Exploring Graph Traversal Algorithms for Knowledge Graphs

Bachelor Thesis

by

Sven Bulach

Degree Course: Industrial Engineering and Management B.Sc.
SPO 2015

Institute of Applied Informatics and Formal Description Methods (AIFB)

KIT Department of Economics and Management

Advisor:	Prof. Dr. Harald Sack
Second Advisor:	Prof. Dr. J. Marius Zöllner
Supervisor:	M.Sc. Russa Biswas
Submitted:	February 20, 2021

Abstract

Knowledge Graphs (KGs) are semantic databases where entities that represent real world objects are related to each other. The entities are provided with attributes and put into thematic context or ontologies. Most KGs follow the RDF standard which makes the knowledge that is stored in such graph machine-understandable. Since KGs like DBpedia pursue the goal of collecting as much information as possible, they take on gigantic dimensions. The KG that is used for this thesis is DBpedia.

One important aspect of graph theory is graph traversal which is generally very well studied, but not specifically for Knowledge Graphs. In this thesis, the two algorithms Dijkstra and A-star are used to find paths between two nodes in a KG. They are compared in terms of path length and search time.

For the Dijkstra to work, the examined graph will be weighted based on the degrees of nodes. In order to apply the A-star algorithm, two heuristics are introduced which use the `rdf:type` information given by DBpedia. The `rdf:type` relation assigns entities that share common characteristics into a corresponding type.

Finally, a feedforward neural network is used to predict the shortest distance between two nodes of a Facebook graph. Knowing the shortest distance between two nodes can be useful information that can be exploited for instance by the A-star algorithm.

Contents

List of Figures

List of Tables

1 Introduction

Knowledge Graphs (KGs) are semantic databases. Here, entities are related to each other, provided with attributes and put into thematic context or ontologies.The initial idea of knowledge graphs was to use graphs, a discrete mathematical concept, as a representation of the contents of medical and sociological texts. Accumulation of the knowledge was to be carried out by constructing larger and larger graphs in such a way that the resulting structure could function as an expert system. Some of today's most popular Knowledge Graphs are for example DBpedia, Freebase, Yago or Googles Knowledge Graph. For instance DBpedia gathers the information from Wikipedia and represents it in a structured way to make it machine-understandable. An important aspect of interacting with such graphs, is the ability to query the connectivity of nodes. Graph traversal algorithms are mechanisms to traverse edges and vertices in a graph in a systematic way.

1.1 Motivation and Background

Since one of the goals of common Knowledge Graphs is to gather as much information as possible, they are now taking on gigantic dimensions. Figure 1 gives an overview of the KGs mentioned earlier. It can be seen, that the number of entities, which are represented as nodes in the KG, goes up to 39 million in the case of Freebase [1]. Graph traversal becomes a very time consuming task with larger graphs and it is therefore needed to find efficient algorithms to solve this issue. To know the shortest distance between two nodes in a graph can be very helpful for graph traversal algorithms. For instance the A-star algorithm selects nodes based on their approximate distance to the target node. Graph walk is one of the fundamentals of graph traversal algorithms. A walk is a sequence of vertices and edges of a graph, with repeated nodes and edges. The walk is a trail if any edge is traversed at most once. A trail is a path if any vertex is visited at most once, except possibly the initial and terminal vertices when they are the same. This is a well-studied problem in the field of Graph Theory [18]. This thesis focuses on exploring different graph traversal algorithms in the pretext of Knowledge Graphs combined with a method for approximating shortest distances between nodes using a neural network.

[1] `https://developers.google.com/freebase/guide/basic_concepts`

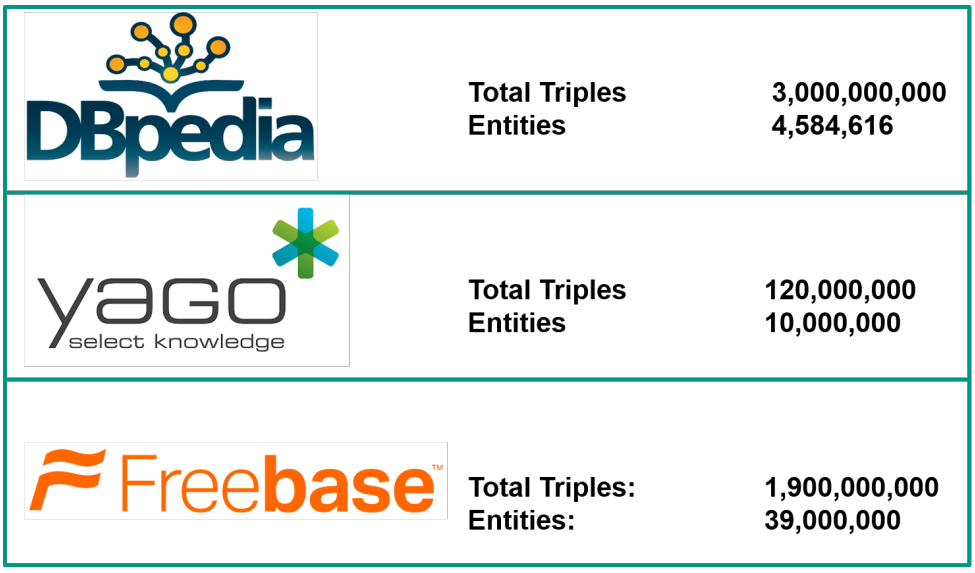

Figure 1: Some statistics of the three popular KGs DBpedia, Yago and Freebase. The numbers are from [2] [1] [3].

1.2 Objective

The objective of this thesis is to explore and evaluate different graph traversal techniques as well as a method for approximating shortest distances between two nodes using a neural network. More precisely, two algorithms Dijkstra and A-star are investigated in detail. Dijkstra is measured on a weighted graph and A-star is executed with two created heuristics.

1.3 Structure

Following structure is given to the thesis: In chapter 2, background information and relevant terms are introduced, which are important for a good understanding of the rest of the thesis. Chapter 3 presents related work that sets this thesis in perspective and gives further research work for the interested reader. Chapter 4 describes the approaches developed in the course of the thesis, followed by a presentation, and evaluation of the results in chapter 5. Finally, chapter 6 concludes the thesis and presents future directions of research.

2 Foundations

2.1 Semantic Web

The Semantic Web is built on top of the current Web and does not link documents but facts with each other. It has the purpose of providing a possibility to combine and aggregate resources on the Web so they could be collectively useful. Furthermore, it is machine-understandable, so machines can understand the meaning of given information and can then respond, using structure and rules embedded in the Semantic Web document. Tim Berners Lee defined it as follows:
"The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users"[4].
Another definition is given by the World Wide Web Consortium(W3C):
"The Semantic web is a major research initiative of the world Wide Web Consortium(W3C) to create a metadata-rich Web of resources that can describe themselves not only by how they should be displayed(HTML) or syntactically(XML), but also by the meaning of the metadata"[11].

2.2 Resourse Description Framework

The *Resource Description Framework* (RDF) is a wide used standard for encoding data and knowledge representation. It is deployed in the Knowledge Graphs that are examined in this thesis. RDF was originally developed by the World Wide Web Consortium (W3C) as a standard to describe metadata. However, it is now seen as a fundamental component within the Semantic Web. It allows to represent information about resources and their relations of the real world in a structured and machine-understandable way by dividing this information into statements with a clearly defined form.

In the RDF model, each statement is formulated from three units which form a triple. These three units are the *subject* the *predicate* and the *object* with the given form <*subject, predicate, object*>. The subject and the object are resources between which the predicate defines a relationship. Resources are seen as unique terms that always stand for a certain thing [21].

Relationships specify how entities in an ontology are related to other entities. Usually these relations are of a special type that defines the meaning of the relation of the two entities. Some relations of the DBpedia KG are for example: `rdfs:subClassOf`, `dbo:influencedBy`, `dbo:influenced`, `dbo:parent`, `dbo:foundedBy` or the very common

`rdf:type` which is further described in 2.4.

In RDF statements, the subject and the predicate have to be identified by a unique internationalised resource identifier (IRI), while the object may contain an IRI or a literal. In the case of being a literal, it describes data values that do not have a separate existence.

RDF statements can represent information but not yet the meaning of it. Since the Semantic Web is meant to enable machines to understand the meaning (semantics) of information on the Web, an extension of the RDF vocabulary the *RDF Schema* (RDFS) was introduced. "RDFS is an extendable knowledge representation language that one can use to create a vocabulary for describing classes, sub-classes and properties of RDF resources"[27]. It provides mechanisms for dividing related resources into groups called classes and describing the relationships between them. "The RDF Schema class and property system is similar to the type systems of object-oriented programming languages such as Java" [10]. The members of a class are known as instances of the class and the classes themselves are resources as well. Classes can be subordinated to other classes with the property `rdfs:subClassOf`, leading to a hierarchical structure.

2.3 Knowledge Graph

The structured knowledge bases (KBs) that are considered in this thesis, foremost DBpedia, but also Freebase can be described as Resource Description Framework (RDF) graphs. They are both derived from the Web encyclopedia Wikipedia and are made available via web applications and services. As defined by the W3C Recommendation, a RDF document or KB is a graph, consisting of a set of RDF triples [16]. Wikipedia is considered a semi-structured KB: It contains unambiguous entities, but most of the information is composed of unstructured data, e.g. the natural language text of the articles describing each entity. In the following only fully structured Knowledge Bases are examined. These structured KBs are called Knowledge Graph (KG) – a term that became popular due to Google's "Knowledge Graph", which serves the entity search result of Googles web search.

A Knowledge Graph is a graph constructed by representing all entities as nodes, and linking those nodes via edges that represent the relationships between them. Accordingly it captures the semantic knowledge of data by modelling information in a structured way. Knowledge Graphs have abundant natural semantics and can contain various and more complete information. Its expression mechanism is close to natural language. The entities that are described in the examined Knowledge Graphs are about real-world objects [8].

It has a long history in logic and artificial intelligence but more recently it has been used in the Semantic Web community with the intention of creating a "web of data" which is machine-readable [4].

The focus of this thesis lies on the Open Knowledge Graph DBpedia which follows the RDF standard [16]. This results in the following description: Given a Knowledge Graph $G = (E, R)$, where E is the set of entities, and R is the set of relations between them. A fact in a KG is represented by a triple in the form $< s, r, o >$, here $s, o \epsilon E$, $r \epsilon R$, where s and o are the subject and object respectively and r is the relation between them. Therefore, KG is a graph in which each edge has a value. A Knowledge Graph is a directed graph. Consider the following RDF triples from DBpedia:

```
<dbr:Albert_Einstein        rdf:type              dbo:Scientist>
<dbr:Albert_Einstein        dbo:deathPlace        dbr:Princeton,_New_Jersey>
<dbr:Albert_Einstein        dbo:birthPlace        dbr:Ulm>
<dbr:Albert_Einstein        dbo:doctoralAdvisor   dbr:Alfred_Kleiner>
<dbr:Princeton,_New_Jersey  rdf:type              dbo:Town>
<dbr:Princeton,_New_Jersey  dbo:timeZone          dbr:Eastern_Time_Zone>
<dbr:Ulm                    rdf:type              dbo:Town>
<dbr:Ulm                    dbo:Country           dbr:Germany>
<dbo:Scientist              rdfs:subClassOf       dbo:Person>
<dbo:Person                 rdfs:subClassOf       dbo:Agent>
<dbo:Town                   rdfs:subClassOf       dbo:Settlement>
<dbo:Settlement             rdfs:subClassOf       dbo:PopulatedPlace>
<dbo:PopulatedPlace         rdfs:subClassOf       dbo:Place>
```

These triples say that:
`dbr:Albert Einstein` is of type `dbo:Scientist`.
The `dbo:birthPlace` of `dbr:Albert Einstein` is `dbr:Ulm`.
The `dbo:deathPlace` of `dbr:Albert Einstein` is in New Jersey.
`dbr:Ulm` is of type `dbo:town`.
`dbr:Ulm` is located in the `dbo:country` `dbr:Germany`.
It can also be deducted, that:
`dbo:Scientist` is a sub-class `dbo:Person` and `dbo:Person` is a sub-class of `dbo:Agent` and therefore `dbr:Albert Einstein` is also of type `dbo:Person` and `dbo:Agent`.
This is based on the hierarchical structure of the underlying ontology of DBpedia as described in 2.4. The graph which can be created from these triples is shown in figure 2.

2.3.1 Entity

From the point of view of the KG, entities are the atomic units, like e.g. the city db:Ulm, that the KG makes statements about. In AI, the need for a proper knowledge representation, and thus an understanding of what an "entity" is, was mainly driven by the

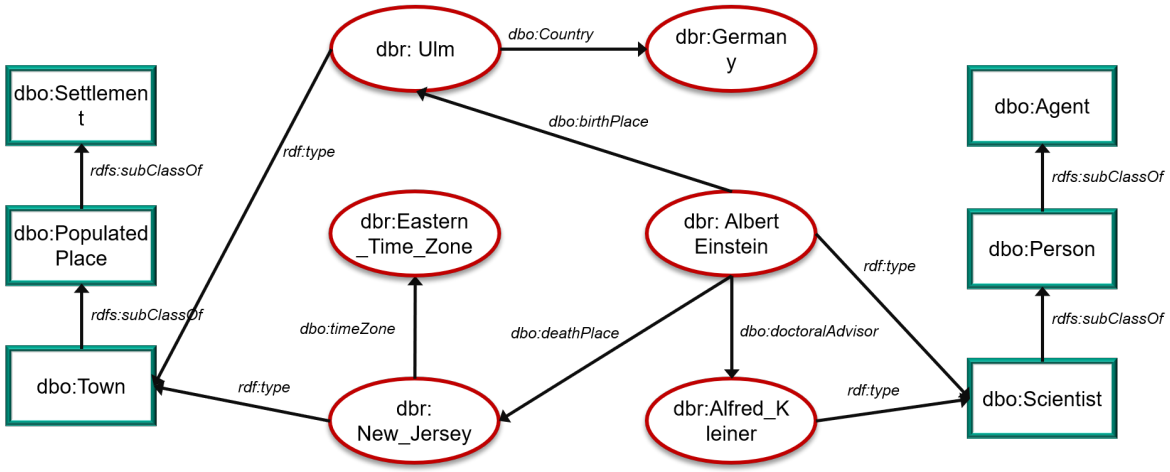

Figure 2: Directed Graph from the introduced triples. For better readability the classes are displayed in rectangles and the entities in ellipses

insight that for matching human performance in tasks such as natural language understanding, but also for building expert systems, the accumulation and use of large amounts of problem-specific knowledge is essential. For most KGs also for DBpedia, an entity is everything one can make statements about[17].

2.3.2 DBpedia

In this thesis, DBpedia is regularly used as a Knowledge Graph. It is a project that extracts information from Wikipedia and makes it accessible on the Web. Wikipedia is a multilingual encyclopedia on the Web where each page usually refers to a single entity and is manually classified in a number of categories. DBpedia represents those entities in a machine-readable RDF version of Wikipedia. It is organised as an RDF graph that contains millions of entities connected with fine-grained explicit semantic relations. It is also well-known for being the central entity repository authority within the linked open data (LOD) world.

It was first established in 2007 and since then, multiple releases have been published. This thesis uses the English release version from 2016-10.

The DBpedia ontology defines several classes and properties that help to structure the resources. It was created by manually arranging the most commonly used infobox templates within the English version of Wikipedia into a subordination based hierarchy of about 760 classes, forming a 7-level-class hierarchy. Infoboxes are the property summarizing tables found on many Wikipedia pages [21][5]. These classes, also called types, are further described in section 2.4.

2.4 Type information

In the DBpedia KG there exists a relation (`rdf:type`; URI: `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`) that denotes that the entity (the subject) is an instance of a certain type (the object).Types can also be regarded as ontology classes (semantic categories) that group together entities with similar properties as described in 2.2. For example, Albert Einstein is or respectively has been a Scientist in the real world, and therefore in the DBpedia KG the following triple exists:

`<dbr:Albert_Einstein, rdf:type, dbo:Scientist>`

This statement assigns the entity `Albert_Einstein` into the class `Scientist`. As described in 2.2, ontology classes are arranged in hierarchical structure, so that entities of type `Scientist` are also of type `Person`. This results in a *type path* like:
`Thing/Agent/Person/Scientist` with `Scientist` being the most fine-grained type and `Thing` being the coarse-grained type. Hierarchical structure in this case means, that the class `Scientist` is a sub-class of `Person`, which in turn is a sub-class of `Agent`, which is a sub-class of `Thing`. Figure 3 illustrates a part of the DBpedia ontology. The whole ontology can be seen at [7]. The property that entities are grouped into classes of different hierarchical levels is exploited in approach 4.1.

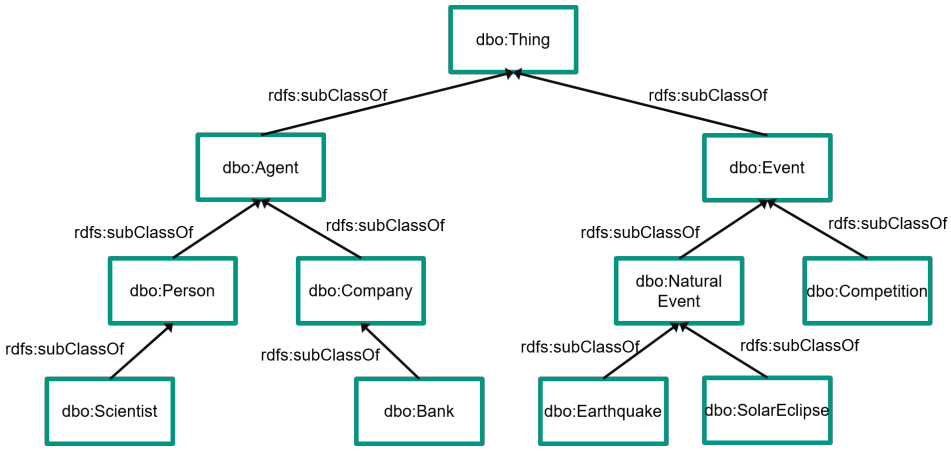

Figure 3: Illustration of a small part from the DBpedia ontology

2.5 Graph Traversal Algorithms

The problem of graph traversal in graph theory refers to the mechanism of traversing every edge and vertex in a graph in a systematic way. In the following some definitions are introduced which are important for a better understanding of the rest of the thesis.

Graph:
A graph is formed by vertices and edges connecting the vertices. Formally, a graph is a

pair of sets (V, E), where V is the set of vertices and E is the set of edges formed by pairs of vertices. E is a multiset, its elements can occur more than once [18].

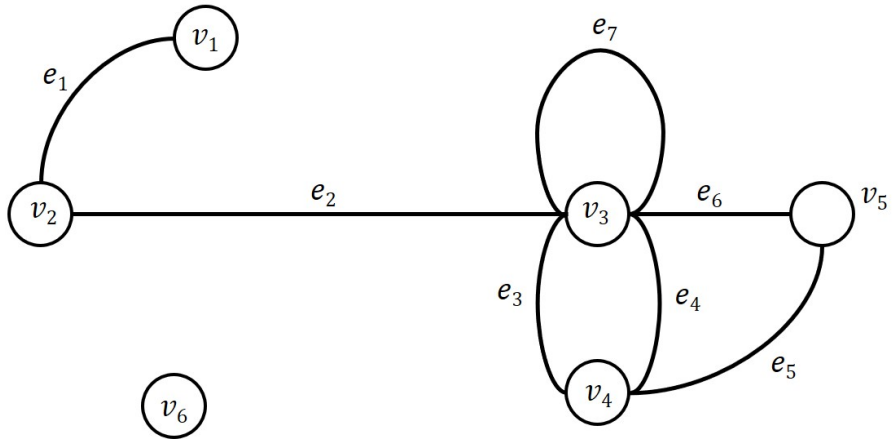

Figure 4: Example of an undirected graph

Figure 4 shows a graph with the vertices $V = \{v_1, ..., v_2\}$ and the edges $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_3,), (v_3, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$. The edges can also be labelled with letters as in figure 4: $e_1, e_2, ...$. Note that the two edges (u, v) and (v, u) are the same which means the pair is not ordered.

Labelled Graph:
By a labelling of edges of the graph, a mapping $\beta : E \to B$ is meant, where B is the label set. If these labels are numbers, then they are called *weights* of the edges and the graph is called weighted graph [18].

Directed Graph:
A directed graph or digraph is formed by vertices connected by directed edges. In directed graphs without multiple edges E forms a subset of all pairs (u, v) resulting from the Cartesian product $V \times V$. The elements of E are now ordered pairs and (u, v) is now the oppositely directed edge to (v, u). An example of a directed graph is shown in figure 5 [18].

Walk:
A walk in the graph $G = (V, E)$ is a finite sequence of the form

$$v_{i_0}, e_{j_1}, v_{i_1}, e_{j_2}, ..., e_{j_k}, v_{i_k}, \tag{2.1}$$

which consists of alternating vertices and edges of G and has to start at a vertex. k is the length of the walk. A zero length walk is just a single vertex v_{i_0}. It is allowed to visit a vertex or go through an edge more than once [18].

Path:
A walk is a path if any vertex and any edge is visited or gone through at most once except

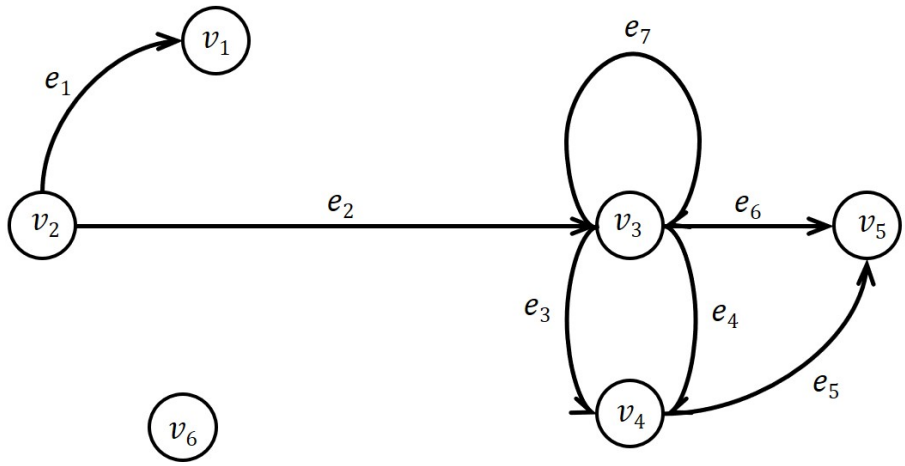

Figure 5: Example of a directed graph

possibly the initial and terminal vertices when they are the same. A path is a circuit if the initial vertex v_{i_0} and the terminal vertex v_{i_k} are the same($v_{i_0} = v_{i_k}$) [18].

2.5.1 Dijkstra

Dijkstra algorithm calculates the costs of the cheapest walks from a starting node to all other nodes in a graph. It starts from a starting node and selects step by step the cheapest walk via the next reachable nodes. It can also make improvements. It does this until all nodes have been visited and no better walk has been found. Because it gradually chooses the next node that promises the greatest gain (in this case the lowest distance from the root) it is called a greedy algorithm.
At the beginning, the algorithm has to be initialised:

- The path to the start node has zero cost
- The path to the other nodes is not known yet, therefore the costs to them are evaluated with infinite. During the algorithm these costs are to be improved, whereby the algorithm always stores the shortest path to every node
- The start node is added to the queue

Now the front-most node is taken from the queue. At the beginning this can only be the start node. Then all neighbouring nodes of the taken node are considered with their respective edges and checked for the following **condition**: Is the node included in the queue and are the costs over the new edge lower than the previous costs?

If this is the case, the costs for this node are reduced to the new value. Otherwise, the system checks for another **condition**: Has the node not been visited yet? If yes, it is now

added to the queue, so that later on, the edges leaving it can be viewed. In addition, this node receives the costs which result from the sum of the costs to its predecessor node and the costs of the newly discovered edge to itself.

Now nodes are taken out of the queue and their neighbouring nodes are checked for these two conditions until there are no more nodes in the queue. When the queue is empty, the algorithm is completed and has found the most favourable ways from the starting node to all other nodes.
In this thesis a modified version of dijkstra is used. This version stops as soon as the target node is found, calculating only the path form the source node to the target node. This is known as a Single-Pair-Shortest-Path problem [25].

2.5.2 A-star

The A-star search algorithm is one of the best and popular techniques used in graph traversals. It aims to find a path from the starting node to the goal node with the lowest possible cost. Cost can mean *least distance travelled, shortest time* etc. In the following, a path with *lowest possible cost* means a path between two nodes, which traverses as few nodes as possible, also called a "shortest path". It fulfils this task by examining vertices first that *probably* lead fast to the destination. To determine *promising* nodes, each known node x is assigned with a value $f(x)$ which is calculated using a task specific *heuristic*. The value indicates a best-case approximation of the distance to the destination when using the considered node. The node with the lowest value is examined first. The value $f(x)$ is calculated as follows:

$$f\left(x\right) = g\left(x\right) + h\left(x\right) \tag{2.2}$$

For a node x, $g(x)$ designates the costs to reach x from the starting node which can be calculated exactly because the path to x is known. $H(x)$ designates the approximated costs to reach the goal node from x. To calculate this cost a heuristic is used.

Functionality:

- *unknown nodes* have not yet been found during the search. There is no known path to them. At the beginning of the algorithm, all nodes except the starting node are *unknown*.
- *known nodes* have been already found and there exists a (possibly sub optimal) path to them. Every known node is stored together with its f-value in the `Open List`. From that list, the node with the lowest f-value is chosen and further examined

- *finally examined nodes* are stored in the `Closed List`, so that they are not examined repeatedly. A shortest path to those nodes is known.

At each step of the algorithm, the node with the lowest f-value is removed from the `Open List` and the f and g values of its neighbours are updated using the specified *heuristic*, and added to the `Open List`. The nodes in the `Open List` can be updated if there is a shorter path to them. To receive the found path after terminating, each examined node has to keep track of its predecessor, by keeping a pointer to it. When the node gets updated, the pointer is updated as well [24].

2.6 Deep Learning

2.6.1 Graph Embeddings

The key idea in the concept of graph embedding is to represent components of the graph (i.e., nodes and edges) as k-dimensional vectors in a continuous vector space in order to simplify manipulation while preserving the structure of the graph. One example of creating vectors from a graph is its own adjacency matrix. The adjacency matrix is a square matrix used to represent a finite graph. The elements reveal whether pairs of vertices are adjacent or not in the graph. Given a simple graph with a vertex set V, the adjacency matrix is a square $|V| \times |V|$ matrix A, such that each element A_{ij} is one if there is an edge from vertex i to vertex j and zero if there is no edge. Each column and each row in the matrix present a node. With n nodes in the graph, a vector representation of a node has n dimensions. The idea of graph embedding however is, to map the graph and its elements to low-dimensional embeddings. The embedding method which was used for this thesis is called node2vec. The idea is to find embeddings of nodes to d-dimension so that "similar" nodes in the graph have embeddings that are close together. The goal is to encode nodes so that similarity in the embedding space (e.g. dot product) approximates similarity in the original network.

2.6.2 node2Vec

Node2vec is an algorithm introduced by Grover and Leskovec [9] that works for directed/undirected and weighted/unweighted graphs. It takes short random walks in a graph and interprets the sequence of nodes seen on such random walks as if they were words appearing together in a sentence. From these walks a corpus is created. After generating the corpus, node2vec uses word2vec's Skip-gram model with Negative Sampling to obtain vector representations of the nodes.

The authors designed a flexible neighborhood sampling strategy which allows to smoothly

interpolate between BFS and DFS which play a key role in producing representations that reflect either *homophily* or *structural* equivalence. They state, that "under the homophily hypothesis nodes that are highly interconnected and belong to similar network clusters or communities should be embedded closely together" [9]. For instance, the nodes $s1$ and u in figure 6) belong to the same network neighbourhood. The structural equivalence hypothesis however, says that the embedding of nodes with similar structural roles in networks should be similar (e.g., nodes u and $s6$ in figure 6).

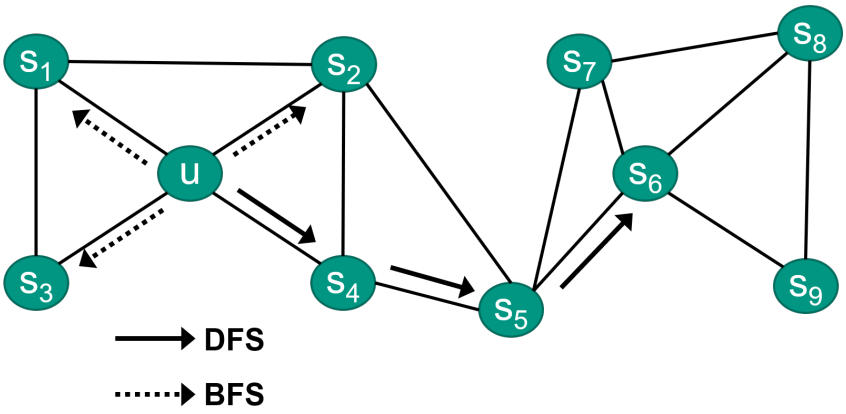

Figure 6: Example of BFS and DFS search strategy from node u with $k = 3$ (Ammended from [9])

Importantly, unlike homophily, structural equivalence does not emphasise connectivity; nodes could be far apart in the network and still have the same structural role. In real-world, these equivalence notions are not exclusive; networks commonly exhibit both behaviours where some nodes exhibit homophily while others reflect structural equivalence" [9]. BFS is a sampling strategy which generates a neighbourhood $N_s(u)$ of a source node u with a maximum of k nodes. The neighbourhood is restricted to nodes which are immediate neighbours of the source node. For example in Figure 1 for a neighbourhood of size $k = 3$, BFS samples nodes $s1, s2, s3$. The neighbourhood created by DFS consists of nodes sequentially sampled at increasing distances from the source node. In Figure 1 with $k = 3$, DFS samples $s4, s5, s6$.

Neighbourhoods that are sampled by BFS, result in embeddings that correspond closely to structural equivalence. The nodes sampled by DFS reflect more of a macro-view of the neighbourhood which is essential for deriving homophily similarities. It can be seen that both DFS and BFS both capture essential characteristics of the graph structure. To exploit both strategies, the authors of node2vec propose an algorithm which generates biased random walks which perform DFS and BFS simultaneously. Therefore two parameters to control the random walk are introduced. The **return parameter** p for controlling the probability of returning to the previous node in the walk and the **In-out parameter** q for controlling the probability of visiting nodes that are further away from the source node.

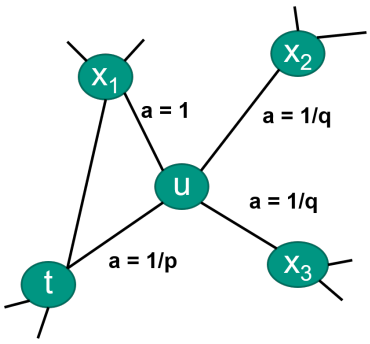

Figure 7: Illustration of the random walk procedure. The walk has just transitioned from u to t (Ammended from [9])

Using these two parameters, a *search bias* α_{pq} is introduced. Consider a part of a graph as shown in figure 7 and a random walk that has just transitioned from node t to node v. For every edge (v, x) leading out of vertex v, a transition probability π_{vx} is given. π_{vx} gives the probability of choosing node x as the successor of v in the walk. The unnormalised probability is defined as $\pi_{vx} = \alpha_{pq}(t, x)w_{vx}$ with w_{vx} being the weight of edge (v, x). α is defined as

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{2.3}$$

d_{tx} indicates the shortest path distance between nodes t and x.
A high value for p ($>$ max(q,1)) ensures that it is less likely to traverse an already visited node in the following two steps. If p is set to a low value ($<$ min(q,1)) the probability of walking back is higher and the walk is kept "local" to the starting node.
If q is set to a value $q > 1$, the walk approximates BFS behaviour and if $q < 1$ the walk represents a more DFS like structure, leading the walk further away from the source node. The random walks are then fed into the Skip-Gram model to update previously randomly generated node embeddings. Skip-Gram is a model originally designed for NLP, to predict the surrounding context words for a given target word. The model relies on the concept of a text corpus. This corpus however is displayed by the random walks which are treated as sentences.

2.6.3 Neural Networks

Artificial neural networks try to imitate the functioning of the human brain. They are trained for a special behaviour and are afterwards able to react appropriately both to the training situations and beyond them to new situations. This ability is achieved by the

interconnection of many (several hundred to several thousand) neurons.

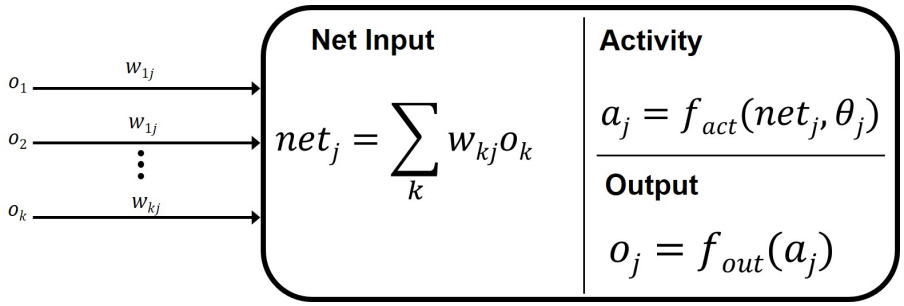

Figure 8: Illustration of a Neuron (Ammended from [15])

A **neuron** is a processing unit, which summarises the incoming values of the weighted connections in a suitable way (propagation function) and determines an activation state by means of an activation function considering a threshold value. From this activation an output function determines the output of the neuron. A basic structure is shown in figure 8. Each neuron includes the following values and functions:

- The **input information** o_k, which neuron j receives from other neurons that are connected with it. This input information is regulated by the respective weights w_{kj} of the connections. The weighted information can also be received from the environment of the network as input values.
- A **propagation function** net_j which links the input information with the weights of the connections to a net input. A net input combines the information coming from the net into the neuron: $net_j = \sum_k w_{kj} * o_k$
- The **activation function** f_{act} determines from the net input, the new activity a_j under consideration of a threshold value b_j: $a_j = f_{act}(net_j, b_j)$.
- An **output function** f_{out} determines from the activity a_j of the neuron, the output value o_j, which is passed on to the following neurons: $o_j = f_{out}(a_j)$. The identity $f(x) = x$ is very often used as output function which leads to $o_j = f_{out}(a_j) = a_j$. Although other functions are not excluded, the identity is assumed in the following for simplification purposes.
- A local storage contains the activity a_j and the threshold value b_j of the neuron.

If several neurons of the type described above are linked together, a network of neurons is created. A connection between two neurons i and j is characterised by a connection weight $w_{ij} \neq 0$. The definition of a network is thus obtained by defining the connections and the connection weights. Neural networks are used to perform supervised learning and

prediction tasks. Input data is provided to the neural network. This input data is named a training dataset. A training dataset contains labelled data, which means that the result is known [15].

2.6.4 Feedforward Neural Network

Feedforward neural networks consist of several layers of neurons, but at least of one input and one output layer. One or more inner layers (hidden layers) can be interconnected. It is important that only neurons of different layers are linked together, however only in the direction of the output layer. Feedforward neural networks are monitored and trained by means of error feedback [15]. If the neurons of a layer receive input from all neurons of its previous layer, it is called a dense layer and is therefore deeply connected. Usually a neural network is used with non linear activation functions and dense layers, as it is done in this thesis. Common non linear activation functions are the rectified linear unit (ReLu) or softplus. ReLu is defined as

$$f(x) = max(0, x) \tag{2.4}$$

and softplus which is a smoother version of ReLu as:

$$f(x) = ln(1 + e^x) \tag{2.5}$$

2.6.5 Backpropagation

A common feature of all neural network models is that a large number of simple units, neurons or particles, together have a task to solve. The interconnection of these units can be realised by weighted connections and/or by a processing function. The goal of the interconnection is to solve tasks that cannot be explicitly described by an algorithm, but whose solution is described by examples. From the examples a network behaviour is trained, which is then able to process unknown inputs as desired. Backpropagation is a common method for teaching artificial neural networks. Let x be a training input, then $y = y(x)$ is the desired output. The neural network has to be trained, so that it approximates $y(x)$. The backpropagation algorithm achieves this by finding weights and biases so that the output of the network approximates $y(x)$ for all training inputs x. To quantify how well this goal is achieved, a *cost function* is defined:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{2.6}$$

Here, w denotes the collection of all weights in the network, b all the biases, n the total number of training inputs, L the number of layers in the network and $a^L = a^L(x)$ the

vector of activation output when x is the input. C is well known as the mean squared error (MSE). The cost $C(w, b)$ becomes small, i.e. $C(w, b) \approx 0$, precisely when $y(x)$ is approximately equal to the output a, for all training inputs x. The training algorithm has done a good job if it can find weights and biases so that $C(w, b) \approx 0$ and it has done a bad job if $C(w, b)$ is large, meaning $y(x)$ is not close to the output a for a large number of inputs. The idea of backpropagation is to use gradient descent to find the weights w_k and biases b_l which minimise the cost in equation 2.6. It does this by computing the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function C with respect to any weight w or bias b in the network [15].

3 Related Work

A series of recent studies has focused on RDF Graph traversals. In this chapter, some related work is presented.

Weighted Shortest Paths for RDF Graphs (WiSP) [23] focus on techniques to find paths between nodes which are likely to be of interest to a user. Various existing methods often rely on enumerating all such paths (up to a fixed length or number) before ranking is applied. In this paper, they instead propose applying a shortest path search on weighted versions of the graph in order to directly compute the most relevant path(s) between two nodes without fixed-length bounds, further obviating the need to enumerate irrelevant paths. They investigate weighting based on node degree, PageRank and edge frequency.

In prior research multiple hops using SPARQL, has been exploited to find paths between two entities or nodes in a Knowledge Graph [20]. RelFinder [12] is another interactive visualized tool which reveals relationships between two objects and displays them as a graph. They focus more on visualization and user experience. Another paper is about discovering unknown connections in the DBpedia Knowledge Graph [13]. Therefore, the tool RelFinder is used.
Other graph traversal algorithms that were not used in this thesis are briefly introduced below.

Depth-first search:
A DFS algorithm traverses the depth of any graph before exploring its breadth. It begins at the root node from where it iteratively transitions from the current node to an adjacent unvisited node. The algorithm does this, until there is no unexplored node to transition to from the current node. It then backtracks along previously visited vertices until it finds a node that is connected to an unexplored node and repeats the previous procedure. It does this until no unvisited nodes are left. Essentially the algorithm picks nodes starting from the root node in a systematic way, where it prefers child nodes over siblings.

Breadth-first search:
Unlike DFS, the Breadth-first search visits sibling nodes before child nodes. It starts with a source node and traverses the graph layerwise by exploring all the nodes that are directly connected to the source node. After the first layer is explored, it moves on to the nodes at the next depth level.

Bellman-Ford algorithm:
The algorithm calculates the cost of the cheapest paths from a starting node to all other nodes in a weighted graph. It can thus construct the cheapest paths itself. The algorithm proceeds iteratively: it starts from a very bad estimate of the costs and improves them until the correct values are found. At the beginning the start node has cost 0 and the

remaining nodes have infinite costs. Then it checks all edges for the following condition: Are the costs of the starting node i of edge (i, j) plus the costs of using the edge (i, j) lower than the costs of the target node j? If this is the case, a shortcut is found and the costs of target node j are updated [22].

Floyd Warshall algorithm:
The Floyd Warshall algorithm solves the All-Pairs-Shortest-Path problem, i.e. it calculates the shortest paths between all node pairs for a given graph. It is based on the principle of dynamic programming. All possible paths between all pairs of nodes are compared step by step, with only the best values being stored.The algorithm assumes the following observation: if the shortest path from u to v passes through w, then the sub paths from u to w and from w to v are also minimal. The Floyd-Warshall algorithm works iteratively [26].

In this thesis, the two graph traversal algorithms Dijkstra and A-star are used. The Dijkstra algorithm is usually performed on weighted graphs. Knowledge Graphs, although labelled, are not weighted. Therefore this thesis introduces a weighting of edges based on the degree of nodes. The A-star algorithm needs an heuristic to work properly, but without knowing the distances between nodes at all, this becomes a hard task. This thesis presents two different heuristics that use the `rdf:type` information that is given in Knowledge Graphs. A third approach presents a method for distance approximation using a neural network, that could potentially be used as a heuristic for the A-star algorithm.

4 Approach

Approach 4.1 and 4.2 describe techniques to find paths between two nodes in Knowledge Graphs. Approach one tries to use the type information of the nodes, while in approach two the graph is precomputed, so that edges are weighted based on the degrees of the nodes they end in.
In approach 4.3 node embeddings are exploited and with the use of a neural network, shortest distances between nodes are approximated.

4.1 Using Type Information

The objective of this approach is to use the type information of the nodes as guidance for the A-star algorithm to find a shortest path between the *starting node* and the *target node*. This is achieved by comparing the ontology classes of the target node with the ontology classes of the *traversed nodes* and thereby preferring nodes which have common ontology classes. *Traversed nodes* are the nodes, that are examined by the algorithm in order to find a shortest path to the *target node*. The nodes that share the most fine-grained type are preferred most, while nodes that share no type except `owl:Thing` are given the lowest priority.

To achieve this goal, the following steps were carried out:

1. The hierarchy of the existing ontology classes has been extracted
2. Each ontology class has been assigned with a hierarchy number:
    a For each step deeper in the hierarchy or as described in 2.4: one step further on the *typepath* towards the more fine-grained type, the hierarchy number has been increased by one. Consider for example the typepath: `Agent/Person/Scientist`, where `Scientist` is a sub-class of `Person` and `Person` is a sub-class of `Agent`. `Agent` however has no super-class and is therefore assigned with the hierarchy number 1. `Person` will be assigned with 2 and `Scientist` with 3
    b A dictionary has been created, which assigns each ontology class with its hierarchy number
3. Each node in the Knowledge Graph has been equipped with a list called `typeinfo`. This list has been filled with all the ontology classes (and the respective hierarchy numbers) that have been assigned to this node. The tuples were in the form: *(ontology class, hierarchy number)*
4. The A-star algorithm has been executed for each pair of nodes. As described in 2.5.2, an heuristic to calculate the expected costs to reach the value is used for each

examined node. In this thesis, two different heuristics have been created to value the similarity of the type information between the *target node* and the *traversed node*.

Heuristic 1:
Prior to the execution of the A-star algorithm coupled with the heuristic, a single operation has to be done: The `typeinfo` list of the target node must be sorted by the hierarchy number, so that the types with higher hierarchy number come first in the list.

When the heuristic is called by the A-star, it iterates over the `typeinfo` list of the target node. Because the list is sorted, it iterates over the types with the highest hierarchy number first. For each iteration, it compares the current type from the list of the target node with the whole `typeinfo` list of the traversed node, and if it finds the type of the target node in the `typeinfo` list of the traversed node, it stops and returns $\frac{1}{hierarchynumber}$ back to the A-star. This results in returning the value (meaning: $\frac{1}{hierarchynumber}$) of the most fine-grained type the two nodes have in common. Because the hierarchy number of more fine-grained types is higher, this leads to the fact, that nodes that have more fine-grained types in common will be valued lower and therefore be preferred by the algorithm.

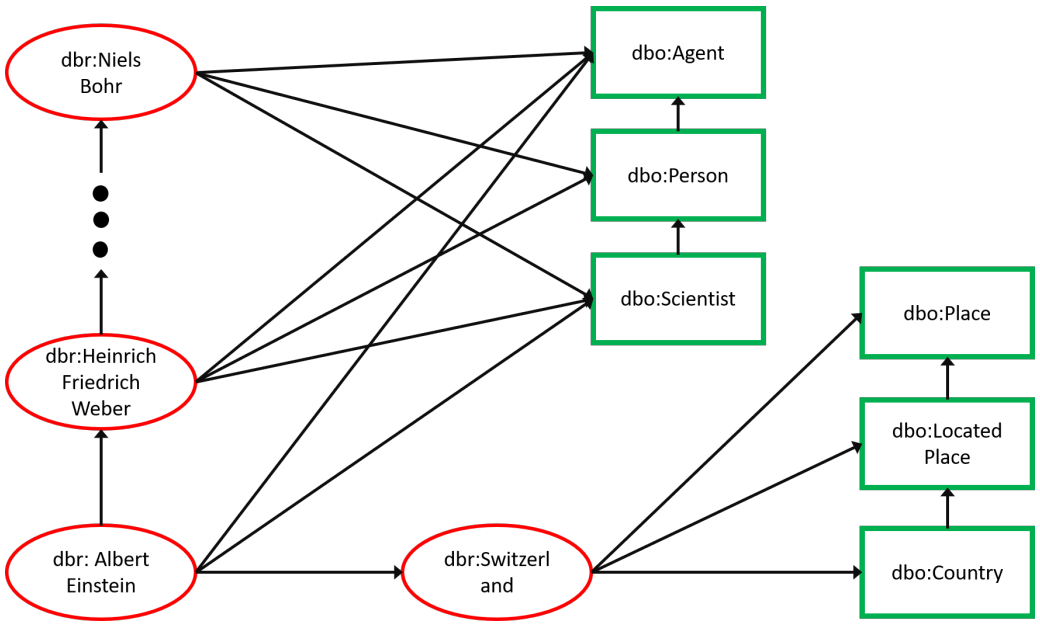

Figure 9: Illustration of parts of a graph

Example: Consider the nodes as depicted in table 9. As start node `dbr:Albert Einstein` and as target node `dbr:Niels Bohr` with the types `dbo:Agent, dbo:Person, dbo:Scientist`.

The A-star starts with Albert Einstein and has then to choose which node with its outgoing edges to examine next. For both nodes the heuristic is called and returns values depending

Table 1: Nodes with Types and the assigned Hierarchy Number

Node	Types	Hierachy Number
Niels Bohr		
	Agent	1
	Person	2
	Scientist	3
Switzerland		
	Place	1
	Located Place	2
	Country	3
Heinrich Friedrich Weber		
	Agent	1
	Person	2
	Scientist	3

on the node types. Since Switzerland does not have any common types with Niels Bohr, the heuristic returns a fix value which is set to two. For Heinrich Friedrich Weber it returns $\frac{1}{hierarchynumber}$ of the type with the highest hierarchy number that it shares with the target node Niels Bohr, which in this case is Scientist with hierarchy number 3. As described in 2.5.2, it calculates the f-value with the following formula:

$$f(x) = g(x) + h(x) \tag{4.1}$$

Since the graph is unweighted, each traversed edge from the start node to the actual node, $g(x)$ is increased by one. Both nodes are exactly one edge away, so $g(x)$ for both is 1. $H(x)$ of Heinrich Friedrich Weber is $\frac{1}{3}$ and for Switzerland it is 2, resulting in $f(HeinrichFriedrichWeber) = 1 + \frac{1}{3}$ and $f(Switzerland) = 1 + 2$. Thus as a next step the node Heinrich Friedrich Weber is chosen to be examined. This shows, that nodes that are of type Scientist will be preferred by the algorithm.

Heuristic 2:
This heuristic as well iterates over the `typeinfo` list of the target node. For each match between the list of the target node and the traversed node, a *counter* will be increased by the respective hierarchy number. When the for loop is finished, $\frac{1}{counter}$ will be returned. This leads to:

- The more types the target node and the traversed node have in common, the lower the returned value
- The higher the hierarchy number of the types that are equal, the lower the returned value

4.2 Using Node Degree

The degree of a node is the number of edges adjacent to it. For a directed graph it is further divided into indegree and outdegree. Indegree is the number of edges ending in the node and outdegree is the number of edges starting at the node.
These properties were exploited in the following for finding paths between nodes. Therefore the modified Dijkstra algorithm, as described in 2.5.1, has been applied.
Before the algorithm could be executed, the graph had to be edited in advance. All edges have been weighted based on the outdegree of the nodes they end in.

Given a graph $G = (V, A)$, where V is the set of nodes and A is the set of directed edges. For $v_1, v_2 \, \epsilon \, V$, the outdegree of v_1 is denoted as $d^+(v_1)$. For a directed edge $a \, \epsilon \, A$ that ends in v_1, e.g. (v_2, v_1) the weight $W(a)$ of the edge will be calculated as follows:

$$W(a) = \frac{1}{d^+(v_1)} \tag{4.2}$$

This is done for every edge and will cause, that edges with an end in nodes with higher outdegree have lower weights. Consider an edge a_1 that ends in a node v_1 with $d^+(v) = 10$ and an edge a_2 that ends in a node v_2 with $d^+(v) = 1$. The introduced method will label edge a_1 with the weight $W(a_1) = \frac{1}{10}$ and edge a_2 with the weight $W(a_2) = \frac{1}{1} = 1$. Since the Dijkstra algorithm is a greedy algorithm and therefore follows the edge which promises the lowest section from the start node, this means, that edges which end in a node with high outdegree are chosen over edges that end in nodes with lower outdegree.

4.3 Shortest Distance Approximation Using a Neural Network

This approach is a reproduction of a paper from Rizi and Granitzer [19]. Let $G = (V, E)$ be an unweighted undirected graph with n nodes and m edges. The used embedding technique node2vec creates a real-valued vector embedding $\phi(v) \epsilon R^d$ for every node $v \epsilon V$. The goal of this approach is to approximate the real shortest distance $d_{u,v}$, between a pair of nodes $u, v \epsilon V$. This distance $\hat{d}$ is approximated using a feedforward neural network. $\hat{d}$ is defined as a function

$$\hat{d} : \phi(u) \times \phi(v) \rightarrow R^+ \tag{4.3}$$

that maps a pair of vector embeddings to a real-valued shortest path distance $d_{u,v}$. Before the neural network can predict those distances, it has to be trained with training pairs from the graph G. Therefore a small number l of nodes as landmarks are chosen. From these landmarks to all other nodes in the graph, the actual shortest distance is computed using BFS. With n nodes in total, this leads to $l(n - l)$ training pairs. For each training pair $< \phi(u), \phi(v) >$ a joint representation as input for the neural network is created using

concatenation: $(\phi(u), \phi(v))$.
The feedforward network consists of an input layer, a hidden layer, and an output layer. The input layer requires $2d$ neurons. For the first two layers, the rectified unit (ReLU) activation function is set. The output layer uses a single unit of softplus. The quality of the predictor is evaluated by Mean Squared Error (MSE).
With n nodes in the graph, learning vector embeddings takes $O(n)$ time. For each landmark a BFS tree has to be computed, to receive the shortest distances to all remaining nodes. With $l(n-l)$ training pairs and time complexity of BFS for unweighted sparse graphs being $O(n+m)$, this results in consuming $O(l(n+m))$ time. A trained feedforward neural network can process an input of two concatenated vector embeddings $< \phi(u), \phi(v) >$ in a small amount of time independent of the graph size, i.e. $O(1)$ time. This means that calculating the shortest path distance from a starting node u to all remaining nodes takes $O(n)$.

Approximation Quality

The results of the prediction are rounded because shortest path distances are discrete real values. Two metrics, namely Mean of Relative Error (MRE) and Mean Absolute Error (MAE) are used to measure the accuracy of the used method. MRE is defined as,

$$MRE = \frac{1}{n} * \sum_{i=1}^{n} \frac{|\hat{d} - d|}{d} \tag{4.4}$$

with d being the actual distance, $\hat{d}$ the approximation and n the number of approximated values. Since the MRE is smaller for larger distance values which is not perfectly fair, the second metric MAE is used which is defined as

$$MAE = \frac{1}{n} * \sum_{i=1}^{n} |\hat{d}_i - d_i| \tag{4.5}$$

being independent of the target value.

5 Evaluation

5.1 Dataset

For the approaches 1 and 2, the following dataset was used:

1. First, all entities of the type `dbo:Scientist` have been extracted. Number of entities: 24948
2. Next, all triples, that contained the entities from 1. as a subject, have been extracted. Number of triples: 174339

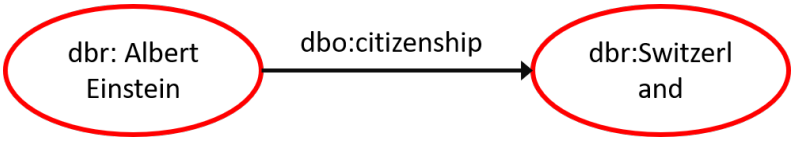

Figure 10: Graph representation of a triple with an entity of type `dbo:Scientist` as subject.

3. The objects of the triples from 2. have been selected and multiple occurrences removed. Number of objects: 51635
4. Triples that embody the selected entities(the former objects) from 3. as subjects have been extracted. Number of Triples: 164044

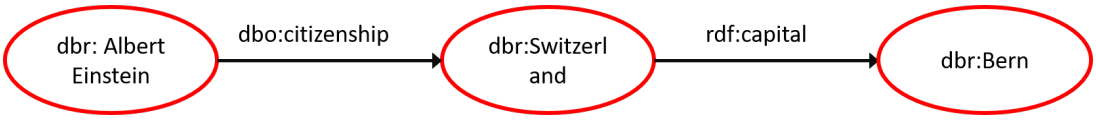

Figure 11: Graph representation of two triples: The object `dbr:Switzerland` of the first triple, is the subject of a triple that was added in 4.

5. The triple from 2. and 4. have been merged together in a dataset and the types of all entities have been added. Triples that occured more than once have been reduced to only one occurrence and types which are not contained in the used type hierarchy from DBpedia [7] were deleted.

The created dataset contains 441868 triples.

Essentially the dataset consists of all the entities of type Scientist with two hops starting from each of the scientists. Additionally the `rdf:type` information is included. Table 2 gives an overview.

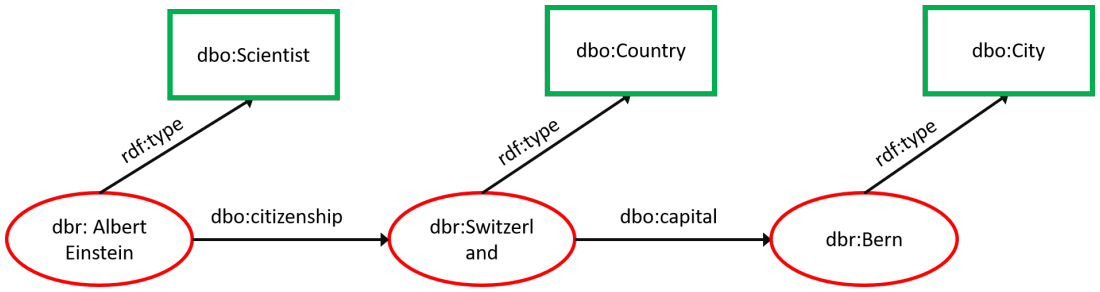

Figure 12: Illustration of two hops with the included `rdf:type` information

Table 2: Statistics of Scientist Dataset

Number of scientists	24948
Number of triples from first hop	174339
Number of triples from third hop	164044
Total number of triples	441868

5.2 Approach 1 and 2

At this section, the approaches 4.1 and 4.2 will be evaluated. This includes the A-star algorithm with the heuristics from section 2.5.2, which will further simply be called "Heuristic 1" and "Heuristic 2". It also includes the modified Dijkstra algorithm which was performed on the graph with modified edge weights. This will be called "Weighted by Outdegree" and "Weighted by Degree". As a baseline the modified Dijkstra algorithm was executed on the "raw" unedited graph. It will be called "No Heuristic" since it was implemented like an A-star algorithm but without a heuristic.
The used dataset is explained in section 5.1. From this dataset two sets of nodes have been created. For each set, the algorithms had to find paths between all the containing nodes. With n nodes this has led to $n * (n-1)$ pairs of nodes between which paths have been searched. The sets had 200 nodes each, resulting in $200 * 199 = 39800$ pairs. Set A contained 200 randomly selected scientists, meaning 200 entities of type `dbo:Scientist`. Set B contained 200 randomly picked entities from the given dataset.
Between the pairs of set A exist 9993 paths, whereas between the pairs of set B only 665 exist. Only the baseline was guaranteed to find a actual shortest path. To compare the used sets (e.g. set A, set B) based on the average shortest path length, one has to compare the path lengths of "No Heuristic".

Set A:
Table 3 gives an overview of the average time and path lengths for each technique. Figures 13 to 21 show detailed results for set A.

Table 3: Overview of the results based on set A

Technique	**Average time per path**	**Average length of paths**	**Average time per path compared to "No Heuristic"**
Heuristic 1	0.112s	13,64	55%
Heuristic 2	0.109s	13,64	53%
No Heuristic	0.203s	9,97	100%
Weigthed by Outdegree	0.122s	10,53	60%
Weigthed by Degree	0.115s	10,53	56%

It can be seen, that heuristic 1 and heuristic 2 performed fastest. For heuristic 2, the average required time per path was only 53% of what was needed for the baseline. However, for both heuristics, the path was in average 3.67 steps longer than the shortest possible path (average shortest path: 9.97 compare "No Heuristic"). The approaches which were based on the node degree, performed slightly slower than the heuristics, but still significantly faster than the baseline. They also found paths which are in average 3.11 steps shorter than the heuristics and only 0.56 steps longer than the average shortest path. For example in figure 15 and 16 it can be seen, that the heuristics came up with some paths that are longer than 30 steps, up to 43, whereas the rest of the techniques had a maximum length of 30. The occurrence of paths of this length is very low as shown in figure 15. Nevertheless, figure 15 also shows a slight shift of the curve to the right for the two heuristics. This explains the larger average of the path lengths.
The node degree based approaches show an increase in search time that is strongly proportional to the increase in path lengths, up to a length of about 18 (compare figure 20 and 21. The baseline also shows a proportionality between path length and search time, but it flattens earlier. The search time of the heuristics does not rise that evenly but also up to a path length of about 14 an increase with higher path length is visible. From a path length of maximum 18, the curves of all approaches show an irregular course but remain within a limited area. The number of paths decreases with increasing path length. This leads to the fact that trends are difficult to recognise with increasing path length, because outliers get more weight.

Set B:
Table 4 gives an overview of the average time and path lengths for each technique. Figures 22 to 30 show detailed results for set B.

Table 4: Overview of the results based on set B

Technique	**Average time per path**	**Average length of paths**	**Average time per path compared to "No Heuristic"**
Heuristic 1	0.225s	14.33	150%
Heuristic 2	0.232s	14.33	154%
No Heuristic	0.150s	12.75	100%
Weigthed by Outdegree	0.196s	13.19	131%
Weigthed by Degree	0.191s	13.34	127%

Compared to set A, all techniques except the baseline performed slower. The A-star techniques performed worse, both in terms of time required and path length. The baseline was faster and the actual shortest paths between pairs of set B are 2.78 steps longer than the ones from set A. The curves which are shown from figure 26 to 30 to a very irregular.

Since the nodes for set B have been chosen completely randomly, it contained 90 nodes which were not assigned into a type. Therefore heuristic 1 and heuristic 2 could not be performed on pairs that included one of the 90 nodes as a target. For that reason all the algorithms were run again on the same set but without the mentioned nodes. With 110 remaining nodes this lead to $110 * 109 = 11990$ pairs which resulted in 431 found paths. Table 5 gives an overview of the average time and path lengths for each technique. Figures 31 to 39 show detailed results for the cleaned set B.

Table 5: Overview of the results based on set B cleaned

Technique	**Average time per path**	**Average length of paths**	**Average time per path compared to "No Heuristic"**
Heuristic 1	0.165s	14.14	124%
Heuristic 2	0.173s	14.14	130%
No Heuristic	0.133s	12.81	100%
Weigthed by Outdegree	0.121s	13.20	91%
Weigthed by Degree	0.123s	13.38	92%

It is visible, that both, the A-star approaches and the node degree approaches had a

significant performance increase after cleaning set B. Performance of heuristic 1 increased by 26.67%, heuristic 2 by 25.43%, outdegree by 38.27%, degree by 35,6% and the baseline by 11.33 %. The node degree approaches performed fastest and the heuristics slowest.

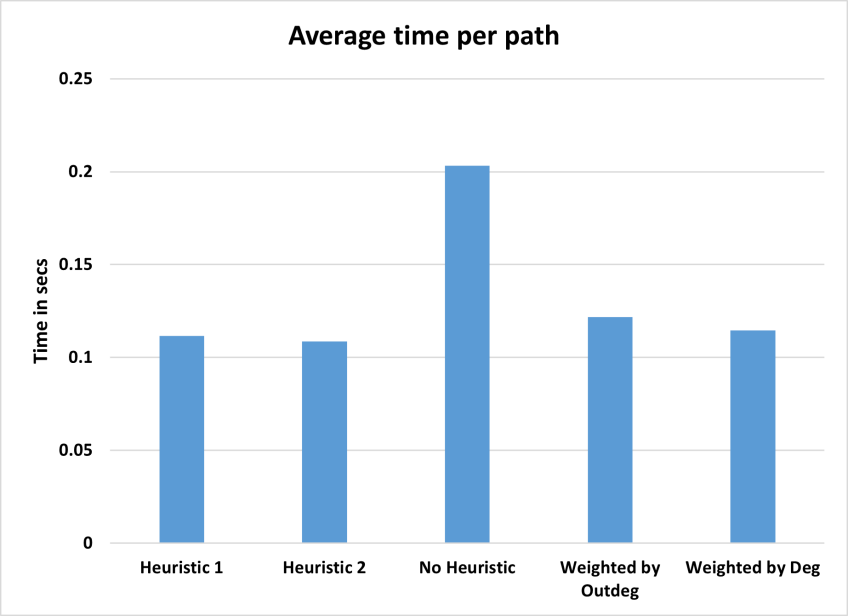

Figure 13: Average time per path for each approach set A

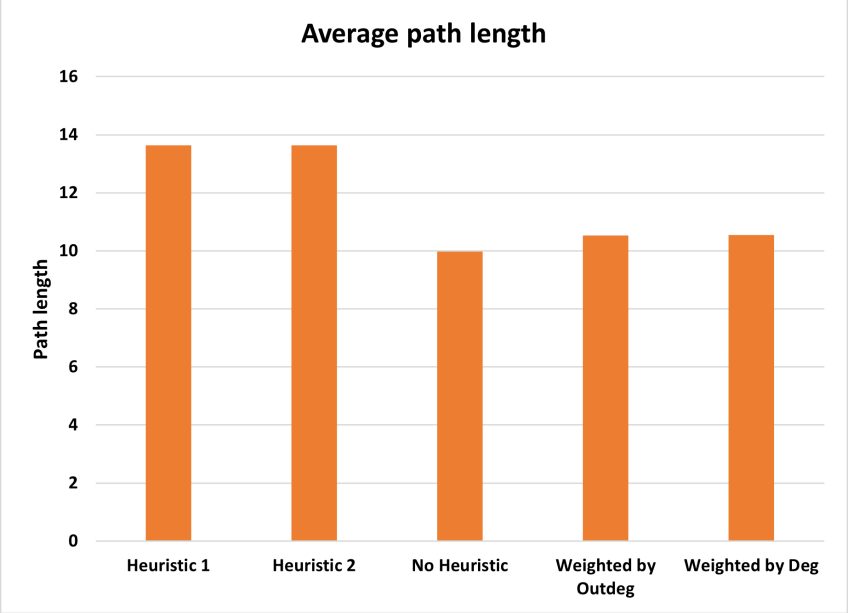

Figure 14: Average path length for each approach set A

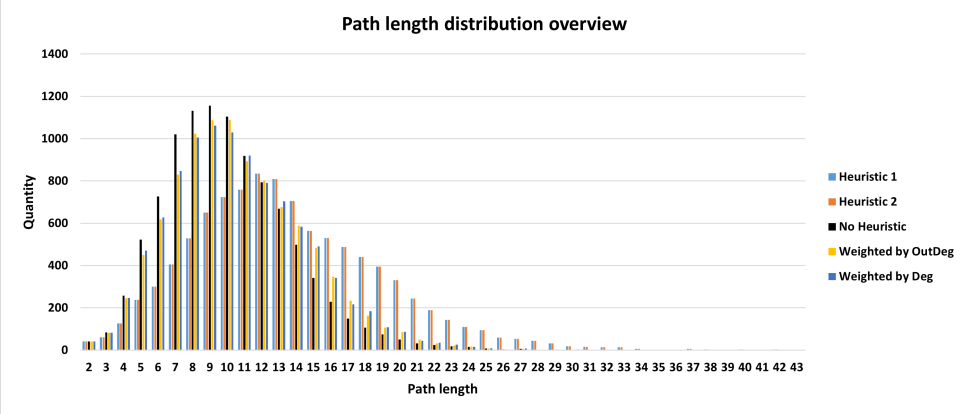

Figure 15: Distribution of path lengths for each approach set A

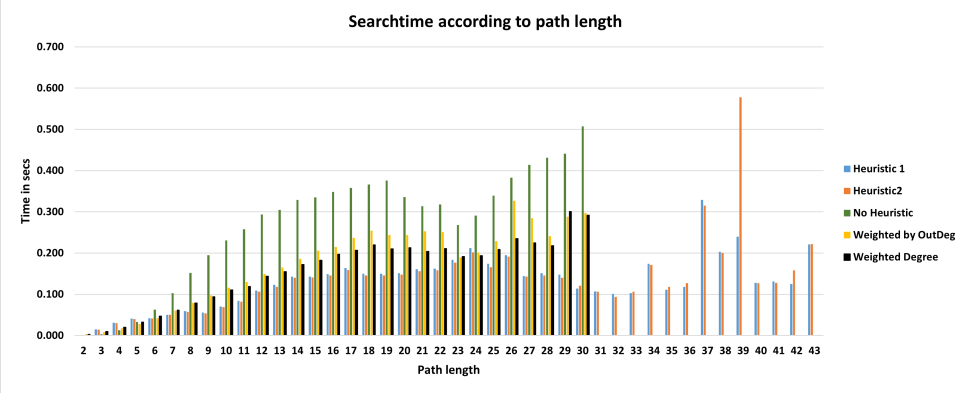

Figure 16: Average search time per path length for each approach set A

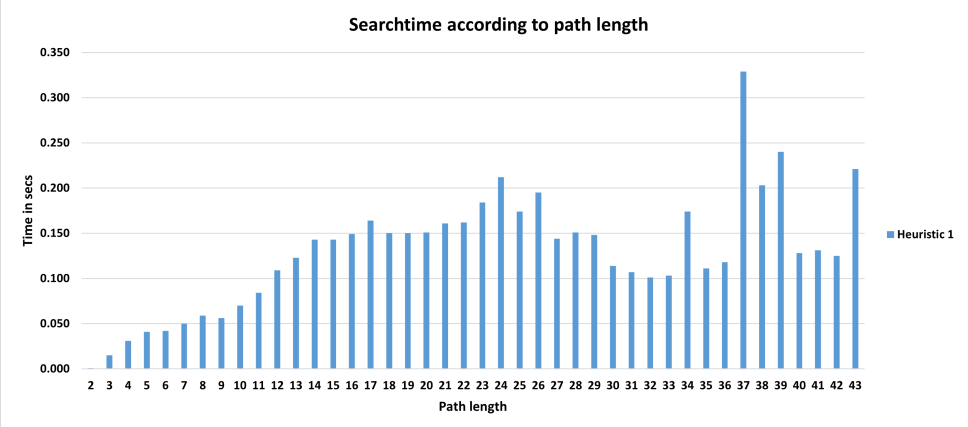

Figure 17: Average time per path heuristic 1 set A

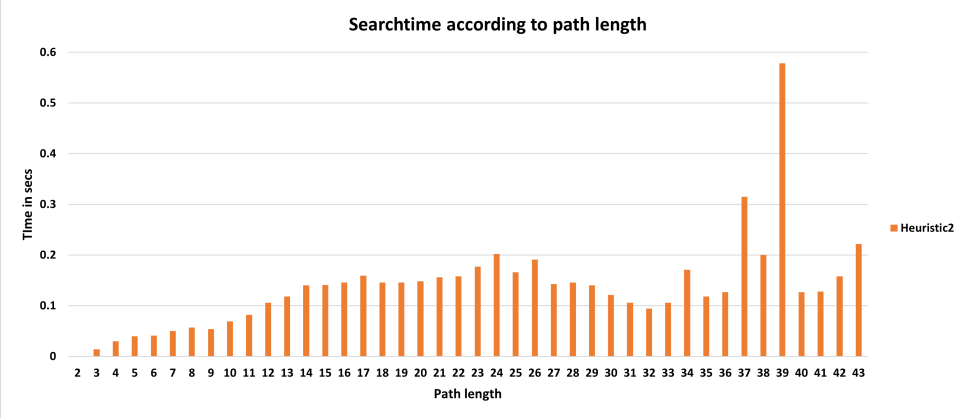

Figure 18: Average time per path heuristic 2 set A

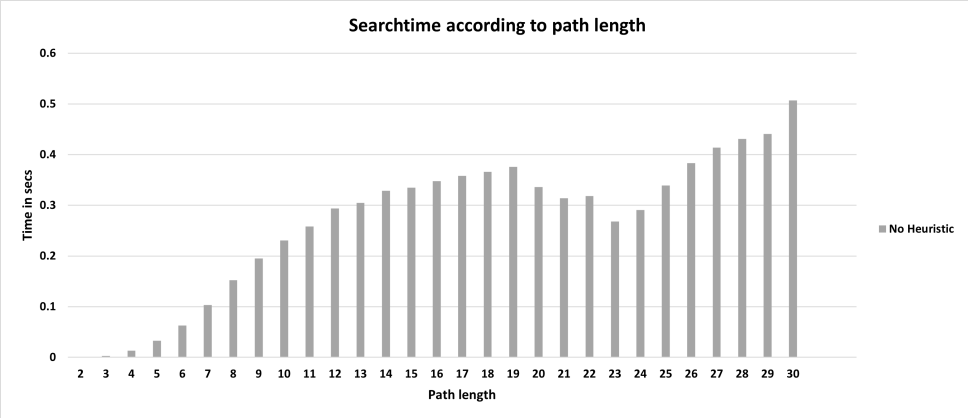

Figure 19: Average time per path no heuristic set A

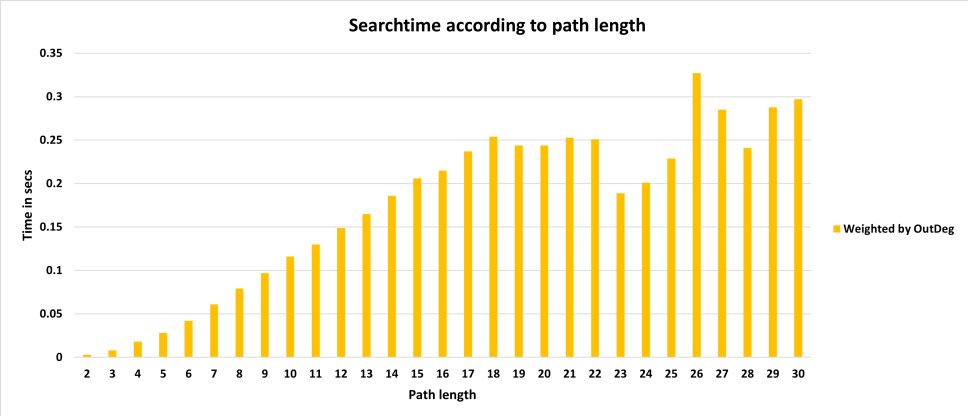

Figure 20: Average time per path weighted by outdegree set A

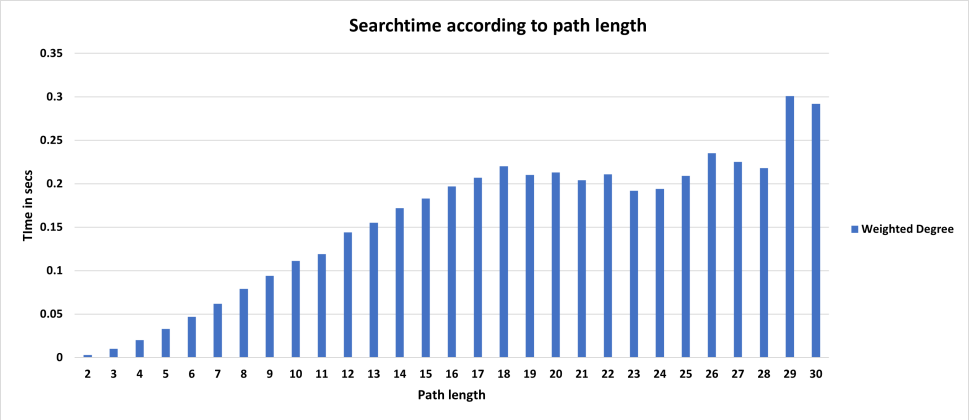

Figure 21: Average time per path weighted by degree set A

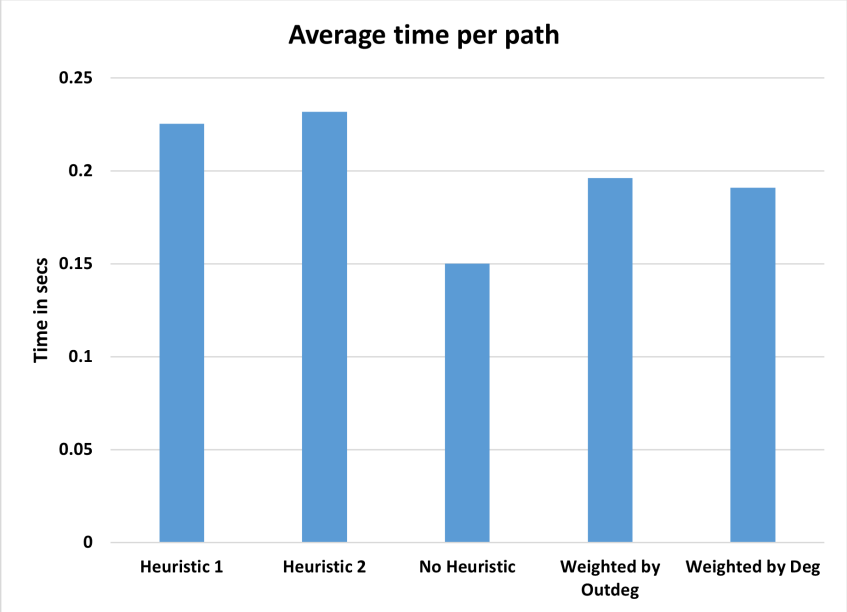

Figure 22: Average time per path for each approach set B

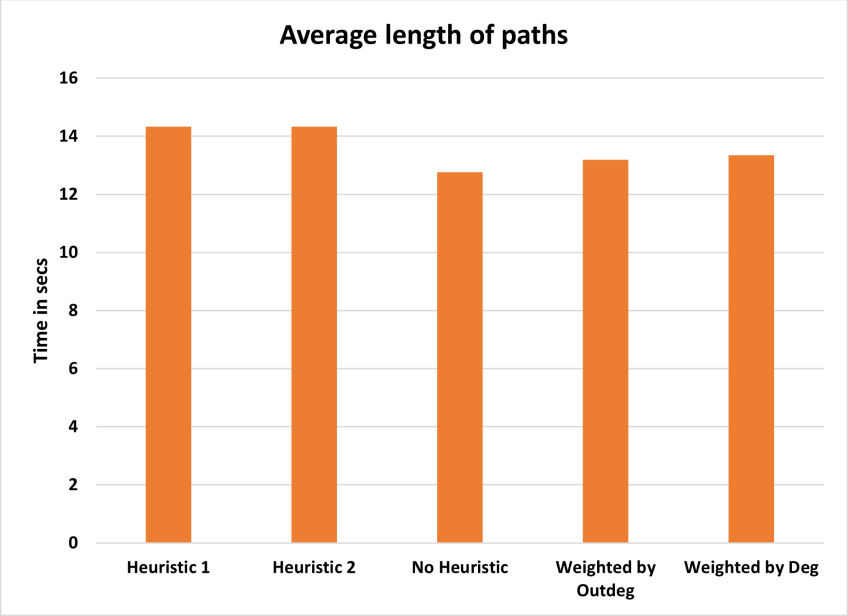

Figure 23: Average path length for each approach set B

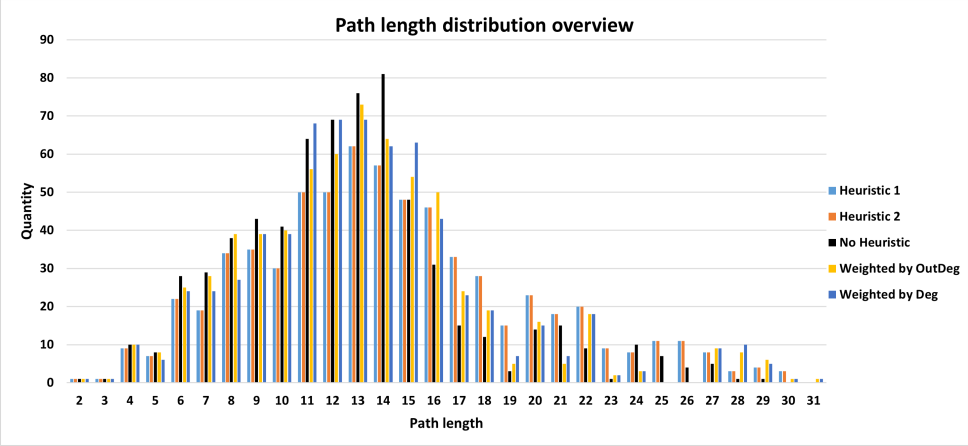

Figure 24: Distribution of path lengths for each approach set B

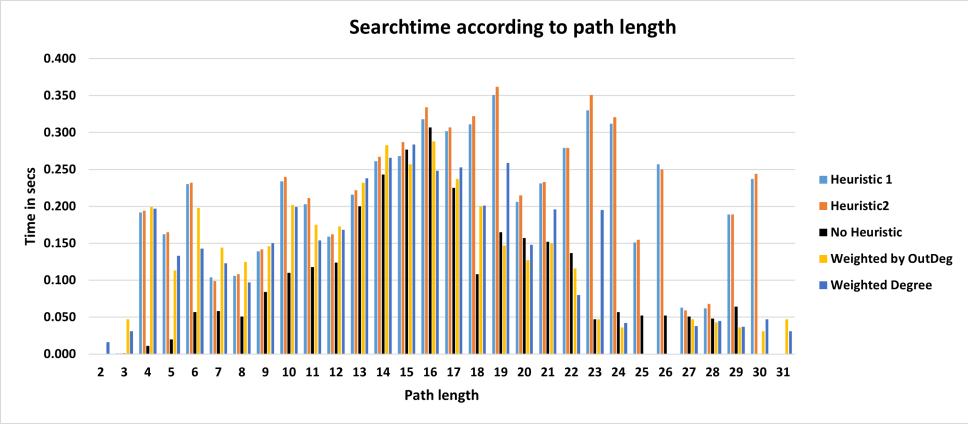

Figure 25: Average search time per path length for each approach set B

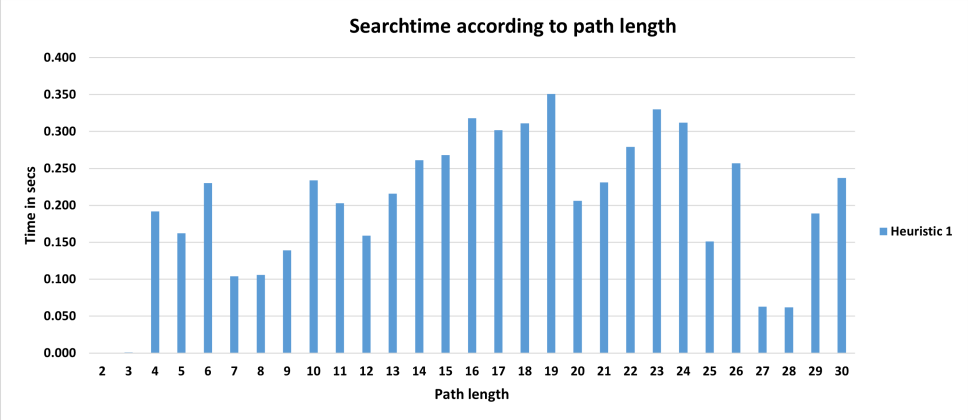

Figure 26: Average time per path heuristic 1 set B

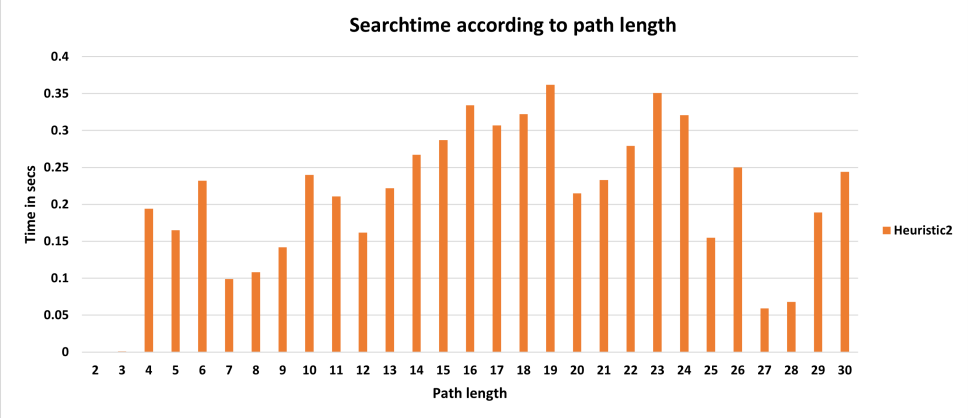

Figure 27: Average time per path heuristic 2 set B

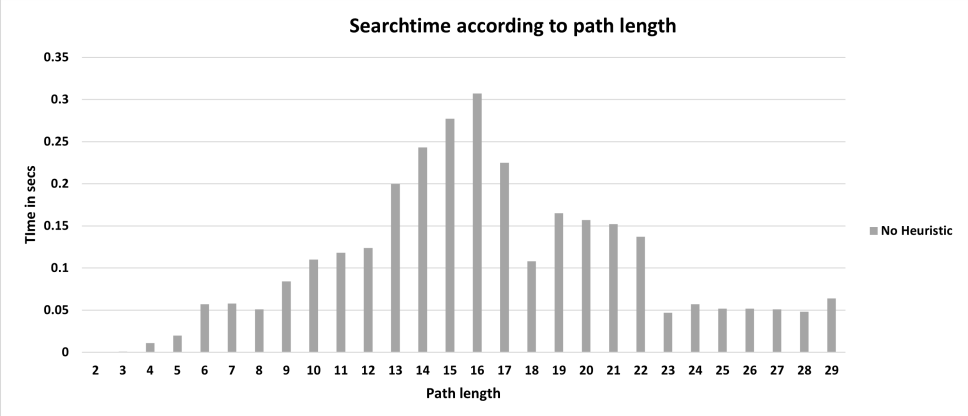

Figure 28: Average time per path no heuristic set B

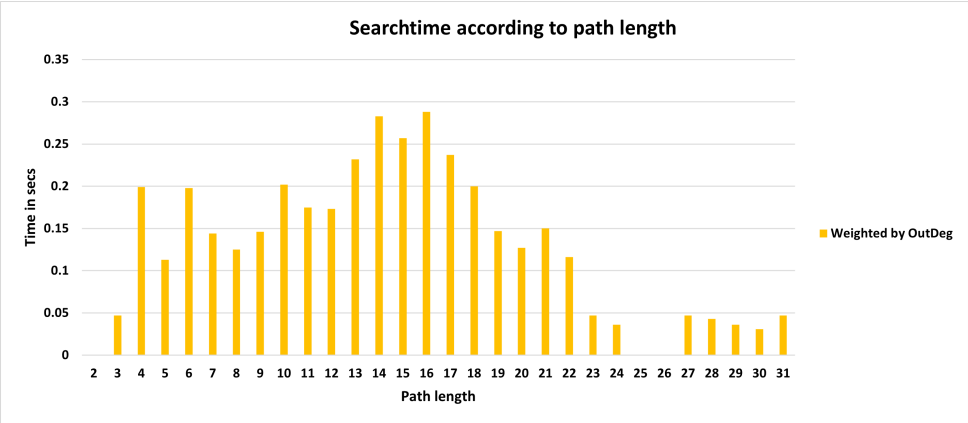

Figure 29: Average time per path weighted by outdegree set B

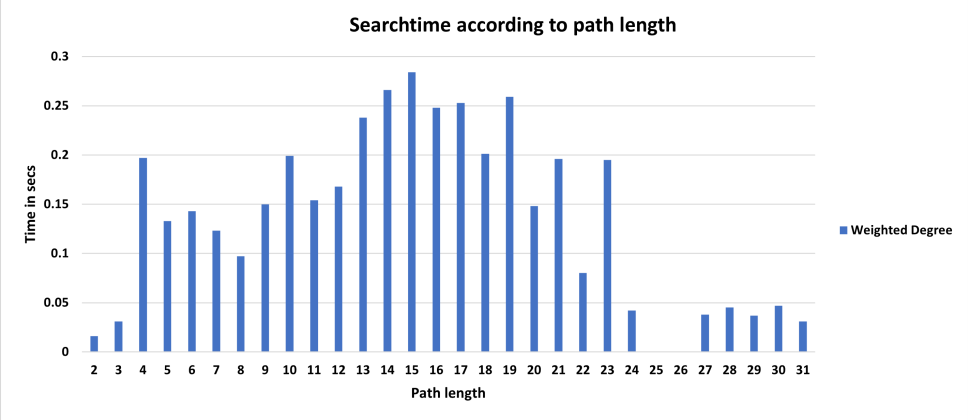

Figure 30: Average time per path weighted by degree set B

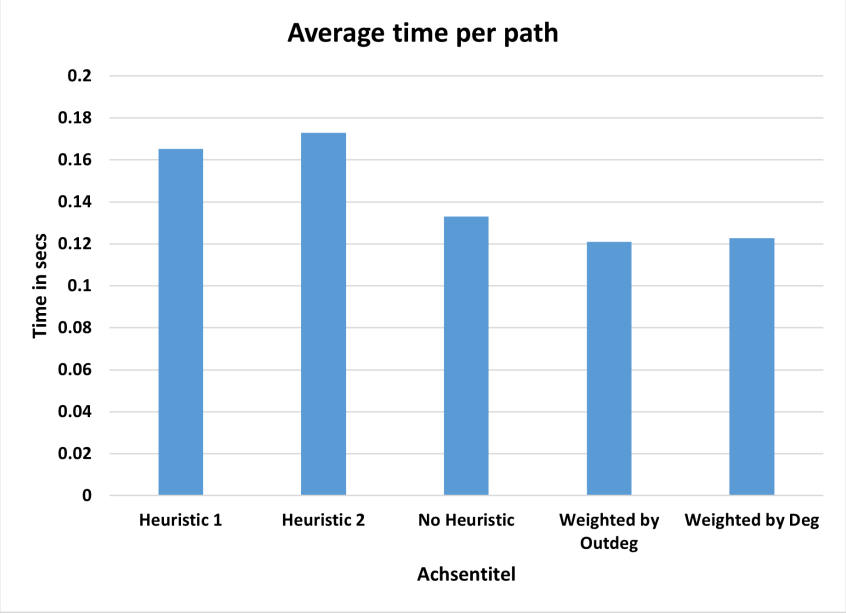

Figure 31: Average time per path for each approach set B2

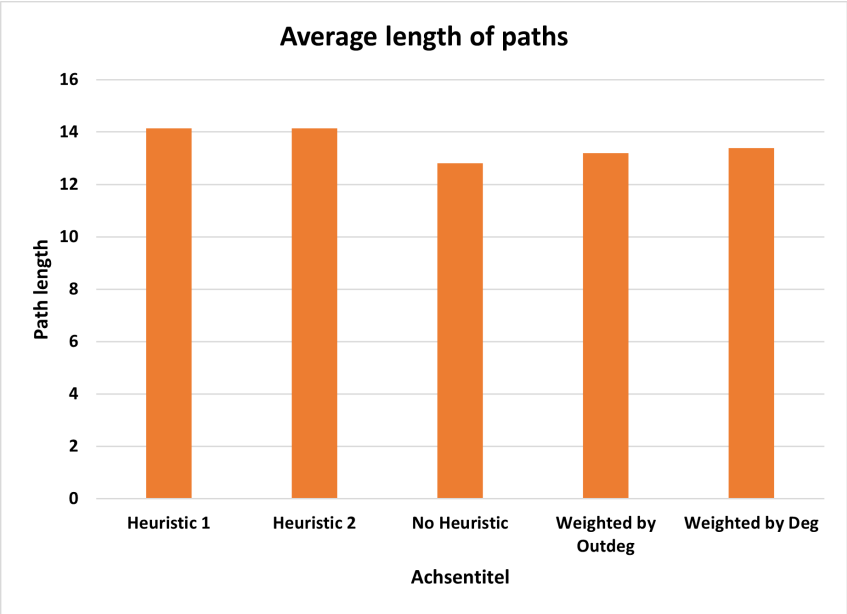

Figure 32: Average path length for each approach set B2

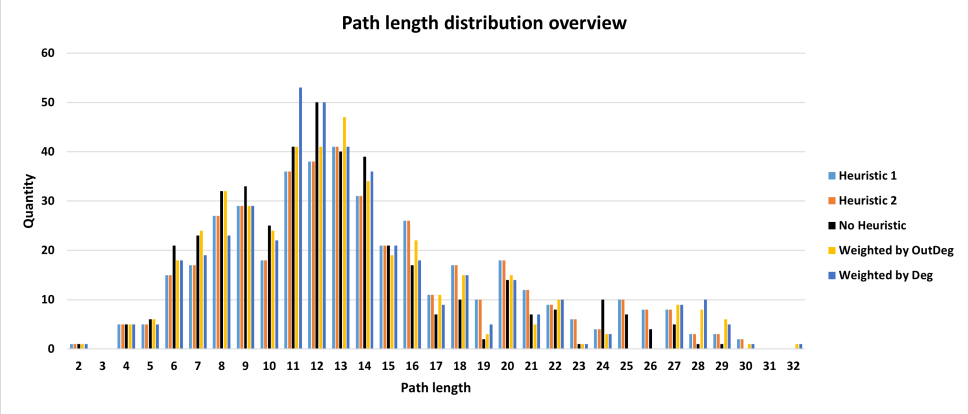

Figure 33: Distribution of path lengths for each approach set B2

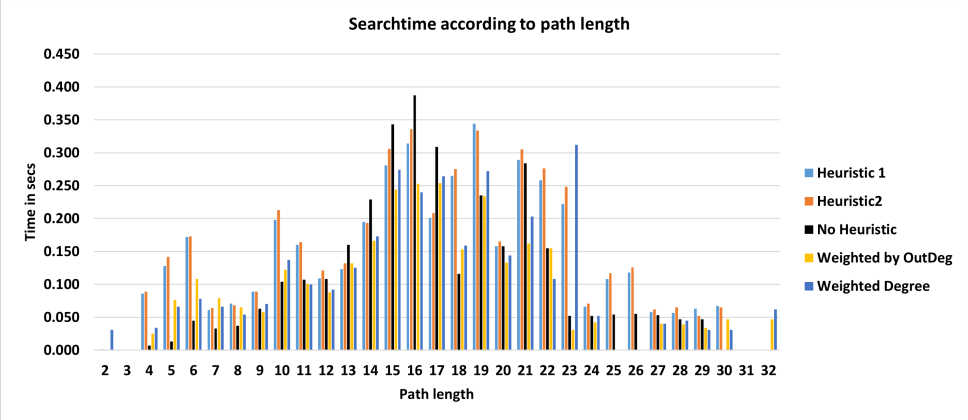

Figure 34: Average search time per path length for each approach set B2

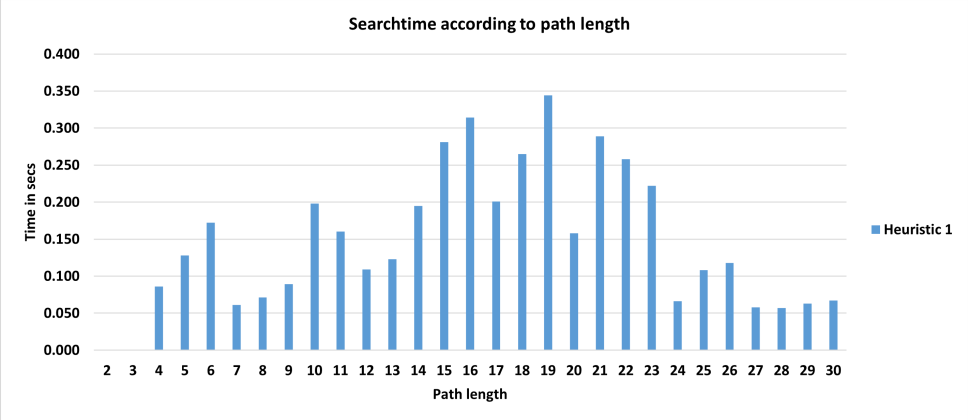

Figure 35: Average time per path heuristic 1 set B2

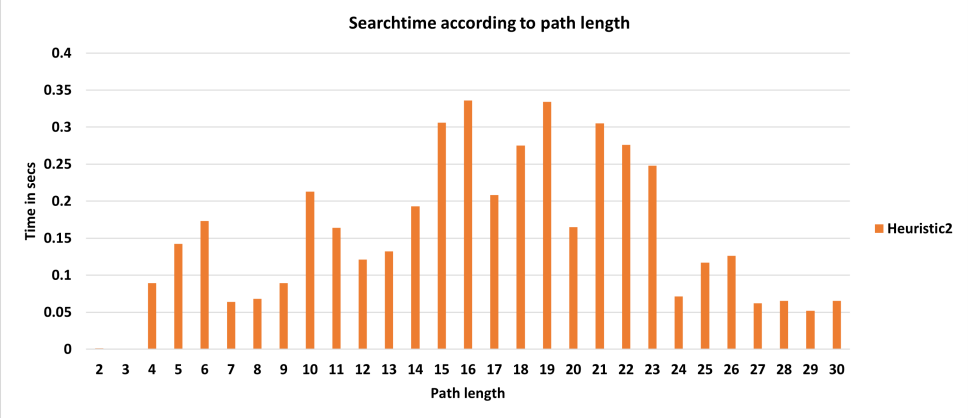

Figure 36: Average time per path heuristic 2 set B2

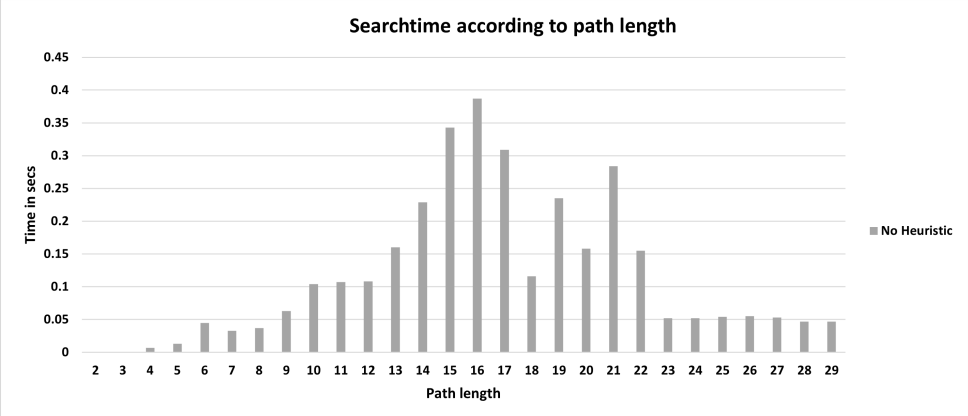

Figure 37: Average time per path no heuristic set B2

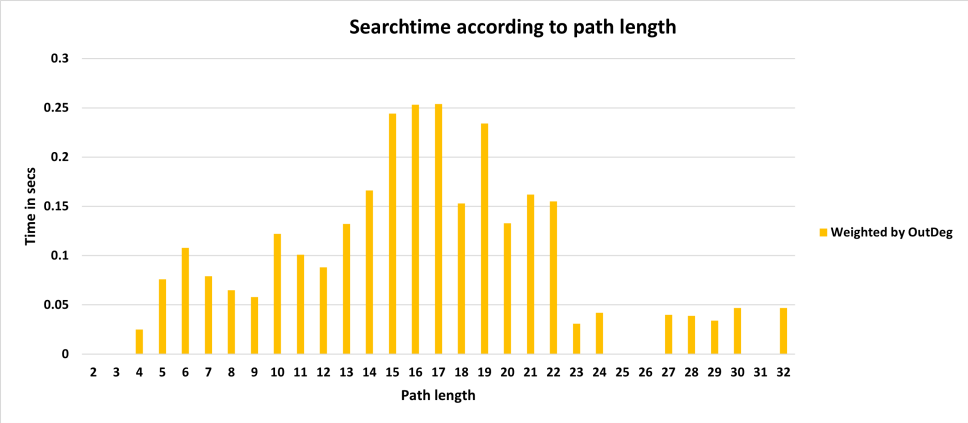

Figure 38: Average time per path weighted by outdegree set B2

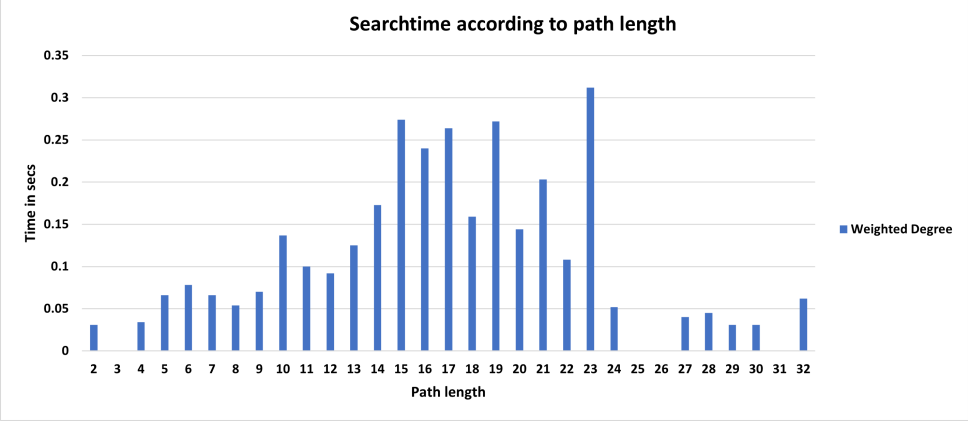

Figure 39: Average time per path weighted by degree set B2

5.3 Approach 3

The graph on which this technique was performed, was a real-world social network graph from Facebook. It was collected from survey participants using a Facebook app and represents a network of connections between friends [14]. It includes 4039 nodes, from which training pairs and test pairs have been created. It was performed three times with three different training and test sets. In order to get a balanced training set, the path lengths were limited differently each time. The originally imbalanced distribution of path lengths occurs, because a shortest path for a given pair carries nodes by keeping shortest distance between them. Respecting that fact, the BFS search also retrieves nodes that are included in the shortest path for the given training pair. Tables 6 to 8 give an overview of the distribution for each training set.

Table 6: Path length distribution

Path length	**Number of paths**
2	389079
3	318832
4	210796
5	71726
6	19747

Table 7: Path length distribution 2

Path length	**Number of paths**
4	227948
5	81958
6	18732

Table 8: Path length distribution

Path length	**Number of paths**
4	222186
5	86943
6	24001
7	7461
8	437

Table 9: Average MAE and RMSE for each training set

	2≤ length≤6	**4≤ length≤6**	**4≤ length≤8**
MAE	0.642	0.336	0.319
RMSE	0.931	0.659	0.635

Table 9 shows the average MAE and the RMSE for each training set. It can be seen, that the errors are very low for all sets. However, the set with a limited path length of $4 \leq length \leq 8$ performed best with an average MAE of 0.319 and a RMSE of 0.635.

6 Conclusion

6.0.1 Summary

In this work, the problem of graph traversal in Knowledge Graphs has been addressed, particularly for DBpedia.

Two graph traversal algorithms namely Dijkstra and A-star have been analysed and compared in terms of search time and path length. For the Dijkstra algorithm to work, the KG of the given dataset was weighted based on the degree of nodes beforehand. The two heuristics that were designed for the A-star algorithm exploited the entity types and the hierarchical structure of the DBpedia classes [7]. Depending on the test set, both approaches could eventually outperform the baseline in search time, but had higher average path lengths.

The introduced method from 4.3 produces shortest distances for the large majority of node pairs, matching the ground truth only with a very small deviation. It is a method that can possibly be applied for large graphs since it returns the results in linear time.

6.0.2 Future Work

Possible future directions of the current work are:

- Perform BFS, DFS, Floyd Warshall and Bellman-Ford on Knowledge Graphs such as DBpedia.
- Perform the suggested methods from 4.1 and 4.2 on other KGs than DBpedia.
- Perform the technique from 4.3 on larger graphs such as the one which was used for approach 4.1 and 4.2.
- Use the technique from 4.3 with different parameters such as embedding size or different node embedding techniques like e.g. HARP [6].

References

[1] Databases and Information Systems. Max Planck Institute for Informatics, URL: https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago, 2018.

[2] Facts and Figures. DBpedia URL: https://wiki.dbpedia.org/about/facts-figures, 2020.

[3] Bast, H., Bäurle, F., Buchhold, B., and Haussmann, E. Easy access to the freebase dataset. pp. 95–98.

[4] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. *Scientific American* (2001).

[5] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. Dbpedia - a crystallization point for the web of data. *Journal of Web Semantics 7*, 3 (2009), 154 – 165. The Web of Data.

[6] Chen, H., Perozzi, B., Hu, Y., and Skiena, S. Harp: Hierarchical representation learning for networks, 2017.

[7] DBpedia. Ontology Classes, 2016. URL: http://mappings.dbpedia.org/server/ontology/classes/.

[8] Duan, Y., Shao, L., Hu, G., Zhou, Z., Zou, Q., and Lin, Z. Specifying architecture of knowledge graph with data graph, information graph, knowledge graph and wisdom graph. In *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)* (2017), pp. 327–332.

[9] Grover, A., and Leskovec, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining* (2016), pp. 855–864.

[10] Hawke, S., Herman, I., Archer, P., and Prud'hommeaux, E. RDF Schema 1.1, 2013.

[11] Hawke, S., Herman, I., Archer, P., and Prud'hommeaux, E. W3c semantic web activity. URL: https://www.w3.org/2001/sw/, 2013.

[12] Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., and Stegemann, T. Relfinder: Revealing relationships in rdf knowledge bases. vol. 5887, pp. 182–187.

[13] Lehmann, J., Schüppel, J., and Auer, S. Discovering unknown connections - the DBpedia relationship finder. In *Proceedings of 1st Conference on Social Semantic Web. Leipzig (CSSW'07), 24.-28. September* (September 2007), vol. P-113 of GI-Edition of *Lecture Notes in Informatics (LNI)*, Bonner Köllen Verlag.

[14] Leskovec, J., and Mcauley, J. J. Learning to Discover Social Circles in Ego Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 539–547.

[15] Lämmel, U., and Cleve, J. *Künstliche Intelligenz*. Carl Hanser Verlag, 2012.

[16] McBride, B., Klyne, G., and Caroll, J. J. Resource Description Framework (RDF): Concepts and Abstract Syntax, 2004.

[17] Mirza, A., Nagori, M., and Kshirsagar, V. Constructing knowledge graph by extracting correlations from wikipedia corpus for optimizing web information retrieval. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (2018), pp. 1–7.

[18] Ruohonen, K. *Graph Theory, Lecture notes for TUT Finlandia*. 2013.

[19] Salehi Rizi, F., Schloetterer, J., and Granitzer, M. Shortest path distance approximation using deep learning techniques. pp. 1007–1014.

[20] Savenkov, V., Mehmood, Q., Umbrich, J., and Polleres, A. Counting to k or how sparql1.1 property paths can be extended to top-k path queries. In *Proceedings of the 13th International Conference on Semantic Systems* (New York, NY, USA, 2017), Semantics2017, Association for Computing Machinery, p. 97–103.

[21] Sofronova, R. Fine-grained Type Prediction of Entities using Knowledge Graph Embeddings. KIT, Karlsruhe.

[22] Stotz, R. Der Bellman-Ford-Algorithmus. Technischen Universität München URL: `https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_de.html`, 2013.

[23] Tartari, G., and Hogan, A. Wisp: Weighted shortest paths for rdf graphs. In *VOILA@ISWC* (2018), V. Ivanova, P. Lambrix, S. Lohmann, and C. Pesquita, Eds., vol. 2187 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 37–52.

[24] Velden, L. Der A* Algorithmus. Technischen Universität München URL: `https://www-m9.ma.tum.de/graph-algorithms/spp-a-star/index_en.html`, 2014.

[25] Velden, L. Der Dijkstra-Algorithmus. Technischen Universität München URL: `https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html`, 2014.

[26] Voroncovs, A. Der Floyd-Warshall-Algorithmus. Technischen Universität München URL: `https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_de.html`, 2015.

[27] Yu, L. *A Developer's Guide to the Semantic Web*. Springer, Berlin, 2011.

Assertion

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, November 2, 2020 VORNAME NACHNAME