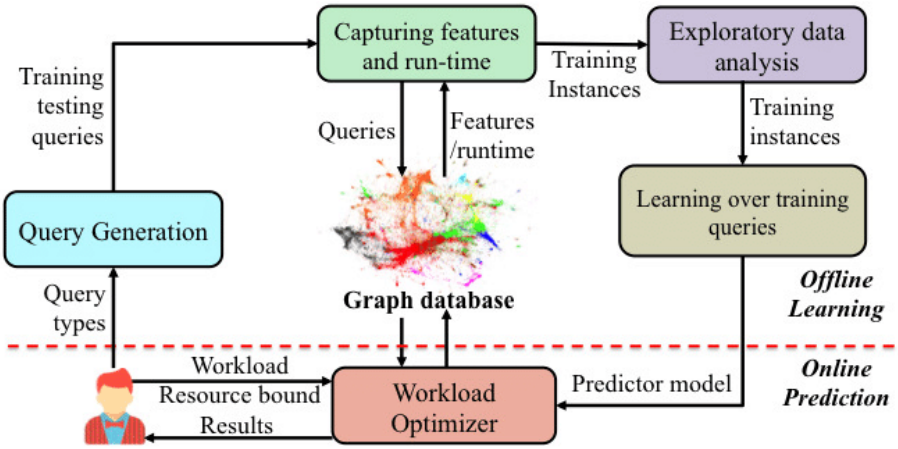


Figure 2.3: Workflow of the framework [7]

### 2.4.2 Model evaluation

The goal of the framework is to accurately predict the performance, in this case, runtime, of a given query over a graph. Performance metrics are a vital part of measuring the accuracy of predictions. Numerous metrics are available to help with the accuracy evaluation; we consider *R-Squared Accuracy* (R2) and *Mean Absolute Error* (MAE) in our implementation.

**Definition 7** (R-Squared Accuracy). R2 explains the proportion of variance of the actual value that the independent variables in the models have explained. It roughly shows how well the model fits the input dataset and how good a prediction the model can produce from that fit. R2 is calculated as follows:

$$R2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

where $y$, $\hat{y}$, and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ are the actual value, predicted value, and the mean of actual values respectively [7]. R2 ranges between a maximum value of 1.0 and minus infinity, because there is no limit to how bad a model can perform. The higher the score is, the more accurate the predictions given by the model are.

**Definition 8** (Mean Absolute Error). MAE is a simple metric that shows the average difference between the actual value and the prediction that the model produces. MAE is calculated as follows:

$$MAE(y, \hat{y}) = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$

where $y$, and $\hat{y}$ are the actual value, and predicted value respectively. MAE serves as a quick and uncomplicated way to evaluate the accuracy across multiple predictive models.

**Definition 9** (Regression). We treat the query performance prediction as a regression problem, where the models have to discover a relationship between the input and the output:

- Input (x): Graph and query to be performed
- Output (y): Prediction about query performance

A function in the form of $f(x) = y$ predicts the output value for a given set of inputs. With the help of the above-mentioned metrics, we explore suitable functions and modifications thereof to deliver the prediction with the highest accuracy.

### 2.4.3 Query feature parameterization

In comparison to traditional queries on a relational database, graph queries on knowledge graphs are more abstract and can not be easily represented using familiar metrics. For example, there is no word length or ambiguity with graph query since knowledge graphs represent a different type of data [11]. Just the query alone would also not provide enough information to be used as input for the predictive models. Therefore the query needs to be parameterized accordingly with features that are specific to graphs. With that in mind, we divide each query into three features: *(Q)uery*, *(S)ketch*, and *(A)lgorithm* [7].

**Query features** represent topological constraints on the query itself. For example, bounds on search degree, or restriction on inputs. These features are similar to those that are also found on traditional relational queries.

**Sketch features** are only found on graphs. They give an estimate of how the nodes are represented on the graph, as well as the connection to other nodes in the neighborhood, using graph statistics. By using these features, we can infer the ambiguity of the query, a feature that can not be retrieved from just the query alone. As verified later in the thesis, these features have a large contribution to the accuracy of the predictive models.

**Algorithm features**, as the name suggests, refer to the actual algorithm that is used to traverse the graph. In the case of multiple algorithms or an algorithm that has modifiable parameters, these features can further improve the accuracy.

### 2.4.4 Predictive models

We used and evaluate various machine learning models during the thesis.

**Definition 10** (Linear regression). Linear regression (LR) is a basic and simple starting point for the framework. In LR, the following equation is solved to find an approximation for the output:

$$y \approx f(x) = \hat{y}$$

$$f(x) = w_1 x_1 + w_2 x_2 + ... + w_n x_n + w_0$$

LR finds the set of coefficient $\{w_1, w_2, ..., w_n\}$ that best solve the equation between the given set of input $\{x_1, x_2, ..., x_n\}$ and output $\hat{y}$. An additional variable $w_0$ is introduced

to help offset the slight error in input data that might come from measurement. As its name suggests, LR is best fit for situations where the explored relationship is linear. With a more complicated relationship, LR might fail to provide a desirable result.

**Definition 11** (Regression trees). Our next candidate is a regression tree (RT), which is a more elaborated model than LR. It can predict continuous value, in contrast to its sibling decision tree, where input are grouped into discrete output classes. A regression tree consists of a single starting root node that splits at multiple levels with additional rules used for splitting depending on the feature of the input set. Each leaf represents the predicted value. RT brings an improvement over LR where the relationship is no longer linear.

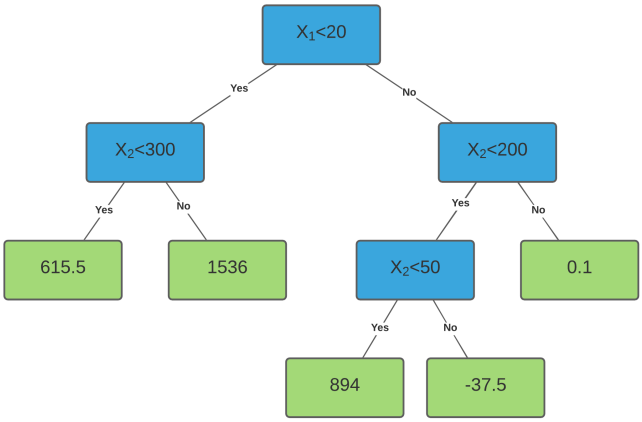


Figure 2.4: Example of a regression tree

Figure 2.4 shows an example of a very simple regression tree. The blue nodes are the input given to the model. It first checks the value of $X_1$ and branches accordingly. After that, depending on the value of $X_2$, the tree branches until it reaches a green leaf node. The value in that leaf node is the prediction of the tree.

**Definition 12** (Random forest). Random forest (RF) is an ensemble learning method that employs multiple RT where each tree is fitted with a different subset of the input data. Each individual RT in the forest makes its own prediction. The mean of those predictions is then taken as the prediction of the RF. By splitting the data set and fitting to multiple trees, this method practically eliminates the problem of overfitting the training set of RT. With a multitude of RTs, the accuracy of predictions is also greatly improved.

**Definition 13** (Multi-layer perceptrons). Multi-layer perceptrons (MLP) is a class of artificial neural networks, a model that takes inspiration from the neuron network in the human brain. An MLP consists of three layers: input, hidden, and output. Multiple nodes make up each layer and each node is activated according to the input they received following an activation function. Each node in one layer is also fully connected with the next layer with appropriate weighting. Once input is given to the model, each layer will

have its nodes activated accordingly and send the result to the next layer. At the last layer, the output layer, the activated node gives the prediction of the model [3]. Figure 2.5 shows an abstract illustration of an MLP.

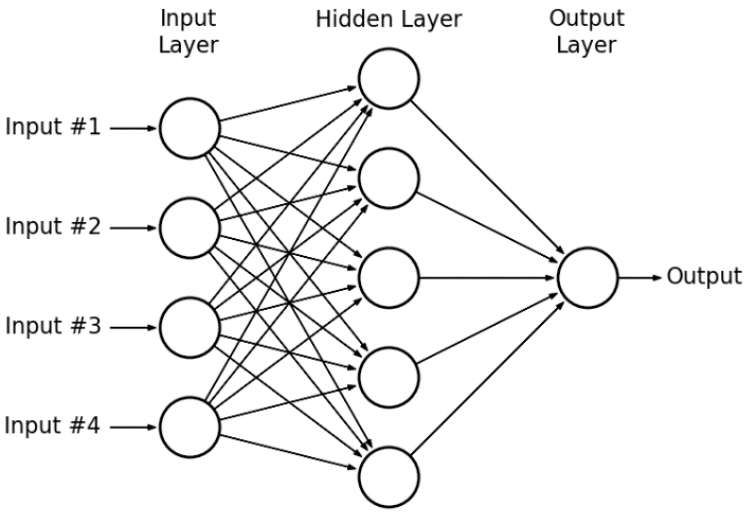


Figure 2.5: An abstract illustration of MLP [4]

In MLP, the model learns and adjusts the appropriate weighting of each perceptron. By comparing the predictions and actual values, the model dynamically adjusts the weighting to achieve the best results that minimize the error [3].

# 3 Implementation

In this chapter, we will discuss an implementation of the framework. That includes the reachability query, the machine learning model, and an application of the model for an optimization problem.

## 3.1 Reachability query

We approach the reachability problem using a well-known graph traversal algorithm: breadth-first search (BFS). BFS works by extending from a root node and visiting all of its neighbors, one layer of depth at a time. In our case, the algorithm terminates when the destination node is encountered during the expansion of BFS or when the depth bound is reached. Algorithm 1 shows a simple implementation of BFS for the reachability problem.

**Algorithm 1:** Reachability algorithm with simple BFS

**Input:** source node *src*, destination node *dst*, search depth bound *d*
**Output:** is *dst* reachable from *src* within *d* steps?

```
1  if src == dst then
2  |  return True
3  depth ← 0
4  queue, explored ← src                    // list for queue and explored nodes
5  next_depth ← empty list                  // helper to gather nodes for next depth
6  while depth < d do
7  |  depth ← depth + 1
8  |  while queue is not empty do
9  |  |  visiting_node ← pop first element of queue
10 |  |  gather neighbor set of visiting_node
11 |  |  if dst is in neighbor then
12 |  |  |  return True
13 |  |  foreach node in neighbor do
14 |  |  |  if node is not in explored then
15 |  |  |  |  add node to explored and next_depth
16 |  queue ← next_depth
17 return False
```

A node is considered a neighbor of another node, if there exists a relationship between those two. This set of neighbors is gathered in line 10 of Algorithm 1. In RDF terms, a triple exists with those two nodes acting as the subject and the object. To be able to retrieve all neighbors of a node, we construct a SPARQL query as in Listing 3.

```
SELECT ?neighbor
WHERE {
    ?startnode ?p ?neighbor.
}
```

Listing 3: SPARQL query to get neighbors of a node

However, this BFS implementation exhibits a problem with large KGs, i.e., the number of nodes to process. The average number of nodes to visit in BFS is $b^n$, where $b$ is the branching factor of the graph and $n$ is the search depth. In a large KG such as DBPedia, there are estimated millions of entities presented with many relations between them. Therefore, the number of nodes to be visited grows exponentially with increasing depth. For example, if a node has on average 150 neighbors, at the depth of 3, BFS has to process $150^3$ nodes, which is almost 3.5M nodes. Therefore, the simple BFS has to be suitably modified to adapt to the new application in large KGs.

With the reachability problem, instead of just knowing one starting node and branching out from it, we also know what the destination node is. That is a property that we can exploit to reduce the load for BFS: instead of branching out from one direction, we can instead search from both directions. After the depth limit has been reached, we can check both sets of visited nodes for common elements. If such an element exists, that means we can go from the source to that node, and from that node to the destination, which in turn means that there is at least one path from the source to the destination nodes. Algorithm 2 illustrates an implementation of this modified BFS, now bi-directional BFS.

**Algorithm 2:** Reachability algorithm with bi-directional BFS

**Input:** source node *src*, destination node *dst*, search depth bound $d$
**Output:** is *dst* reachable from *src* within $d$ steps?

```
if src == dst then
    return True
depth_forward, depth_backward ← 0
while depth_forward + depth_backward < d do
    if queue_forward ≤ queue_backward then
        depth_forward ← depth_forward + 1
        forward BFS from src
    else
        depth_backward ← depth_backward + 1
        backward BFS from dst
check for common element from two visited sets, terminate if found
```

The forward search is identical to normal BFS, but a small modification needs to be made for the backward search. Since KGs are directed graphs, meaning relationships only apply one-way, instead of searching for nodes where the starting node is the subject,

the starting node now has to be the object in the triple. Accordingly, Listing 4 shows the modified SPARQL query.

```
SELECT ?neighbor
WHERE {
    ?neighbor ?p ?startnode.
}
```

Listing 4: Modified SPARQL query to get backward neighbor

By branching out from both directions, we do two BFSs with the depth of $\frac{d}{2}$ each instead of one BFS with the depth of $d$. The average number of nodes to process is now $2b^{\frac{d}{2}}$ instead of $b^d$. With the example above, only around 50K nodes have to be visited instead of 3.5M. This significantly reduces the number of nodes that have to be processed while still delivering the desired result.

## 3.2 Predictive model

The next part of the implementation is the predictive model. As discussed earlier, we explore several methods of unsupervised learning to solve our problem of predicting query runtime. Without additional data preprocessing, the only information we can use as input to the model is the name of the source and destination nodes, as well as the bound of the query. That information alone is not enough for the model to make a meaningful or accurate prediction for our purpose. However, as a unique feature of graphs, each node in a graph can be parameterized into statistical information that better describes them.

The hop bound d itself is already a numerical value, which is already suitable for the model. We categorize the hop bound as a topological feature of the query. The task now is to parameterize the source and destination node into numerical values which additionally will provide more information about them. In a KG, each node has many relationships to other nodes, the number of neighbors that each node has will in turn tell us how popular each node is. Additionally, since KGs are directed graphs, we can further split those relationships into two types: in- and out-degree. Instead of representing a node by its name, now it can be represented in terms of numbers of in- and outcoming relationships with other neighboring nodes. These two parameters are classified as the sketch category of query feature. Listing 5 shows the required SPARQL query to gather additional features of a node.

This counts all of the nodes that have any arbitrary incoming relationship, i.e, any predicate $p$ in the RDF triple, with the source node. Analogously, the same can be done to count out-degree, only the position of the source and neighbor variables needs to be swapped.

```
SELECT count(?neighbor) as ?indegree
WHERE {
    ?source ?p ?neighbor.
}
```

Listing 5: SPARQL query to gather in-degree of a node

With the method to suitably convert input data established, they can be used to feed into the machine learning models to produce predictions. The workflow of the predictive model is as follows:

1. Generate random query as input data set to use as training. The initial set consists of three parts of information: hop bound, source, and destination nodes.
2. Convert the source and destination nodes respectively into in- and out-degree.
3. Train the model using the preprocessed dataset.
4. Make predictions over newly generated datasets.

As discussed in Section 2.4.4, we deploy several predictive models and compare their accuracy and performance. The data preprocessing and model implementation (LR, RT, RF, and MLP) are all done within Python. With the exception of MLP, all models are implemented with *scikit-learn*. In respect to the order above, the following classes in *scikit-learn* are used to implement the models: *LinearRegression*, *DecisionTreeRegressor*, and *RandomForestRegressor*. For MLP, we utilized the library *TensorFlow* to help create our own model.

*scikit-learn* is a popular machine learning library for Python. It contains many implementations for various machine learning models. Each implementation also provides many different hyperparameters to work with and to be changed to the need of the user.

*TensorFlow* is also a popular library for machine learning and artificial intelligence. *TensorFlow* focuses on deep learning with neural models. In contrast, *scikit-learn* focuses more on traditional machine learning algorithms. With *TensorFlow*, we can build our model from scratch, tuning and optimizing almost all aspects of the model to fit our needs. However, more freedom in implementation also means more complications in using and generating an optimized model.

As the most simple model, LR does not have many hyperparameters to change. Most of them are to help with better fitting the data to the model, as well as additional preprocessing for the dataset. Since the data preprocessing has already been done, additional help from those hyperparameters is not needed. We use all default settings.

RT provides many hyperparameters that are useful for fine-tuning the tree. Multiple algorithms can be used to decide where to split, as well as to measure the quality of each split. The depth of the tree can also be determined. However, for our case, changing those parameters only resulted in differences within the margin of error, if not worse. Listing 6 shows the initialization and data fitting of LR and RT.

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
LR = LinearRegression() #default hyperparameters
LR.fit(X_train, y_train) #fit model to training data
RT = DecisionTreeRegressor() #default hyperparameters
RT.fit(X_train, y_train) #fit model to training data
```

Listing 6: Initializing and fitting LR and RT

As RF is essentially a collection of many RT, therefore the hyperparameters are similar. We test the effectiveness of the ensemble method by varying the number of trees used in the ensemble. We vary the number of trees between 2 and 100, as shown in Listing 7.

```
from sklearn.ensemble import RandomForestRegressor
RF = RandomForestRegressor(n_estimators=2) #n_estimators ranges from 2 to 100
RF.fit(X_train, y_train) #fit model to training data
```

Listing 7: Initializing and fitting RF

With MLP, since we now build the model from the ground up, we can choose and change its configuration however we want. There exist many options to choose from, e.g., the solver, loss function, or the activation function. Tuning the size of the hidden layers is a manual process as there is no concrete way to set it for each situation. Not only the number of layers but also the number of perceptrons within each layer can be tuned accordingly. Through a process of trial and error, we settled on a model with 4 hidden layers with 100 neutrons each that get activated with the *relu* function. The loss function is *huber*, with the *adam* optimizer, and the verification metric *MAE* and *MSE*. The model is trained over 100 epochs. The sampled code is given in Listing 8.

```
from tensorflow.keras.layers import Input, Dense
from keras.models import Sequential
model = Sequential()
model.add(Input(shape=(X.shape[1],))) #input
for i in range(4): #4 hidden layers
    model.add(Dense(100, activation='relu')) #100 perceptrons in each layer
model.add(Dense(1)) #output
#hyperparameters tuning
model.compile(loss='huber', optimizer='adam', metrics=["MAE","MSE"])
model.fit(X_train, y_train, epochs=100, validation_split=0.2, verbose=0)
```

Listing 8: Building MLP and tuning its hyperparameters

After the predictive models have been implemented and trained, they can be serialized using Python built-in *pickle* module. Later, the workload optimizer can invoke an appropriate model to create predictions without training and verifying them again.

## 3.3 Workload optimization

Now that a suitable model has been implemented, we can utilize its predictions to help with optimizing resources allocation. One use case that is closely related to real-world optimization is query selection for servers. More specifically, the problem is to choose and execute queries from a list of incoming queries such that the highest profit is gained while we stay within the bounds of available resources. Therefore, a workload optimizer using the predictive models is implemented to tackle the optimization problem in resource-bound querying scenarios [1].

The problem is as follows [7]: Given a series of queries $\mathcal{W} = \{Q_1, Q_2, ..., Q_n\}$, where each query $Q_i$ has a given profit $p_i$. The series $\mathcal{W}$ is to be executed over graph $G$ within a time bound $T$. The goal is to find a subset $\mathcal{W}'$ of $\mathcal{W}$ such that the time to execute $\mathcal{W}$ is bounded by $T$, and the total profit gained $\sum_{Q_i \in \mathcal{W}'} p_i$ is maximized. In this scenario, profit can be an arbitrary metric such as the priority or importance of queries.

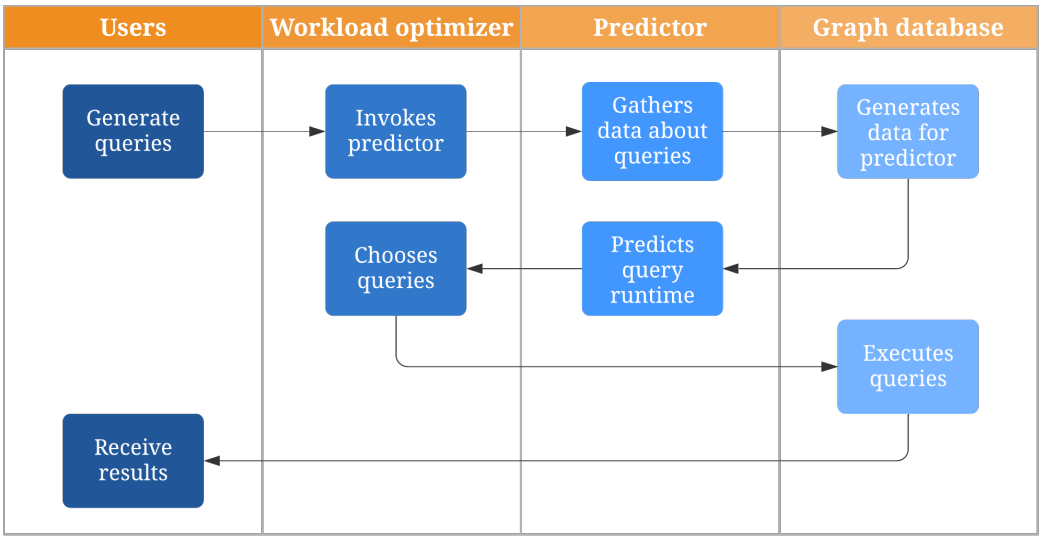


Figure 3.1: Workflow for the optimizer

Figure 3.1 shows a general workflow for the optimizer. While using the database, the users generate a set of queries $\mathcal{W}$, where each query in the set has a profit $p_i$, to be performed. The optimizer receives this set and invokes the predictor to predict the runtime $t_i$ of each query in $\mathcal{W}$. With the predictions ready, the optimizer tries to greedily solve the knapsack problem: with $T$ as the size of the knapsack, $t_i$ as the item weight,

and $p_i$ as the item value, the optimizer greedily chooses the next queries to be executed until the bound $T$ is reached. The chosen subset $\mathcal{W}'$ of $\mathcal{W}$ is then performed on the database and the users receive the query results.

Additionally, to further verify the result of the workload optimizer, we implement and compare different optimizing strategies. They represent different scenarios where databases have different optimizers in place.

**Opt_RF**. This optimizer uses the RF predictive model to predict and choose queries to be executed following the workflow above. Listing 9 shows our implementation of **Opt_RF**.

**Opt_Rnd**. A baseline optimizer, it will randomly choose a query from the set $\mathcal{W}$ and execute it until the bound $T$ is reached. This is a situation where there is no optimizer in place, and the database run the queries on a first come, first serve basis.

**Opt_True**. The counterpart of **Opt_RF**, where instead of using prediction, it uses the actual runtime of the query. This simulates an ideal scenario where the runtimes of the queries are known beforehand and we can use that information to choose the appropriate queries to be executed. **Opt_True** is used to verify the effectiveness of **Opt_RF** and **Opt_Rnd**.

```
1   import pandas as pd
2
3   def Opt_RF(query, prediction_model, bounded_time):
4       query_set = pd.read_csv(query)              #set of queries to be executed
5       model = pd.read_pickle(prediction_model)    #prediction model
6
7       #predict the runtime of the received set of queries
8       query_set['runtime_prediction'] = model.predict(query_set.filter(items=
9                                         ['in_src', 'out_src', 'in_dst',
10                                        'out_dst', 'bound'], axis=1))
11
12      #calculate profit to runtime ratio of each query
13      for index, row in query_set.iterrows():
14          query_set.at[index, 'ratio']  = row['profit']/row['runtime_prediction']
15
16      #sort ratio and update index
17      query_set.sort_values(by=['ratio'], ascending=False, inplace=True)
18      query_set = query_set.reset_index(drop=True)
19
20      #Knapsack solver (based on predicted runtime)
21      actual_time = 0
22      profit = 0
23      for index, row in query_set.iterrows():
24          actual_time += row['runtime_prediction']
25          profit += row['profit']
26          if actual_time > bounded_time:
27              break
28
29      return(profit)
```

Listing 9: Implementation of **Opt_RF**

# 4 Experiments

This chapter presents the results of the experiments done to verify the framework. We also show the process of data gathering and the observations as well as anomalies that have been encountered during the experiments.

## 4.1 Data source

As discussed in the earlier part of this thesis, we planned to implement and test the framework on the database of DBPedia. As an open project, DBPedia provides a SPARQL endpoint, available at `https://dbpedia.org/sparql/`, for the public to query their KG. We deployed our implementation of the framework and conducted an initial test at the public endpoint as the first step.

The Python library *RDFLib* and its extension *SPARQLWrapper* provide means to query a SPARQL endpoint. Therefore, we can use Python to send our SPARQL queries over to DBPedia's public endpoint. A sample of the Python code used to query the database to get the ID of an article is given in Listing 10.

```
1   from SPARQLWrapper import SPARQLWrapper, CSV
2
3   sparql = SPARQLWrapper('https://dbpedia.org/sparql')
4   sparql.setQuery(f'''
5       SELECT ?id
6       WHERE {{
7       <{article}> dbo:wikiPageID ?id
8       }}
9   ''') #get the id of the given article
10
11  sparql.setReturnFormat(CSV) #set format to CSV
12  qres = sparql.query().convert().decode('u8')
```

Listing 10: Sample code to send SPARQL query to endpoint

However, as a public service, the endpoint has some restrictions in place to ensure a fair use policy is observed. For example, the number of requests that can be made is restricted to 100 queries per second. With those restrictions in mind, we tried several reachability queries on DBPedia's endpoint and observed that a single request takes around 110ms to complete on average. The average round-trip ping to DBPedia's server from our network is 10ms, which also adds to the runtime of the request.

Each reachability query usually has to visit and retrieve information on estimated hundreds if not thousands of nodes. Even if a single request only takes a fraction of a second, the required waiting times quickly add up at this scale. One reachability query can take up to several minutes to complete.

A possible explanation for the lower performance on DBPedia's endpoint, aside from the latency to their server and the query limitations, is the large size of their internal database. The complete DBPedia database consists of over 21B RDF triples [5], which describe many elaborated relationships between entities. However, for the reachability query, these extra pieces of information do not bring any meaningful improvement to the query quality. Therefore, a smaller, curated database is better suited than a complete database. DBPedia frequently releases the core part of its database every quarter. These are smaller subsets of the complete database that contains factual data from articles and infoboxes of Wikipedia.

With our current implementation, we consider a KG that has the following specification: each English Wikipedia article is a node; if an article has a link to another article, they are connected in the graph with the respective direction, represented by the *WikiPageWikiLink* relationship between two articles; each article has a unique ID, represented by the *WikiPageID* relationship between the article name and its ID number.

From the core release of DBPedia, version 2021-09, we downloaded the following subsets: *ids*, *wikilinks*. They come in the *Turtle* file format (*.ttl), which is a plain text file that contains just one RDF triple in each line of text. They can be directly parsed and processed by *RDFLib* in Python. However, *RDFLib* is inherently unable to handle large RDF files directly. The extracted dataset is a text file at over 30GB in size, which is too large for *RDFLib* to parse and save as a graph in memory. A different, more efficient approach is needed to query those data efficiently.

The next step is to set up our own SPARQL endpoint, hosted locally on our machine that utilizes the downloaded dataset. The software that we use to deploy our local endpoint is OpenLink Virtuoso. OpenLink Virtuoso is an open-source database engine with a wide range of functionality that combines traditional relational databases with other types of databases, in our case graph databases. For their public endpoint, DBPedia also uses a version of Virtuoso, which makes the process of switching to a local endpoint seamless and the result comparable. OpenLink Virtuoso provides many useful database-related functions. However, for our purpose, we are only interested in the triplestore and the endpoint deployment ability.

The process of deploying a local endpoint is straightforward. After OpenLink Virtuoso is installed, the downloaded dataset above is imported into its triplestore. The resulting database has 262M triples in total, which comprises a KG with 242M edges and 19M vertices. Some parameters of the server are also tuned to utilize more of the available computing resource for faster querying. Now that a local endpoint is deployed, we no longer have to follow the query limitations at the public endpoint. While our hardware might not be comparable to DBPedia's server, performance has increased significantly with a more curated database and local access. We set the return format to CSV to further increase the performance, which returns a single CSV stream containing our result. Unlike other formats such as JSON or XML, CSV only keeps the minimum formatting for the results needed for our query. This further lowers the processing time by a few milliseconds. As a result, we received a significant improvement in query execution time. A single SPARQL query only takes on average 4ms to return a result.

## 4.2 Experiment results

With the local endpoint set up and ready to operate, we can begin our experiment to demonstrate the ability of the framework. Our first step is to gather a large number of queries to be used for model training. For the reachability query $Q(G, s, t, d)$, we sampled 5K pairs of nodes with an out-degree of at least 40, i.e., articles that have at least 40 links to another article. We also set a random $d$ for each query.

For the algorithms, we implemented the bi-directional BFS as well as the query workload optimizer as discussed in Chapter 3 in Python.

We also implemented the predictive models in Chapter 3, with the hyperparameters tuned as discussed, also in Python. To verify and compare the effectiveness of different models, we used the metrics R2 and MAE.

Each experiment was performed multiple times, and the average result was taken. We conducted the experiments on a machine with the following specification:

- CPU: AMD Ryzen 3600X @ 4.2 GHz
- Memory: 16GB DDR4 @ 3600 MHz
- OS: Windows 10 version 21H1
- Virtuoso version 07.20.3233, Python version 3.8.5

### 4.2.1 Performance of predictive models

**Query runtime and distribution**. The first set of results is the runtime of the sampled reachability queries. As Table 4.1 illustrates, we have a mix of short, average, and long-running queries. The average runtime of a reachability query is 380ms, which is a considerable improvement over using the public endpoint, where one such query can take minutes to return a result. Figure 4.1 shows the runtime distribution of those queries. As expected, generally the higher the bound is, the longer it takes a query to execute. While there are some outliers, the results lie within our expectations.

| Total 5K queries | Min | Average | Max |
|---|---|---|---|
| Runtime (ms) | 3 | 380 | 21,175 |

Table 4.1: Response time of query samples

**R-Squared Accuracy**. RF has the highest performance in comparison to other models. Figure 4.2 shows R2 scores ranging from as low as 0.29 for LR to as high as 0.96 for RF. MLP comes in a close second with 0.95 and RT comes third with 0.91. RF achieved the highest accuracy thanks to its ensemble methods that make it compatible with a wide range of input values. MLP, while also coming close to RF, needs many of its hyperparameters to be tuned to get to this result. For a quick and simple implementation, RF is the superior choice here. LR has the lowest score because it works best with input

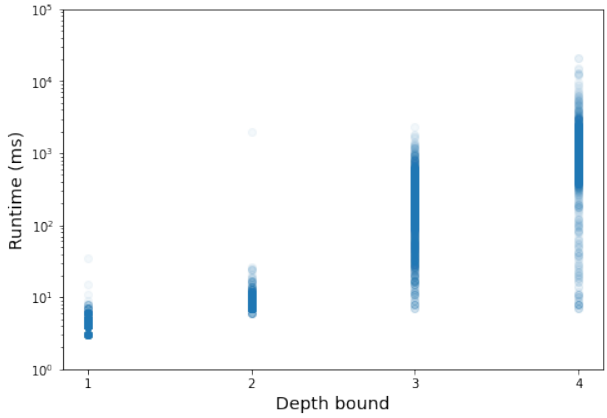


Figure 4.1: Distribution of query runtime by bound

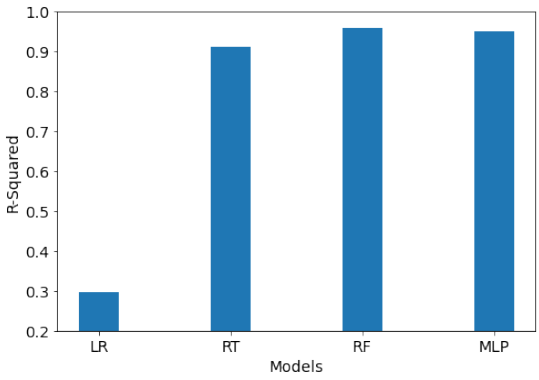


Figure 4.2: R-Squared Accuracy

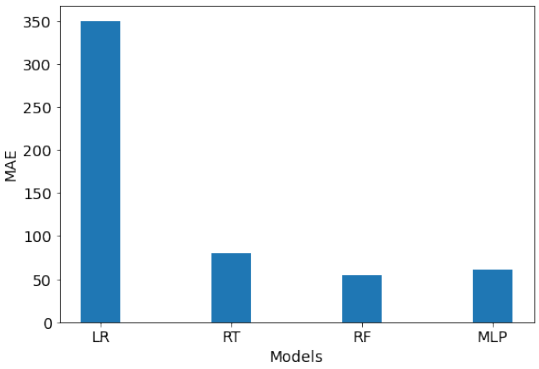


Figure 4.3: Mean Absolute Error

with high linearity. In our experiment, the independent variables that are fed into the model show a high degree of non-linearity.

**Alternative metric**. With the alternative metric MAE as shown in Figure 4.3, aside from LR, all models show a difference between real and predicted runtime of under 80ms. With a high variance in query runtime as shown in Table 4.1, we can see that the models are robust against different metrics [7].

**Models training and verifying time**. Different models need different times to fit the training data. Figure 4.4 shows the time it takes to fit and train each type of model. In this experiment, the set of 5K queries sampled as above is split into two sets: 4K queries for training, and 1K for verifying. Data from the first set is fit into the model, the remaining queries are used to verify the prediction of the model. Except for MLP, all models need just under one second to train and verify themself. With RF, we take a look at the two extremes of the number of RT: 2 and 100. Even with 100 trees to fit and train, it only takes an additional 0.3 seconds. Therefore, we keep the default setting of 100 trees for the later part of the experiment. This further shows that RF is our best

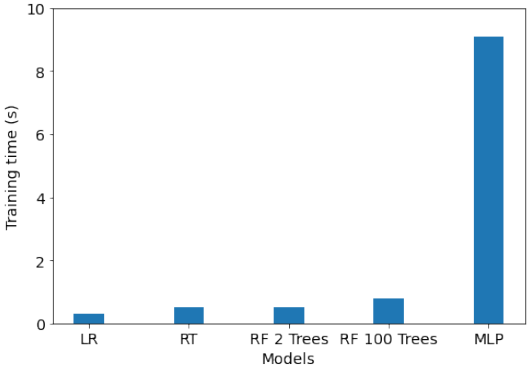


Figure 4.4: Models training time

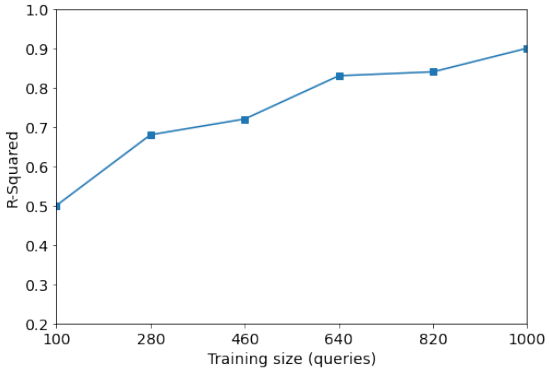


Figure 4.5: Training size

model.

**Training size**. With the same dataset as above, we varied the number of queries used to train the model from 100 to 1000, as shown in Figure 4.5. RF is used for this experiment. As expected, the accuracy of the model rises with more instances of training queries, from 0.5 at 100 queries to 0.9 at 1000. Furthermore, the model does not need a large set of training queries to achieve high accuracy. The R2 score reaches 0.8 at only around 600 instances.

| Feature | Type | RT (%) | RF (%) |
|---|---|---|---|
| in-degree of src | S | 0.29 | 0.69 |
| out-degree of src | S | 8.33 | 6.04 |
| in-degree of dst | S | 61.09 | 61.55 |
| out-degree of dst | S | 0.31 | 1.21 |
| hop bound d | Q | 29.98 | 30.51 |

Table 4.2: Relative feature importance

**Feature importance**. Table 4.2 shows the evaluation of the relative feature importance by RT and RF. Because LR only tries to find the coefficient to solve the equation between input and output, it does not have a metric for feature importance. The most important feature is the in-degree of the destination node. Since we already sampled nodes with an out-degree of at least 40, the in-degree feature is an important metric to show how popular a node is. Additionally, we expand forward from source and backward from destination; therefore, the in-degree of the source node and out-degree of the destination node are not highly relevant. The hop bound d is the second most important because a higher bound means more nodes to explore.

### 4.2.2 Workload optimization

In this section, we apply the predictive model to aid in a real-world application. The model of choice is RF, and accordingly we implemented the workload optimizer **Opt_RF**. The effectiveness of **Opt_RF** is calculated as follows:

$$\frac{\sum_{Q_i \in \mathcal{W}} p_i}{\sum_{Q_i \in \mathcal{W}'} p'_i}$$

where $\mathcal{W}$ (and $\mathcal{W}'$) is the query choice of **Opt_RF** (and **Opt_True**) [7]. It shows the ratio of profit gained by **Opt_RF** to the total profit gained by **Opt_True**. We calculate the effectiveness of **Opt_Rnd** similarly.

To simulate the workload for the optimizer, 4K queries were sampled in the same way as the model training. Additionally, we assigned a profit to each sampled query using the normal distribution $\mathcal{N}(\mu, \sigma^2)$. These queries were sent to the optimizer together with a time bound to choose the best queries to be executed.

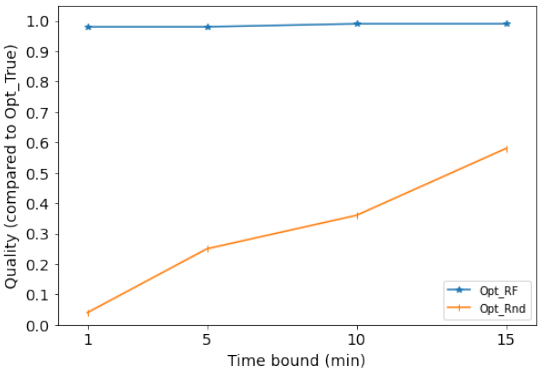


Figure 4.6: Optimizer quality (fixed profit)

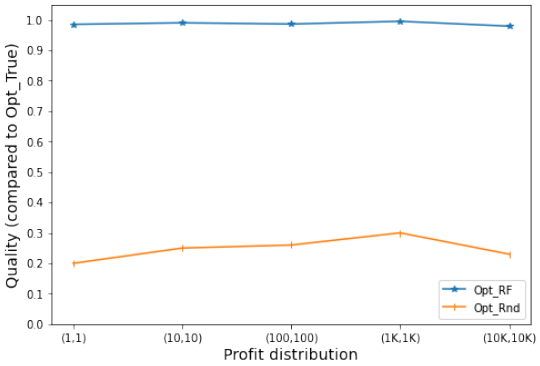


Figure 4.7: Optimizer quality (fixed time)

In the first experiment, the profit of each query is fixed while the time bound is varied from 1 to 15 minutes, as shown in Figure 4.6. We see that **Opt_RF** keeps a very high performance, always more than 98% of **Opt_True**. While at the same time, **Opt_Rnd** can only gain more profit with a more relaxed bound. **Opt_Rnd** failed to gain a reasonable profit from the given queries at a lower bound.

For the second experiment, the time bound is fixed. We then change the profit distribution, $\mathcal{N}(\mu, \sigma^2)$ from (1,1) to (10K,10K), as shown in Figure 4.7. Again, **Opt_RF** shows a high percentage of profit gained while **Opt_Rnd** fluctuates up and down.

After those two experiments, we can confirm that **Opt_RF** is remarkably robust against fluctuation in both time bound and query profit. The performance of **Opt_RF** is almost the same in every experiment setting. As shown in Section 4.2.1, the time to train the predictive model and make predictions is just under one second. Therefore there is no need to fall back to the **Opt_Rnd** strategy of choosing random queries for execution. Both **Opt_RF** and **Opt_True** can only deliver an approximation of the best solution since the Knapsack problem falls into the class of NP-Complete problems.

## 4.3 Observation and anomaly

After running the experiments, we find that our initial results are in line with what the original paper presented. However, after further inspection, we found that our implementation of BFS was not exactly optimized to deliver the fastest result. Currently, our BFS acts as follows: it first retrieves all of the neighboring nodes of both the source and destination until the bound d is reached; only after that does it check for common elements of both sets to return a result. The bound d may be set to a higher value than the actual distance of source and destination pair, for example d = 4 when source and destination are direct neighbors. In this case, our previous algorithm has to retrieve all of the neighboring nodes until the depth of 4, then it checks if both sets have common elements. While ideally it only has to retrieve neighbors of degree 1 to see that source and destination are directly connected.

To avoid unnecessary workload, we modified the algorithm to check for common elements after each time it retrieves the next depth of neighbors. This is illustrated in Algorithm 3. Note that the common element check is now inside the while loop.

**Algorithm 3:** Optimized reachability algorithm

**Input:** source node *src*, destination node *dst*, search depth bound *d*
**Output:** is *dst* reachable from *src* within *d* steps?

```
if src == dst then
    return True
depth_forward, depth_backward ← 0
while depth_forward + depth_backward < d do
    if queue_forward ≤ queue_backward then
        depth_forward ← depth_forward + 1
        forward BFS from src
    else
        depth_backward ← depth_backward + 1
        backward BFS from dst
    check for common element from two visited sets, terminate if found
```

With the optimization in place, the average runtime of the reachability query went down to just 261ms, as shown in Table 4.3. However, when we applied the new optimized variant of BFS to gather data to train the predictive models, we failed to reach the same accuracy as before. Even with RF, our best candidate, the R2 score only hovered around 0.4, less than half of what it used to achieve.

| Total 6K queries | Min | Average | Max |
|---|---|---|---|
| Runtime (ms) | 3 | 261 | 10,551 |

Table 4.3: Response time of query samples with optimized algorithm

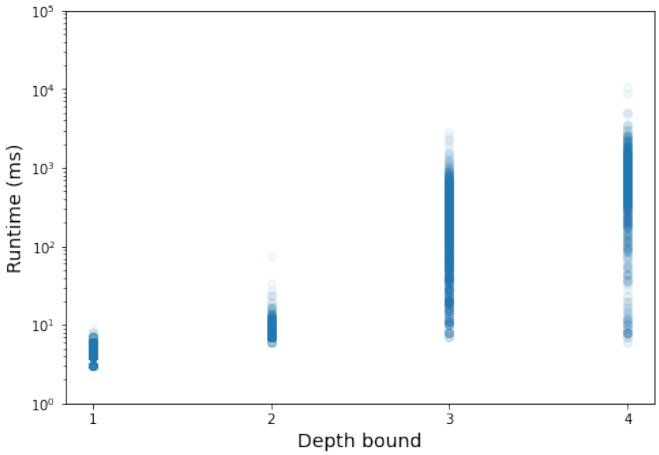


Figure 4.8: Distribution of query runtime by bound with optimized algorithm

One possible explanation for this anomaly is that the search bound, which is one of the important inputs for the models, no longer scales with the cost of the query. Earlier it was straightforward, that the amount of work the query needs to do rises with a higher search bound. However, now with the new algorithm, the predictive model still obtains the same bound as the input, but in reality, the query might end much sooner than expected. As with the example above, a query with a bound of 4 might have the runtime of one with a bound of just 1. This introduces much more noise into the predictive models, making it more difficult to achieve high accuracy for their prediction. Figure 4.8 illustrates the problem we encountered. The distribution of query runtime is much tighter than before.

We again collect queries with the new algorithm to use as input for the models to verify our prognosis. However, we use the search depth when the algorithm terminates as input instead of using the bound. As expected, the predictive models regain their high accuracy as before. This confirms our hypothesis of why the models have such a low accuracy with the new algorithm. Nevertheless, to collect the search depth of a query when it terminates, the query needs to be executed beforehand. With our goal of predicting the runtime of a given query, the need to run the actual query ahead to collect vital information defeats the purpose of the prediction in the first place.

With the root of the problem identified, we now try to implement a solution to help eliminate the issue. Since the models regain their high accuracy score when the search depth is used instead of the bound, we try to predict the distance of the node pair and use that prediction instead of the bound. In BFS, the depth when the algorithm terminates is the distance between the two nodes. For this task, we again employ machine learning. However, instead of regression, the machine will need to solve a classification problem.

The model of choice this time is again a random forest for its versatility and high accuracy. The class *RandomForestClassifier* from *scikit-learn* is used, as shown in Listing 11. Its working principle is similar to *RandomForestRegressor*, but instead of predicting continuous values, it now classifies inputs into several predefined groups.

```
from sklearn.ensemble import RandomForestClassifier

RF = RandomForestClassifier() #default hyperparameters
RF.fit(X_train, y_train) #fit model to training data
```

Listing 11: Initialization and fitting of classification model

We again utilized the same inputs as in Section 3.2 for the model: in- and out-degree of source and destination node. This saves us the work of calculating extra information for the new model. The output this time is the distance between the nodes. Again, 5K random pairs are sampled and their distance is calculated using BFS. The accuracy of this new model is evaluated using the built-in *accuracy_score* metric of scikit-learn. This simple metric calculates the fraction of correct predictions made by the model. With $y_i$ as the real value and $\hat{y}_i$ as the prediction:

$$accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$

After training the classification model, we received an accuracy score of around 65%. This means more than half of the predictions made by the model are correct. The base workflow is the same as in Section 3.2, an additional step is added however. The classification model will now predict the node distance, these predictions are then used as input for the regression model to predict query runtime in place of the query bound. With the new improvement, the R2 score of the regression model rises to 0.85. This is still not as high as the prediction with the unoptimized BFS, but a huge improvement in comparison to the 0.4 of before.

# 5 Conclusion and outlook

The main goal of the thesis was to analyze and predict the performance of graph queries and apply the prediction to help with real-world scenarios of optimizing workload. Namaki et al. have developed an innovative framework [7], and one implementation of it was discussed in detail in this thesis.

Graph query, particularly reachability query, has been studied extensively to achieve this goal. We have shown that with clever exploitation of efficiently calculable features of graph query - the sketches of each node in the graph - the machine learning models can predict the performance of those queries with high accuracy. These models can be trained and make predictions in just under a second. Furthermore, they require very little training data to reach a satisfactory predictive result.

A workload optimizer has also been implemented when applying the models to aid with resource allocation and optimization. Experiments have shown that the optimizer could deliver an excellent result, even when pitched against an optimizer with actual knowledge of query runtime, which is unobtainable before query execution. Further experiments also proved the robustness of the optimizer under different time bounds or highly varied profit distribution.

While we encountered anomalies during the implementation of the framework, a possible solution has been proposed to help the models regain their accuracy. Together with the newly optimized reachability algorithm that reduces the runtime considerably, the new model would undoubtedly significantly improve query performance and resource allocation for database servers.

Knowledge graphs are steadily rising as the new standard in the generation of Web 4.0. With the internet evolving more extensively than ever before, more and more graph databases with massive KGs are becoming the norm. Nowadays, web users are also becoming ever more abundant and have stricter requirements for the quality of service. The optimization of both query performance and resource allocation is a crucial task for such databases. Our study has shown the effectiveness of the predictors and optimizers with real-world data. The framework is therefore ready to bring significant improvement in such application.

We have discussed an implementation of the framework for *reachability* queries, which is just one class of graph queries. Different query classes such as *top-k* or *dual-simulation* queries are still left open. Furthermore, additional exploitation of query and graph features, such as log files or user interactions, can improve the predictive models and indirectly the optimizer. The original paper also mentioned a plan to develop the framework further to support ad-hoc online learning without offline training on a predetermined dataset.

Additionally, the anomalies encountered during our implementation can be further investigated. We have proposed a workaround for the problem; however, additional improvements are always possible. Further exploitations of graph and query features can be used to better predict the distance. Our implementation and the dataset gathered

for the thesis are readily available online via GitHub[1].

[1] https://github.com/mentatss/thesis_bachelor

# Bibliography

[1] Wenfei Fan, Xin Wang and Yinghui Wu. "Querying Big Graphs within Bounded Resources". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 301–312. ISBN: 9781450323765. DOI: 10.1145/2588555.2610513.

[2] David Ferrucci et al. "Building Watson: An Overview of the DeepQA Project". In: *AI Magazine* 31.3 (July 2010), pp. 59–79. DOI: 10.1609/aimag.v31i3.2303.

[3] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow: Concepts, Tools, and techniques to Build Intelligent Systems*. 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Sept. 2019. 819 pp. ISBN: 978-1-492-03264-9.

[4] Hassan Hassan et al. "ASSESSMENT OF ARTIFICIAL NEURAL NETWORK FOR BATHYMETRY ESTIMATION USING HIGH RESOLUTION SATELLITE IMAGERY IN SHALLOW LAKES: CASE STUDY EL BURULLUS LAKE." In: *International Water Technology Journal* 5.4 (Dec. 2015). URL: https://www.researchgate.net/publication/303875065 (visited on 19/01/2022).

[5] Marvin Hofer et al. "The New DBpedia Release Cycle: Increasing Agility and Efficiency in Knowledge Extraction Workflows". In: *Semantic Systems. In the Era of Knowledge Graphs*. Ed. by Eva Blomqvist et al. Cham: Springer International Publishing, 2020, pp. 1–18. ISBN: 978-3-030-59833-4. DOI: 10.1007/978-3-030-59833-4_1.

[6] Mehryar Mohri, Afshin Rostamizadeh and Ameet Talwalkar. *Foundations of Machine Learning*. 2nd ed. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2018. 504 pp. ISBN: 978-0-262-03940-6.

[7] Mohammad Hossein Namaki et al. "Performance Prediction for Graph Queries". In: *Proceedings of the 2nd International Workshop on Network Data Analytics*. NDA'17. Chicago, IL, USA: Association for Computing Machinery, 2017. ISBN: 9781450349901. DOI: 10.1145/3068943.3068947.

[8] Jorge Pérez, Marcelo Arenas and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009). ISSN: 0362-5915. DOI: 10.1145/1567274.1567278.

[9] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. W3C, Jan. 2008. URL: https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ (visited on 19/01/2022).

[10] Stuart J. Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. London, UK: Pearson, 2020. 1136 pp. ISBN: 0-13-461099-7.

[11] Keyvan Sasani et al. “Multi-metric Graph Query Performance Prediction”. In: *Database Systems for Advanced Applications*. Ed. by Jian Pei et al. Cham: Springer International Publishing, 2018, pp. 289–306. ISBN: 978-3-319-91452-7. DOI: 10.1007/978-3-319-91452-7_19.