Path-based Graph Indexing for Keyword Search on RDF data

Discovering Concepts Through Community
Detection

Audun Arnessønn Sæther

Master of Science in Informatics
Submission date: March 2017
Supervisor: Trond Aalberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

RDF graphs are structured data that is designed to be used by computers for reasoning. However, such data may be of interest to people as well. Therefore has keyword search on RDF graphs been introduced. The current approaches to keyword search, on both RDF graphs and other graphs, typically finds solutions that connects keyword matching nodes with minimal cost. Minimal costs could, for instance, be the shortest paths between the nodes. Minimal solutions, however, may not always be the most informative answers to a query. Returning parts of the graph that constitute concepts corresponding to how humans conceptualize the world, could give answers that better satisfies the information need. This thesis has explored if such concepts can be discovered automatically in an RDF graph. To this end, we have employed algorithms for finding closely connected groups of nodes in a graph. These groups of nodes are called *communities*, and finding them is called *community detection*.

A state of the art review examining approaches to keyword search on graphs in general, and RDF graphs in particular, was carried out. Additionally, we performed a review on current algorithms for finding overlapping communities in a graph. Based on this review, we did a preliminary study where the most relevant of these algorithms were applied to some RDF graphs. Through the study, we found some weaknesses in the direct application on RDF graphs, with regard to our stated objective of finding concepts. This was related to the special nature of RDF graphs, compared to other graphs. In particular, we found inconsistencies in the discovered communities with regard to class information, and the meaning that is found in predicate edges. These inconsistencies prevented us from describing the found communities as logical concepts.

To eliminate these weaknesses, we developed a novel community detection algorithm for finding concepts in an RDF graph. This new algorithm defines communities in RDF graphs as a set of paths, not as a collection of nodes. We run our community detection algorithm on a sample of the nodes in the graph to find the paths included in communities, and then aggregate the results. The paths that are most frequently included in a community are kept, and used to build concepts. Our algorithm ensures that the concepts are both consistent internally in a community, and consistent across the dataset.

The feasibility of our approach were shown through experiments. Furthermore, the usefulness of the approach is argued through a proof-of-concept search solution that uses the concepts found by the community detection algorithm. We find that the concepts discovered by our algorithm can enhance the answers to queries, when compared to corresponding minimal solutions.

Sammendrag

RDF-grafer er strukturert data som er designet for datamaskiner slik at disse kan trekke slutninger ut fra dataene. Denne typen data er likevel interessant for mennesker. Derfor har fritekstsøk i RDF-grafer blitt introdusert. Nåværende tilnærminger til fritekstsøk, både i RDF-grafer og andre grafer, finner typisk løsninger som kobler sammen de nodene som inneholder søkeord med mål om at svaret skal ha minimal kostnad. Minimal kostnad kan for eksempel være kortest mulig lengde på stiene mellom nodene. Minimale løsninger er ikke nødvendigvis alltid de mest informative svarene på en spørring. I stedet kan deler av grafen som svarer til konsepter slik mennesker konseptualiserer verden, gi svar som bedre tilfredsstiller informasjonsbehovet. Denne masteroppgaven har derfor undersøkt om slike konsepter kan oppdages automatisk i en RDF-graf. For å oppnå dette har vi anvendt algoritmer for å finne tett sammenkoblede grupper av noder (*communities*) i grafer. Prosessen med å finne communities kalles for *community detection*.

En gjennomgang av ulike tilnærminger til fritekstsøk i grafer generelt og RDF-grafer spesielt har blitt utført. I tillegg har vi sett på algoritmer for å finne overlappende communities i grafer. Basert på denne gjennomgangen gjorde vi en forstudie der noen av algoritmene ble testet ut på RDF-grafer. Forstudien avdekket svakheter ved å bruke slike algoritmer direkte på RDF-grafer, med hensyn til målet vårt om å finne konsepter. Dette var knyttet til RDF-grafers særegenheter sammenlignet med andre grafer. Vi fant uregelmessigheter i de oppdagede communities som særlig gjaldt klasseinformasjon og informasjon i predikatkantene. Uregelmessighetene gjaldt både innad i et community, og på tvers av ulike communities. Dette gjorde at vi ikke kunne beskrive de communities vi fant som logiske konsepter.

For å bøte på disse svakhetene utviklet vi vår egen algoritme for å finne communities i en RDF-graf. Denne nye algoritmen definerer communities i RDF-grafer som et sett av stier, og ikke som en samling noder. For å finne disse stiene kjøres algoritmen på et utvalg av nodene i grafen før resultatene summeres opp. De stiene som oftest blir inkludert i et community beholdes, og brukes til å bygge konsepter. Algoritmen sørger for at konseptene er konsistente både internt i et community og over hele datasettet.

Gjennomførbarheten til tilnærmingen har vi vist gjennom eksperimenter. I tillegg har vi laget en proof-of-concept søkeløsning som benytter konseptene funnet av vår algoritme, og som viser nytteverdien av tilnærmingen. Vi finner at konseptene oppdaget av vår algoritme kan føre til bedre svar på fritekstspørringer sammenlignet med tilsvarende minimale løsninger.

Preface

This master thesis was written in the fall of 2016 and the spring of 2017 at the Norwegian University of Science and Technology (NTNU). It is the culmination of my degree in Master of Science in Informatics, with databases and search as field of study. The supervisor of this thesis has been Trond Aalberg at the Department of Computer and Information Science.

I would like to thank my supervisor for input and feedback throughout the process. Additionally, I would like to thank my family for continuous support. A special thanks go to my sister for proofreading the thesis.

Audun A. Sæther

Trondheim, March 2017

Contents

List of Tables

List of Figures

List of Algorithms

1 | Introduction

1.1 Motivation

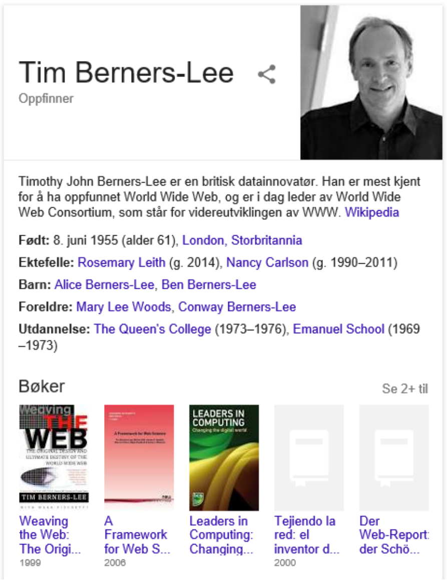
The Semantic Web is a term coined by Tim Berners-Lee that encompasses a vision of a web where computers can understand the meaning of web content and use this to carry out advanced tasks [2]. Berners-Lee noted that the current web mainly consists of documents meant to be understood by people, and not information that machines can process. To provide machines with the power to reason about data, a number of components of the Semantic Web have been introduced. For this thesis, the *Resource Description Framework* (RDF)[1] is the most relevant. RDF data is sets of triples that makes assertions about things. For instance, a triple could be <Tim Berners-Lee *isAuthorOf* "The Semantic Web">. Tim Berners-Lee and "The Semantic Web" are things that have the relationship *isAuthorOf*. RDF data is often presented as graphs. A more thorough description of RDF follows in Section 2.2.

While RDF is designed with machines in mind, such structured data is also of interest to people. However, as is often the case with structured data, exploring it requires knowledge of not just the data, but also how it is stored and how to query it. Additionally, structured data requires the user to be exact to get the desired information. In order to remedy this, keyword search on RDF graphs has been introduced [6, 9, 11, 19, 24, 31, 46]. Keyword search on RDF graphs aims to increase the usability of RDF data by returning ranked results. Results are typically substructures (either subtree or subgraph) of the graph. The goal is usually to find a minimal substructure that covers the input keywords. Current approaches often build solutions to queries at runtime, which requires indexing approaches that avoids performance bottlenecks. In this thesis, we claim that the quality of answers could be improved by finding structures in the graph that corresponds to logical concepts, rather than minimal solutions.

Google noted, when introducing their Knowledge Graph, that the keywords in a query are not only strings, they are also things [44]. The goal of search then becomes to return information about the thing(s) represented in the keywords, not a ranked list of resources that best matches the strings (words). Therefore, a good answer to the keyword query "Tim Berners-Lee" is not a list of resources that may or may not have some interesting

[1] `https://www.w3.org/RDF/`

information about him, or the minimum substructure that contains all terms in the query, but instead information about Tim Berners-Lee; how old he is, what he is famous for, etc. Google's Knowledge Graph provides this information directly, and gives a summary of the most relevant information about him (see Figure 1.1). The Knowledge Graph also provides links to other objects in the graph, and in that sense it is a realization of a web of data.



Tim Berners-Lee
Oppfinner

Timothy John Berners-Lee er en britisk datainnovatør. Han er mest kjent for å ha oppfunnet World Wide Web, og er i dag leder av World Wide Web Consortium, som står for videreutviklingen av WWW. Wikipedia

Født: 8. juni 1955 (alder 61), London, Storbritannia

Ektefelle: Rosemary Leith (g. 2014), Nancy Carlson (g. 1990–2011)

Barn: Alice Berners-Lee, Ben Berners-Lee

Foreldre: Mary Lee Woods, Conway Berners-Lee

Utdannelse: The Queen's College (1973–1976), Emanuel School (1969 –1973)

Bøker — Se 2+ til

Weaving the Web: The Origi... 1999
A Framework for Web S... 2006
Leaders in Computing: Changing...
Tejiendo la red: el inventor d... 2000
Der Web-Report: der Schö...

Figure 1.1: Google Knowledge Graph search result for query "Tim Berners-Lee"

Google has the luxury of having a large number of users, and an even larger number of searches. Through this, they can decide what the most relevant information for a query is (for instance through user clicks). Lacking a search solution, the most relevant information must be inferred from the data itself. This can be a hard task. The goal, however, should be the same: to move the heavy lifting of search from the user to the search system.

1.2 Research Questions

Following the motivation given above, the aim of this thesis is to explore new ways to improve keyword search on Semantic Web data, that is, RDF graphs. The main research question that guide the work is:

- **How can indexing of RDF graphs be implemented to better support keyword search and retrieval that corresponds to how human conceptualize the world?**

The goal is to develop new methods, or algorithms, that take into account that the information need in a query can be greater than what a couple of keywords suggest. The idea is that a query may refer to things, or concepts, where the user has a notion of which pieces of information constitutes the concept. For instance, the query "Avatar" should return a result that includes information on who directed the movie, as the concept *movie* includes the director as an important attribute. However, in RDF, we may have one node representing the movie, and another representing the director. The challenge is that the query "Avatar" match the movie node, but not the director node. Approaches that seeks minimal solutions will only return the movie node. If we are to present the user with who directed this movie, we must decide which nodes around the Avatar node should be included in the answer to the query. This is not trivial to deduce automatically from data. For instance, the movie may be a realization of a book. Even though both the director and the book could be just one step away from the movie in the graph, it may be argued that the concept of a movie is more strongly connected to director than book. This is because all movies have a director, but not all movies are realizations of a book. In addition, to be a director you have to direct one or more movies, while books can exists independently of movies. The fact that humans probably not agree on which building blocks constitute different concepts only increases the difficulty of the task.

In order to answer the main research question, the following subquestions are addressed:

- **RQ1: What is the state of the art for keyword search on RDF graphs?**
- **RQ2: How can graph theory be used to automatically find concepts in RDF graphs?**
- **RQ3: How can automatically found concepts best be indexed to support keyword search on RDF data?**

The review of the state of the art look at both general approaches to keyword search on graphs and approaches directed specifically towards RDF graphs. Included in the review is a look at current RDF triple stores, and the possibilities for keyword search found there.

The main goal of this thesis, then, is to explore if it is possible to automatically discover concepts in RDF graphs. Relevant graph theory is explored to this end. Given the concepts found by the use of graph theory, we look at how these can be indexed so that keyword search is best supported.

1.3 Research Approach

The goal of the research is to design new methods, which is consistent with the research strategy of design and creation. As Briony J. Oates has described, design and creation is about developing new IT artifacts, including methods and instantiations [35]. In this thesis, both new methods and an instantiation demonstrating the methods are developed. Oates describes five stages of the design and creation process: awareness, suggestion, development, evaluation and conclusion. Our research process follows these stages in what resembles an iterative cycle.

Based on a state of the art review, existing solutions, and ideas emerging from discussions with the supervisor in the awareness stage, suggestions on new methods are made. Then, the methods are developed and evaluated, corresponding to the development and evaluation stages. The development includes both a new method and an instantiation of it. The evaluation is done against existing approaches, and against each other. In the early iterations, the evaluations are carried out in less rigorous fashion than in later iterations. The goal is to quickly judge if the method in question is worth further development, or if a new path of development should be explored. Each iteration does not necessarily start with a fresh approach. It can be a slight adjustment of a previous method. The conclusion stage of each iteration decides the course of action in the next iteration. Hopefully, the quality of the methods improves in the course of the iterations. In the end, the most promising method(s) are reviewed and tested more thoroughly on different datasets. From these tests, a conclusion on the usefulness of the method(s) is reached.

As the goal of the research is to improve answers to keyword queries that contains human conceptualizations, the data analysis is partly qualitative. Whether or not *director* belongs to the concept *movie* does not have a clear-cut answer. Studying the datasets and judging if the returned result is satisfactory can be an interpretative task for the researcher. Information retrieval metrics, a quantitative approach, is also employed. In addition, the methods is evaluated on performance, which is a quantitative analysis.

1.4 Thesis Structure

Chapter 2 provides the necessary theoretical background with regard to graph theory and RDF. Additionally, a state of the art review for keyword search on graphs, both RDF graphs and others, is performed. Keyword search capabilities of existing triple stores are also reviewed.

Chapter 3 describes a preliminary study where existing community detection algorithms are tested on RDF graphs.

Chapter 4 presents our new community detection algorithm for indexing RDF graphs.

Chapter 5 presents a proof-of-concept search solution utilizing the new algorithm.

Chapter 6 summarizes and discusses the results of our work. A conclusion on the research questions is reached.

2 | Background

2.1 Graph Theory

In this section, relevant concepts and definitions related to graphs are presented. We follow the definitions and notation from [15].

2.1.1 Definitions

A **graph** $G = (V, E)$ consists of two sets V and E, where V are **vertices** (or **nodes**) and E are **edges**. An edge is associated with a set of one or two nodes. Two nodes are **adjacent** if they are joined by an edge. If an edge connects a node to itself, we call it a **loop**. A **multi-edge** is a collection of two or more edges with identical node sets. A **simple graph** is a graph with no loops or multi-edges. The **degree** of a node in a simple graph is the number of neighbors, while it in the general case is the number of edges incident on the node plus twice the number of loops.

A **directed graph** is a graph where each of the edges has direction. A **directed edge** is directed from one node u (the **tail**) to another node v (the **head**). Such an edge is an ordered pair of nodes, often denoted (u, v). In a directed graph, we may have sources and sinks. A **source** is a node with zero indegree, which means that no edges are directed to it. If a node has no edges directed from it, and thus outdegree zero, it is a **sink**. In addition to direction, an edge could also be assigned a weight, creating a **weighted graph**. The weight can, for instance, be used to indicate the length of the edge, or the cost of traversing it.

A **walk** in a graph is an alternating sequence of nodes and edges,

$$W = v_0, e_1, v_1, e_2, ..., e_{n-1}, v_{n-1}, e_n, v_n$$

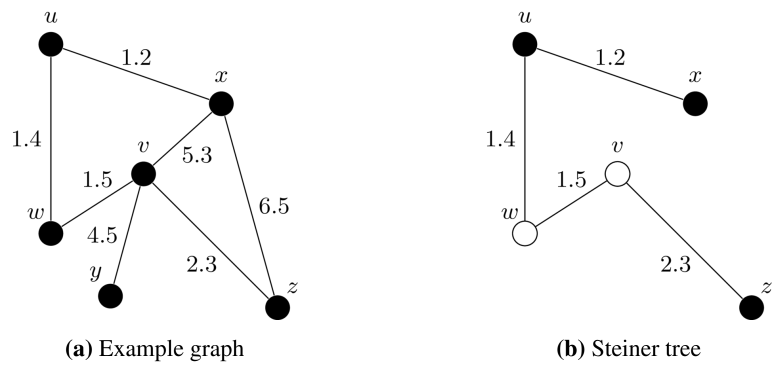
such that for $j = 1, ..., n$, the nodes v_{j-1} and v_j are endpoints of the edge e_j. The **length** of a walk is the number of edges from the start node (v_0) to the end node (v_n). The distance between two nodes is the length of the shortest walk between them. If the initial node is also the final node in a walk, we have a **closed** walk. If no edge occurs more than once in a walk and no internal node (all nodes except v_0 and v_n) is repeated, we have a **path**. A closed path with length of at least 1 is called a **cycle**.

A graph is **connected** if for every pair of nodes there exists a walk between the nodes. If we have a graph G and a graph H such that $V_H \subset V_G$ and $E_H \subset E_G$, then H is a **subgraph** of G. An **induced subgraph** is a graph where $V_H \subset V_G$, and which contains all edges in G whose endpoints are in V_H. A simple graph is **complete**, if there for every pair of nodes exists an edge that joins them. If a subgraph is complete, we call it a **clique**. For some purposes, cliques are required to be maximal, that is, no nodes can be added to it and still keep it a clique.

A graph is **dense** when the number of edges is close to $|V|^2$ and **sparse** when the number of edges is much less than $|V|^2$. The density $D(G)$ of a simple undirected graph G is given by

$$D(G) = \frac{2|E|}{|V|(|V|-1)} \tag{2.1}$$

A **tree** is a connected graph with no cycles. If a tree, produced from a graph, connects all nodes in the graph with minimum total edge weight, we call it a **minimum spanning tree**. A **Steiner tree** is a tree of minimum weight which connects a set of nodes. This set of nodes, called terminals, is a subset of V. If the subset consist of two terminals, the problem of finding a Steiner tree is equivalent to finding the shortest path between the two. If the subset is equal to V, the problem is equivalent to finding the minimum spanning tree. The requirement for a Steiner tree is that it contains all the terminals, but it can contain additional nodes from the graph not in the terminals set. It is also allowed to construct connections points that reduce the cost. Non-terminals that are constructed or added are called Steiner points. Figure 2.1 shows an example of a Steiner tree for the terminals set $\{u, x, z\}$ which includes the Steiner points $\{v, w\}$. The minimal Steiner tree problem is NP-complete [22].



(a) Example graph **(b)** Steiner tree

Figure 2.1: A weighted graph and a Steiner tree for the terminals set $\{u, x, z\}$

2.1.2 Graph Traversal

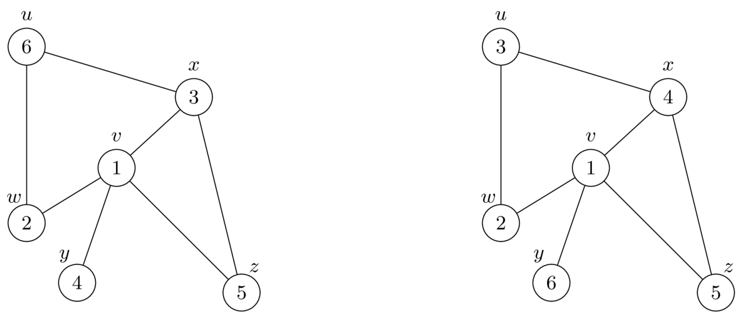
Breadth-first search

Breadth-first search (BFS) traverses the graph from a start node by visiting all nodes at depth d before visiting the nodes at depth $d + 1$. Depth is the number of edges from the start node. Implementations of the algorithm often uses a queue, initialized with the start node, to hold unexplored nodes. As long as the queue is not empty, the first node of the queue is removed from the queue and explored for neighboring nodes. The neighboring nodes that are not already explored are added to the queue, so that all nodes at depth $d+1$ are behind the nodes at depth d in the queue. BFS has time complexity $O(|V| + |E|)$ for a graph $G = (V, E)$. Another way of describing the run time of BFS is $O(b^d)$, where b is the branching factor (the number of neighboring nodes of each node). Figure 2.2a shows an example of BFS.

BFS is the basis of other algorithms, one of which is Dijkstra's single-source shortest-paths algorithm. This algorithm finds the shortest path from a source to all other nodes in a graph [10]. It may be seen as a generalization of BFS, since BFS find the shortest paths if all edges have weight 1, while Dijkstra's algorithm works for all nonnegative edge weights. The time complexity is $O(|E| + |V| \log |V|)$, which is worse than BFS because a priority queue is needed instead of a regular queue, so that unvisited nodes can be sorted in ascending order of total edge weight.

Depth-first search

Depth-first search (DFS) repeatedly explores an edge incident to the most recently visited node that still has unexplored edges. Thus, instead of newly discovered edges finding their place at the end of the queue like in BFS, they are pushed to the top of the stack. Again, the time complexity for traversing the graph is $O(|V|+|E|)$. An example of DFS is shown in Figure 2.2b, and can be compared to the BFS approach.



(a) Breadth-first search **(b)** Depth-first search

Figure 2.2: Example of BFS and DFS from start node v. Numbers indicate order of discovery. Nodes are visited alphabetically if otherwise equal.

2.2 RDF

RDF is a data model used to represent subject-predicate-object triples, where a predicate describes a relationship between the subject and object [11]. Table 2.1 shows an example collection of RDF triples, also called statements. A triple collection forms a directed, labeled graph, with subjects and objects as nodes, and predicates as edges. The graph representation of the data in Table 2.1 is shown in Figure 2.4. An RDF graph can contain both loops and multi-edges, and is therefore not a simple graph.

RDF graphs have some special characteristics that need to be taken into account. As can be seen from the graph in figure 2.4 and the table of RDF data, there are different kinds of nodes in RDF. The W3C[1] notes that are three kinds of nodes: IRIs[2], literals and blank nodes [47]. IRIs and literals are resources, that is, something that exists in the world. While IRIs are referents (typically an URL), literals contain data with a data type (such as string or date) and a defined range. Blank nodes do not refer to resources, and are used to state that a relationship exists without having to name it. An important feature of RDF graphs are the labeled edges. An edge carry information that can be used to interpret the nodes which it joins. In the example dataset we have the literal node 2009, which is connected to the IRI node for the movie Avatar. Without knowing that the label on the edge is "year_of_release", the value of 2009 can be interpreted differently. It could, for instance, have been the length of the movie measured in seconds, or it could have been the income the movie made in million dollars.

Querying an RDF graph is usually done by matching triple patterns using the standard RDF query language SPARQL[3]. For instance, to find the names of movies directed by James Cameron released in 1984, the SPARQL query would be:

```
SELECT ?movie
WHERE {
        ?movie year_of_release 1984 .
        ?movie director ?director .
        ?director name "James Cameron"
}
```

Figure 2.3: SPARQL query for finding movies directed by James Cameron released in 1984

The "?" indicates variables that can take on any value that match the given triples. In this case, the "?movie"-variable needs to have a year_of_release literal with value 1984 and a director relationship. The object of the director relationship is denoted by the variable "?director" and needs to have a name literal with value "James Cameron".

[1]The World Wide Web Consortium. A community that develops open standards on the web.
[2]Internationalized Resource Identifiers : `https://tools.ietf.org/html/rfc3987`
[3]`https://www.w3.org/TR/sparql11-overview/`

Table 2.1: Example collection of RDF triples

Subject	**Predicate**	**Object**
film/terminator	name	"The Terminator"
film/terminator	year_of_release	1984
film/terminator	rdf:type	Movie
director/cameron	director	film/terminator
director/cameron	director	film/avatar
director/cameron	name	"James Cameron"
director/cameron	rdf:type	Director
Director	rdfs:subClassOf	Person
film/avatar	name	"Avatar"
film/avatar	year_of_release	2009
film/avatar	rdf:type	Movie

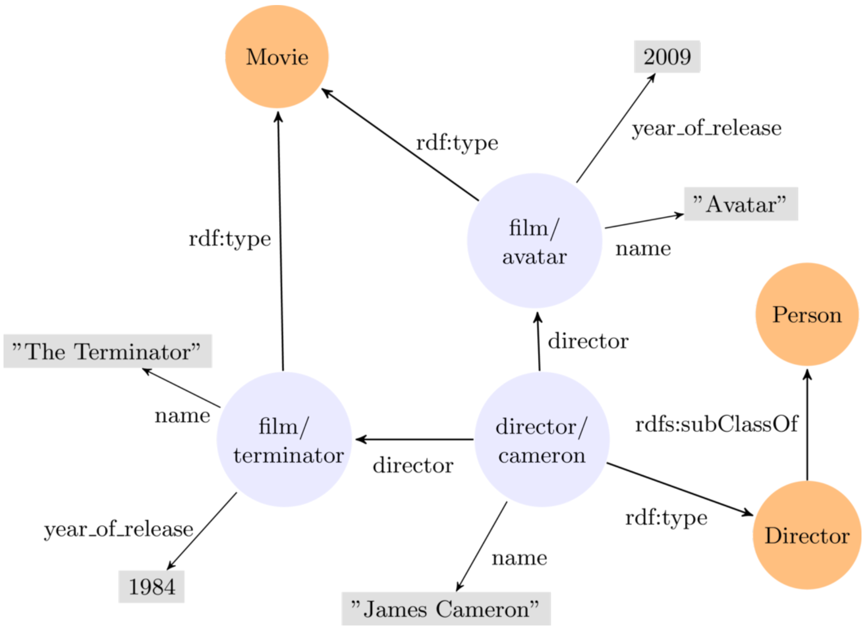


Figure 2.4: Graph representation of example RDF collection

2.2.1 RDF Schema

RDF Schema (RDFS)[4] is an extension of RDF that provides a vocabulary for data modeling the RDF data. RDFS is a somewhat lightweight version of an ontology language. An ontology language defines concepts and relationships in some area, and can be quite formal. The language facilitates inferring for machines, and is thus an important part of the vision of the Semantic Web. In the Semantic Web, OWL 2[5] is the standard ontology language. RDFS is less formal than OWL 2, but provides the possibility to describe classes and properties, and their domain and range. A class is stated using the rdf:type property[6]. Resources of a particular class are instances of that class. In Figure 2.4, Terminator and Avatar are instances of the class Movie, while James Cameron is an instance of Director. Classes can have subclasses. The property rdfs:subClassOf is used to state that all instances of one class are also instances of another. In the example, we see that Director is a subclass of Person, which implies that James Cameron is an instance of the class Person.

2.3 General Approaches to Keyword Search on Graphs

The problem of keyword search on RDF graphs can be generalized to the problem of keyword search on any graph: based on input query keywords, one or more substructures (subgraphs or subtrees) containing some or all of the keywords are returned [21]. Different proposed solutions often follow the same idea: to match query keywords to elements in the graph and create substructures based on connections between these elements [46]. A scoring function is then used to find the top-*k* answers to the query. Typically, smaller results are preferred, for instance minimal Steiner trees. As we saw in Section 2.1, a Steiner tree is a minimum spanning tree for a set of terminals, which may include non-terminals. For the keyword search problem, terminals can be nodes matching the query keywords, and the non-terminals can be roots or other nodes in the answer trees used to connect the keyword matching nodes. Since finding minimal Steiner trees is NP-complete, using polynomial approximation algorithms is often suggested. However, some of these algorithms require an examination of the whole graph, which may be both unnecessary and expensive [20]. Thus, approaches to keyword search on graphs often seek to explore as small a part of the graph as possible.

Many different kinds of data can be modeled as graphs. Relational data can be modeled as graphs, with tuples as nodes and edges corresponding to foreign key relationships. XML data is often represented as a tree with a root, but since XML documents can be linked together by IDREF/ID links they can also form graphs.

An approach to keyword search over XML documents was presented by Guo et al. [16]. In their system, XRANK, is a result of a query an XML element, not the entire document. The result XML element is the most specific result that contains all the query keywords.

[4]`https://www.w3.org/TR/rdf-schema/`

[5]The Web Ontology Language: `https://www.w3.org/TR/owl2-overview/`

[6]The prefix rdf refers to the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

Specific refers to the requirement that the XML element does not contain any subelement that also contains all the query keywords. Since the result of a query could be very specific and lacking context, XRANK supports user navigation to ancestors of the element and also the possibility for predefined answer nodes. Predefined answer nodes could ensure that a XML element such as <title> is always returned with its parent to provide context. The authors note that the creation of such answer nodes may require domain experts.

XML documents have a hierarchical structure that make them different from keyword search over HTML documents. In HTML documents, the distance between keywords is essential, while in XML it is also important to view the ancestor distance, that is, the height of the XML tree. Two keywords may be inside a common XML element, but one of them may be a direct child of the element, while the other may be far down in the hierarchy. While the keyword distance between them could be small, the ancestor distance could be large. Due to this, XRANK ranks more specific results higher than less specific results.

Liu and Chen [27] addressed the problem of returning meaningful results for keyword search on XML documents. They argued that the most specific match to a query is not always the most informative. By analyzing the XML data structure and the keyword match patterns, they seek to identify return nodes. Their approach starts by finding what they call VLCA (variant of lowest common ancestor) nodes. These are nodes whose subtree contains at least one match to every keyword. In their system, XSeek, they distinguish between nodes representing entities, and nodes representing attributes. To infer which nodes belongs to which category, they use heuristics. Specifically, they state that if a node has a one-to-many relationship with other nodes of name n, then n is most likely entity nodes. If a node has only one child, this child is most likely an attribute. Nodes that have neither of these characteristics, are connection nodes.

In addition to analyzing the XML data structure, XSeek also looks at keyword match patterns. In particular, they are interested in classifying keywords as search predicates or return nodes. Search predicates are keywords that restrict the search (similar to the *where* clause in SQL), while return nodes are keywords that state the desired output of the search (similar to the *select* clause in SQL). An input keyword k_1 is classified as a return node if the keyword matches a node name u, and if there do not exists an input keyword k_2 matching a node v such that u is an ancestor of v. If this requirement does not hold, the input keyword indicates a predicate.

Based on the analysis of the XML data structure and the keyword match patterns, search results are outputted. Both matches to return nodes and search predicates are outputted. The search predicates output the paths from a VLCA node to its descendant predicate matches. The return nodes are displayed on the basis of category; entity, attribute or connection node. For entity or connections nodes, all attribute children are outputted together with links to child entities. This is done to limit the search results and increase the user friendliness. Attribute nodes are outputted as name and value.

Qin et al. [40] propose a technique for keyword search on relational databases. Given a set of keywords, they find induced subgraphs that contain all keywords within a given distance. They call the induced subgraphs for communities[7]. The authors observe that finding minimal connecting trees can result in many answers, and answers that miss important information. Three types of nodes are described: (1) keyword nodes, which are nodes that matches a keyword; (2) center nodes, which are nodes that connect to every keyword node within a given radius (shortest path distance); (3) path nodes, which are additional nodes that appear on a path from a center node to a keyword node. A community is a directed graph with multiple center nodes. The authors argue that communities can give fewer and more informative answers to queries.

Another solution for keyword search on relational databases was proposed by Hristidis and Papakonstantinou [18]. Their DISCOVER system defines a solution to a query as minimal joining sequences of tuples. The joining of tuples happens from foreign keys in one table to primary keys in another table. Shorter sequences (that is, fewer number of joins) are considered better. If more than two keywords are present, the sequence may be representing as a network or tree of tuple joins. A maximum number of allowed joins is supplied by the user. The result of a query is a set of joining networks that are total (each keyword is contained in the network) and minimal (no tuple can be removed so that the network is still total).

Bhalotia et al. [3] presented an algorithm for graphs in general that searches backwards from a set of node sets, S, containing keyword matching nodes. S contains as many node sets as there are keywords, where S_i contains nodes relevant for the keyword term i. The backward search algorithm starts from each node in S and runs Dijkstra's shortest path algorithm concurrently from all. The graph in question is directed, and the algorithm traverses edges backwards from each start node. A result of a search is a rooted directed tree, where the root is a common node that has a path to at least one node in each node set in S. Each run of Dijkstra's algorithm has an iterator. At each iteration of the algorithm, one of these iterators is picked to expand one step further backwards. This selection is done based on the iterator whose next node has the shortest path to its source node. After this node is explored, the intersection of node lists for all the iterators is found. If there exists some common node(s) that reach all keywords, a result is found.

The backward search approach may explore an unnecessary large part of the graph if the keywords match many nodes, or if an iterator reaches a node with many links to other nodes. Kacholia et al. [20] therefore proposed a bidirectional search algorithm. This approach uses only one iterator, called an incoming iterator, for the backward search. When deciding which node to visit next, the incoming iterator prioritizes the node with highest spreading activation. Spreading activation is based on edge weights and the number of links to other nodes, and not shortest path as in the backward search algorithm. The fewer links, the higher activation is spread from a node to its neighbors. Each node spreads a fraction of its received activation to its neighbors, and divides it in inverse proportion to the weight of the edge. This gives lower activation scores to nodes far away from a keyword node, and to nodes with many neighbors. The activation for a node is initialized based on the number of nodes in S_i, so that node in a node set with few other nodes gets higher initial activation than nodes in larger node sets.

[7]Note that this is not the same as the community concept we describe in Section 3.1. For this thesis, community/communities will always refer to the graph theory concept presented later, except in this paragraph.

As the name indicates, the bidirectional algorithm searches in both directions. Thus, in addition to having an incoming iterator for backward search, an outgoing iterator is also employed. The outgoing iterator starts forward search from potential roots. Potential roots are the nodes reached by the incoming iterator. The outgoing iterator also uses spreading activation to decide which node to visit next. Nodes close to the potential root get higher activation, again by dividing it with inverse proportion to the edge weights. The distance from an explored node to each keyword node is kept and updated. When a node has paths to all keyword nodes, an answer tree can be built.

To illustrate the difference between backward and bidirectional search, we show an example in Figure 2.5 which is developed from an similar example in [20]. Let the query be "Bowie Queen 1982". We see that the answer tree will be rooted at the "Under pressure"-node. The dotted arrows indicate the steps in the search. The red node set match the query word "1982", while "Bowie" (green node) and "Queen" (blue node) match a single node each. The backward search would start an iterator from each of the colored nodes. If we assume the distance between each node to be 1, all the albums of David Bowie would have to be visited before we can start exploring the tracks and find the "Under pressure"-node. For the bidirectional search we would start considering nodes "Queen" and "David Bowie" because the "1982"-nodes are in a larger node set and therefore have lower activation. Spreading the activation will cause "Hot space" to have higher activation than the other album nodes, because the activation from "David Bowie" will be spread across all his albums, while the activation from "Queen" solely goes to "Hot space". Step 1 in the search is therefore from "Queen" to "Hot space". From "Hot space", we iterate backwards to find "Under pressure" (Step 2) and forwards to find the "David Bowie"-node (Step 3). From "Under pressure" we can iterate forward to find a "1982" node (Step 4). Then all node sets are covered, and we have a result tree with root "Under pressure". With bidirectional search we only have to explore one album, while with backward search we must explore all albums because we have no concept of different importance of nodes.

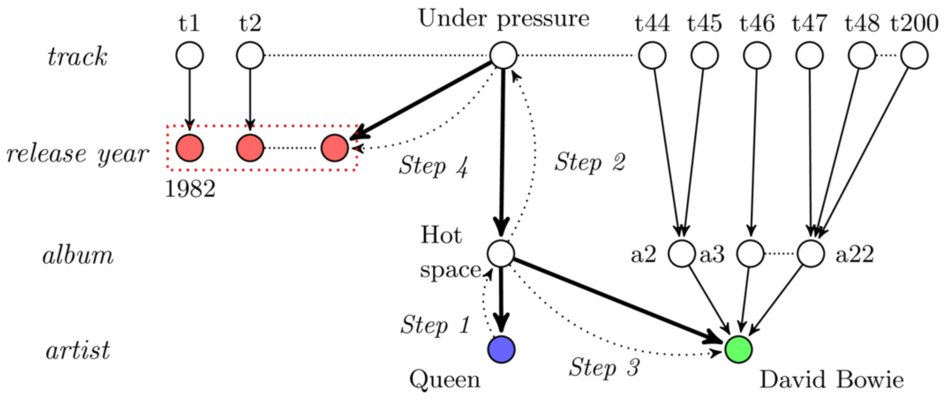


Figure 2.5: Bidirectional search example for query "Bowie Queen 1982"

The approach of He et al. [17] is to partition the graph into subgraphs, or blocks, and have an index that stores a summary of the information found in the block. In order to avoid some of the performance drawbacks of other approaches to keyword search on graphs, the authors argue that indexing the content of nodes is not sufficient. They propose to index information on the structure of the graph, in particular the shortest paths between nodes. Their system, BLINKS, provides efficiency by having summary information at each block so that navigation between blocks is faster, and by having more detailed information inside each block. This use of an index makes it possible to make jumps during search, instead of having to step-by-step expand an answer tree like the backward and bidirectional search approaches. The index at block-level also narrows the starting point for expansion as only the relevant blocks are considered (that is, those that contain the keywords in question). An answer to a query is a subtree, where the root reaches all the input keywords. To avoid similar answers, they follow the distinct-root assumption. This assumption ensures that the top-k answers have distinct roots. When ranking the answer subtrees, the shortest paths from the root to the nodes containing the keywords are used, in addition to information retrieval (IR)-style scores for matching nodes.

Instead of finding trees, Kargar and An [21] propose a general method for finding top r-cliques in a graph. An r-clique is a set of content nodes that cover all of the input keywords, and where the distance between two nodes is equal to or smaller than r. Content nodes are nodes that contain one or more input keywords. By only considering the content nodes, the graph to explore is significantly reduced. The concept of r-cliques ensures that the content nodes are close to each other, which should give more informative answers than other approaches. When searching for r-cliques, the graph is divided into subgraphs (search spaces), and the best answer from each subgraph compete with each other to find the best answer. The best answer is the set of content nodes for which the sum of distances between them is the least, and where all distances are less than or equal to r. To find the second best answer in the next step, the subgraph where the current best answer was found is searched using the same procedure of dividing the graph. The answers found in these subgraphs compete with each other *and* the leftover answers from the previous step. A priority queue is used, where the answers are ranked in ascending order according to their weights. The top answer is removed from the queue and outputted as answer, and its corresponding subgraph divided and searched for new best answers. When the queue is empty (which would happen because not all subgraphs contain a best answer), the procedure of finding answers is done. The answers are presented as Steiner trees to the user. From the r-cliques found, a Steiner tree from the input graph is found that connects all the content nodes in the r-clique.

Sozio and Gionis [45] define the community-search problem. This problem concerns finding communities in a graph given a set of query nodes. The goal is to discover subgraphs in the graph on the basis of which nodes a query match. The subgraphs should be densely connected. In contrast to community detection algorithms (which we review in the next chapter), this approach does not find communities only from an input graph, but also uses a set of query nodes. Their algorithm is a greedy algorithm that starts with the whole graph, and then, in each iteration, removes the node with minimum degree. The termination condition is fulfilled when the node with minimum degree is in the set of query nodes, or when the removal of a node results in the query nodes no longer being connected.

2.4 Keyword Search on RDF Graphs

2.4.1 Characteristics of RDF Graphs

RDF graphs have some characteristics that make the general approaches described above less suitable. This particularly concerns the edges. While edges in the general graph case are treated as unlabeled and weighted, RDF edges (or predicates) are labeled and not weighted [24]. In RDF, the edges denote relationships between nodes and carry information that can be important to take into account when doing keyword search over the graph. As two nodes may be connected by multiple edges, distinguishing between the edges by inspecting the labels gives valuable information about the relationship between the nodes.

Edges are not weighted, as two individual nodes either have a relationship (represented by a edge), or not at all. In RDF, it does not make sense that two nodes should be connected twice (or more) by the same relationship, since this would just be redundant information (two persons connected by *one* isSiblings-relationship are not *less* siblings than two persons connected by *two* isSiblings-relationships). Two nodes may have a number of edges connecting them, but this represents different relationships, and cannot necessarily be interpreted as a measure of how tightly connected the two nodes are. The nature of the relationship (that is, the information found in the edge labels) is also important to take into account.

While RDF graphs often are described, correctly, as directed, the direction of edges do not carry that much meaning, as they usually are invertible [19]. That is, a isStudentAt-relationship might as well be expressed as a hasStudent-relationship (with switched object and subject) without changing the nature of the relationship between a student and a university. The matching triple pattern of SPARQL also makes the direction less meaningful, as one can match a variable just as easily to a subject as an object.

In addition to this, it should be mentioned that RDF graphs may be very large. Indeed, the vision of the Semantic Web is to create a web of linked data[8], and the bigger, the better. Even subsets of RDF data can be very large, and this should be taken into account when creating index and search algorithms.

2.4.2 Approaches to Keyword Search on RDF Graphs

To avoid the problem of dealing with too large RDF graphs when searching, a number of proposed solutions creates a summary of the graph in the indexing phase. The approach of He et al. [17] described above is an example of this. However, the special nature of the labeled, unweighted edges in RDF graphs is not taken into account there, as they focus on node-labeled graphs. In addition, their indexing scheme requires a lot of data to be stored, which is a problem when dealing with large graphs [8, 24] .

[8]https://www.w3.org/standards/semanticweb/

A specific RDF approach by Le et al. [24] makes use of the structure of RDF data, where an entity typically has a class and a number of connections to literals. Based on this, a condensed graph is created, where a node consists of the entity value (an URI), the class, and the associated keywords found in the literals. In line with [17], the condensed graph is partitioned so that it is possible to quickly prune parts of the graph that are uninteresting for the search. Then, backward search is employed on the most promising subgraphs. Each of the partitions have a minimal set of class-based structures that summarizes the partition, which are used in the pruning process. The summary allows for computing upper and lower bounds on distance traversed by a backward search, which can be used to find the most promising partitions (those with least probable distance to traverse). The search is performed on two levels: first, the summary-level is backward searched for the most promising connected partitions that contains all keywords, and then those partitions are searched with backward expansion.

Tran et al. [46] propose an approach that produces top-*k* query candidates based on the input keywords, and then presents the candidates to the user. A keyword index is held in the graph, and permits mapping from the keywords to elements in the graph. Based on this mapping, subgraphs are found, and queries to obtain these subgraphs are produced by mapping nodes to objects and subjects, and edges to predicates. This means that the results can be graphs, and not only trees as in other approaches. Traditional tree-exploration algorithms such as breadth-first search, backward search, or bidirectional search are therefore not sufficient. The input keywords are mapped to elements in the RDF graph, before a connecting element is found. This connecting element is an element that is connected to all keyword elements either directly or through some path. The subgraph is constructed using the connecting element and paths to the keyword elements. The keyword index utilizes IR-techniques (for instance stemming and stop word removal) to find keyword from elements in the graph, and additionally uses WordNet[9] to fetch semantically similar words. In order to find query candidates fast, an augmented summary graph is created with information on the elements and structure of the graph. In the augmented summary graph, every node is an aggregation of all nodes having a specific class, and the edges are connections between instances of the classes. At query time, the summary graph is augmented with keyword matching elements (which are edges between entity and data nodes, or data nodes themselves). The augmented summary graph contains sufficient information to build query candidates. When building query candidates, the objective is to find minimal subgraphs that include at least one occurrence of each query keyword. The candidate query is a conjunction of all predicates in a given subgraph.

The approach of Elbassuoni and Blanco [11] returns a ranked list of answer subgraphs. Each triple in the graph is indexed in an virtual document. This document contains a set of keywords extracted from the subject, object and predicate. The index is used to find matching triples, which are then joined with other triples to form subgraphs which are unique and maximal. That is, no retrieved subgraph is a subset of any other retrieved subgraph. The subgraphs are ranked using a statistical language model, which tries to interpret the information need in the query. Their model find the most likely triple-pattern query by examining the data and finding which triple patterns the keywords often appear

[9]A lexical database for English: `https://wordnet.princeton.edu/`

in. The subgraphs returned as answers in this approach does not necessarily contain all keywords in the query.

Cappellari et al. [6] index the RDF graph by paths. In particular, they are interested in edges and sinks in the graph, as the keywords entered by users most likely will refer to the information found there, and not in the URIs. The paths they are examining are full paths, meaning that they start from sources and ends in sinks. After discovering the paths, they are grouped together in clusters if they share the same template. A template is a sequence of edge labels. As an example, a template could be #-*author*-#-*type*-#, where *author* and *type* are edge labels. Nodes are stored with reference to the path template they match, and their position in the template. In the example, there are three possible positions (denoted by #). The solution subgraphs are built by intersecting promising paths (those with highest score) from the clusters [9]. This is based on matches to nodes, and in turn which templates these nodes match. The scoring function takes into account the length of the path(s) in a subgraph (either the candidate paths or the solution subgraphs), and the relevance of the information in nodes and edges with regard to the query. For relevance, a variant of TF/IDF[10] is used.

The SemSearch system of Lei et al. [26] transforms input keywords to formal queries. In order to do this, the system interprets the semantic meaning of the keywords. A keyword can match a concept (classes), semantic relations between concepts, or entities. Predominantly, the labels of entities are searched for keyword matches. Based on what the keywords match, different formal query templates are employed. For instance, if there are two keywords that match classes, we can expect the results to be instances of one class connected to instances of the other class. The predefined templates are used and ran against a semantic data repository. Finally, the system ranks the results.

Ning et al. [34] developed a keyword searching model that finds minimal connected subgraphs in an RDF graph, so that all input keywords are covered. This is similar to some of the approaches to keyword search on general graphs, but here particular concern is given to the problem that there are many potential answers on a large data graph, which RDF graphs often are. To limit the answers, only the top-*k* results are returned. These are ranked based on cost: smaller cost is better. The cost is calculated based on edge weights. The top-*k* results are built by first creating groups of nodes for each keyword. Starting from a node in a group, nodes are added to the answer trees based on shortest path to nodes who belong to groups not already represented in the tree. When all groups (and thus all keywords) are represented in the tree, it is added to a result queue which is ordered by cost. This process is repeated for all nodes in the chosen start group. The top-*k* results are outputted.

Cheng and Qu [7] propose to create virtual documents for each linked data object. A virtual document includes textual descriptions of objects linked to the given object, and the name of the links. An inverted index is built from the terms in each object's virtual document. Snippets from the RDF descriptions which highlights matched input keywords are provided in the search interface, so that the user quickly can decide if the results are relevant. The ranking is performed based on how well the object matches the keywords,

[10]term frequency/inverse document frequency. A measure of the importance of a word in a collection.

and also how popular the object is.

An approach to offering full-text search capabilities to RDF data is using existing information retrieval techniques. Minack et al. [31] propose to introduce full-text search in RDF storage systems by utilizing Lucene[11], a text search library, combined with Sesame[12], a framework for storage, inference and querying of RDF data. Their approach is to maintain and use the existing query language parser of Sesame by combining the full-text query with the structured query. The authors identify two indexing strategies for mapping RDF data to Lucene documents: one where a Lucene document represents a resource and another where a Lucene document represents a triple. They chose the resource-based approach. The Lucene document have the fields: URI, all (all literals of a resource), and one for each predicate where the object is a literal. Using Lucene, score and snippets can be returned as results along with triples.

2.5 Keyword Search in Triple Stores

The approach of Minack et al. described in the previous section introduced full-text search into SPARQL queries. A number of triple stores also supports keyword search. In the following section, we describe some of the most well-known triple stores and their capabilities in this regard. Apache Lucene or Solr[13], or in some cases both, are used to index the text.

Apache Jena

Apache Jena supports keyword search through Lucene or Solr[14]. The indexed data could be either literals from the RDF data or external content. Most commonly, a property is mapped to a text index field, but multiple properties per URI is possible. Lucene can be configured with different analyzers and tokenizers. Results are a list of resources. The index do not support combining triples to larger index entities. If one want a more informative answer, SPARQL triple matching can be used together with the keyword query.

Blazegraph

Blazegraph offers keyword search by tokenizing literals to create an index[15]. An analyzer from Lucene is used to do this. The created index is a B+-tree, which makes retrieval fast. The results are ranked according to relevance.

[11]http://lucene.apache.org/core/

[12]Now called rdf4j: http://rdf4j.org/

[13]Search platform built on Lucene. http://lucene.apache.org/solr/

[14]https://jena.apache.org/documentation/query/text-query.html

[15]https://wiki.blazegraph.com/wiki/index.php/FullTextSearch

The external keyword search option is more sophisticated, allowing a search to be directed to an Apache Solr index running externally. Results returned from Solr are either set to be URI or literal. The Solr index could for instance consists of JSON documents which correspond to literals in some RDF data (not necessarily data stored in Blazegraph).

Virtuoso

Virtuoso supports keyword search through indexing of object values[16]. The search capability is added to the SPARQL query through the *bif:contains* predicate. This predicate enable free text search on variables in a query. Which triples should be indexed is configurable through indexing rules. These rules are manually decided by the user. The object value needs to be a string to be indexed. An example of an indexing rule is a rule that indexed all triples that match a specific predicate (for instance, *https://www.w3.org/2000/01/rdf-schema#label*).

AllegroGraph

AllegroGraph supports indexing either natively or through Solr[17]. The native indexer could be built to include none, some or all literals, and none, some or all parts of URIs, or a combination of different parts of a triple. Some filters are provided: stemming for English, removal of accented characters, and the opportunity to index words by how they sound rather than how they are spelled. Fuzzy matching using the Levenshtein distance[18] is supported in the search, as are wildcards.

The Solr indexer runs separately from AllegroGraph, which means that it is slower than the native indexer and that the user needs to make sure it is kept in sync. Solr provides more features than the native indexer, such as different languages and relevance ranking[19].

GraphDB

GraphDB from Ontotext provides keyword search through Lucene[20]. Since Lucene provides search on text documents, the RDF graph must be turned into a number of text documents. This is done by each node being associated with an RDF molecule and stored as a text document. The user has to configure how the RDF molecule is built. Different parameters can be used to configure an RDF molecule: what types of nodes to index (URIs, literals, blanks), the size of the molecule associated with a node (that is, how many

[16]`http://docs.openlinksw.com/virtuoso/rdfsparqlrulefulltext/`

[17]`http://franz.com/agraph/support/documentation/current/text-index.html`

[18]A measure of difference between two strings

[19]See `http://franz.com/agraph/support/documentation/current/solr-index.html` for a more comprehensive list

[20]`http://graphdb.ontotext.com/documentation/free/full-text-search.html`

predicate hops from the node we should go), and what should be included in the molecule. In addition, alternative Lucene analyzers and scorers can be employed. The search returns ranked results of URIs, literals and blank nodes, depending on what was chosen to be indexed.

The GraphDB Lucene connector[21] provides another way to do keyword search. Here, a list of types can be given to state which entities are to be synchronized. The fields to index are given through property chains, which typically end in literals. It is possible to have multiple property chains per field. The result of a search is a ranked list of entities, but the query can be combined with SPARQL triple matching to give more informative results. Property chains are manually configured by the user.

2.6 Summary

We have presented approaches to keyword search from the research angle, and described actual implementations in triple stores. With regard to our research questions, we find that most approaches seek minimal solutions to keyword queries. This goes for the implementations of keyword search as well, as solutions to a query typically is a list of resources, not larger graph structures. In some cases, larger structures are supported, but these has to be defined manually.

We therefore find that our target of automatically finding RDF graph structures that corresponds to human conceptualizations of the world, is something not previously addressed. In the following chapter, we will explore how we can achieve this target.

[21]`http://graphdb.ontotext.com/documentation/free/lucene-graphdb-connector.html`

3 | Preliminary Study

The crucial part of the main research question is to automatically find concepts in RDF graphs that corresponds to how humans conceptualize the world. This goal could be translated into the more specific task of grouping nodes in an RDF graph that somehow "belong together". Such groups of nodes have been given different names, of which the most usual are clusters, communities and modules [12]. The different choice of words is also reflected in what the algorithms for finding these node groups are called, namely clustering, community detection and modularity optimization. In general, *data clustering* is concerned with grouping data elements based on a similarity measure, while *graph clustering* is concerned with grouping nodes based on the edge structure [43]. When we refer to clustering here, we are referring to graph clustering, which we understand as equivalent to community detection. Modularity optimization tries to optimize the modularity measure. Modularity is the number of edges falling within modules minus the expected number in an equivalent graph where edges are placed at random [33]. An equivalent graph is a graph with the same structural properties, for instance the same number of edges and/or the same degree distribution [12]. Modularity optimization seeks to find the division of the graph that is furthest away from what could be considered random.

The use of different terminology may be rooted in the fact that the study of graphs - sometimes called networks - is spread across a number of disciplines, such as math, computer science, biology, sociology, physics and more. In this thesis, we mainly stick to the terminology of *community* and *community detection*.

A related concept is that of graph partitioning. The goal of graph partitioning is usually to find a division of a graph such that minimal communication between the partitions occurs [33]. Minimal communication may for instance be achieved by minimal number of edges between partitions, or minimal cost of the edges crossing partitions. A difference between this and community detection algorithms, is that graph partitioning often require a partition of the graph with a predefined number of partitions [12], while community detection algorithms only creates a partition if it fulfills some quality measure. One problem that is often solved by graph partitioning is that of parallel computing, where the number of computers is known in advance. Graph partitioning algorithms are therefore not relevant in our case, since we do not know the desired number of partitions for a given graph in advance.

3.1 Community Detection

The definition of the term community is somewhat unclear [48, 50], but it is often informally described as a set of nodes with more internal connections and fewer external connections [1, 12, 25]. Yang and Leskovec [50] distinguish between structural and functional definitions of communities, where the structural definition is based on connectivity between nodes, and the functional definition is based on the idea that nodes in an community share a common function. They argue that the premise underlying community detection algorithms is that functional communities have structure, and that finding these communities can be achieved by examining the graph structure.

Some research on community detection identifies only disjoint communities, that is, a node belong to one, and only one, community. In this thesis, we look at the research concerned with identifying overlapping communities. This is because we find the Semantic Web best described as a set of overlapping and connected RDF graphs, where a resource can belong to multiple communities.

3.1.1 Overlapping Community Detection

A number of algorithms for detecting overlapping communities have been proposed. Xie et al. [48] provide an overview of the state of the art, identifying five categories based on different approaches. The categories are: clique percolation, line graph and link partitioning, local expansion and optimization, fuzzy detection, and finally, agent-based and dynamical algorithms.

Clique percolation methods finds communities by identifying adjacent cliques. Palla et al. [37] define a community as the union of all cliques of size k that can be reached through adjacent k-cliques. If cliques share $k-1$ members, they are adjacent. Algorithms in this category are suited for dense graphs, as sparse graphs most likely do not contain many cliques.

Line graph and link partitioning algorithms use links instead of nodes as building blocks for a community. Ahn et al. [1] state that the link approach can reveal both the overlapping and hierarchical structure in a graph. Their intuition is not that a community consists of a group of nodes, but that it consists of interrelated links. By clustering the links based on link similarity, they build a dendrogram which can be cut at different thresholds to produce communities. Link similarity for links e_{ik} and e_{jk}, which share the node k, is

$$S(e_{ik}, e_{jk}) = \frac{|n_+(i) \cap n_+(j)|}{|n_+(i) \cup n_+(j)|}, \tag{3.1}$$

where $n_+(i)$ is the set of node i and its neighbors, and $n_+(j)$ the set of node j and its neighbors. A node have membership to the communities that its links belong to. Each link can only belong to one community, but because nodes can have multiple links, they can belong to multiple communities.

Local expansion and optimization algorithms try to grow communities naturally. One method is to find some seed nodes to expand from. Given a local fitness function, a community can be grown from the seed node by judging if neighboring nodes improve the function. When no node can be added to improve the fitness, a community has been found. Some of the algorithms allow for removing nodes at each step, if the removal would increase the fitness of the community. The fitness function F_C, for community C, from [23] is

$$F_C = \frac{k^C_{in}}{(k^C_{in} + k^C_{out})^\alpha}, \tag{3.2}$$

where k^C_{in} is the internal degree, and k^C_{out} is the external degree. k^C_{in} is the number of internal links in C multiplied by 2 (to account for the fact the a link connects two nodes), while k^C_{out} is the number of edges that cross the community boundary. The parameter α is used to control the size of the community; lower values yields larger communities. The authors in [25] suggest values between 0.9 and 1.5.

There are different approaches to choosing seeds. Lee et al. [25] use maximal cliques as seeds. Their argument is that cliques are typical components of communities, and that they are rare structures. The choice of cliques as starting point for community detection implies that all found communities includes a clique. Thus, strongly connected parts of the graph that resembles a community is not found unless they contain a clique. The authors argue that the clique requirement are not too strong, and points to benchmark testing. After finding all maximal cliques, the largest unexpanded seed is chosen and greedily expanded as long as the fitness increase. If a detected community is a near-duplicate of an already found community, the detected community is discarded. A way to decide whether a community is a duplicate of another, is to measure the proportion of nodes in the smaller community that are not in the larger community. This overlap coefficient between communities C_1 and C_2 is defined as

$$f(C_1, C_2) = 1 - \frac{|C_1 \cap C_2|}{min(|C_1|, |C_2|)} \tag{3.3}$$

If this value is below a parameter ϵ, interpreted as the minimum community distance parameter, the two communities are regarded as near-duplicates. A value of 0.6 for ϵ was suggested.

Fuzzy detection algorithms differ from the other categories in that each node has a belonging factor to all the communities. This factor denotes the degree of membership in a community. Nepusz et al. [32] generalize the partition matrix $\mathbf{U} = [u_{ik}]$ where each node k either belong to a community i ($u_{ik} = 1$) or not ($u_{ik} = 0$), to allow u_{ik} to be any real value in the interval $[0, 1]$. A node's total membership degree is 1, but this can be distributed among the communities. Nodes are assigned to communities based on node similarity. The presence of an edge between two nodes indicates similarity. A graph's adjacency matrix is used as the actual similarity between nodes, and the goal is to optimize a partition matrix so that the computed similarities between nodes approximate the actual similarities found in the adjacency matrix. The computed similarity between two nodes i

and *j* is:

$$s_{ij} = \sum_{k=1}^{c} u_{ki} u_{kj}, \tag{3.4}$$

where *c* is the number of communities. The number of communities is predefined, starting with 2. When the optimization has reach a minimum (the best approximation of the partition matrix to the adjacency matrix), *c* is increased to see if this yields a better community structure. To assess the community structure, a fuzzified modularity function is used. The modularity function compares the number of edges within communities in the actual graph, with what we could expect with edges placed at random in a similar graph [33]. Overlapping communities are allowed for by using a fuzzified version of modularity. The algorithm terminates when adding to *c* does not improve the community structure.

Psorakis et al. [39] propose a probabilistic approach based on Bayesian non-negative matrix factorization. The approach quantifies how strongly a node belong to a community, and opens up for overlapping memberships. The idea is to use an adjacency matrix of interactions between nodes to find classes of nodes (communities) that affect the interaction between two nodes. If two individuals interact a lot, it is likely that they belong to the same community.

Agent-based and dynamical algorithms. The use of community labels on nodes is a way to create communities. Nodes with the same label are in the same community. To find overlapping communities, each node need to be allowed to be associated with more than one community label. Gregory [14] proposed an algorithm for assigning community labels to nodes which allows for overlapping communities. Each node is labeled with a set of pairs (c, b), where c is the community identifier and b indicates the node's belonging coefficient to c. The belonging coefficient for a node over all communities sum up to 1. A node's label is set on the basis of the labels of its neighbors. For a given community c_i, the label (c_i, b) for a node is updated by summing all the neighbors' (c_i, b) labels belonging coefficient, and normalizing by the number of neighbors. To avoid producing as many communities as there are nodes, the belonging coefficient needs to be above the threshold $1/v$ to be kept for a node. The parameter v expresses the maximum number of communities a node can belong to. If $b < 1/v$, then the (c, b) pair is deleted from the node, unless it is the largest b for this particular node. This means that community identifiers may be gone at the end of the algorithm. Assigning and updating belonging coefficients are run for a number of iterations. The termination condition for the algorithm is a heuristic based on the number of community identifiers in use for a given iteration. The minimum number of nodes labeled with each community identifier since the number of community identifiers was last reduced is found. If this is equal in subsequent iterations, the algorithm is terminated.

Assigning labels to nodes can be done by utilizing interaction rules between pairs of nodes. A node remembers a label assigned to it in a previous iteration, and the probability of seeing a label in a node's memory is the membership strength. One example of such a algorithm, is Xie and Szymanskis SLPA [49]. In SLPA, each node is instantiated with a unique label. A node is selected as listener, and ask each of its neighbors to select a label randomly with probability proportional to the occurrence frequency of the labels in its memory. The selected label is sent to the listener, which adds the most popular label among its neighbors to its memory. This is repeated for a user defined maximum number of times. A post-processing step is performed to output the communities based on labels in the nodes' memory. The labels represent community identifiers, and each node will at the end of the algorithm have a strength associated with a number of communities. With the use of a threshold, labels can be removed so that the node only retain the labels of the communities it associates most strongly with. When the threshold is above 0.5, SLPA will produce disjoint communities.

3.2 Node Centrality

As described above, local expansion algorithms may grow communities from seed nodes. One way to find seed nodes, is to select the most central nodes in the graph. Different centrality metrics have been developed for discovering these nodes. A classic categorization of centrality metrics was proposed by Freeman, who identified three types of centrality: one based on degree, another based on closeness, and finally, one based on betweenness [13].

Degree centrality is the most simple metric. It measures the number of nodes a given node is directly linked to. The idea is that a node with many links is central in the graph. Degree centrality was first described for binary graphs, and calculating this metric in that scenario consists of simply counting the number of links in and out of a node (if a directed graph). This measure does not take into account the strength of the link between two nodes. If the graph has edges with weights, the node strength is the sum of the weights of edges connected to the node [36]. As both the number of connections (the degree) and weight of those connections (the strength) can be useful when measuring centrality of a node, Opsahl et al. [36] proposed a metric combining them:

$$C_D^{W\alpha} = k_i * \left(\frac{s_i}{k_i}\right)^{\alpha} \tag{3.5}$$

where k_i is the degree and s_i is the strength. α is a tuning parameter that can be used to control which of the two measures should be emphasized.

Closeness centrality is a measure of the cost for a node to reach all other nodes in the graph. By taking the inverse of the sum of the shortest possible distance to all other nodes, nodes that can reach the other nodes with little cost are considered most central. Closeness centrality can be computed by applying Dijkstra's shortest path algorithm. Dijkstra's algorithm finds the path with least cost, which is not what we are interested in when finding central nodes. On the contrary, the shortest path in that context is the path with highest total weight, implying that it is easy to move between those two nodes. Thus, to utilize Dijkstra's algorithm, the weights has to be inverted so that the maximum weight becomes the minimum weight. After computing the shortest path to all other nodes, the weights of the paths are added together, and then inverted again so that the highest sum indicates the most central node. Again, Opsahl et al. [36] introduce a tuning parameter on the weights to allow for preference on whether the weights or the number of steps in the path is more important.

Betweenness centrality also uses the shortest path between nodes, and state that nodes that most frequently occur in such paths are the most central in the graph. One way of describing this is that such nodes control the flow in the graph. Betweenness centrality also utilizes the paths found by Dijkstra's algorithm. The centrality for a node is calculated by looking at the paths where the node in question is neither the start nor end point. If the node occurs as an intermediate node in a path, the weight of the path is counted, otherwise not. After examining all paths, a sum of weights of paths that includes the given node is found. This sum is divided by the total sum of path weights for paths that do not have the node as start or end point. So, if a node occurs as intermediate node in all the paths, the betweenness centrality is the maximum value of 1.

Another measure of node centrality is eigenvector centrality. This measure uses the largest eigenvalue of the adjacency matrix for a graph, and the related eigenvector to describe the centrality of the nodes [5, 42]. Eigenvector centrality take into account the pattern of the whole graph, and do not treat each link between nodes equally. Instead, nodes connected to important nodes are seen as more important than nodes connected to less important nodes.

3.3 Discovering Concepts through Community Detection

At the heart of the main research question is that what we described as concepts corresponding to how human conceptualize the world should in some way be extracted from the data. Preferably automatically, without the need of field experts. Also, the method for extracting concepts should work on datasets with different characteristics. Communities in a graph, as presented in Section 3.1.1, can informally be described as a set of nodes that "belong" together. Yang and Leskovec [50] noted that the premise underlying community detection algorithms is that functional communities have a structure that can be extracted from the graph. Human concepts correspond to functional communities. In this section, we explore if this underlying premise can be said to hold for RDF graphs. We explore community detection in both class graphs and instance graphs.

3.3.1 Experimental Setup

For experimentation here and in subsequent chapters, different datasets with different qualities have been used. The different datasets and some facts about them are shown in Table 3.1. The number of class instances refer to unique instances in the dataset. A relationship between two classes exists if there is a statement where the subject is of one of the classes and the object is of the other class. Note that the number of class relationships refers to the existence of a relationship between two classes, it does not sum up different predicates

Table 3.1: Datasets

Dataset	Statements	Class instances	Classes	Class relationships
Linkedmdb	6 147 975	738 890	53	54
Musicbrainz	186 007 290	34 810 813	13	24
murder.rdf	4 331	425	4	4

that may exists between two classes. The Musicbrainz dataset[1] contains data about music (see [41] for a description of the ontology) and is quite large. This makes it suitable for testing the performance of different index and search approaches. On the other hand, it has relatively few classes and class relationships. In Appendix A, a class graph showing the class relationships in the Musicbrainz dataset can be found. The Linkedmdb dataset[2] concerns movie information[3]. Though it is smaller than the Musicbrainz dataset, it has more classes and more varied relationships between the classes. See Appendix B for a list of the classes in the Linkedmdb dataset.

The murder.rdf dataset, provided by the supervisor, is small and mostly used because it was easy to get an overview of all the data. This made it quicker to judge some of the algorithms on quality, if not on performance. Because the dataset has edges in both directions (that is, both Person-*authorOf*-Work and Work-*author*-Person statements), it also meant that we could test the algorithms under such circumstances. The dataset contains bibliographical data about works that have some relation to "murder", for instance books with the word "murder" in the title or books with murder as a theme.

Each dataset is stored in its own repository in a GraphDB instance from Ontotext. The free version 7.1[4] is employed on a server provided by NTNU running Ubuntu with 30 GB RAM. The experiments in this and subsequent chapters are run on another Ubuntu machine with 6 GB RAM, which communicates SPARQL queries to the triple store over HTTP. All algorithms are implemented in Java 8.

3.3.2 Choice of Community Detection Algorithm

In Section 3.1.1 we described five categories of community detection algorithms. Clique percolation methods do not fit RDF data particularly well, as such data usually is quite sparse and cliques are hard to come by. The fuzzy detection algorithms require each node to have belonging factor to all communities. This becomes problematic for larger graphs. The same goes for the link partitioning approach, where a dendrogram for a very large number of nodes has to be built. Algorithms that utilize local qualities in the graph - local expansion algorithms and agent-based algorithms such as SLPA are examples - are thus

[1]Fetched from `ftp://ftp.musicbrainz.org/pub/musicbrainz/rdf/20131125/`. Accessed 07.11.2016

[2]Fetched from `http://www.cs.toronto.edu/~oktie/linkedmdb/`. Accessed 07.11.2016

[3]See `http://www.linkedmdb.org/`

[4]Documentation at `http://graphdb.ontotext.com/documentation/7.1/free/`

to be preferred. This matches the empirical findings of Xie et al. [48], which note that these algorithms have both higher performance on larger graphs and are better for graphs with low overlapping density (where few nodes are part of multiple communities). An issue with SLPA is that it is non-deterministic, meaning that the resulting communities may differ for different runs of the algorithm. In order to avoid that a random result that is suboptimal is chosen, the algorithm could be run multiple times. However, that would increase the performance cost of indexing. We therefore choose a local expansion algorithm. Following the general technique of greedy local optimization as described in [25], the pseudocode for finding communities is shown in Algorithm 1.

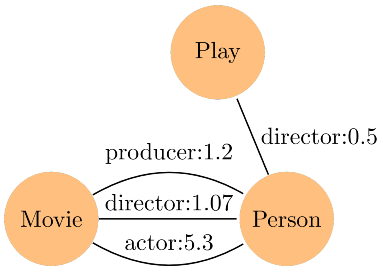
Algorithm 1 General algorithm for community detection using greedy local optimization

```
Input: $graph$
Output: $com$
1:  $seeds \leftarrow$ GETSEEDS($graph$)
2:  $com \leftarrow \emptyset$                                ▷ The set of found communities
3:  for all $S \in seeds$ do
4:      $C \leftarrow S$                                  ▷ Create new community with seed
5:      $N_F \leftarrow$ GETFRONTIERNODES($C, graph$)
6:      $F_C \leftarrow$ GETCOMMUNITYFITNESS($C$)
7:      $n \leftarrow$ GETFITTESTNODE($N_F, F_C, C$)
8:      while $n.fitness > 0$ do
9:          $C \leftarrow C \cup n$
10:         $N_F \leftarrow$ GETFRONTIERNODES($C, graph$)
11:         $F_C \leftarrow$ GETCOMMUNITYFITNESS($C$)
12:         $n \leftarrow$ GETFITTESTNODE($N_F, F_C, C$)
13:     $com \leftarrow$ CHECKFORNEARDUPLICATES($C, com$)
14: return $com$
```

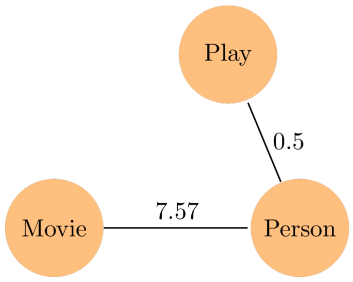
At line 1, we fetch the seeds from the graph. This is the nodes from which communities are grown. How to decide what the seeds should be, varies with the approaches we will present. As does the fitness function. For each seed, we create a new community C (line 4). Then we keep adding nodes to C as long as they increase the fitness (checked at line 8). To find which nodes to consider, we fetch the frontier nodes, N_F, around the community (lines 5 and 10). From these nodes, we find the node n that will increase the fitness most. The increase in fitness is calculated against the fitness of C, which we get at lines 6 and 11. When no more nodes can be added to C, we are done finding the community for this seed. To make sure that not too similar communities are found, we remove near-duplicates at line 13 using the overlap coefficient in Equation 3.3. If a community is near-duplicate to another, we keep the larger community. This means that the community we just found is not necessarily added, it may also be discarded. The check at line 13 may also result in previously accepted communities being removed from com.

3.3.3 Community Detection in RDF Class Graphs

We have noted that RDF data can be large, and this creates a performance challenge. The graph is often represented by a smaller graph to avoid this. We have seen that class information can be used to summarize the graph (for instance in [24, 46]). One way of doing this is to have each class appear exactly once in a summation graph, and aggregate the predicates between the nodes found in the data. This will create a weighted graph from which communities can be detected. A weight between two classes could be as simple as the number of links between them. We found it more appropriate to use the average number of weights per instance, since it provides info on the cardinality between the classes. Using the number of links could for instance hide that a couple of instances are responsible for most of the links, and therefore that this relationship is more rare than it seems. Looking at Figure 3.1a, we see that each instance of a Movie on average has 1.07 links to a Person via the director predicate (most movies have 1 director, but some may have more). The weights of different predicates are then summed up, as showed in Figure 3.1b.



(a) Class-based summation graph without summed up predicate weights



(b) Class-based summation graph with summed up predicate weights

Figure 3.1: Example of class-based summation graphs

The intuition behind the class-based summation graph approach is that the weights indicate how closely connected two classes are, and that the fitness function of community detection indicates how unique the connection is. That is, since the Person class in Figure 3.1 does not have many other connections than to Movie *and* the weight indicates a strong relationship, we can assume that these two classes belong together. Person has a connection to Play as well, but the weight indicate a less strong relationship. The weights therefore allow us to distinguish between links, and not just treat them as binary relations.

When using weights, the fitness function in Equation 3.2 is changed to use internal and external weights instead of degree. Then k_{in}^{C} is the sum of the internal edge weights, and k_{out}^{C} is the sum of the weights on edges crossing the community boundary.

We also test with degree as community decider. Instead of weights, the degree of the nodes in the class-summation graph can be used. Then, the most central classes would be in different communities and be joined by the less central classes around them. The fitness function would be the same as in Equation 3.2. Finally, when the communities at class level have been discovered, they can be used on the instance level to create index entities.

Implementation

Given the approach outlined, Algorithm 1 was implemented with Equations 3.2-3.3 for fitness and near-duplicate check. Since the class graph most likely will be small, it is feasible to use the more sophisticated node centrality measures of closeness, betweenness and eigenvector centrality to find seed nodes to expand from. The seeds were chosen using the eigenvector centrality measure[5]. Those with a normalized eigenvalue above $\frac{1}{|V|}$ were chosen as seeds, with the intuition that these nodes are more central in the graph than can be expected. $|V|$ is the number of classes in the dataset. Note that before the algorithm is run, some statistics and information are fetched from the dataset. This includes which classes exists, and the relationships between them. Based on this, the weights of the relationships are calculated to produce the class-based summation graph.

The experiment was run on multiple datasets, using both weights and degree in the fitness function, and with different values for α (see Equation 3.2). The near-duplicate limit, ϵ, was set to 0.6 as suggested in [25].

Evaluation

Table 3.2 shows the results for the Musicbrainz dataset. Seven classes were found to be seeds based on the eigenvector centrality: SoloMusicArtist, MusicArtist, Release, SignalGroup, MusicGroup, SpatialThing and Composition. Looking at the result for the Musicbrainz dataset, we see that small values of α are not able to stop the growing of a community until it contains the entire graph. The expectation of larger communities for smaller values hold. Also, larger values results in more communities. This is natural, as it is less likely to find duplicates for smaller communities. For instance, CW1 for $\alpha = 1.5$ does not include Composition and MusicalWork like it does for $\alpha = 1.2$, which opens up for those classes to be a separate community (CW3). It is not a linear relationship, where increasing values of α necessarily leads to more communities (for instance, $\alpha = 2.5$ yields two communities). At some point, though, the value become so large that no seed can grow, and we are left with seven communities which only contains the seed.

The Musicbrainz dataset has a hidden inheritance relationship (hidden in the sense that it is not encoded in the dataset). According to the specification of the music ontology[6], MusicArtist is a parent class for MusicGroup and SoloMusicArtist. This provides a tool to reason about what communities we should expect to see. The expectation is that these three classes appear together in communities. Looking at the result, we see that they in most cases are grouped together. The exceptions are community CD2 for $\alpha = 1.5$ where MusicGroup appears alone, and community CW2 for $\alpha = 1.5$ where MusicArtist

[5]The jblas library was used for this: `http://jblas.org/`

[6]See `http://musicontology.com/specification/` or `https://github.com/motools/musicontology`

Table 3.2: Test results for community detection on class-based summation graph for the Musicbrainz dataset

Approach	α	No. of communities	Communities
Weight	0.9	1	One community with all thirteen classes
	1.0	1	One community with all thirteen classes

	1.2	2	CW1 = SoloMusicArtist, Track, Record, MusicArtist, Signal, SignalGroup, Release, Composition, MusicalWork, MusicGroup
			CW2 = SignalGroup, Release, ReleaseEvent, SpatialThing, Label

	1.5	3	CW1 = SoloMusicArtist, Track, Record, MusicArtist, Signal, MusicGroup
			CW2 = MusicArtist, SpatialThing, ReleaseEvent, Label, Release, SignalGroup
			CW3 = Composition, MusicalWork

Degree	0.9	1	One community with all thirteen classes
	1.0	1	One community with all thirteen classes
	1.2	1	CD1 = Release, SoloMusicArtist, MusicArtist, SignalGroup, SpatialThing, Record, ReleaseEvent, Label, MusicGroup

	1.5	2	CD1 = Release, Record, ReleaseEvent, SpatialThing, Label
			CD2 = Composition, MusicalWork, MusicGroup, SignalGroup

is alone. Since MusicArtist is the parent class, the latter does not violate expectations. MusicGroup appearing alone is more problematic. Following the subclass property in RDFS, all instances of MusicGroup are instances of MusicArtist. Thus, we would expect MusicArtist to be in the community with MusicGroup.

Of the results in Table 3.2, the one for $\alpha = 1.5$ using the weight approach is probably the best. Here, we can identify three aspects of the music making process: the work in question (CW3), the recording of the work (CW1), and the release of the work (CW2). Comparing it to the class graph (see Appendix A), no obvious concept is missing.

For the Linkedmdb dataset, similar tendencies as for Musicbrainz was found, even though small values for α worked slightly better here. The algorithm was able to detect some smaller communities, at least with the degree approach. With the weight approach, the number of found communities increased with increasing α-values, while it remained quite stable with the degree approach. Regardless of approach and α-values, the algorithm produced one large community in addition to the smaller ones. The large community was grown from the Film-class, which is the most central class both in terms of how many classes it has links to, and also how numerous the number of links are for each class relationship. Therefore, it is natural that this community encompasses many of the classes found in the dataset. The communities returned by the algorithm seemed to be fairly logical, and can be said to refer to concepts such as: the film itself, film distribution, film crew, performance in a film, and film company. One concept that exists in the data but was not found, is that of film festival. There are two classes, film festival and film festival event, that are connected to each other but no other class. This illustrates a problem with the seed approach: some parts of the graph may not be explored. RDF graphs may be sparse or even disconnected, and there is no guarantee that distant or disconnected parts contain a node that qualify as seed.

A more fundamental problem with the class-based summation graph approach is that a lot of information is lost, probably too much. In particular, the summation of predicates hides the different kinds of relationship that can exist between two instances of a class. Figure 3.1 showed an example of how different relationships can be hidden if predicate weights are summed up. The weights in Figure 3.1b suggests that Person is more strongly connected to Movie than Play. However, it may be that instances of Person are frequently connected to both Movie and Play *if* the instances are directors. To have all instances of a class grouped together is problematic when the class can be said to contain subgroups. In our example, Person probably has three subgroups: actor, director and producer. These subgroups can be found looking at the predicates. The graphs in Figure 3.1 remove the opportunity to assess the possible different communities of actors and directors. For the Musicbrainz dataset this is not problematic, as most classes are connected only through one predicate. However, the Linkedmdb dataset has this issue, with the Film and Person classes being linked by four predicates: actor, director, editor and cinematographer.

Another issue is that using weights on RDF data is problematic. Figure 3.1a may suggest that Movie is more strongly connected to Person through the actor predicate than through the director predicate. However, this is just a result of the cardinality that is inherit in the concept of a movie: there are usually more actors than directors in a movie.

3.3.4 Community Detection in RDF Instance Graphs

The class-based approach is coarse, ignoring one of the characteristics of RDF data: the labeled edges. A way to take different kinds of class relationships into account, is simply to run Algorithm 1 on the instance graph instead of the class graph. In light of the problem of using weights, we only consider the degree approach here.

Implementation

Eigenvector centrality is not feasible to use on large graphs. Instead, degree centrality can be used, as this only depends on local factors in the graph. We use instances with a degree above a threshold as starting seeds. The intuition is that central nodes are those with more links than the average. The node centrality threshold is thus:

$$\text{node centrality threshold} = \frac{2|E|}{|V|} \tag{3.6}$$

In this approach, we only consider the nodes that are instances of a class, not literals or class nodes. While we previously stated that literals nodes contained important information for search, we do not need this information to find concepts in the graph. The information found in the literals nodes may be indexed after communities are found.

Evaluation

Performance wise, this approach is quite obviously slower than the class-based approach, considering the difference in graph size. In the Musicbrainz graph, for instance, is the number of nodes 13 versus 34 810 813 (compare the "Classes" and "Class instances" columns in Table 3.1). Because the number of seeds is larger, the possible fitness increase of adding a node to a community has to be computed many times. The duplicate check is also more performance intensive as the number of communities to check grows. In general, many operations on sets has to be undertaken.

Figure 3.2 shows a community (represented by the darker nodes) that is grown from the "Murder on the Orient Express" node in the murder.rdf dataset. This example illustrates some challenges with the approach. First, we note that "And then there were none" is included in the community because a Russian book includes both works, and is translated by the same person. As all the other parts of the community is clearly linked to "Murder on the Orient Express", other novels do not belong. It is not that "Murder on the Orient Express" and "And then there were none" are inherently unfit to be in the same community: in a community grown around Agatha Christie, they should probable both be present (in fact, they are *not*. That community only contains "And then there were none" of the two). The reason why "And then there were none" is added to the community, is that there are no nodes with many links along the way to stop the growing. A way to fix this is to increase the α-value. Indeed, $\alpha = 1.2$ cuts off the whole Russian branch of the community along with the "And then there were none" node. However, then we are left with a community with the French and Esperanto versions of the novel, and that does not easily correspond to a human concept.

The second problem is the inconsistency in which relations are added to the community. We see that "Murder on the Orient Express" is connected to six other nodes through the expression predicate. If a community should reflect a concept, either all of those nodes should be in the community or none of them, at least when "Murder on the Orient Express" is the seed. It could be argued that a concept could be the French expression of the work, but it is hard to think of a concept that encompasses the French, Russian and Esperanto version of a novel, and not the Latvian and English. The reason why the English expressions of text and spoken word are not included is pretty clear: they have too many further links. Thus, the community fitness would decrease by adding them. Why the Latvian expression is not added, while the French and Russian expressions are, is harder to see. When the French node is added, both the Russian and Latvian node have the same fitness gain as the French. Since the French is considered first, it is added. After the "French text" node was added, its translator and manifestation was also added. The Latvian expression had lower fitness gain than "Louis Postif" and "Book, 1966" in these iterations of the algorithm. Finally, when the Latvian expression was the fittest node again, the community

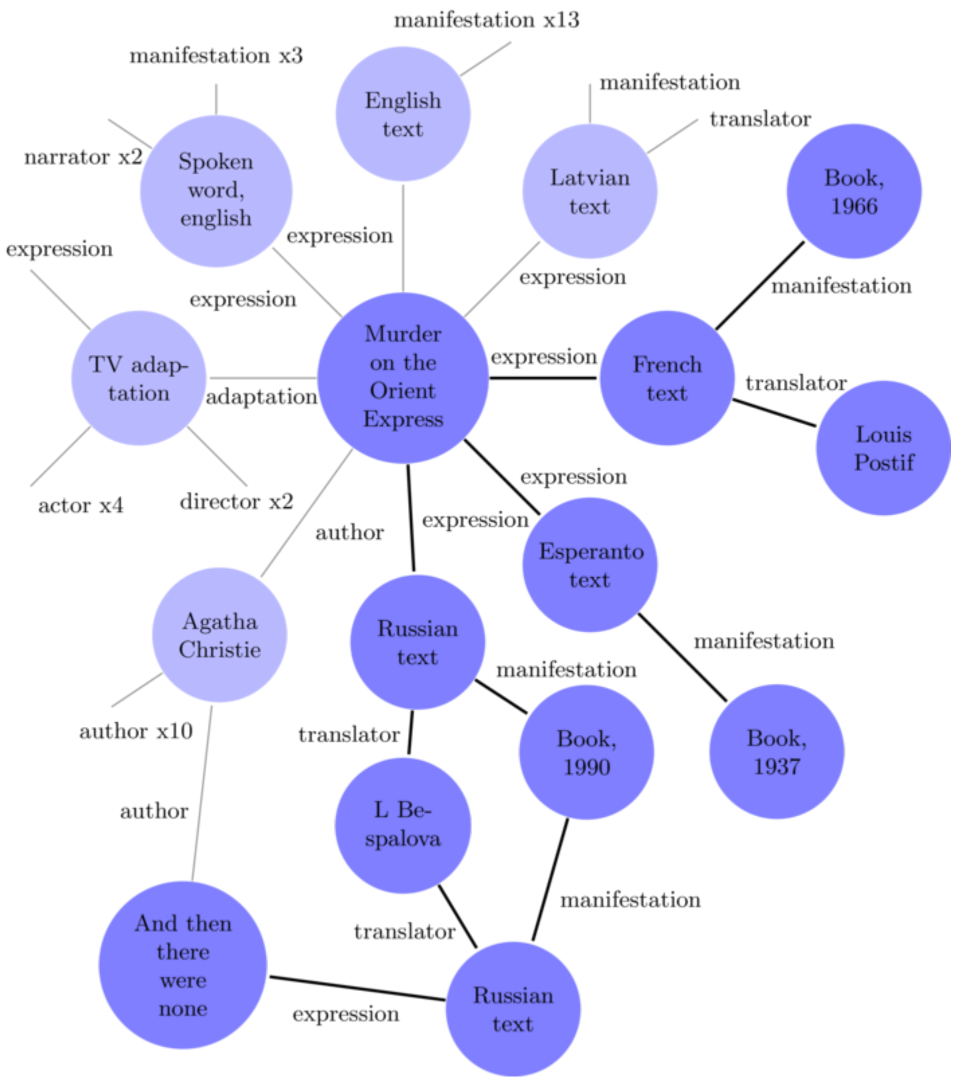


Figure 3.2: Example of a community (the darker nodes) detected from the murder.rdf dataset for seed "Murder on the Orient Express" with $\alpha = 1.0$. "Manifestation x13" means that there are 13 different links with predicate "Manifestation" from the node.

had changed (the whole french and russian part of the graph had been added), and adding it would decrease the fitness. Thus, the French and Latvian expression got their fitness evaluated against different communities. To avoid this, a solution could have been to add all top nodes if several have the same fitness gain. However, that would not have helped in the case of the English expressions.

3.4 Summary

Our preliminary study has shown the weaknesses of applying existing community detection algorithms directly to RDF graphs. In particular, the algorithms do not take into account the special nature of RDF. Most prominently we see this in the way edges and nodes are treated purely as graph elements, not as elements carrying information. Without considering the information edges and nodes represents, we point out two inconsistencies that can arise:

- An identical class relationship may be treated differently in the *same* community detection process
- An identical class relationship may be treated differently *across* community detection processes

The first point refers to the example described above where different instances of the same relationship (for instance Work-*expression*-Expression) may be treated differently for the same seed node. The second point refers to the issue that we may have consistency for one seed node, but the same relationship may be judged differently for another seed node. Consequently, we may have a community around the work "And then there were none" that includes all Work-*expression*-Expression relationships, while the community around the work "Murder on the Orient Express" does not. If either or both of the inconsistencies arise, it can hardly be said that we have discovered a logical concept for the class Work.

4 | Path-based Graph Indexing

Because of the issues with applying existing community detection algorithms directly to RDF graphs, we create a novel community detection algorithm for finding concepts in an RDF graph. The details of the algorithm are presented, and the feasibility is shown through experiments and complexity analysis.

4.1 Path-based Community Detection in Instance Graphs

4.1.1 Approach

In order to avoid the inconsistencies described in the previous chapter, we define a community by a set of paths found from the seed node, not by the nodes themselves. The inclusion of nodes in a community is governed by which paths are deemed to be in the set of community paths. Nodes that are reachable through these paths belong to the community. We define a path similar to how we did in Section 2.1.1, as an alternating sequence of nodes and edges. The requirement for a path was that no edge occurs more than once in the sequence, and that no internal node was repeated. We extend the last requirement to cover all nodes, not just the internal ones. We now define an RDF path, which is a path in the context of RDF graphs.

Definition 4.1.1. *RDF Path.* An RDF path is an alternating sequence of nodes and edges, where the nodes represents RDF resources that are instances of classes and edges represents predicates. The nodes in the path are labeled with the name of their class. We require that no RDF resource is repeated in the path.

What is meant by the definition, is that the RDF path for the relation between "Murder on the Orient Express" and "French text" in Figure 3.2 is Work-*expression*-Expression, because "Murder on the Orient Express" is an instance of the class Work and "French text" an instance of the class Expression. This definition means that the same class and predicate may appear multiple times in a path[1], but that they represent different instances

[1]From now on, when referring to a path we mean RDF path as defined in Definition 4.1.1

of nodes and edges. For instance, we may have the path Artist-*made*-Track-*madeBy*-Artist-*made*-Record-*contains*-Track. This path is acceptable if the second Artist is a different instance to the first Artist instance, and if the first Track is different to the last. Thus, in contrast to [6], a path is not only defined by edge labels, but also by the class of the nodes connected by edges. Artist-*made*-Track is a different path to Artist-*made*-Record, even though the edge labels are the same. In our approach, an edge is a triple $\{fromClass, edgeName, toClass\}$, where $fromClass$ and $toClass$ are the class names of the nodes the edge joins, and $edgeName$ is the predicate.

Algorithm 2 shows the path-based approach. At line 1 we collect information about the graph: the classes and their relationships, and how many edges and nodes there are. We also store the nodes, from which we shall select seeds. At line 2 we calculate the node centrality threshold for the graph, a value that is used in our fitness function (this calculation is the same as Equation 3.6). Given the classes and the different relationships between them, we obtain at line 3 the set of possible paths in the graph from each class up to a specified length k (to be discussed below). Note that $paths$ may contain paths that does not exists in the data. This is because we combine class relationship at the class level, not the instance level. For instance, we may have a relationship Work-*expression*-Expression and another relationship Expression-*narrator*-Person. This would then be combined to the path Work-*expression*-Expression-*narrator*-Person. However, we have no guarantee that there exists an Expression instance that joins these two relationships. The reason why we do it this way is that it would be computationally expensive to find such joining instances. Because the class graph is much smaller, joining class relationships is cheaper.

Algorithm 2 Path-based Community Detection in Instance Graph

```
 1: $graph \leftarrow$ GETGRAPHINFORMATION()
 2: $threshold \leftarrow (2 * |graph.edges|)/|graph.nodes|$          ▷ Node centrality threshold
 3: $paths \leftarrow$ GETPATHS($k, graph$)
 4: $stats \leftarrow \emptyset$
 5: for all $class \in graph.classes$ do
 6:     $seedNodes \leftarrow$ GETSEEDNODES($class, graph$)
 7:     $paths_c \leftarrow paths[class]$          ▷ Paths starting from $class$
 8:     for all $seedNode \in seedNodes$ do
 9:         $paths_r, paths_i \leftarrow$ FINDPATHCOMMUNITY($seedNode, paths_c, threshold$)
10:         for all $path \in paths_r$ do
11:             $stats[path][possible] \leftarrow stats[path][possible] + 1$
12:         for all $path \in paths_i$ do
13:             $stats[path][possible] \leftarrow stats[path][possible] + 1$
14:             $stats[path][actual] \leftarrow stats[path][actual] + 1$
15: CREATEPATHCOMMUNITIES($stats$)
```

The idea is to run our community detection algorithm for a number of randomly selected seed nodes for each class in the dataset. The seed nodes are selected at line 6. For each class we obtain the paths starting with the given class ($paths_c$), that is, all possible paths for the $seedNodes$. Each run of the community detection algorithm at line 9 returns a set of paths included in the community ($paths_i$), and a set of paths that could have been included, but was not ($paths_r$). We have that $paths_i \cap paths_r = \emptyset$ and that $paths_i \cup paths_r$ is all the paths up to length k from $seedNode$. Note that this is not necessarily the same as $paths_c$ since there may be possible paths not present for a particular $seedNode$, but we have that $paths_i \cup paths_r \subseteq paths_c$. From the two path sets, we create statistics that state the number of times a path was included (line 14) as a ratio of how many times it was possible to be included (lines 11 and 13).

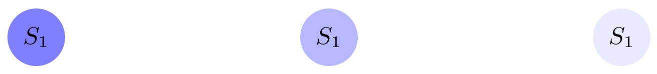
Table 4.1 shows an example illustration of what the statistics could look like. For the path Work-*author*-Person, 67 of the selected seed nodes of class Work had a relation to one or more instances of class Person through the *author*-predicate. Of these 67 cases, 35 runs of the $FindPathCommunity$ algorithm resulted in the Person instances to be included, while 32 of the runs resulted in the Person instances being rejected. The idea is that these numbers tell us how closely classes are connected through different predicates. The statistics are used at Line 15 to build path communities from the most frequent paths. These can then be used to build query solutions at index or search time.

Table 4.1: Example of path statistics

Path	Actual communities	Possible communities
Work-*expression*-Expression	34	56
Work-*author*-Person	35	67
Person-*narrator*-Expression	4	4

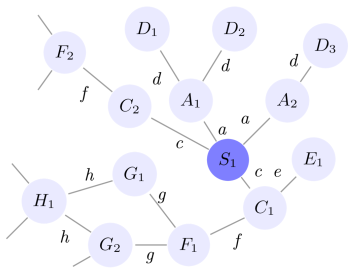
Figure 4.2 shows an example of lines 9-14 in the algorithm, where community detection is done for each of the seed nodes with max path length of 2. The colors on the nodes in the example represents the status of the node. Figure 4.1 shows the colors for the three types of nodes: (a) those included in the community, (b) those rejected from the community, and (c) those who are not (yet) tested for inclusion.

The example is in the middle of a run of Algorithm 2. The seed node S_1 belongs to the class S, and the possible paths originating from S is showed in the four path statistics tables in the figure. This is $paths_c$. $paths_i$ and $paths_r$ are updated after each path is checked, depending on which of the two sets the path in consideration belong to. If the path do not exists for this seed, it will be in neither of the two sets. After testing each of the paths in $paths_c$, we update the path statistics on the basis of which of the two sets a path is in. The procedure in the figure is repeated for all the selected seed nodes of each class in the dataset. The details of how a path is tested and placed in either of the two sets is presented in subsequent sections.



(a) Included node **(b)** Rejected node **(c)** Not tested node

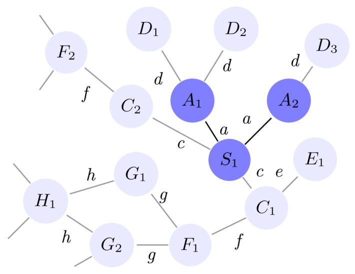
Figure 4.1: Color codes for nodes



Path	**A**	**P**
S-*a*-A	34	56
S-*a*-A-*d*-D	23	42
S-*b*-B	7	7
S-*c*-C	45	78
S-*c*-C-*e*-E	5	8
S-*c*-C-*f*-F	21	29

$paths_i$: { }
$paths_r$: { }

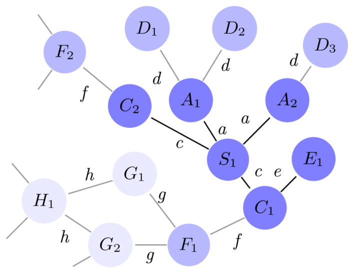
(a) Before community detection for S_1



Path	**A**	**P**
S-*a*-A	34	56
S-*a*-A-*d*-D	23	42
S-*b*-B	7	7
S-*c*-C	45	78
S-*c*-C-*e*-E	5	8
S-*c*-C-*f*-F	21	29

$paths_i$:
{S–*a*–A}
$paths_r$: { }

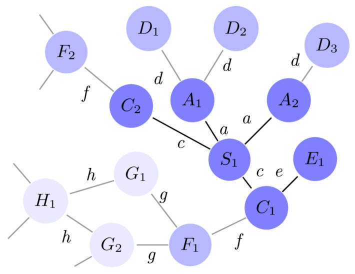
(b) After testing first path S-*a*-A



Path	**A**	**P**
S-*a*-A	34	56
S-*a*-A-*d*-D	23	42
S-*b*-B	7	7
S-*c*-C	45	78
S-*c*-C-*e*-E	5	8
S-*c*-C-*f*-F	21	29

$paths_i$:
{S–*a*–A,
S–*c*–C,
S–*c*–C–*e*–E}
$paths_r$:
{S–*a*–A–*d*–D,
S–*c*–C–*f*–F}

(c) After testing last path S-*c*-C-*f*-F



Path	**A**	**P**
S-*a*-A	***35***	***57***
S-*a*-A-*d*-D	23	***43***
S-*b*-B	7	7
S-*c*-C	***46***	***79***
S-*c*-C-*e*-E	***6***	***9***
S-*c*-C-*f*-F	21	***30***

$paths_i$:
{S–*a*–A,
S–*c*–C,
S–*c*–C–*e*–E}
$paths_r$:
{S–*a*–A–*d*–D,
S–*c*–C–*f*–F}

(d) Update path statistics

Figure 4.2: Example of the overall approach for the seed node S_1 with path length = 2. Note that the nodes in the figure represent instances marked with a number subscript, while the different letters represents the class the instance belong to. Darker nodes indicate nodes included in the community around S. The path statistics tables are similar to the one showed in Table 4.1, with A="Actual communities" and P="Possible communities". S-*c*-C-*f*-F-*g*-G is not tested because it is longer than the set max path length. S-*b*-B is not updated because that path did not exists for this seed node.

4.1.2 Seed Node Selection

The selection of seed nodes is done randomly for a given class. This approach closely resembles the stratified sampling approach, where the sample space is divided into a number of strata from which a random sample is collected [30]. The stratas are supposed to be disjoint, a requirement that may seem to be violated here because an instance can belong to multiple classes. It should be noted, though, that $paths_c$ is disjoint across classes, so even if the same instance is picked as seed for different classes, $paths_c$ will have no paths in common. The data we are sampling then, is the set of instance-class tuples, from which the stratas will be disjoint.

We have to decide on a sample size. A simple approach would be to choose an absolute number to pick from each class. We call this baseline approach **classSampling**, and will test different numbers later. Given that classes have great variance in the number of instances (for instance, in the Linkedmdb dataset we have 10 instances of the class "film_distribution_medium" and about 190 000 instances of the class "performance"), a proportional approach may be better. We then need to find a class sample percentage. Since some classes are small, like "film_distribution_medium", they may not be sampled at all. We therefore provided a minimum sample size of 50 for each class. We define the sample size n_c for a class c as:

$$n_c = max\left(50, \left(\frac{N_c \cdot classSamplePercentage}{100}\right)\right), \tag{4.1}$$

where N_c is the total number of instances of this class. We call this approach **proportionalClassSampling**. Later, in the experiments section, we will test with different values for $classSamplePercentage$.

The random selection of seed nodes for a $class$ is done using the SPARQL query in Figure 4.3.

```
SELECT distinct ?s WHERE {
        ?s a class .
} ORDER BY RAND() LIMIT n_c ;
```

Figure 4.3: SPARQL query for randomly selecting seed nodes for $class$, with class sample size n_c

The goal of the sampling is to represent all possible paths sufficiently. While it may seem intuitive with higher sample sizes for larger classes, we cannot know that the largest classes have the greatest variance in paths. As an illustration, we may imagine that there are 50 000 instances of the class film, and 25 000 instances of the class actor. Using the proportional approach, the film class will have more instances sampled. However, let us assume that all the instances in the film class has a path Film-*director*-Person and that one third of actors has the path Actor-*dubbed*-Film, another third the path Actor-*playedIn*-Film, and the last third Actor-*wonOscarFor*-Film. For the sake of the example, these three groups are disjoint. If we have $classSamplePercentage = 10\%$, we would sample 5 000 films and 2 500 actors. The Film-*director*-Person path would be sampled 5000 times, and the three actor paths would be sampled on average 833 times. These numbers may seem reasonable, but the problem is that the actor paths are on average, which means that it does not provide any guarantee of sufficient sampling. It may happen that one of the paths is not sampled at all.

Increasing the percentage in the **proportionalClassSampling** approach, or the number in the **classSampling** approach, should increase the likelihood of representing all paths sufficiently. However, such an approach also results in some paths being tested more than strictly necessary, which would decrease the performance. A more refined approach should sample nodes in such a way that we know that all paths are tested. One solution could be to do a depth-first or breadth-first search from all nodes in the dataset to find which nodes satisfies which paths. Then we could select nodes and be sure each path is tested. However, this would affect the performance, and almost defeat the purpose of sampling.

Instead, we propose an approach we call **pathsNotCoveredSampling**. In this approach, we first do community detection on a selected number of instances of the class (based on either **classSampling** or **proportionalClassSampling**). Then, we check if all paths originating from the class are sufficiently covered (a term we shall define soon). If all paths are sufficiently covered, we continue to the next class. If not, we select more seed nodes of this class to do community detection on. In each iteration, we add as many new seed nodes as we initially selected. That is, if we started with 1 000 seed nodes and found that these did not sufficiently covered all paths, we add another 1 000 seeds. We continue to add more seed nodes until either

1. All paths are sufficiently covered *or*
2. We have no more seed nodes to chose from *or*
3. We two times in a row get identical sets of not sufficiently covered paths *or*
4. We reach a limit on the number of seed nodes we sample from each class

For the last point, we would like to avoid that the total sample size becomes too large. We state this as a percentage, and found 30 % to be a reasonable limit after some initial testing. Note that for some classes, the selected number of instances we add in the first iteration will be more than 30 %. We allow this for those classes. The limit is mostly there to avoid too much of the large classes being sampled.

The reasoning behind the third point is that we shall not keep testing the same paths over and over again. This can happen if we are dealing with large classes, and the paths need a lot of testing to be sufficiently covered. In addition, as we noted above, $paths_c$ may include paths not present in the dataset. We should avoid to repeatedly try to sample paths that do not exist. When no change occurs between iterations, we take this as a sign that the paths in the set of not covered paths do not exist in the dataset. With regard to the second requirement, the sampling is clearly over when there are no more nodes to sample. This happens for classes with few instances.

For the first point, we need to define what we mean by a sufficiently covered path. First, we define the $pathSampleSize$ for a path p as:

$$pathSampleSize(p) = max\left(50, \left(\frac{n_{pc} \cdot pathSamplePercentage}{100}\right)\right), \qquad (4.2)$$

where c is the class of the first node in p, and n_{pc} the number of unique instances of class c satisfying p. The $pathSamplePercentage$ is a value we will make subject to testing later. Ideally, we would like to get the exact value for n_{pc}, but that would be expensive to find out. Instead, we have that n_{pc} is the number of instances of c satisfying the first edge of p, that is, the maximal number of instances of c satisfying p. This number is not too expensive to find. 50 is the minimum sample size, and serve as a lower bound to avoid a too small $pathSampleSize$.

A path p is sufficiently covered if any of the following occurs:

C1 $stats[p][possible] \geq pathSampleSize$ *or*

C2 $n_{pc} \leq stats[p][possible]$ *or*

C3 It is impossible for p to be a frequent path *or*

C4 p is hopeless

C1 states that p is sufficiently covered if it has been tested enough according to the $pathSampleSize$. A path is also sufficiently covered if there are no more nodes to test (**C2**). **C3** refers to the fact that some paths can be pruned away if they cannot be frequent paths (we will define what frequent paths are later). For instance, say that we are considered the path Film-*actor*-Person, and that there are 500 instances of Film that satisfies this path ($n_{pc} = 500$). Let us assume that we require for a path to be frequent that at least 80% of the instances includes the path in their community. If we have tested 101 nodes with this path, and none of them included the path, we already know that this path will be infrequent (the maximum frequency would be $399/500 = 0.798$). Thus, we do not have to test it anymore. **C4** is based on some initial experiments, which showed that some paths that appeared hopeless was repeatably tested. In particular, for large classes, n_{pc} and $pathSampleSize$ will be large. Thus, we can have that a path have been tested for instance 4 000 times, of which none have resulted in inclusion for the path. But if the class contains tens of thousands of instances, we keep testing it. Given that we chose instances at random, we can be reasonably sure this path will never be frequent. We consider a path p hopeless if we have:

$$(stats[p][possible] \geq 50) \text{ and } \left(\frac{stats[p][actual]}{stats[p][possible]} < 0.05\right) \qquad (4.3)$$

This definition ensure that we do not consider infrequently tested paths as not covered. In Section 4.2 we will test these different sampling approaches, and evaluate which is most suitable.

4.1.3 Path Community Detection

The most critical step in Algorithm 2 is line 9 and the call to FindPathCommunity. This method decide which paths should be included in the seed node's community and which paths should be rejected. A path is included in the community if all nodes at the end of the path from the seed node are fit to be in the community. If one or more of these nodes are unfit, then the path is rejected, and none of the nodes included in the community. The pseudocode for this method is shown in Algorithm 3.

We loop through the possible path set $paths_c$ in ascending order by length. This breadth-first like graph traversal is necessary because the fitness of a node at depth l is calculated against the community of nodes at depth $l - 1$ away from the seed node. Therefore, all paths of length $l - 1$ has to be checked before any path of length l. Having the same point of reference for all nodes at the same depth avoids the problem of order influencing which nodes or paths are added to the community. Lines 1-6 initializes some variables. There are three different kinds of paths for a seed node: paths that are included in the community ($paths_i$), paths that are rejected ($paths_r$), and empty paths ($paths_e$). The empty path set is used to avoid unnecessary checks and database calls. For instance, if we know that Work-*expression*-Expression is an empty path (that is, there are no instances of the class Expression linked to the seed node of class Work through the *expression*-predicate), we also know that Work-*expression*-Expression-*narrator*-Person is empty for the seed node. In addition, we store the instances at the end of a successful path to avoid unnecessary database calls (line 34). These provides starting points for testing continuations of an included path (line 18).

For each $path$ we loop through, one, and only one, of these four possibilities occurs:

- $path$ is a continuation of an **empty** path (checked at line 10) *or*
- $path$ is a continuation of a **rejected** path (checked at line 14) *or*
- $path$ is a continuation of an **included** path *or*
- $path$ is a path of length 1 (checked at line 15)

To check for continuation, we obtain a path p_l at line 9 that is the current $path$ without the last edge. Given our breadth first traversal, p_l is either in one of the three path sets, or it is empty. For the last two possibilities listed above, p_l is neither empty nor rejected. Consequently, $path$ can possibly be included in the community and therefore we do our fitness check on lines 15-24. If $path$ is of length 1 and thus p_l is empty, the only start instance is the seed node (line 16). Otherwise, the start instances are the instances at the end of path p_l. These are fetched at line 18. We loop through each of the instances at line 19 and find the objects at the end of the edge at line 20. For each $path$, the only new information is the last edge of the $path$ and we find the relations that match this edge. This is done through the SPARQL query seen in Figure 4.4, which is run for each $startNode$. While we treat edges as undirected, the edges may be directed in the dataset, so we check for connections in both directions. The $GetNodes$-call at line 20 essentially return the result of the SPARQL query, but removes the nodes already in the community at $depth - 1$ by using d, which is passed as an argument. d is a map where the keys are depths

Algorithm 3 FindPathCommunity

```
Input: $seedNode, paths_c, threshold$
Output: $paths_i, paths_r$
 1: $paths_i \leftarrow \emptyset$                                          ▷ Included paths
 2: $paths_r \leftarrow \emptyset$                                          ▷ Rejected paths
 3: $paths_e \leftarrow \emptyset$                                          ▷ Empty paths
 4: $i \leftarrow \emptyset$                              ▷ Map with nodes at end of each path
 5: $d \leftarrow \emptyset$                        ▷ Degrees and nodes at each depth from seed
 6: $d[0] \leftarrow$ UPDATEDEPTHSNAPSHOT($d, 0, seedNode$)
 7: for all $path \in paths_c$ do
 8:     $edge \leftarrow lastElementOf(path)$
 9:     $p_l \leftarrow path - edge$                      ▷ Obtain path without last edge
10:     if $p_l \notin paths_e$ then
11:         $rejected \leftarrow$ false
12:         $pNodes \leftarrow \emptyset$                     ▷ Nodes at the end of this path
13:         $depth \leftarrow sizeOf(path)$                       ▷ Depth from seed node
14:         if $p_l \notin paths_r$ then
15:             if $p_l == \emptyset$ then
16:                 $startNodes \leftarrow seedNode$
17:             else
18:                 $startNodes \leftarrow i[p_l]$
19:             for all $startNode \in startNodes$ do
20:                 $pNodes \leftarrow pNodes \cup$ GETNODES($startNode, edge, d, depth$)
21:             for all $node \in pNodes$ do
22:                 $change \leftarrow$ GETFITNESSCHANGE($node, d, depth, threshold$)
23:                 if $change < 0$ then
24:                     $rejected \leftarrow$ true
25:         else
26:             $rejected \leftarrow$ true
27:             $pNodes \leftarrow$ GETNODESIFEXISTS($seedNode, path$)
28:         if $pNodes \neq \emptyset$ then
29:             if $rejected$ then                    ▷ The path has some unfit nodes
30:                 $paths_r \leftarrow paths_r \cup path$
31:             else                                    ▷ The path has only fit nodes
32:                 $paths_i \leftarrow paths_i \cup path$
33:                 $d[depth] \leftarrow$ UPDATEDEPTHSNAPSHOT($d, depth, pNodes$)
34:                 $i[path] \leftarrow pNodes$
35:         else                                      ▷ The end of the path is empty
36:             $paths_e \leftarrow paths_e \cup path$
37:     else
38:         $paths_e \leftarrow paths_e \cup path$
39: return $paths_i, paths_r$
```

from 0 to $k - 1$, and where each entry contains the set of nodes in the community for that depth and below, and the internal and external degree given the set of nodes.

```
SELECT ?o WHERE {
        {startNode edge.edgeName ?o  .
        ?o a edge.toClass }
        UNION
        {?o edge.edgeName startNode  .
        ?o a edge.toClass }
}
```

Figure 4.4: SPARQL query for finding nodes at the end of an $edge$ for a $startNode$

For each of the resulting nodes, $pNodes$, at the end of $path$ that does not cause circles, we check the fitness change at line 22 (we will describe this method later). If the fitness change is negative at line 23, $path$ is rejected and none of the nodes are added to the community. Note that if we get to line 24, we can break out of the inner for loop, because there is no point checking the rest of the $pNodes$ if one of them is already rejected. If we looped through all the $startNodes$ without finding any $pNodes$, we have an empty path. The path is added to the empty path set (line 36). A successful path is found if we for all $pNodes$ never found a node that decreased the community fitness. When this happens, we store the path in $paths_i$. We also update d with the nodes we found, and set the external and internal degree of the community for this depth (lines 32-34).

If p_l is in the set of empty paths, $path$ is a continuation of an empty path, and thus empty itself. It is then added to the empty path set at line 38. Even if $path$ is a continuation of a rejected path, we are still interested in which paths further down this path are rejected or empty. If the path Work-*expression*-Expression is rejected, we would like to know if the path Work-*expression*-Expression-*narrator*-Person is also rejected, or if it is empty. This provide us with more information to decide which paths are frequently rejected, and thus improves the data on which communities are built. The $GetNodesIfExists$-call at Line 27 does this check. This method is a SPARQL query checking if the given $path$ returns any result for the $seedNode$. We are only interested if there exists nodes along a continued rejected path, not to get all nodes that do, so $pNodes$ at line 27 is either empty or contains one element. Since rejected is true, we are not going to use this element. Checking if a continuation of a rejected path is empty or not has a performance cost. We will discuss if this cost is worth it below.

The $UpdateDepthSnapshot$-call on lines 6 and 33 updates the internal and external degree for a given $depth$. Algorithm 4 shows this method. If $depth$ is 0, which only happens at the call at line 6 from Algorithm 3, we must initialize a new snapshot for $depth = 0$ (line 2). Otherwise, we fetch the existing snapshot for $depth$ (line 4). If there exists no snapshot for $depth$ and $depth > 0$, it means we must start from the snapshot at one smaller depth (line 6). A snapshot at depth l includes all the nodes from the snapshot at depth $l-1$, plus the nodes at depth l that are included in the community. The new nodes are added at line 7, and the internal and external degree calculated at lines 8 and 9. As before, the internal degree is the number of internal edges in the community multiplied by 2, while the external degree is the number of edges crossing the community boundary. Finally, we return the updated snapshot.

Algorithm 4 UpdateDepthSnapshot

```
Input: $d, depth, pNodes$
Output: $depthSnapshot$
1: if $depth == 0$ then
2:     $depthSnapshot \leftarrow \emptyset$
3: else
4:     $depthSnapshot \leftarrow d[depth]$
5:     if $depthSnapshot == \emptyset$ then
6:         $depthSnapshot \leftarrow d[depth - 1]$
7: $depthSnapshot.nodes \leftarrow depthSnapshot.nodes \cup pNodes$
8: $depthSnapshot.d_{in} \leftarrow$ CALCULATEINTERNALDEGREE($depthSnapshot$)
9: $depthSnapshot.d_{out} \leftarrow$ CALCULATEEXTERNALDEGREE($depthSnapshot$)
10: return $depthSnapshot$
```

4.1.4 Fitness Function

Because the internal degree at depth 0 is 0, the fitness of the community at this depth is 0 as well. In our approach, this will cause all nodes and their corresponding paths at depth 1 to be added, because the fitness will increase from 0 regardless of how many further links a node has. Obviously, this is a problem that can disrupt the notion of a community as representation of a logical concept. One way to solve this is to use a node centrality threshold similar to what we did in section 3.3.4 for paths of length 1. Only paths where all nodes have degree centrality below the threshold are added. Another approach could be to set the internal degree equal to the external degree. For $\alpha = 1.0$, the fitness of a seed node would then be 0.5. This could be a workable approximation if we expected the resulting communities to have a mean at about 0.5. Whether this expectation hold for RDF datasets in general is hard to tell and test because there is no way to decide what the correct communities in a dataset are, and hence the correct fitness cannot be known either. On one hand, RDF datasets are often sparse. That suggests that the fitness of communities will rarely approach 1. On the other hand, the sparseness suggests that communities found will have relatively few external links, and we also know that the dataset could be a number of disconnected graphs. Figure 4.5 provides an example of how nodes at depth 1 are treated with this scheme for initial fitness. The seed node, S, has three links to other nodes. Let us assume that these links are of different predicates, each letter *a*-*g* representing a unique predicate. Under this scheme, none of the nodes would be added. Looking at the graph, it seem reasonable that at least S, W, and X should be grouped together. With this scheme it will not happen, neither with seed nodes S, W or X. We conclude that an initial fitness of 0.5 does not work.

The node centrality threshold ($\frac{2|E|}{|V|}$) for the example graph is 2.33. With the approach that

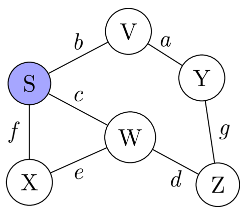


Figure 4.5: Example of inclusion of nodes at depth 1 from seed node S

nodes at depth 1 must be below the node centrality threshold, S-b-V and S-f-X will be included. We do not get W in the community through the direct path S-c-W, but the path via X will make sure W is included. It should be note though, that we are not really interested in the nodes included in the community, but rather the paths. This example illustrates that not all possible paths inside the community of nodes are regarded as included, only the ones actually used to include nodes. Here, the c edge will connect two nodes inside the community, but the path S-c-W will be in the set of rejected paths. From our perspective, this is not an error of the algorithm, but rather a feature. We are interested in exploring some common features of the path S-c-W across the dataset, and it would be wrong to let the presence of a path S-f-X-e-W for an instance influence this. With this in mind, we extend our fitness function to require that nodes at depth 1 have degree centrality below the node centrality threshold.

Algorithm 5 shows how the $GetFitnessChange$ call at line 22 in Algorithm 3 works with node centrality threshold as fitness function for nodes at depth 1, and Equation 3.2 for the other nodes. We test the fitness for an individual node based on its neighbors, obtained at line 1. For nodes one step away from the seed node, we check the fitness change using the number of neighbors, as can be seen at line 4. If this is below the node centrality threshold, the change is positive and $node$ is fit for the community.

A node at $depth > 1$ from the seed node is judged by how adding it changes the community fitness with regards to the fitness at $depth - 1$. α in the fitness function is a global parameter set before running Algorithm 2. We use the d map at line 6 to get the nodes included in the community at $depth - 1$, and the internal and external degree. Then, we calculate the fitness for this community, $f_{current}$. To check if the fitness will increase or decrease by adding $node$, we simulate such an adding. This is done by calculating the new internal and external degree at lines 10 and 11 if the node was added (lines 8 and 9 simulate the adding). When calculating the internal degree, we take the current internal degree and add the number of neighbors of $node$ already in the community times 2. The reason we are multiplying by 2, is that one connection between $node$ and a neighboring node inside the community increment the internal degree for both of them. When calculating the new external degree, we need to add the connections from $node$ to nodes outside the community, but at the same time remove the connections that was previously external but become internal by the adding of $node$. With the new external and internal degrees we calculate the new fitness, f_{new}, and the $change$ to see if adding this node will increase the fitness. Finally, the $change$ is returned and then used to decide if $node$ is fit or not in Algorithm 3.

Algorithm 5 GetFitnessChange

```
Input: $node, d, depth, threshold$
Output: $change$
1: $neighbors \leftarrow$ GetNeighborsOfNode($node$)
2: $change = 0$
3: if $depth == 1$ then
4:     $change \leftarrow threshold - neighbors.size$
5: else
6:     $d_{in}, d_{out}, nodes \leftarrow d[depth - 1]$    ▷ In and out degree, and nodes in community
7:     $f_{current} \leftarrow d_{in}/(d_{in} + d_{out})^{\alpha}$    ▷ Current fitness
8:     $ni \leftarrow nodes \cap neighbors$    ▷ Neighbors already in community
9:     $no \leftarrow neighbors \setminus ni$    ▷ Neighbors outside community
10:    $nd_{in} \leftarrow d_{in} + (ni.size * 2)$    ▷ New internal degree
11:    $nd_{out} \leftarrow d_{out} - ni.size + no.size$    ▷ New external degree
12:    $f_{new} \leftarrow nd_{in}/(nd_{in} + nd_{out})^{\alpha}$    ▷ New fitness
13:    $change \leftarrow f_{new} - f_{current}$
14: return $change$
```

4.1.5 Max Path Length

To avoid traversing the whole graph for each seed node, a max length k has to be set on path lengths. Since we run the algorithm for instances of all the classes in the dataset, long path lengths are most likely not necessary to explore all the different paths. It might seem enough to check only paths of length 1. For instance, if we check the path Artist-*made*-Release, we can refer the check of Release-*on*-Label to instances of Release. When the algorithm is finished, we can then combine paths of length 1 to longer paths. However, since instances are chosen randomly, some possible paths may not be explored. In addition, finding that Label is fit to a community containing both Artist and Release, gives more information than if the community just contains Release. We therefore require $k \geq 2$, so that some overlap occurs.

If the class graph is dense, we would expect that a shorter max path length will provide a good cover of possible paths. This is because the paths can be tested from different angles. Figure 4.6 shows an example of a dense class graph. For such a graph, the path length does not have to be long for a path to be checked for community inclusion for many different classes. For instance, the *actor* edge between Movie and Person is reachable for all the classes in the graph with $k = 2$. This means that a number of instance seeds of different classes will check this path as a subpath of other paths.

In contrast, Figure 4.7 shows an sparse class graph. If $k = 2$ here, the path Release-*contain*-Track will only be checked for instances of class Release or Record. We might be interested in exploring if communities grown from seed of class Artist will include Track or not. As mentioned, we could combine different paths, but the result may be misleading since the set of instances that was used to check the path from Artist to Release may have no overlap with the instances used to check the path from Release to Track. Ideally, then,

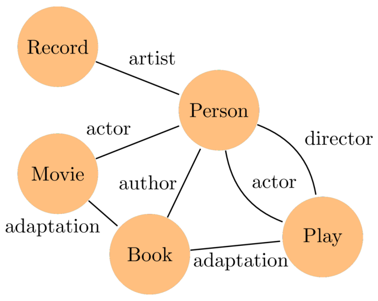


Figure 4.6: Dense class graph

should the maximum path length be equal to the number of different relationships between classes (7 in Figure 4.6 and 4 in Figure 4.7). However, this number quickly gets large and creates a performance bottleneck.

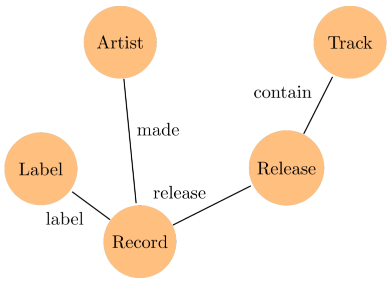


Figure 4.7: Sparse class graph

With the intuition that sparser graphs requires higher max path length, we state that the max path length k for the graph G is

$$k = 1 + \left\lfloor \left(\frac{|E|}{D(G)} \right)^{\beta} \right\rfloor, \tag{4.4}$$

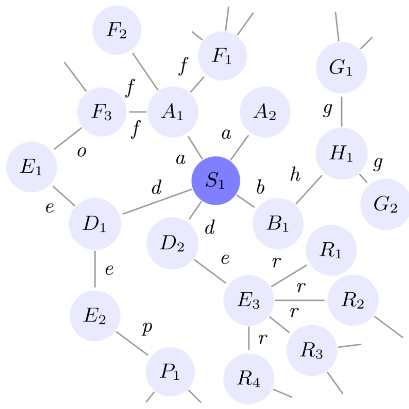
where $|E|$ is the number of edges in the class graph, $D(G)$ is the graph density in the class relationship graph found by Equation 2.1[2], and β is a scaling parameter. The expression inside the floor will always be larger than 1 (as long as $|E| > 0$), so $k \geq 2$. We found $\beta = 0.1$ to give reasonable path lengths for our datasets. We will test our algorithm with different path lengths in the experiments section below.

4.1.6 Example

To illustrate how Algorithm 3 works, we walk through an example in Figures 4.8 to 4.15. The color codes are the same as the ones in Figure 4.1. We set max path length, k, to 2,

[2]We can use this formula, which is for simple graphs, because we disregard loops in the graph and because we do not consider multi-edges

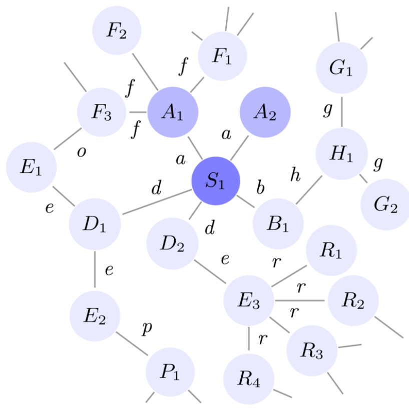
$\alpha = 1.0$, and the node centrality threshold to 3.4. The letters in the nodes indicate the class of the node, while the subscript number indicates different instances. Thus, A_1 and A_2 are two node instances (or IRI nodes in the RDF graph) of the class A. Figure 4.8 shows the initialization, or lines 1-6 in the FindPathCommunity algorithm. $paths_c$ us the different paths we are going to test for the seed node S_1. The d map is initialized with the seed node at depth 0, and the corresponding degrees. S_1 has five neighbors, so $d_{out} = 5$.



$paths_c$: {S–a–A, S–b–B, S–c–C,
S–d–D, S–a–A–f–F,
S–b–B–h–H, S–c–C–k–K,
S–d–D–e–E, S–d–D–m–M}
$paths_i$: { }
$paths_r$: { }
$paths_e$: { }
i : { }
d : 0={nodes:{S_1}, d_{in}:0, d_{out}:5}
threshold : 3.4

Figure 4.8: Initialization

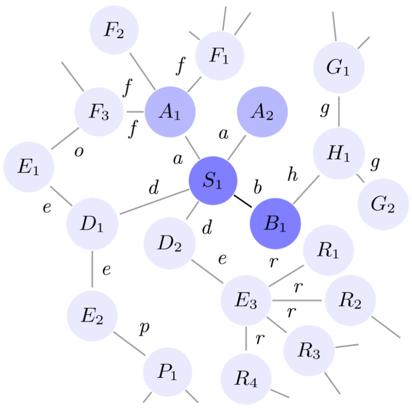
$paths_c$ is sorted ascending by length, and the first path we test is S-a-A. As we can see in Figure 4.9, this path is rejected because A_1 has 4 neighbors and therefore is above the node centrality threshold (when finding neighbors, we look at all neighbors, including those already in the community). A_2 is fit to be in the community, but that does not matter because we want consistency for all instances at the end of a given path. We see that node A_1 and A_2 are colored as rejected, and the path S-a-A is added to the rejected path set $paths_r$.



$paths_c$: {S–a–A, S–b–B, S–c–C,
S–d–D, S–a–A–f–F,
S–b–B–h–H, S–c–C–k–K,
S–d–D–e–E, S–d–D–m–M}
$paths_i$: { }
$paths_r$: {S–a–A}
$paths_e$: { }
i : { }
d : 0={nodes:{S_1}, d_{in}:0, d_{out}:5}
threshold : 3.4

Figure 4.9: Testing, and rejecting, first path S-a-A

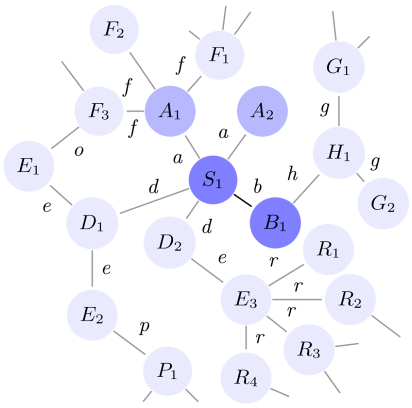
The next path to test is S-b-B, showed in Figure 4.10. The only instance to consider is B_1, and this is below the node centrality threshold. B_1 is colored as included, and S-b-B added to the included path set $paths_i$. We also make sure to store B_1 in i, so we can use it when we later test path S-b-B-h-H. d is updated on the current depth, which is 1. There was no snapshot at this depth present, so we copy from the snapshot at $depth - 1$ and then add the node to the new snapshot. The degrees are calculated by using the previous degrees and adding the new internal and external degree for the new node B_1. In this case, d_{out} was 5. We add B_1's links to nodes outside the new community (the link to H_1) and subtract the now internal link between S_1 and B_1. Then we are still at 5. The internal link makes the internal degree 2, since both S_1 and B_1 have internal degree 1.



$paths_c$: {S–a–A, S–b–B, S–c–C,
S–d–D, S–a–A–f–F,
S–b–B–h–H, S–c–C–k–K,
S–d–D–e–E, S–d–D–m–M}
$paths_i$: {S–b–B}
$paths_r$: {S–a–A}
$paths_e$: { }
i : {S–b–B = {B_1}}
d : 0={nodes:{S_1}, d_{in}:0, d_{out}:5}
1={nodes:{S_1, B_1}, d_{in}:2, d_{out}:5}
threshold : 3.4

Figure 4.10: Testing, and including, path S-b-B

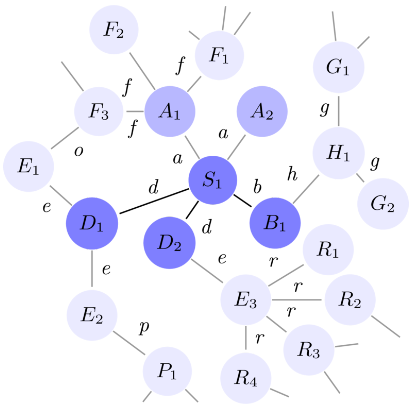
In Figure 4.11 we test the path S-c-C, which turn out to be empty (that is, $pNodes$ is empty at line 28 in Algorithm 3). The only thing that happens, is that the path is added to the empty path set $paths_e$ (line 36 in the algorithm).



$paths_c$: {S–a–A, S–b–B, S–c–C,
S–d–D, S–a–A–f–F,
S–b–B–h–H, S–c–C–k–K,
S–d–D–e–E, S–d–D–m–M}
$paths_i$: {S–b–B}
$paths_r$: {S–a–A}
$paths_e$: {S–c–C}
i : {S–b–B = {B_1}}
d : 0={nodes:{S_1}, d_{in}:0, d_{out}:5}
1={nodes:{S_1, B_1}, d_{in}:2, d_{out}:5}
threshold : 3.4

Figure 4.11: Testing the empty path S-c-C

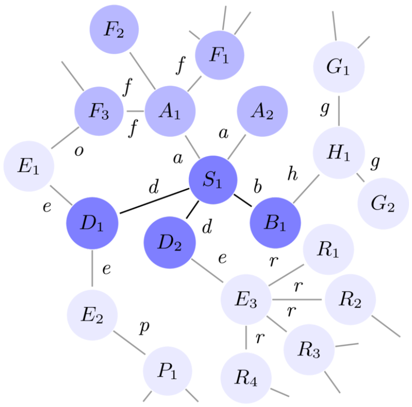
The last path of length 1, S-d-D, is tested in Figure 4.12. There are two instances of class D, D_1 and D_2. Both are below the threshold, and thus included in the community. The data structures are updated accordingly. $d_{in} = 6$, because the internal degrees are $D_1 = 1$, $D_2 = 1$, $B_1 = 1$ and $S_1 = 3$.



$paths_c$: {S–a–A, S–b–B, S–c–C,
S–d–D, S–a–A–f–F,
S–b–B–h–H, S–c–C–k–K,
S–d–D–e–E, S–d–D–m–M}
$paths_i$: {S–b–B, S–d–D}
$paths_r$: {S–a–A}
$paths_e$: {S–c–C}
i : {S–b–B = {B_1}, S–d–D = {D_1, D_2}}
d : 0={nodes:{S_1}, d_{in}:0, d_{out}:5}
1={nodes:{S_1, B_1, D_1, D_2},
d_{in}:6, d_{out}:6}
threshold : 3.4

Figure 4.12: Testing path S-d-D

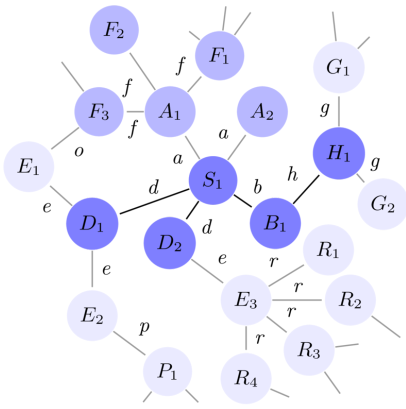
Figure 4.13 shows the test for path S-a-A-f-F. We already know that S-a-A was rejected, so this cannot be included (the check at line 14 in Algorithm 3). In this example, we choose to check for continuations of rejected paths, so we check to see if there exists some instances of class F at the end of the path. It does, meaning that the path is placed in $paths_r$. If there had not been any instances of F, it would have been an empty path.



$paths_c$: {S–a–A, S–b–B, S–c–C,
S–d–D, S–a–A–f–F,
S–b–B–h–H, S–c–C–k–K,
S–d–D–e–E, S–d–D–m–M}
$paths_i$: {S–b–B, S–d–D}
$paths_r$: {S–a–A, S–a–A–f–F}
$paths_e$: {S–c–C}
i : {S–b–B = {B_1}, S–d–D = {D_1, D_2}}
d : 0={nodes:{S_1}, d_{in}:0, d_{out}:5}
1={nodes:{S_1, B_1, D_1, D_2},
d_{in}:6, d_{out}:6}
threshold : 3.4

Figure 4.13: Testing path S-a-A-f-F, a continuation of a rejected path

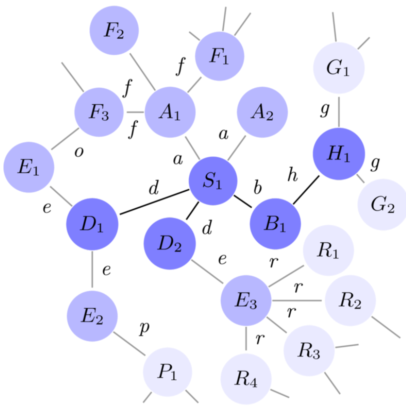
Path S-b-B-h-H is included, as we can see in Figure 4.14. We use i to get the nodes of class B satisfying the path S-b-B (the path called p_l in Algorithm 3), which in this example is B_1. From B_1 we find H_1 at depth 2, and check if adding this node increase the fitness compared to the fitness at depth 1. Since we now have depth > 1, we use the other fitness function. From d we get the fitness at depth 1, which is $\frac{6}{6+6} = 0.5$. Adding node H_1, which have two further external links, gives fitness $\frac{8}{8+7} = 0.53$. In other words, the fitness increases, and node H_1 is fit for the community. This is the only node for the path, so the path is added to $paths_i$. Since the max path length, k, is the same as the length of this path, we do not have to update i and d. d is only used with key $depth - 1$, so we have no use of $d[k]$, unless we for some reason should be interested in the fitness of the final community.



```
$paths_c$ : {S–$a$–A, S–$b$–B, S–$c$–C,
            S–$d$–D, S–$a$–A–$f$–F,
            S–$b$–B–$h$–H, S–$c$–C–$k$–K,
            S–$d$–D–$e$–E, S–$d$–D–$m$–M}
$paths_i$ : {S–$b$–B, S–$d$–D, S–$b$–B–$h$–H}
$paths_r$ : {S–$a$–A, S–$a$–A–$f$–F}
$paths_e$ : {S–$c$–C}
i : {S–$b$–B = {$B_1$}, S–$d$–D = {$D_1$, $D_2$}}
d : 0={nodes:{$S_1$}, $d_{in}$:0, $d_{out}$:5}
    1={nodes:{$S_1$, $B_1$, $D_1$, $D_2$},
              $d_{in}$:6, $d_{out}$:6}
threshold : 3.4
```

Figure 4.14: Testing path S-*b*-B-*h*-H, a continuation of a included path

Figure 4.15 shows the graph and the data structures after all paths are tested. S-*c*-C-*k*-K is empty because S-*c*-C was empty (tested at line 10 in Algorithm 3). Since S-*d*-D was included, we needed to check the path S-*d*-D-*m*-M. From i we got D_1 and D_2 to test for, but none of them have any link to a node of class M. This path was therefore empty. The path S-*d*-D-*e*-E was not empty. Both E_1 and E_2 were fit to be included in the community, but E_3 was not (the fitness of E_3 would be $\frac{8}{8+9} = 0.47$).



```
$paths_c$ : {S–$a$–A, S–$b$–B, S–$c$–C,
            S–$d$–D, S–$a$–A–$f$–F,
            S–$b$–B–$h$–H, S–$c$–C–$k$–K,
            S–$d$–D–$e$–E, S–$d$–D–$m$–M}
$paths_i$ : {S–$b$–B, S–$d$–D, S–$b$–B–$h$–H}
$paths_r$ : {S–$a$–A, S–$a$–A–$f$–F, S–$d$–D–$e$–E}
$paths_e$ : {S–$c$–C, S–$c$–C–$k$–K, S–$d$–D–$m$–M}
i : {S–$b$–B = {$B_1$}, S–$d$–D = {$D_1$, $D_2$}}
d : 0={nodes:{$S_1$}, $d_{in}$:0, $d_{out}$:5}
    1={nodes:{$S_1$, $B_1$, $D_1$, $D_2$},
              $d_{in}$:6, $d_{out}$:6}
threshold : 3.4
```

Figure 4.15: After testing all paths

The nodes of classes G, P and R are not tested because they are at depth 3 from the seed node, which is longer than the max path length we decided on. Finally, $paths_i$ and $paths_r$ are returned to Algorithm 2, and used to update the path statistics.

4.1.7 Creating Path Communities from Path Statistics

At the end of Algorithm 2 (the $createPathCommunities$ call at line 15) we have to decide which paths are interesting and which are not. Another parameter is thus the path threshold p_t. A path is considered interesting when the path frequency

$$p_f(path) = \frac{|path[actual]|}{|path[possible]|}, \tag{4.5}$$

is above p_t. The set of frequent paths, $paths_f$, is then:

$$paths_f = \{path \in stats : p_f(path) > p_t\} \tag{4.6}$$

We refer discussions of what p_t should be to the experiments section.

A problem with this approach is that we can risk having frequent paths whose starting subpaths are infrequent. As an example, say that there are 50 possible Person-*author*-Work paths, of which 17 are included. For the values of p_t we have chosen below (0.5 and 0.8), this path would be an infrequent path. Consider then the path Person-*author*-Work-*expression*-Expression, for which there are 21 possible paths. Now, given the fact that 17 of the tested instances included the first part of the path in their path communities, the path frequency would be somewhere between $\frac{0}{21}$ and $\frac{17}{21}$. Thus, it should be clear that we can have a situation where Person-*author*-Work-*expression*-Expression is frequent, while the first part of the path is not. The question is how this should be interpreted. We could interpret it in such a way that we only include Person-*author*-Work in a community, if we also have a continuation from Work to Expression. However, this quickly creates situations where the inconsistencies we are trying to avoid appear again. If a Person has both of the paths in question, only one of the Works will be added, which cause similar problems as the ones we illustrated in Figure 3.2. Two basic solutions arise: either add Person-*author*-Work to the set of frequent paths, or remove Person-*author*-Work-*expression*-Expression. The first option seems less than ideal, since we have run community detection on this path and found it to be infrequent. In addition, the community detection for this path is run on more instances than the longer path, so we should weight that result more. Thus, we choose the second option of removing a frequent path if not its subpaths originating from the start is frequent too. By "originating from the start" we mean that if a path Person-*author*-Work-*expression*-Expression is frequent, we are only interested in checking if Person-*author*-Work is frequent too, not Work-*expression*-Expression.

After removing the infrequent paths and the frequent paths with infrequent subpaths, we are left with paths from which path communities can be built for index and search purposes. A question that arises is whether we should combine paths to longer paths, but as we noted above, this may lead to the creation of paths that do not exist in the data. A related issue is what to include in a community around a class. The following may for instance be frequent paths found from the Person class: Person-*author*-Work, Person-*author*-Work-*expression*-Expression, Person-*actor*-Expression, Person-*narrator*-Expression-*expression*-Work. Should all these paths be included in the community around Person? This seems to be dependent on the query. If the query is "David Suchet", perhaps we want all available information on the Person. However, if the query is "David Suchet film", we are probably not interested in the audio books he has narrated.

In light of the discussion above, we refrain from combining paths on basis of the class information available, and instead pass this task further along the index and search process to when instance information becomes available.

4.2 Experiments

In this section, we are interested in analyzing the following questions:

E1: Which values for the path threshold p_t are reasonable?

E2: How does the path length affect the performance?

E3: How does the check of continuations of rejected paths affect the performance and the quality of the results?

E4: How does the random seed selection affect the resulting set of frequent paths?

Questions **E1** and **E2** are discussed in Section 4.2.2, while **E4** is discussed in Section 4.2.3. **E3** is discussed in both sections.

When doing the experiments, we run the tests multiple times and report mean values. We set $\alpha = 1.0$ in the fitness function for all experiments to reduce the number of variables. Table 4.2 shows the max path length k for each dataset, calculated using Equation 4.4. Additionally, we show the node centrality threshold.

Table 4.2: Max path length for datasets

Dataset	**Max path length (k)**	**Node centrality threshold**
Linkedmdb	3	4.6
Musicbrainz	2	3.3
murder.rdf	2	4.9

Table 4.3 shows the number of class paths for different max path lengths for the two datasets.

Table 4.3: Number of class paths per path length for datasets

Dataset	Path length	Number of class paths
Linkedmdb	2	2 750
	3	23 506
	4	299 072
Musicbrainz	2	582
	3	4 869
	4	41 110

4.2.1 Evaluation Metrics

When testing and reporting performance of the algorithm, the exact run times are not themselves that interesting. More interesting is the relationship between the run times for test with different parameters or approaches. The exact run times are mostly interesting when we evaluate if the performance of the algorithm is sufficiently good to be acceptable. What constitutes acceptable performance will be discussed when actual run times occur below.

Sampling of seed nodes makes the algorithm non-deterministic. To evaluate the effect this has, we look at the consistency of frequent paths over multiple runs. This makes it possible to answer question **E4**. We find the consistency by looking at how many paths are common for all tests as a fraction of all frequent paths. Let p_i be the paths returned in test run $i = 1, 2, ..., n$ and $p_c = p_1 \cap p_2 \cap .... \cap p_n$. That is, p_c is the paths that are common for all runs. Let p_a be the union of all paths: $p_a = p_1 \cup p_2 \cup ... \cup p_n$. Then we define the consistency, c, for a set of test runs as

$$c = \frac{|p_c|}{|p_a|} \tag{4.7}$$

If the returned path sets for all runs are the same, then $c = 1$, and if the runs have no common paths, $c = 0$. The seed selection is the only random part of the algorithm, so if we run the algorithm with all nodes as seed nodes, c would be 1.

In some cases, it is feasible to run the algorithm for all nodes in the dataset. For those cases, we can treat the returned set of frequent paths as the correct answer. When sampling the dataset, we wish the set of frequent paths to be as close to the correct answer as possible. By measuring how correctly the algorithm classifies different paths as either frequent or not, we can evaluate the quality of our sampling approach. The traditional information retrieval metrics of precision, recall and F-measure (sometimes called F-score) are used. Let $paths_a$ be the set of frequent paths that are considered the correct set, and $paths_f$ be the set of frequent paths returned by a given run of the algorithm, like we defined above. Adapting the formulas from [28] to concern paths instead of documents, precision for a run of the algorithm is then defined as:

$$Precision = \frac{|\text{relevant paths retrieved}|}{|\text{paths retrieved}|} = \frac{|paths_a \cap paths_f|}{|paths_f|} \tag{4.8}$$

Recall is defined as:

$$Recall = \frac{|\text{relevant paths retrieved}|}{|\text{relevant paths}|} = \frac{|paths_a \cap paths_f|}{|paths_a|} \tag{4.9}$$

F-measure is defined as:

$$F = \frac{(\beta^2 + 1) Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} \tag{4.10}$$

We choose to weight precision and recall evenly by setting $\beta = 1$. F-measure with this β is called F_1 and defined as:

$$F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{4.11}$$

4.2.2 Parameters and Performance Tests

First, we discuss question **E1**. Which values are reasonable for the path threshold p_t is strongly related to question **E3**, because the path frequency should be higher if we check continuations of already rejected path than if we do not check. This is simply because checking such paths lead to lower values of $|path[possible]|$ in Equation 4.5. When we do not check paths that are continuations of rejected paths, we count these paths as rejected. Intuitively, p_t should be somewhere between 0.5 and 1. Through some initial experiments we found 0.5 to be a reasonable value if we do *not* check these paths, while 0.8 might be a good value if we *do* check them. Note that paths of length 1 are not affected by whether we check rejected paths or not; all these paths will be checked. We therefore set $p_t = 0.8$ for paths of length 1 for both cases.

For question **E3**, we test with and without line 27 in Algorithm 3. This is the line that check continuations of rejected paths. Figure 4.16 shows the average run time for the algorithm with and without checking continuations of rejected paths and for different max path lengths. The class sample size is 1 000, meaning that we do community detection with 1 000 seed nodes from each class. We did not manage to complete the test for Musicbrainz with checking of continuations of rejected paths and path length of 4. However, testing on some of the classes suggested that it took about 15 times as long as for path length 3. As we can see, when checking for continuations of rejected paths, the run time resembles a exponential function. This is as expected, since each node that is visited usually has links to a number of other nodes which in turn has links to other nodes that also has to be checked out later. It is a bit more surprising that the tests without checking continuations are almost flat. The flatness suggests that, for these datasets, few paths of length 3 and 4 are worth considering. The result of the tests confirm this, as relatively few paths of length 3 and 4 are including in the final set of frequent paths. For **E2**, then, we find that the path length affect the performance quite heavily when we check continuations of rejected paths, and that the effect is much smaller when we do not check continuations. This may be the case due to characteristics of our datasets, and is perhaps not the general case. However, it seems reasonable that more paths of short length should be included in a community than longer paths, since length is one indication of how close instances are conceptually.

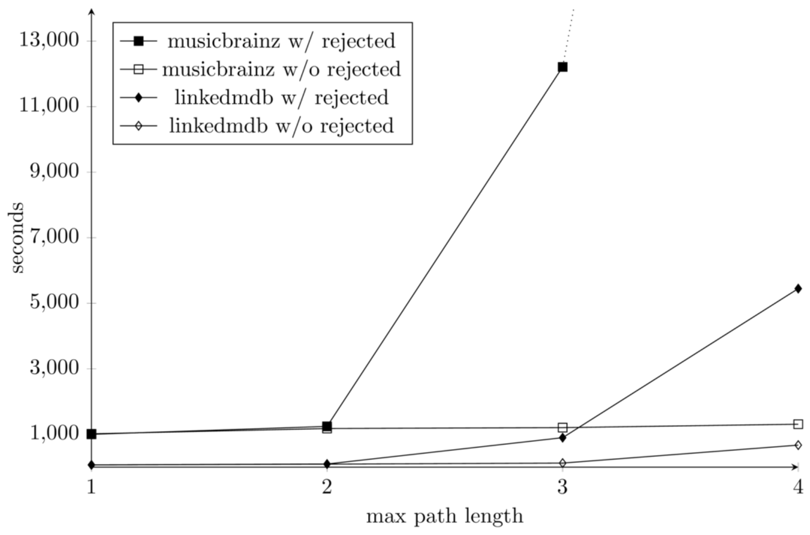


Figure 4.16: Average run time for algorithm for different path lengths with sample size of 1000 per class.

4.2.3 Sampling Tests

To motivate why we need to sample the datasets, we first discuss running the algorithm for all nodes in the dataset. For the Linkedmdb dataset with max path length $k = 3$ without checking continuations of rejected paths, the total time was about 1.5 hours, while with checking rejected paths if took about 14 hours[3]. The frequent paths found for Linkedmdb were the same regardless of whether we checked continuations of rejected paths or not. The only difference between the two approaches was a slightly smaller number of possible communities for the test with rejected path check. This is as expected, since checking the continuations of rejected paths would reveal some of these to be empty, while if we do not check it we simply treat all as rejected. All frequent paths for Linkedmdb are listed in Appendix C.

[3]For this last run time, the test was run only once. Therefore, it is possible that it is an outlier run time not representative for the actual average run time. However, we have run our algorithm multiple times throughout the work on this thesis and have yet to see extreme outliers with regard to run time.

For the Musicbrainz dataset, the algorithm took too long to finish, or we ran into memory issues. This might have to do with the implementation of the test or the algorithm itself, or the hardware of the machine used, or all of the above. Regardless, some initial results suggested that running Algorithm 3 for a node in the Musicbrainz dataset with $k = 2$ took on average about 50 millisecond. Given that there are about 35 000 000 class instances (see Table 3.1), this would have taken something like 20 days. While this indexing algorithm is not something that would have to be run that often, 20 days is too long. Thus, the need for sampling should be clear.

Table 4.4 shows the results of tests for different class sample sizes using the baseline **classSampling** approach for the Linkedmdb dataset. The path length is set to 3, following Table 4.2. One consistency test is calculated by five runs of the algorithm (that is, $n = 5$ in for the consistency expression in Equation 4.7). For each sample size, we run ten consistency tests and calculate average consistency[4]. In other words, to get the value **0.53** for average consistency with class sample size 1 000 with checking continuations of rejected paths, the algorithm is run 50 times. We also report minimum and maximum consistency, to get a feel for the span in which the consistency varies. The average run time in seconds of one run of Algorithm 2 is showed in the table, to illustrate the costs of increasing the class sample size. The number of seed nodes in the table indicates how many times Algorithm 3 is run for each run of Algorithm 2. For Linkedmdb, we know the set of frequent paths for running the algorithm on all instances. We can therefore calculate recall, precision and F_1. The average values are listed in the table.

We see that the number of seed nodes cannot be calculated by multiplying the number of classes with the class sample size. This means that there are classes in the Linkedmdb dataset that have fewer instances than the given class sample sizes. In fact, there are only 12 out of 47 classes that have more instances than 10 000 (see Appendix B). In other words, class sample size 10 000 will lead to 35 of the classes having all its instances run as seeds for community detection. A consequence of this is that all the frequent paths starting with any of these classes will be found. We should be therefore be careful to see the high values of precision and recall as proof that 10 000 instances per class is the right choice of sampling strategy. However, looking at the actual frequent paths (Appendix C), we note that most of them come from the classes with higher instance count than 10 000: only 18 of the 51 frequent paths starts from classes with instance count < 10 000. Thus, we only know that the recall cannot be lower than 0.35. It should therefore still be possible to evaluate the sampling quality.

The table expose a weakness in using consistency as a quality measure. We see that the consistency is high for class sample size of 100. However, recall and F_1 are low. Looking at the frequent paths returned in these experiments, we find that the paths typically start with a class containing few instances. This is unsurprising, since these classes have a higher proportion of their instances sampled. While the consistency is high, we would not see this result as particularly good, since most of the frequent paths are absent. In general, we find that high values of F_1 implies high consistency, but not the other way around. Therefore, when judging datasets for which we cannot compute F_1 scores (like

[4]For the slower test cases we have restricted ourselves to five consistency test. That is, 25 runs of the algorithm.

Table 4.4: Test of Linkedmdb with and without checking continuations of rejected paths with different class sample sizes using the **classSampling** approach. Max path length (k) is 3.

		class sample size		
		100	1 000	10 000
	no. of seed nodes	3 970	26 453	147 149
w/o rejected	*avg run time (sec)*	80	230	1 041
	min consistency	0.78	0.46	0.71
	avg consistency	**0.85**	**0.60**	**0.79**
	max consistency	1	0.76	0.86
	avg recall	0.25	0.51	0.83
	avg precision	0.87	0.69	0.90
	avg F_1	**0.39**	**0.58**	**0.86**
w/ rejected	*avg run time (sec)*	425	1 023	5 336
	min consistency	0.75	0.30	0.80
	avg consistency	**0.83**	**0.53**	**0.85**
	max consistency	1	0.67	0.93
	avg recall	0.25	0.51	0.82
	avg precision	1	0.72	0.97
	avg F_1	**0.40**	**0.59**	**0.89**

the Musicbrainz dataset), we must look at, and compare, the actual paths returned in tests with different class sample sizes, not just consider the consistency. What the consistency does indicate, is the level of stability. The goal, of course, is stable results of high quality, where quality is measured by how close to the correct answer the frequent path set is. In the absence of a correct answer, we manually scan the frequent paths returned in experiments to evaluate the quality.

With regard to question **E3**, we have been unable to detect any significant differences between checking and not checking continuations of rejected paths. The frequent path sets we have obtained through the running of the algorithm both for the run time experiment in Figure 4.16, and the sampling experiment for the **classSampling** approach, reveal no difference in consistency. We find that the approach of not checking continuations of rejected paths returned slightly more paths of length 2 and 3, but this was often infrequently tested paths. Additionally, as we can see in Table 4.4, the precision is slightly higher when checking continuations of rejected paths. This suggests the checking continuations of rejected paths is a bit better at avoiding false positives. However, the recall is found to be similar, and so are the $F1$ scores. We conclude that not checking continuations of rejected paths do not decrease the quality of the result in any significant way. Hence, the remaining difference between the approaches is the performance gain of not checking continuations, which is evident in both Figure 4.16 and Table 4.4. In the following, we refrain from testing paths that are continuations of rejected paths, and simply count these as rejected.

Table 4.5 shows the results of consistency tests for the Musicbrainz data using the **class-Sampling** approach. One expected consequence of increasing the class sample size, is higher average consistency, since we should expect the likelihood of covering the different paths to increase. Looking at the numbers, this expectation seems to hold for the Musicbrainz dataset, while it does not hold for the Linkedmdb dataset. An explanation could be found in Table 4.3, where we see that Linkedmdb has many more different paths between classes. It seems that Musicbrainz have more homogeneous data, that is, most of the instances of a class have the same class relationships. In Linkedmdb, this varies more, and some paths are quite infrequent. For instance, only 15 of the 85 000 instance of the class Film have a costume designer (see Appendix B and C). Finding nodes that satisfies this path by randomly sampling 100 or 1 000 of the instances requires a bit of luck.

Looking at the actual paths returned for different class sample sizes for Musicbrainz, they are mostly the same for class sample size 100 and 10 000. Higher class sample size result in more stable results, with a slightly larger core of paths that are constantly found. The paths that are not consistent over runs (that is, $p_a \setminus p_c$), typically have small values of possible communities (see Table 4.1). This means that these paths are only tested a few times. For instance, one of the paths that appear on and off as a frequent path for the Musicbrainz dataset is the path MusicGroup-*composer*-Composition. When this path is included, it is typically with a path statistic like "actual communities=2 and possible communities=2" or "actual communities=3 and possible communities=3". Of course, this could be interpreted as if there are only a few connections between MusicGroup and Composition through the *composer*-predicate, and that it is a strong connection when it occurs. The actual occurrence tell another story, though. There are in fact 1 730 MusicGroups connected to Composition through this predicate. Using only 2 or 3 instances to decide for all the 1 730 instances, suggests a flaw in the **classSampling** strategy. A counterargument could point to the fact the are about 195 000 instances of MusicGroup in the dataset, and that this path is insignificant anyway. Viewed this way, this sampling strategy does not guarantee that all paths are found and tested, but the paths that *are* found could be interpreted as the most important. If we follow this reasoning, we could introduce a post-processing step that removes paths that are infrequently tested, such as the example path here. Then we would be left with paths that are frequently included in communities, and that are thoroughly tested. With regards to our stated goal of finding human concepts, this may be a good way to make sure that the concepts we find are strong.

However, we argue that the frequency of a connection is not equal to its conceptual strength. For instance, in the Musicbrainz dataset we also have the paths MusicArtist-*composer*-Composition and SoloMusicArtist-*composer*-Composition. These paths are much more frequent than the MusicGroup-*composer*-Composition path. Thus, for a given run of the algorithm, these paths will most likely be tested more. Conceptually, however, we

Table 4.5: Test of Musicbrainz without checking continuations of rejected paths with different class sample sizes using the **classSampling** approach. Max path length (k) is 2.

	class sample size		
	100	1 000	10 000
no. of seed nodes	1 300	13 000	130 000
avg run time (sec)	764	1 467	7 449
min consistency	0.32	0.64	0.75
avg consistency	**0.48**	**0.76**	**0.82**
max consistency	0.63	0.82	0.92

would argue that MusicGroup, MusicArtist and SoloMusicArtist have equal strength in their relationship to Composition.

Table 4.6 shows tests for the **proportionalClassSampling** approach, where we instead of sampling an absolute number of each class sample a percentage of the instances. We test for the percentages 0.1%, 1% and 10%. Again, since we provide a minimum sample size, the number of seed nodes cannot necessarily be calculated by using the percentage of the total number of instances. Comparing this table to Table 4.4 and Table 4.5, we find no improvement in using a proportional approach. The results seem to be roughly the same. For Musicbrainz with sample percentage of 1% we note that the consistency is the same in all tests run. When we examine the actual paths returned in these tests, we find that there are 11 paths that are always found, while 3 paths appear on and off. These three paths are of length 2 and originating from the MusicGroup class. They connect to the Composition class via the *composer* predicate, and from Composition to MusicArtist, SoloMusicArtist, and MusicGroup, again via the *composer* predicate. In other words, these paths finds the compositions of MusicGroups which are also composed by other artists. Unsurprisingly, this is not the most common relationship in the dataset. When it appears as a frequent path, it is typically tested 5 to 12 times and with path frequency of around 0.6. This may suggest that 0.5 is a too low path frequency threshold. However, a couple of times we found the frequencies to be 0.8, so raising it might not solve the problem entirely. Additionally, deciding the path threshold on the basis of three paths from one dataset is not a good strategy. While it is not feasible to do community detection for all nodes in the Musicbrainz dataset, it is feasible to do community detection with all MusicGroups as seeds. Doing this, we find that the three mentioned paths are infrequent paths. Thus, the should not appear in the set of frequent paths.

Our last proposed sampling strategy was called **pathsNotCovered**, where we keep adding seed nodes from a class until our proposed heuristics indicate that the paths are sufficiently covered. Table 4.7 shows the result for tests on this approach for the Linkedmdb dataset. We have used proportional class sampling to decide how many seed nodes should be added in each iteration. Two path sample percentages, 20% and 50%, are tested (see Equation

Table 4.6: Tests of the **proportionalClassSampling** approach using different class sample percentages. Max path length (k) is 2 for Musicbrainz and 3 for Linkedmdb.

		class sample percentage		
		0.1	1	10[1]
Musicbrainz	*no. of seed nodes*	34 805	348 103	-
	avg run time (sec)	1 931	10 127	-
	min consistency	0.64	0 79	-
	avg consistency	**0.70**	**0.79**	**-**
	max consistency	0.79	0.79	-
Linkedmdb	*no. of seed nodes*	2 409	8 602	74 392
	avg run time (sec)	60	100	772
	min consistency	0.63	0.31	0.68
	avg consistency	**0.68**	**0.44**	**0.74**
	max consistency	0.81	0.65	0.68
	avg recall	0.23	0.30	0.70
	avg precision	0.82	0.58	0.87
	avg F_1	**0.36**	**0.39**	**0.77**

[1] We do not test this class sample percentage for the Musicbrainz dataset because it would yield too many seed nodes, and therefore take more time than acceptable both for the purpose of sampling and for testing it (one run of the algorithm would most likely take > 1 day. Thus, testing it properly with 25 runs would take at least a month).

4.2).

We first note that this approach manage to get higher F_1 values for significantly lower number of seed nodes than the two previous approaches. For instance, the **proportionalClassSampling** approach has about the same F_1 value for 74 392 seed nodes as the **pathsNotCovered** approach has for 6 000 seed nodes. It seems that targeting the classes which we infer has more paths to test, cause the intended effect of a more representative sample. Another observation we make is that this approach causes some performance overhead compared to previous approaches, when we look at the number of seed nodes. However, getting a F_1 value at around 0.75 is faster for this approach than the others. The different path percentages we test seem to have no impact on the results, whether it concerns consistency, or precision, recall and F_1.

This approach give less control over the run time of a particular run of the algorithm. For instance, for class sample percentage of 1% and path sample percentage of 20%, the fastest run time was 355 seconds while the slowest run time was 1 851 seconds. The other

Table 4.7: Tests for Linkedmdb dataset using the **pathsNotCovered** approach

	class sample percentage					
	0.1		1		10	
	path sample percentage					
	20	50	20	50	20	50
avg no. of seed nodes	6 002	6 005	28 069	25 615	160 698	155 260
avg run time (sec)	203	207	612	545	1 635	1 583
min consistency	0.54	0.54	0.66	0.67	0.75	0.79
avg consistency	**0.66**	**0.61**	**0.70**	**0.70**	**0.84**	**0.84**
max consistency	0.78	0.71	0.79	0.77	0.98	0.96
avg recall	0.65	0.66	0.74	0.77	0.87	0.89
avg precision	0.88	0.88	0.88	0.89	0.89	0.90
avg F_1	**0.75**	**0.75**	**0.81**	**0.82**	**0.88**	**0.89**

sampling approaches has more predictable run times, because the number of seed nodes is known in advance. In the **pathsNotCovered** approach there is a limit on the number of seed nodes (set to 30%), but in most cases will our other heuristics stop the iterations long before we reach the limit. This particular run time of 1 851 seconds is an outlier (the second slowest run time was 1 003 seconds), and we have not found similar outliers for other tests (for instance, for class sample percentage of 10% and path sample percentage of 20%, the run time varies between 1 581 seconds and 1 705 seconds for the 50 runs of the algorithm).

While we find this approach to be better than the other ones, we still get false positives (frequent paths found by the algorithm that are not in the set of correct frequent paths) and false negatives (frequent paths not found by the algorithm that are in the set of correct frequent paths). The false positives typically have frequencies of about 0.6. That these paths yields false positives may suggests that our path threshold p_t of 0.5 is too low. From the set of correct frequent paths, we see that most of them has frequency of 1.0, and those who do not is still above 0.8. It may be that this is a particular feature of the Linkedmdb dataset, but it could also be further indication of $p_t = 0.5$ being too low.

The false negatives are, unsurprisingly, the rare paths. The path http://xmlns.com/foaf/0.1/Agent-*linkedmdb:movie/film_company*-linkedmdb:movie/film_film_company_relationship, for instance, exist for only 1 of the 691 instances of the Agent class. Our sampling approaches are for the most part unable to catch this path. The **pathsNotCovered** approach misses it because it terminates if we for two consecutive iterations have the same set of paths not covered. For this particular path, it may not have the most severe consequences since it only concerns one instance. A solution may be to run a SPARQL query for the paths who are not sufficiently covered after the termination. The SPARQL queries would find instances that satisfies the not covered paths. Since this in most cases would be a small portion of the set of possible paths, it may be feasible to do. For paths of length 1 it is not that performance intensive to find which instances satisfies a path, but for longer paths and for large classes we found that such SPARQL queries can be very slow. Therefore, we do not find it to be a satisfactory solution.

Figure 4.8 shows the results for the **pathsNotCovered** approach for Musicbrainz. Since we have not found any effect of different path percentages, we only show result for path percentage of 20%. We find that this approach is slightly more consistent, but not by much. The paths that are inconsistently included as frequent paths in this experiment, are again the paths originating from MusicGroup. MusicGroup is often one of the classes with most iterations. This means that our approach picks up that this class has paths that need more testing. However, we struggle to find correlation between the number of iterations and the result set of frequent paths. Meaning, sometimes the paths from MusicGroup were included when we did many iterations, and sometimes they were included when we did few iterations. We find the these paths are more rarely included than for the other approaches. Since these paths are not frequent, this is a good thing, and suggests that the **pathsNotCovered** approach is an improvement.

Table 4.8: Test of Musicbrainz without checking continuations of rejected paths with different class sample sizes using the **pathsNotCovered** approach. Max path length (k) is 2. Path sample percentage is 20%.

	class sample size		
	100	1 000	10 000
avg no. of seed nodes	3 220	26 760	250 400
avg run time (sec)	1 898	3 078	11 206
min consistency	0.64	0.69	0.75
avg consistency	**0.70**	**0.76**	**0.89**
max consistency	0.79	1	0.92

With regard to question **E4**, we have seen that the random sampling of seed nodes does affect the results, both in terms of how stable the results are and the quality. Overall, we find the **pathsNotCovered** approach to be sufficiently good to be useful. Without knowing which instances satisfies a given path, we use the already tested instances to infer if more instances are needed for this class. The tests here suggests that our heuristics for inferring this has some value. However, because the tests are limited to a couple of datasets and our quality measure is only used on the Linkedmdb dataset, we cannot say that certain values of class sample percentages or path sample percentages are the correct ones to use in the general case.

4.3 Complexity Analysis

We now discuss the time complexity of Algorithm 2. Fetching information about the graph in question (line 1), such as number of edges, classes, class relationship and node instances (from which we can select seed nodes) requires a traversal of the graph. The typical ways to do this, BFS and DFS, both have complexity $O(n + m)$, where $n = |V|$ and $m = |E|$. Each node and each edge has to be visited one time.

In our setup, we loop through each of the classes in the dataset and fetch seed nodes of each class. Since RDF have no restrictions on the number of classes a instance can be of, we might in the theoretical worst-case fetch all the nodes in the graph for each class. If the number of classes is g, we have at most gn seed nodes. This is highly theoretical though, datasets where all nodes belong to all possible classes are not likely to occur. Usually an instance is of at most a couple of classes.

For each seed node we loop through $paths_c$ in Algorithm 3. We have $|paths_c| = h$. For each path, we have to do a number of operations. The most important are:

(a) Fetching $startNodes$ to explore the current path from (line 18)

(b) Finding nodes at the end of the current path (lines 19-20)

(c) Doing fitness check for the nodes we found in (b) (lines 21-24)

(d) Updating the depth snapshot (line 33)

For (a), we fetch the nodes from a hash table. A hash table has average constant time inserting and lookup, but worst-case complexity $O(n)$ for both[5]. Thus (a) is $O(n)$. For (b), we have to do a breadth-first search to depth 1 from all $startNodes$. The number of start nodes may in the worst-case scenario be all the nodes in the graph, n. We add the start nodes to a queue, and then explore their neighbors. The neighbors of the start nodes are added to the queue, and when all the start nodes have been dequeued, the queue contains the $pNodes$. Now, we have to remove the nodes in $pNodes$ already in the community. This is a set difference operation, which is $O(n)$. The BFS at has to visit at most all nodes and edges, and is thus $O(n + m)$. Because of the start nodes and the set intersection, we might have to visit all nodes three times, meaning that the complexity for (b) is $O(3n + m) = O(n + m)$.

For (c), we need to check the fitness change for each of the nodes at the end of path, which could be $n - 1$ nodes. Calculating the fitness change (Algorithm 5) requires intersection and difference operations on sets, both of which have time complexity $O(n)$. (c) thus has complexity $O((n - 1)n) = O(n^2)$. (d) requires us to add the new nodes to the depth snapshot. This is $O(n)$, because we check if the snapshot already contains the node before we add it. Additionally, we have to calculate the new internal and external degree for the snapshot. For this, we need to do the same set intersection and difference as in Algorithm 5, which is $O(n)$. (d) therefore has complexity $O(n + n) = O(n)$.

[5]Dependent on the hash function. A good hash function will avoid collisions. Our implementation uses Strings as keys and Java's default hash code: `http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode--`.

This means that the complexity of Algorithm 3 is $O(h(n + (n + m) + n^2 + n)) = O(hn^2 + hm))$. Since the number of seed nodes is at worst gn, the worst-case time complexity of Algorithm 2 is $O(n+m)+O(gn(hn^2+hm)) = O(ghn^3+ghnm)$.

This worst-case time complexity illustrates the need for sampling. Even if we are sampling, our sampling approaches cannot guarantee that the number of seed nodes is not gn. We can argue, though, that this only will happen when the graph is small. All our sampling approaches only allows all nodes of a class to be sampled if the class contains very few instances. Thus, for all nodes to be sampled, we must have that all classes have few instances. The only way this could lead to a large graph, is if we have a very large number of classes. We do not consider this to be likely. In practice, the number of seed nodes will be much smaller than gn. This may be seen by comparing the number of seed nodes in the experiments in the previous sections, to the total number of class instances in Table 3.1. In addition to this, we find that the number of $startNodes$ from which we do our BFS in (b) is far from being n. Also, the BFS will not explore the whole graph for each start node, and $pNodes$ will not be close to n either.

4.4 Summary

In this chapter we have presented an algorithm for finding frequent paths in an RDF graph. Our algorithm avoids the inconsistencies described in the previous chapter by defining a community as a set of paths. First, we require that all nodes at the end of a given path are fit to be included in the community for the path to be included. This avoids the first inconsistency of having identical class relationship be treated differently in the *same* community. Second, we make use of aggregation over multiple communities to avoid inconsistencies *across* communities. In this way, a path is treated the same way for every instance of the same class, even though individual community detection processes has yielded different results. The frequent paths that are found can be combined based on the starting class, and in that way form communities.

5 | Proof-of-Concept Solution

We now present a proof-of-concept search solution that utilizes our path-based graph indexing algorithm. The purpose of the solution is to illustrate how the path-based graph indexing algorithm finds concepts from the frequent paths, and how they may look through a user interface. Our user interface is a basic Google-like interface implemented with the Play framework[1]. We chose to index the Linkedmdb dataset, so all queries are related to movies.

5.1 Creating Queryable Objects

After finding the most frequent paths, there are a number of possibilities for how they may be used for search purposes. One possibility may be to find the nodes that best match the input keywords, and from these nodes build queryable objects by utilizing the paths originating from the classes of the different nodes. Another possibility may be to create these objects at index time. Then, the solutions could be ranked by how well the keywords match different objects, not just nodes. For more complex queries, different objects may have to be combined. This could be done in some of the ways described in Chapter 2: backward search, bidirectional search, or intersection of paths. This, however, is out of scope for this thesis, so we instead chose to index objects as a way to show how these look.

The way we create queryable objects is simple. We loop through all class instances in the dataset. For each instance, we do the following:

1. Get literals connected to the instance
2. Find the class of the instance
3. Get the frequent paths starting with this class
4. Get the nodes and corresponding literals that can be reach through the frequent paths from the instance

[1] `https://www.playframework.com/`

Combining the fetched nodes and edges give us a queryable object, which is a subgraph of the RDF graph. From this, we create an XML document for each node with the node as root. The string representation of an XML document is stored in a text field in Lucene and analyzed with the StandardAnalyzer[2]. Note that the predicates are stored as well, so a term like actor is indexed by Lucene even if it is not a literal. Type information such as film and performance is also stored and indexed. Figure 5.1 shows an excerpt from the XML document created from the instance "http://data.linkedmdb.org/resource/film/39829" (the movie *Days of Thunder*) in the Linkedmdb dataset. Tom Cruise' performance in this movie is added because "http://data.linkedmdb.org/resource/movie/film-*http://data.linkedmdb.org/resource/movie/performance*-http://data.linkedmdb.org/resource/movie/performance" is a frequent path. All frequent paths are listed in Appendix C.

```
<?xml version="1.0" encoding="UTF-8"?>
<root URL="http://data.linkedmdb.org/resource/film/39829"
    type="http://data.linkedmdb.org/resource/movie/film">
   <literal predicate="http://www.w3.org/2000/01/rdf-schema#label">
     Days of Thunder</literal>
   <literal predicate="http://data.linkedmdb.org/resource/movie/
     initial_release_date">1990-06-27</literal>
   <literal predicate="http://purl.org/dc/terms/date">1990-06-27
   </literal>
   <literal predicate="http://data.linkedmdb.org/resource/movie/
     filmid">39829</literal>
   <literal predicate="http://purl.org/dc/terms/title">
     Days of Thunder</literal>
   <child URL="http://data.linkedmdb.org/resource/performance/8258"
     predicate="http://data.linkedmdb.org/resource/movie/performance"
     type="http://data.linkedmdb.org/resource/movie/performance">
           <literal predicate="http://data.linkedmdb.org/resource/
            movie/performance_performanceid">8258</literal>
           <literal predicate="http://data.linkedmdb.org/resource/
            movie/performance_actor">Tom Cruise</literal>
           <literal predicate="http://www.w3.org/2000/01/rdf-schema#
            label">performance #8258</literal>
           <literal predicate="http://data.linkedmdb.org/resource/
            movie/performance_film">Days of Thunder</literal>
   </child>
</root>
```

Figure 5.1: Excerpt of XML document representing film object *Days of Thunder*

[2]https://lucene.apache.org/core/6_3_0/core/org/apache/lucene/analysis/standard/StandardAnalyzer.html

5.2 Query Examples

Queries can be classified according to the user's purpose. One category is simple fact-finding or lookup, where the purpose is to get an answer to a question and where the system should present the answer so that the user quickly can find it [29]. Other queries can be more exploratory, where the correct answer to a query is unknown beforehand (and possibly afterwards as well), and where learning and discovering are important goals. A classification specifically related to the Semantic Web includes the category entity search, which finds resources in the RDF graph that represents entities [4]. This may be seen as the equivalent to classic document search on the web, but with linked data objects as results instead.

Pound et al. [38] describe five categories of query types for object retrieval on the Semantic Web:

- **Entity query:** Intention is to find a particular entity.
- **Type query**: Intention is to find entities of a particular class.
- **Attribute query:** Intention is to find values of a particular attribute of an entity or class.
- **Relation query:** Intention is to find how two or more entities or classes are related.
- **Other:** Those queries that do not fit any of the above categories.

To test our search solution, we formulate queries from the different search paradigms (fact-finding and exploration), and from the different categories of Semantic Web object retrieval. We list our queries, state the category, and describe the information need:

Q1: *"the dark knight".*
This is somewhere between fact-finding and exploration. It is a entity query because we wish to find the entity representing this movie, but we also want to learn and discover something about it.

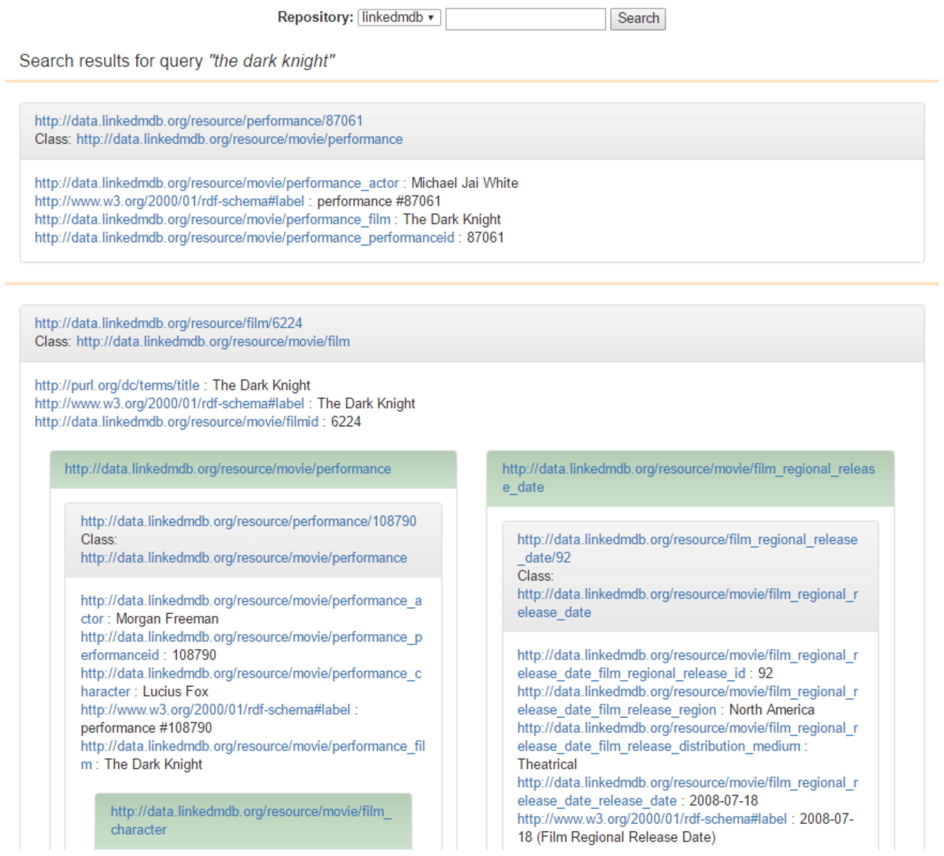
Q2: *"the dark knight freeman".*
This is more clearly a fact-finding query, and it could be categorized as an attribute query. We are interested in Morgan Freeman's role in this particular movie.

Q3: *"the dark knight director".*
Again, we are interested in a particular fact: who directed this movie? Since director is a type/class in our dataset, we can categorize it as a type query. It is however also concerned with the relation between an entity of one class (*"the dark knight"*) and a class (*"director"*), so it might also be a relation query (director is also a predicate in the dataset).

Q4: *"morgan freeman".*
Through this query, we would like to explore and learn about the entity *Morgan Freeman*.

Q5: *"morgan freeman" AND "christian bale"*.
This is a relation query, as we would like to see how these two entities are connected. In particular, we would like to retrieve the movies where both of these actors participate. As such, it is fact-finding.

The first results for **Q1** are shown in Figure 5.2. The ranking of results is according to the scoring function of Lucene. We have used the default scoring without any fine-tuning. The exact ranking is not the most important part here (although we would like the "correct" answers to appear close to the top of course), what we are interested in is the structure of the objects. We see that the "correct" answer to **Q1** comes up as the second result. This result presents some general information about the movie: the performances in the movie (which are more than what is shown in the screenshot) and the characters portrayed, and the release date in some geographical regions.



Search results for query *"the dark knight"*

http://data.linkedmdb.org/resource/performance/87061
Class: http://data.linkedmdb.org/resource/movie/performance

http://data.linkedmdb.org/resource/movie/performance_actor : Michael Jai White
http://www.w3.org/2000/01/rdf-schema#label : performance #87061
http://data.linkedmdb.org/resource/movie/performance_film : The Dark Knight
http://data.linkedmdb.org/resource/movie/performance_performanceid : 87061

http://data.linkedmdb.org/resource/film/6224
Class: http://data.linkedmdb.org/resource/movie/film

http://purl.org/dc/terms/title : The Dark Knight
http://www.w3.org/2000/01/rdf-schema#label : The Dark Knight
http://data.linkedmdb.org/resource/movie/filmid : 6224

http://data.linkedmdb.org/resource/movie/performance

http://data.linkedmdb.org/resource/performance/108790
Class:
http://data.linkedmdb.org/resource/movie/performance

http://data.linkedmdb.org/resource/movie/performance_actor : Morgan Freeman
http://data.linkedmdb.org/resource/movie/performance_performanceid : 108790
http://data.linkedmdb.org/resource/movie/performance_character : Lucius Fox
http://www.w3.org/2000/01/rdf-schema#label : performance #108790
http://data.linkedmdb.org/resource/movie/performance_film : The Dark Knight

http://data.linkedmdb.org/resource/movie/film_character

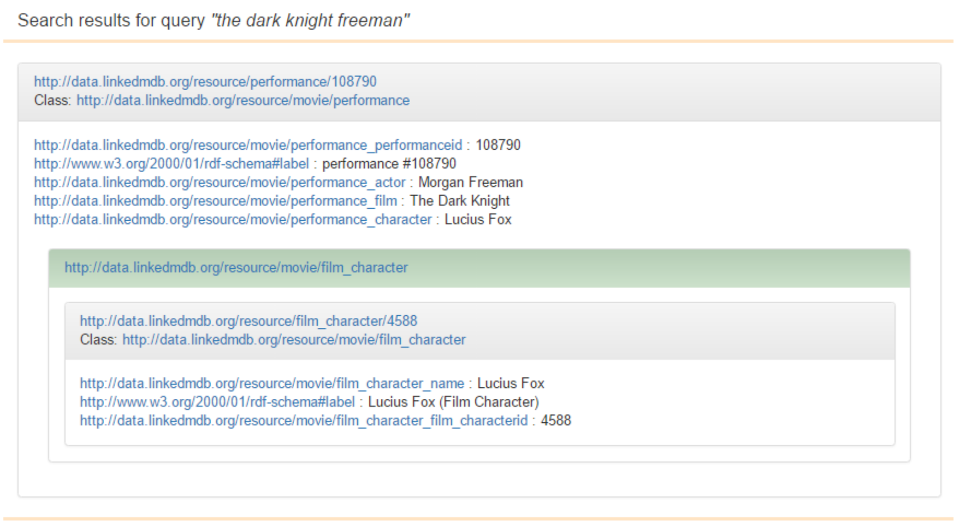
http://data.linkedmdb.org/resource/movie/film_regional_release_date

http://data.linkedmdb.org/resource/film_regional_release_date/92
Class:
http://data.linkedmdb.org/resource/movie/film_regional_release_date

http://data.linkedmdb.org/resource/movie/film_regional_release_date_film_regional_release_id : 92
http://data.linkedmdb.org/resource/movie/film_regional_release_date_film_release_region : North America
http://data.linkedmdb.org/resource/movie/film_regional_release_date_film_release_distribution_medium : Theatrical
http://data.linkedmdb.org/resource/movie/film_regional_release_date_release_date : 2008-07-18
http://www.w3.org/2000/01/rdf-schema#label : 2008-07-18 (Film Regional Release Date)

Figure 5.2: Excerpt of top two results for query **Q1:** *"the dark knight"*

In **Q2** we are not interested in all this information, but rather just Morgan Freeman's role in the movie. Figure 5.3 shows the first result for this query. Other results for this query includes the two results in Figure 5.2 and results connected to the name "freeman".


Search results for query *"the dark knight freeman"*

http://data.linkedmdb.org/resource/performance/108790
Class: http://data.linkedmdb.org/resource/movie/performance

http://data.linkedmdb.org/resource/movie/performance_performanceid : 108790
http://www.w3.org/2000/01/rdf-schema#label : performance #108790
http://data.linkedmdb.org/resource/movie/performance_actor : Morgan Freeman
http://data.linkedmdb.org/resource/movie/performance_film : The Dark Knight
http://data.linkedmdb.org/resource/movie/performance_character : Lucius Fox

http://data.linkedmdb.org/resource/movie/film_character

http://data.linkedmdb.org/resource/film_character/4588
Class: http://data.linkedmdb.org/resource/movie/film_character

http://data.linkedmdb.org/resource/movie/film_character_name : Lucius Fox
http://www.w3.org/2000/01/rdf-schema#label : Lucius Fox (Film Character)
http://data.linkedmdb.org/resource/movie/film_character_film_characterid : 4588

Figure 5.3: Top result for query **Q2:** *"the dark knight freeman"*

One piece of information that is not included in the *The Dark Knight* object, is the director. **Q3** explore if we can obtain this information by adding the keyword "director". This should be possible since there is a director-predicate in the dataset, and since the director of *The Dark Knight*, Christopher Nolan, is connected to the movie in the dataset. However, this query is unable to locate Christopher Nolan, at least in the top 50 results (which is dominated by directors by the name Knight, and *The Dark Knight* actor performances). Some searching establishes that the Christopher Nolan object is a single object not connected to any film.

The results of **Q4** consists of producer and director objects for Morgan Freeman, and his actor performances individually. A big actor object containing all his performances is returned after the individual performances. **Q5** returns two results: the *The Dark Knight* object we saw in Figure 5.2 and a similar *Batman Begins* object, which we find to be the correct answer[3].

5.3 Discussion on the Quality of the Results

The Linkedmdb dataset is somewhat limited in scope, and the relationships are not the most complex. Additionally, we have limited ourselves to five queries, though of different complexity. Given these limitations, we should be careful to make too strong conclusions on the merits of our indexing algorithm. However, overall we find the objects in the results to be relevant. We have not been able to discern any object that contains information that is clearly superfluous for the query. It may be argued that clearly superfluous information

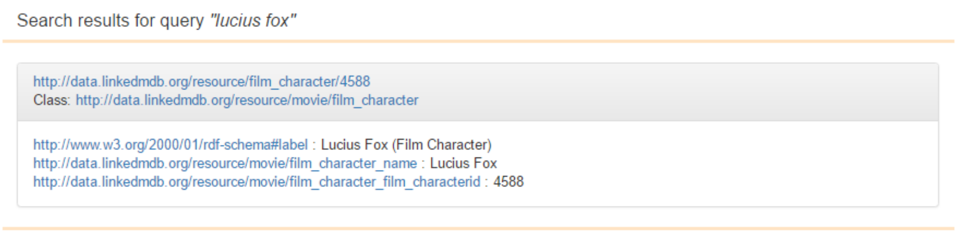
[3]The linkedmdb dataset is from 2012, apparently before *The Dark Knight Rises* was released

does not exists in a linked dataset, since the predicates connecting resources have meaning. By superfluous information, we mean information that is unnecessary in relation to the query. For instance, if the only result to **Q2** was the whole *The Dark Knight* object, it would satisfy the information need, but the relevant information would be drowned in less important information.

We find that our search solution mostly managed to find the information we sought in the example queries. In the more specific queries, like **Q2** and **Q5**, the system satisfied the information need, while at the same time provided context to the answers. For instance, the results of query **Q5** was not only two strings *"the dark knight"* and *"batman begins"*, but two objects which showed information about the objects. Additionally, the results provided "proof" that the results were indeed correct, since both actors performance were listed in the movie objects.

The first two results of **Q4**, telling us that Morgan Freeman is both a producer and director, are both relevant from a exploratory search perspective. We might prefer to have the actor object before the individual performances, because this gives a quicker overview and better structured information. One issue is that the results have quite some duplicate information. This could either be handled in the search phase or the indexing phase. If we handle it at index time, we would have to decide if performance should be a single object, as well as part of the actor object, or just one of the options. We argue that the big actor object may be the best answer to a query like **Q4**, while an individual performance object best answer more specific queries like **Q2**. Therefore, we find that duplicate information should be handled at search time.

In Figure 5.4 we show the top result for query *"lucius fox"*. The second result for this query is the one in Figure 5.3. If we are to produce a minimal substructure, the first result may be the best since it is the minimal result for the query. The argument we are making in this thesis, is that the second result is better both in itself and as a building block of combined solutions. This is because the added information of who is playing the film character is always interesting, and add context to the result by linking the character to an actor (which also contains information of which movie the character appear in).


Search results for query *"lucius fox"*

http://data.linkedmdb.org/resource/film_character/4588
Class: http://data.linkedmdb.org/resource/movie/film_character

http://www.w3.org/2000/01/rdf-schema#label : Lucius Fox (Film Character)
http://data.linkedmdb.org/resource/movie/film_character_name : Lucius Fox
http://data.linkedmdb.org/resource/movie/film_character_film_characterid : 4588

Figure 5.4: Top result for query *"lucius fox"*

The exception among our test queries was **Q3**, which was unable to satisfy our information need. This is the most complex query because one of the keywords refers to a relation between two entities, and one of those entities are not mentioned in the query. The query fails because our indexing algorithm has judged that director and film are not in the same community. This might be seen as a flaw of our indexing algorithm (in fact, we specifically named movie and director as close concepts in the introduction). Another interpretation is that *film* and *director* are two such important classes in a dataset that exclusively concerns movies, that they should be kept apart and only combined if specific queries (like this one) ask for it. In a dataset that contains information from multiple domains, we might expect most of the movie information to be grouped together. When the dataset is specifically about movies, more narrow queries may be expected, and it might be reasonable to have smaller objects that can return more fine-grained answers dependent on the query, and can be building blocks for larger objects.

Thus, whether or not our objects fit as results to queries seems to be dependent on the actual query. Sometimes they are too limited to fully answer the information need. In those cases, some combination of objects may be necessary. For other queries, such as query **Q2**, the returned object may provide enough information. In light of this, it is reasonable that the indexing algorithm creates smaller objects. Big objects may overload the user with information that are beyond a specific information need. We therefore argue that the failure of **Q3** is related to the implementation of the proof-of-concept search solution, not the indexing algorithm.

6 | Conclusion

6.1 Summary

This thesis has explored how we can utilize community detection algorithms to find concepts in an RDF graph that corresponds to how humans would conceptualize the world. We have created an index scheme for the purpose of better support to keyword search on RDF graphs. Current approaches to keyword search, both on RDF graphs and other graphs, typically find solutions that connects keyword matching nodes with minimal cost. This could for instance be the shortest paths between nodes matching the keywords. We have claimed, however, that returning concepts the corresponds to how humans conceptualize the world could make answers to queries more informative.

A state of the art review examining approaches to keyword search on graphs in general, and RDF graphs in particular, was carried out in Chapter 2. Additionally, we did a review on current algorithms for finding overlapping communities in a graph in Section 3.1. Based on this review, we performed a preliminary study where the most relevant of these algorithms were applied to some RDF graphs in Section 3.3. Through this study, we found some weaknesses in the direct application on RDF graphs. This was related to the special nature of RDF graphs, compared to other graphs. In particular, this concerned class information, and the meaning that is found in predicate edges. To eliminate these weaknesses, we developed a novel community detection algorithm for finding concepts in an RDF graph in Chapter 4. This new algorithm defines communities in RDF graphs as a set of paths, not as a collection of nodes. We run our community detection algorithm on a sample of the nodes in graph, and then aggregate the results. The paths that are most frequently included in a community, are kept and used to build concepts. Our algorithm ensures that the concepts are both consistent internally in a community, and consistent across the dataset.

The feasibility of our approach were shown through experiments. Furthermore, the usefulness of the approach is argued through a proof-of-concept search solution in Chapter 5 that uses the concepts found by the community detection algorithm.

6.2 Discussion

Our target was to find groups of nodes that correspond to how humans would conceptualize the world. Looking at the paths returned for different datasets, we would say we are halfway there. We would not describe the paths we have found as always corresponding to complete concepts. Instead, they seem to be relationships that are so strong that they rarely are more informative shown as its individual parts. Whether or not the paths, or combination of paths into objects, are directly applicable as answer to keyword search queries seems to be highly dependent on the query. In some cases, the object is enough to fulfill a information need, while in other cases we found a need to combine objects.

We may ask, then, what the merits of finding frequent paths actually are, if we still need algorithms at search time to create answers. As we have seen, most of the algorithms for keyword search on graphs find minimal substructures connecting nodes that match the keywords. With the objects produced by our indexing algorithm, the task could be to find minimal substructures connecting *objects* that match the keywords. The argument put forward in this thesis is that objects built by frequent paths can be more informative than single keyword matching nodes, and consequently that substructures built by objects can be more informative than substructures built by nodes.

Our experiments have been performed on two real world sized RDF datasets and one small RDF dataset. These datasets all concern cultural data, related to films, music or books. We should therefore bear in mind that they could have characteristics that make the results and conclusions less applicable in the general case.

Of particular concern for our algorithm was the sampling of seed nodes to grow communities from. Given the size of many RDF graphs, running the algorithm for all nodes is infeasible. The approach of choosing a number of instances from a class and the approach of choosing a percentage of each class, were both found to be less than ideal. For these approaches, the only way to get a sample that cover all the potential paths is to increase the sample size, which in turn decreases the performance. Our more refined approach of inferring which classes need more seed nodes to cover the paths, showed promising results on the Linkedmdb dataset. We cannot, however, state that this should be the sampling method of choice. The reason for this is that it is insufficiently tested. A problem with our experiments on sampling methods is that our measure of consistency proved to be purely a measure of stability, and not a measure of quality. For Linkedmdb we could, due to its size, run the algorithm for all nodes in the dataset. This allowed us to get a correct answer set of frequent paths which we could use to measure the quality of our sampling methods. Since this lacked for the Musicbrainz dataset, we must be careful to interpret the results. This applies both to how well the sampling works for this particular dataset, and whether we can generalize the results from the Linkedmdb dataset or not.

We do, however, argue that our proof-of-concept search solution demonstrates the potential of this approach to RDF data indexing. Even though our search solution was implemented without fine-tuning and without considering search result rankings (we left this solely to the default implementation in Lucene), we found that our information need for the most part was satisfied. In particular, the additional information included in an object as a consequence of the frequent paths discovered by our algorithm, provided context to the answers to queries. We found this to improve the answers. If we are to be certain of this, though, we would first need to create a more refined search solution. Secondly, this solution would need to be tested on real users. Since this is out of scope for this thesis, we are satisfied with demonstrating the feasibility, usefulness, and potential of our path-based graph indexing algorithm.

6.3 Conclusion

Our main research question was the following:

> **How can indexing of RDF graphs be implemented to better support keyword search and retrieval that corresponds to how human conceptualize the world?**

In order to answer this question we formulated three subquestions. Before we conclude on the main research question, we address these subquestions. The first was as follows:

> **RQ1: What is the state of the art for keyword search on RDF graphs?**

We presented the state of the art in Chapter 2. We found that most of the approaches to keyword search on (RDF) graphs seek minimal solutions [3, 16, 17, 18, 20, 21, 24, 34, 40, 45, 46], that is, subgraphs or subtrees that connects nodes matching the query such that some measure is minimal. This measure could for instance be the shortest path between the keyword matching nodes. Some approaches did not seek minimal solutions, and proposed to add additional information to provide context to answers [7, 27]. Closest to our research question was probably [7], where the data objects that are created include links to neighboring objects. However, this is not the same as finding concepts, since these can include relationships further away than one link. Additionally, all links one step away do not have to belong to a concept. We therefore found that our research question explores a research path not previously explored for RDF graphs.

The second subquestion was:

> **RQ2: How can graph theory be used to automatically find concepts in RDF graphs?**

Overlapping community detection algorithms were found to be the most relevant topic from graph theory to answer this question. We tested a local greedy optimization algorithm on both an RDF class graph and an instance graph. This preliminary study helped us identify some challenges unique to RDF graphs. In response to these challenges, a greedy community detection algorithm was developed that made it possible to identify frequent paths in an RDF graph. Since a concept should be consistent internally in a community, and consistent across the whole dataset, we aggregated the paths found from different community detection processes. By using a frequency threshold we could identify the most frequent paths. Paths that starts from the same class could then be said to describe a concept in the RDF graph.

Our solution utilizes community detection to index RDF graphs. To the best of our knowledge, this is a novel approach. The approach of Sozio and Gionis [45] do employ a community detection algorithm to answer a keyword query over a graph, but this is done at search time and not index time. Furthermore, this approach is concerned with the general graph case, not RDF graphs. It also seeks minimal solutions, contrary to what we do. The works of Cappellari et al. [6] and Virgilio et al. [9] are related to our proposed solution ([9] builds on [6]). This solution indexes RDF graphs by paths, which is similar to what we do. The definition of paths differ, though. A path is defined by a sequence of edge labels in their approach, while we define a path as an alternating sequence of class names and edge labels. Additionally, their approach is only concerned with paths that start from sources and ends in sinks, while we consider all types of paths. The indexing process of paths also differs significantly. Their approach is concerned with discovering paths and clustering them in different templates, while we are concerned with testing a path's relationship strength through a number of community detection processes.

Finally, our third subquestion was:

> **RQ3: How can automatically found concepts best be indexed to support keyword search on RDF data?**

This question has been answered by the implementation of a proof-of-concept search solution. Our solution uses the frequent paths to create objects, which are then turned into Lucene documents that are queryable. While we found that this method does in fact support keyword search on RDF data, we cannot claim that it is the best way. This is simply because we have not tested other methods. What we have demonstrated is the feasibility and usefulness of our approach.

When it comes to the main research question, then, we would argue that our algorithm shows promise with regards to supporting queries that contains terms corresponding to human concepts. In particular, we find that the concepts found by our algorithm can be used to give more informative answers to such queries than those approaches that seek minimal solutions.

6.4 Future Work

Given that we have implemented a proof-of-concept solution, an obvious way forward is to develop this further. We suggests three different paths forward:

> **Ranking of search results.** Our proof-of-concept solution does ranking solely based on the default Lucene scoring. As we have seen, this may lead to duplicate information. It should be investigated how this is best solved. Additionally, we suggest that answers where the keyword match the root may be better results than when keywords match nodes further away from the root node.
>
> **Employ different index and search schemes.** As we have noted, the frequent paths can be used in different index and search schemes. A line of further research could therefore be to examine different schemes, such as backward search or bidirectional

search, and how the concepts found from the frequent paths could fit into these schemes. In particular, we have found that queryable objects built from frequent paths may have to combined to fully answer certain queries. Whether this should be done at index or search time needs to be examined.

Evaluation on real users. Whether or not the search solution gives meaningful answers needs to be evaluated on real users. Such an evaluation could give important feedback on how the search solution may be improved, and also what the best ways to utilize the frequent paths for indexing are.

Bibliography

[1] Yong-Yeol Ahn, James P Bagrow, and Sune Lehmann. “Link communities reveal multiscale complexity in networks”. In: *Nature* 466.7307 (2010), pp. 761–764.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. “The semantic web”. In: *Scientific American* 284.5 (2001), pp. 28–37.

[3] Gaurav Bhalotia et al. “Keyword searching and browsing in databases using BANKS”. In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE. 2002, pp. 431–440.

[4] Roi Blanco, Peter Mika, and Sebastiano Vigna. “Effective and efficient entity search in RDF data”. In: *International Semantic Web Conference*. Springer. 2011, pp. 83–97.

[5] Phillip Bonacich. “Some unique properties of eigenvector centrality”. In: *Social networks* 29.4 (2007), pp. 555–564.

[6] Paolo Cappellari et al. “A path-oriented rdf index for keyword search query processing”. In: *International Conference on Database and Expert Systems Applications*. Springer. 2011, pp. 366–380.

[7] Gong Cheng and Yuzhong Qu. “Searching linked objects with falcons: Approach, implementation and evaluation”. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.3 (2009), pp. 49–70.

[8] Joel Coffman and Alfred C Weaver. “An empirical performance evaluation of relational keyword search techniques”. In: *IEEE Transactions on Knowledge and Data Engineering* 26.1 (2014), pp. 30–42.

[9] Roberto De Virgilio et al. “Path-Oriented Keyword Search Query over RDF”. In: *Semantic Search over the Web*. Springer, 2012, pp. 81–107.

[10] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[11] Shady Elbassuoni and Roi Blanco. “Keyword search over RDF graphs”. In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM. 2011, pp. 237–242.

[12] Santo Fortunato. “Community detection in graphs”. In: *Physics reports* 486.3 (2010), pp. 75–174.

[13] Linton C Freeman. “Centrality in social networks conceptual clarification”. In: *Social networks* 1.3 (1978), pp. 215–239.

[14] Steve Gregory. “Finding overlapping communities in networks by label propagation”. In: *New Journal of Physics* 12.10 (2010), p. 103018.

[15] Jonathan L Gross, Jay Yellen, and Ping Zhang. *Handbook of graph theory*. 2nd ed. CRC press, 2015. ISBN: 978-1-138-19966-8.

[16] Lin Guo et al. “XRANK: Ranked keyword search over XML documents”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 16–27.

[17] Hao He et al. “BLINKS: ranked keyword searches on graphs”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 305–316.

[18] Vagelis Hristidis and Yannis Papakonstantinou. “Discover: Keyword search in relational databases”. In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 670–681.

[19] Mengxia Jiang et al. “Interactive predicate suggestion for keyword search on RDF graphs”. In: *International Conference on Advanced Data Mining and Applications*. Springer. 2011, pp. 96–109.

[20] Varun Kacholia et al. “Bidirectional expansion for keyword search on graph databases”. In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 505–516.

[21] Mehdi Kargar and Aijun An. “Keyword search in graphs: Finding r-cliques”. In: *Proceedings of the VLDB Endowment* 4.10 (2011), pp. 681–692.

[22] L Kou, George Markowsky, and Leonard Berman. “A fast algorithm for Steiner trees”. In: *Acta informatica* 15.2 (1981), pp. 141–145.

[23] Andrea Lancichinetti, Santo Fortunato, and János Kertész. “Detecting the overlapping and hierarchical community structure in complex networks”. In: *New Journal of Physics* 11.3 (2009), p. 033015.

[24] Wangchao Le et al. “Scalable keyword search on large RDF data”. In: *IEEE Transactions on knowledge and data engineering* 26.11 (2014), pp. 2774–2788.

[25] Conrad Lee et al. “Detecting highly overlapping community structure by greedy clique expansion”. In: *arXiv preprint arXiv:1002.1827* (2010).

[26] Yuangui Lei, Victoria Uren, and Enrico Motta. “Semsearch: A search engine for the semantic web”. In: *International Conference on Knowledge Engineering and Knowledge Management*. Springer. 2006, pp. 238–245.

[27] Ziyang Liu and Yi Chen. “Identifying meaningful return information for XML keyword search”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 329–340.

[28] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*. Vol. 1. 1. Cambridge university press Cambridge, 2008.

[29] Gary Marchionini. “Exploratory search: from finding to understanding”. In: *Communications of the ACM* 49.4 (2006), pp. 41–46.

[30] Michael D McKay, Richard J Beckman, and William J Conover. “A comparison of three methods for selecting values of input variables in the analysis of output from a computer code”. In: *Technometrics* 42.1 (2000), pp. 55–61.

[31] Enrico Minack et al. “The Sesame LuceneSail: RDF queries with full-text search”. In: *NEPOMUK Consortium, Technical Report* 1 (2008).

[32] Tamás Nepusz et al. "Fuzzy communities and the concept of bridgeness in complex networks". In: *Physical Review E* 77.1 (2008), p. 016107.

[33] Mark EJ Newman. "Modularity and community structure in networks". In: *Proceedings of the national academy of sciences* 103.23 (2006), pp. 8577–8582.

[34] Xiaomin Ning et al. "Practical and effective IR-style keyword search over semantic web". In: *Information processing & management* 45.2 (2009), pp. 263–271.

[35] Briony J Oates. *Researching Information Systems and Computing*. Sage, 2006.

[36] Tore Opsahl, Filip Agneessens, and John Skvoretz. "Node centrality in weighted networks: Generalizing degree and shortest paths". In: *Social networks* 32.3 (2010), pp. 245–251.

[37] Gergely Palla et al. "Uncovering the overlapping community structure of complex networks in nature and society". In: *Nature* 435.7043 (2005), pp. 814–818.

[38] Jeffrey Pound, Peter Mika, and Hugo Zaragoza. "Ad-hoc object retrieval in the web of data". In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 771–780.

[39] Ioannis Psorakis et al. "Overlapping community detection using bayesian non-negative matrix factorization". In: *Physical Review E* 83.6 (2011), p. 066114.

[40] Lu Qin et al. "Querying communities in relational databases". In: *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE. 2009, pp. 724–735.

[41] Yves Raimond et al. "The Music Ontology." In: *ISMIR*. Vol. 422. Citeseer. 2007.

[42] Britta Ruhnau. "Eigenvector-centrality - a node-centrality?" In: *Social networks* 22.4 (2000), pp. 357–365.

[43] Satu Elisa Schaeffer. "Graph clustering". In: *Computer science review* 1.1 (2007), pp. 27–64.

[44] Amit Singhal. *Introducing the Knowledge Graph: things, not strings*. 2012. URL: `https://googleblog.blogspot.no/2012/05/introducing-knowledge-graph-things-not.html` (visited on 01/11/2016).

[45] Mauro Sozio and Aristides Gionis. "The Community-search Problem and How to Plan a Successful Cocktail Party". In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2010, pp. 939–948.

[46] Thanh Tran et al. "Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 405–416.

[47] W3C. *RDF 1.1 Concepts and Abstract Syntax*. 2014. URL: `https://www.w3.org/TR/rdf11-concepts/`.

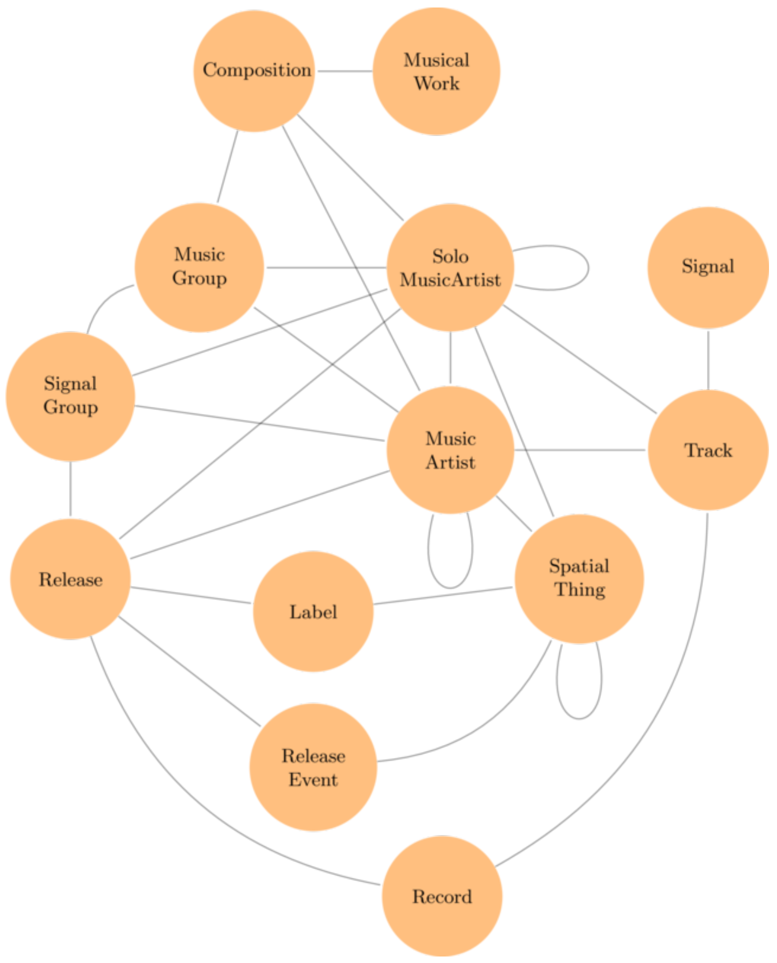
[48] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. "Overlapping community detection in networks: The state-of-the-art and comparative study". In: *ACM Computing Surveys (csur)* 45.4 (2013), p. 43.

[49] Jierui Xie and Boleslaw K Szymanski. "Towards linear time overlapping community detection in social networks". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2012, pp. 25–36.

[50] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth". In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.

A | Musicbrainz Class Graph

See [41] or `http://musicontology.com/specification/` for a description of the ontology.



B | Linkedmdb Classes

We list the classes in the Linkedmdb dataset in descending order by the number of instances of the class[1]. linkedmdb is short for http://data.linkedmdb.org/resource/. The oddlinker classes are links to other datasets, for instance DBpedia.

Class	**Number of instances**
linkedmdb:movie/performance	197 271
linkedmdb:oddlinker/interlink	162 199
linkedmdb:movie/film	85 135
http://xmlns.com/foaf/0.1/Person	74 012
linkedmdb:movie/actor	50 278
linkedmdb:movie/film_cut	45 259
linkedmdb:movie/writer	17 335
linkedmdb:movie/film_crew_gig	17 237
linkedmdb:movie/director	17 156
linkedmdb:movie/film_character	15 752
linkedmdb:movie/film_film_distributor_relationship	15 256
linkedmdb:movie/producer	14 882
linkedmdb:movie/music_contributor	4 498
linkedmdb:movie/film_job	4 004
linkedmdb:movie/editor	3 290
linkedmdb:movie/cinematographer	3 263
linkedmdb:movie/production_company	1 926
linkedmdb:movie/film_location	1 315
linkedmdb:movie/film_subject	1 249
linkedmdb:movie/personal_film_appearance	1 120

[1]The keenly observant reader will note that there are only 47 classes listed here, while we state that there are 53 classes in Table 3.1. The reason for this is that there are 6 classes with no links to other classes in the dataset. These classes are not considered for community detection.

Class	**Number of instances**
linkedmdb:movie/film_story_contributor	740
http://xmlns.com/foaf/0.1/Agent	691
linkedmdb:movie/film_festival_event	685
linkedmdb:movie/film_festival	566
linkedmdb:movie/film_regional_release_date	418
linkedmdb:movie/film_genre	409
linkedmdb:movie/film_art_director	365
linkedmdb:movie/film_costume_designer	353
linkedmdb:movie/film_company	338
linkedmdb:movie/film_production_designer	293
linkedmdb:movie/film_casting_director	273
linkedmdb:movie/country	247
linkedmdb:movie/film_set_designer	206
linkedmdb:movie/film_series	205
linkedmdb:movie/film_distributor	169
linkedmdb:movie/dubbing_performance	118
linkedmdb:movie/content_rating	107
linkedmdb:movie/film_featured_song	72
linkedmdb:movie/film_format	57
linkedmdb:movie/content_rating_system	46
linkedmdb:movie/film_film_company_relationship	36
linkedmdb:movie/film_crewmember	25
linkedmdb:movie/special_film_performance_type	11
linkedmdb:movie/film_distribution_medium	10
linkedmdb:oddlinker/linkage_run	7
linkedmdb:movie/personal_film_appearance_type	5
linkedmdb:movie/film_collection	1

C | Linkedmdb Frequent Paths

Here we list the 51 (out of 23 506 possible) frequent paths found for Linkedmdb, with $\alpha = 1$ for the fitness function, and max path length = 3. Possible is the number of instances of the path's starting class that satisfies the given path, while actual is the number of these instances who included the path in its community. The same paths were returned regardless of whether we checked continuations of rejected paths or not. The only difference was some of the numbers in the Possible column. The numbers listed here are from the test were we did not check continuations of rejected paths. As before, the italics indicates predicates. linkedmdb is short for http://data.linkedmdb.org/resource/. The interlinks are links to other datasets, such as DBpedia.

Path	**Actual**	**Possible**
linkedmdb:movie/film- *linkedmdb:movie/costume_designer*- linkedmdb:movie/film_costume_designer	15	15
linkedmdb:movie/content_rating_system- *linkedmdb:movie/content_rating_system*- linkedmdb:movie/content_rating	17	19
linkedmdb:movie/actor- *linkedmdb:oddlinker/link_source*- linkedmdb:oddlinker/interlink	2220	2220
linkedmdb:movie/film- *linkedmdb:movie/film_regional_release_date*- linkedmdb:movie/film_regional_release_date	224	224
linkedmdb:movie/film_company- *linkedmdb:movie/film_company*- linkedmdb:/movie/film_film_company_relationship	1	1
linkedmdb:movie/film_festival- *linkedmdb:movie/film_festival_event*- linkedmdb:movie/film_festival_event	170	170

Path	Actual	Possible
linkedmdb:movie/actor- *linkedmdb:movie/performance*- linkedmdb:movie/performance	43177	43616
linkedmdb:movie/film_job- *linkedmdb:movie/film_crew_gig_film_job*- linkedmdb:movie/film_crew_gig	3595	3595
http://xmlns.com/foaf/0.1/Person- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crew_gig	24	24
linkedmdb:movie/film- *linkedmdb:movie/film_of_distributor*- linkedmdb:movie/film_film_distributor_relationship	13861	13861
linkedmdb:movie/film_distributor- *linkedmdb:movie/film_distributor*- linkedmdb:movie/film_film_distributor_relationship	166	166
linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crewmember	24	24
linkedmdb:movie/film_character- *linkedmdb:movie/film_character*- linkedmdb:movie/performance	15131	15314
linkedmdb:movie/film- *linkedmdb:movie/film_cut*- linkedmdb:movie/film_cut	11628	11628
linkedmdb:movie/film- *linkedmdb:movie/personal_film_appearance*- linkedmdb:movie/personal_film_appearance	328	328
http://xmlns.com/foaf/0.1/Person- *linkedmdb:oddlinker/link_source*- linkedmdb:oddlinker/interlink	2220	2220
linkedmdb:movie/music_contributor- *linkedmdb:oddlinker/link_source*- linkedmdb:oddlinker/interlink	678	678
linkedmdb:movie/film- *linkedmdb:movie/film_film_company_relationship*- linkedmdb:movie/film_film_company_relationship	22	22

Path	Actual	Possible
linkedmdb:movie/film_distribution_medium- *linkedmdb:movie/film_distribution_medium*- linkedmdb:movie/film_film_distributor_relationship	7	7
linkedmdb:movie/country- *linkedmdb:oddlinker/link_source*- linkedmdb:oddlinker/interlink	247	247
linkedmdb:movie/writer- *linkedmdb:oddlinker/link_source*- linkedmdb:oddlinker/interlink	6661	6661
linkedmdb:movie/film_crewmember- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crew_gig	24	24
linkedmdb:movie/film_film_company_relationship- *linkedmdb:movie/film_company*- http://xmlns.com/foaf/0.1/Agent	1	1
linkedmdb:movie/personal_film_appearance_type- *linkedmdb:movie/personal_film_appearance_type*- linkedmdb:movie/personal_film_appearance	5	5
linkedmdb:movie/film- *linkedmdb:movie/costume_designer*- http://xmlns.com/foaf/0.1/Agent	15	15
linkedmdb:movie/film- *linkedmdb:movie/performance*- linkedmdb:movie/performance	54597	54835
linkedmdb:movie/performance- *linkedmdb:movie/film_character*- linkedmdb:movie/film_character	16571	17822
linkedmdb:movie/film_film_company_relationship- *linkedmdb:movie/film_company*- linkedmdb:movie/film_company	1	1
linkedmdb:movie/film- *linkedmdb:movie/film_crew_gig_film*- linkedmdb:movie/film_crew_gig	389	389
linkedmdb:movie/film- *linkedmdb:movie/film_featured_song*- linkedmdb:movie/film_featured_song	66	66

Path	**Actual**	**Possible**
linkedmdb:oddlinker/linkage_run-*linkedmdb:oddlinker/linkage_run*-linkedmdb:oddlinker/interlink	7	7
linkedmdb:movie/film-*linkedmdb:oddlinker/link_source*-linkedmdb:oddlinker/interlink	21364	21364
linkedmdb:movie/film-*linkedmdb:movie/dubbing_performance*-linkedmdb:movie/dubbing_performance	60	60
linkedmdb:movie/film_crew_gig-*linkedmdb:movie/film_crewmember*-http://xmlns.com/foaf/0.1/Person	24	24
http://xmlns.com/foaf/0.1/Agent-*linkedmdb:movie/film_company*-linkedmdb:movie/film_film_company_relationship	1	1
linkedmdb:movie/actor-*linkedmdb:movie/performance*-linkedmdb:movie/performance-*linkedmdb:movie/film_character*-linkedmdb:movie/film_character	7865	8951
linkedmdb:movie/film_job-*linkedmdb:movie/film_crew_gig_film_job*-linkedmdb:movie/film_crew_gig-*linkedmdb:movie/film_crewmember*-linkedmdb:movie/film_crewmember	22	22
linkedmdb:movie/film-*linkedmdb:movie/film_film_company_relationship*-linkedmdb:movie/film_film_company_relationship-*linkedmdb:movie/film_company*-linkedmdb:movie/film_company	1	1
linkedmdb:movie/film-*linkedmdb:movie/film_crew_gig_film*-linkedmdb:movie/film_crew_gig-*linkedmdb:movie/film_crewmember*-http://xmlns.com/foaf/0.1/Person	2	2

Path	**Actual**	**Possible**
linkedmdb:movie/film- *linkedmdb:movie/film_film_company_relationship*- linkedmdb:movie/film_film_company_relationship- *linkedmdb:movie/film_company*- http://xmlns.com/foaf/0.1/Agent	1	1
linkedmdb:movie/film- *linkedmdb:movie/performance*- linkedmdb:movie/performance- *linkedmdb:movie/film_character*- linkedmdb:movie/film_character	4653	5478
linkedmdb:movie/film_job- *linkedmdb:movie/film_crew_gig_film_job*- linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- http://xmlns.com/foaf/0.1/Person	22	22
linkedmdb:movie/film- *linkedmdb:movie/film_crew_gig_film*- linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crewmember	2	2
linkedmdb:movie/film_job- *linkedmdb:movie/film_crew_gig_film_job*- linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- http://xmlns.com/foaf/0.1/Person- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crew_gig	22	22
linkedmdb:movie/film- *linkedmdb:movie/film_film_company_relationship*- linkedmdb:movie/film_film_company_relationship- *linkedmdb:movie/film_company*- linkedmdb:movie/film_company- *linkedmdb:movie/film_company*- linkedmdb:movie/film_film_company_relationship	1	1
linkedmdb:movie/film- *linkedmdb:movie/film_crew_gig_film*- linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- http://xmlns.com/foaf/0.1/Person- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crew_gig	2	2

Path	Actual	Possible
linkedmdb:movie/film- *linkedmdb:movie/performance*- linkedmdb:movie/performance- *linkedmdb:movie/film_character*- linkedmdb:movie/film_character- *linkedmdb:movie/film_character*- linkedmdb:movie/performance	4498	5478
linkedmdb:movie/actor- *linkedmdb:movie/performance*- linkedmdb:movie/performance- *linkedmdb:movie/film_character*- linkedmdb:movie/film_character- *linkedmdb:movie/film_character*- linkedmdb:movie/performance	7433	8951
linkedmdb:movie/film- *linkedmdb:movie/film_crew_gig_film*- linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crewmember- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crew_gig	2	2
linkedmdb:movie/film_job- *linkedmdb:movie/film_crew_gig_film_job*- linkedmdb:movie/film_crew_gig- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crewmember- *linkedmdb:movie/film_crewmember*- linkedmdb:movie/film_crew_gig	22	22
linkedmdb:movie/film- *linkedmdb:movie/film_film_company_relationship*- linkedmdb:movie/film_film_company_relationship- *linkedmdb:movie/film_company*- http://xmlns.com/foaf/0.1/Agent- *linkedmdb:movie/film_company*- linkedmdb:movie/film_film_company_relationship	1	1