Hans Sandbu

Spatio-temporal Keyword search on RDF graphs

Master's Thesis in Computer Science, Spring 2020

Norwegian University of Science and Technology

Abstract

This thesis looks at methods for using keywords for search in graph structured data, with the goal of finding what aspects are needed for accuracy and speed. To accomplish this, some previous methods for keyword searching will be recreated, and then extended to incorporate both spatial and temporal searches.

Sammendrag

Husk at hvis du er en norsk student og skriver masteren din på engelsk, så *må* du lage et sammendrag på norsk.

(If you are a non-Norwegian student, it is not obligatory to include an abstract in Norwegian.)

Preface

Spatiotemporal keyword search on RDF graphs.

Hans Sandbu
Trondheim, 10th May 2020

Contents

List of Figures

List of Tables

1 Introduction

In computer science, graphs are structures sed for representing relationships between objects. A graph consists of vertices connected by edges, where the vertices represent the objects, and the edges represents the relationship. A graph can take on different shapes, giving the graph special properties. By giving the edges a direction the graph can take on further properties.

When using graphs as a form of storage, the vertices hold the data, and the edges are given an extra property describing the relation. Combining such a data store with common traversal methods for graphs results in an efficient extraction model for relational data. Comparing graph storage to relational storage, such as SQL, graph storage generally outperforms relational storage on structural queries, but not on data queries [15].

Using standardized models such as the Resource Description Framework (RDF), which is created specifically for graph based data, have made the distribution of graph data easier. One pard of RDF describes so called triples. A triple is a description of nodes, and the connection, using the structure “subject-predicate-object”. In this structure the subject and object can be considered vertices, and the predicate is the edge. In addition to forming the relation between the two vertices, the predicate also holds a type of relation. This makes it possible to apply reason to the data set, and derive new information from what is already there.

There are few applications using RDF as an information storage system. Some of the more well known are DBPedia[2], YAGO[12], and Creative Commons. These applications often consists of a large linked data set, or in the case of Creative Commons, it is used for embedding licenses. Creating methods that allows for easy access to information in RDF data makes it possible to create applications with utility. In this thesis methods for efficient keyword search on spatial and temporal data will be explored.

1.1 Background and Motivation

Traditionally relationships between pieces of data have been difficult for computers to model. Regular SQL databases requires expensive joining to be able to display relationships between entities. As more data is generated, there are also more relations between the data. Exploring these relations can be done by using RDF or other graphs, but for a regular person this can be difficult. By proving that fast and accurate keyword search of spatiotemporal RDF graphs is possible new utilities for exploration can be built.

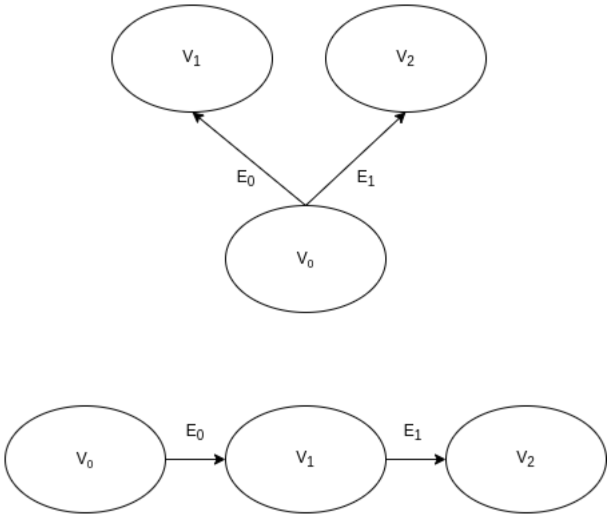


Figure 1.1: Simple Directed graphs.

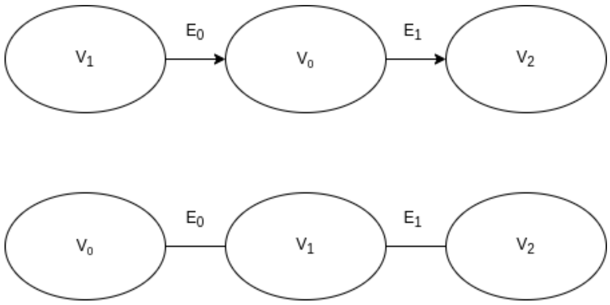


Figure 1.2: Example of Directed an non-directed graphs.

1.2 Goals and Research Questions

Goal *With this thesis the goal is to determine if and how spatiotemporal keyword searching in large scale RDF graphs is possible to do in fast and accurate.*

Accomplishing this goal proves that RDF graphs can be used as a tool for structuring data related to real world places. There is a lot of data that can be placed at one or more real locations, either directly, or indirectly. By using RDF graphs it is possible to model how different places are connected through some pice of data, and how different pieces data can be related to real places.

Research question 1 *How can spatiotemporal data be integrated into exiting keyword query methods for RDF data.*

Research question 2 *What methods can be used to create more effective queries on RDF data.*

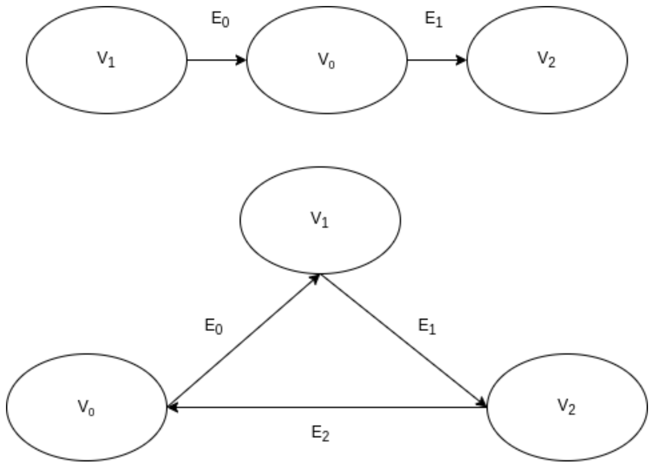


Figure 1.3: Directed graph with and without cycles.

Research question 3 *How do spatial and temporal RDF query methods differ from from other query methods.*

1.3 Research Method

This thesis will build on previous keyword search approaches for RDF graphs, and determine what methods can be expanded to incorporate spatiotemporal queries. A method for spatiotemporal search will be created, and methods for improvement will be tested, with the goal of finding what aspects of the search method have most effect for speed and accuracy.

2 Related Work

2.1 Keyword search on graphs

Keyword search on RDF graphs often follows a set of common strategies. One method is to find nodes containing one or more keyword, then following the edges from the nodes to explore the graph, and find subgraphs where the combined nodes contain as many keywords as possible while also spreading as little as possible. BLINKS [7] propose such a method, in combination with indexing and cost balancing for expanding clusters of accessed nodes. A similar approach is used by the authors in [4] where each node has an associated document containing terms from the triple. When querying the keywords are matched with these documents, creating lists based on the matching keywords, and a subgraph is constructed by joining matches from different lists.

Another strategy for keyword search in RDF graphs is to infer triples from the query. One such query system is used in AquaLog [8]. This method processes the input into a triple based representation, based on a linguistic model, and then further processes the triplets into what they call “query triples.” Creating structured queries through inference is also done in [14]. Here the query is first used to find nodes containing some part of the query, then the graph is explored to find a connection between the nodes. The result is a series of subgraphs connecting nodes that contain part of the query. Each of the subgraphs are in turn used to create a conjunctive query with edges mapped to predicates, and nodes to subjects or objects.

Ranking and scoring the results of a search is also needed for evaluating the different methods and algorithms. A common element for ranking the results is to look at the span of the subgraphs or trees returned from the search. The shorter distance between all nodes, the more accurate a result should be. Of the above mentioned papers, three [7, 4, 14] use some form of minimum spanning tree or graph when scoring or ranking the results. In addition to the minimum spanning graph, the results can be ranked by other factors.

BLINKS adds a scoring system where shared vertices are counted multiple time, once for each node connected to it. This is done to score trees with nodes close to the root higher than nodes further away, even if the further nodes have many shared edges. The content of the nodes are also scored based on an IR style TF/IDF method. In the paper by [4] the minimum tree are ranked by a probabilistic model, and a language model. The probabilistic model scores a result based on the average probability for a term to occur in a triple in the subgraph. In addition, the language model is used to score some keywords higher based on what part of the triple they are found in. This triple scoring is done by weighting words based on the structure of the triples they are found in, so keywords that occur more often in predicates are scored higher if they are found in a predicate. The final paper, [14], adds popularity, and keyword matching to the minimum spanning graph. Popularity is calculated based on how many edges a node has, so that the more connected a node is, the less cost a path through that node has. The keyword matching score is based on keyword matches in a node, but is also weighted based on syntactic and semantic similarity, which is in turn done by using WordNet data.

2.2 Spatial search on RDF graphs

Keyword searching and ranking spatial RDF graphs is quite similar to regular RDF keyword searching. [11] outlines methods that incorporates spatial data into the search. These methods is similar to some of those previously mentioned here [14, 4]. The most substantial difference is the use of R-trees to index the spatial dimension of the graph. This is done so that a subgraph can be rooted both at a real point in the world, and nodes in the graph close to the real world point. The root is used as a point for traversing the graph, and like the previous methods, the goal is to find a minimum subgraphs. The subgraphs are ranked based on how close the root node is to the selected point, the size of the tree, and how well the result tree fits the query words.

R-trees are designed around the spatial dimension, but some work on modifications to include a temporal dimension has been done. Some research done on spatiotemporal tree structures include [13, 16]. Most of the spatiotemporal trees are however created to be able to query and index moving or frequently updated objects. To include a temporal dimension some differences in the tree structures are made. Most approaches builds on the R-tree, but modifies the tree in different aspects. For the purpose of this thesis a different tree structure is not needed.

3 Background Theory

3.1 Definitions

Result tree: *Result r for a given query q with tokens Qt on an RDF graph* $G\langle V, E\rangle$*, where the result forms a minimum spanning tree* $Tm\langle V', E'\rangle$*, V' contains a set of tokens Tt, Tm is rooted in a place vertex p, so that* $V' \subseteq V$*,* $E' \subseteq E$*, and* $Tt \subseteq Qt$

All results from a query is given as a minimum spanning tree. This tree is call a *Result Tree*. A query can have multiple result trees, where each result tree is rooted in a unique place node, and each tree contains at least one query word.

Root: *Any GeoEntity node used as a start when traversing the graph, and as the place node in a result tree*

Accuracy: *Score given to a result tree, where score is based on the number of nodes n in the result tree, node distance nd from root, query q, and number of query words hit h so that* $\mathcal{F}_h = (h_i / |q| : i \in \mathcal{I})$ *and* $\frac{\sum \mathcal{F}_h}{n*(nd+1)}$

Knowledge base, ontology and knowledge graph are terms with multiple definitions, and are often used interchangeably. The definitions here are used to clarify what they mean in this paper, and is not a definitive definition.

3.1.1 Knowledge Base

The term knowledge base can have many definitions, often because the terms knowledge base, ontology, and knowledge graph is used interchangeably [3]. In this paper the term knowledge graph will be defined as follows: "A knowledge base is a data set with some formal semantics. This could include multiple axioms, definitions, rules, facts, statements, and other primitives." [1]

3.1.2 Ontology

Like knowledge base, ontology does not have one clear definition. This is because the term have many different interpretations, and is often confused with other terms [6]. Most definitions of ontologies say that ontologies represent a schema, basic theory, or conceptualization of a domain, which knowledge bases do not. [1] To differentiate ontologies and knowledge bases in this paper the definition of an ontology will be "An ontology is an extended knowledge base that allows for semantic modeling of knowledge." With this definition, ontologies can be considered a specialized form of knowledge bases.

3.1.3 Knowledge Graphs

A broad definition of knowledge graph is “A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge.” [3] This definition is encompasses multiple technologies. In this paper we will use the more specific definition that fits the data used “We define a Knowledge Graph as an RDF graph. An RDF graph consists of a set of RDF triples where each RDF triple (s, p, o) is an ordered set of the following RDF terms: a subject $s \in U \cup B$, a predicate $p \in U$, and an object $U \cup B$ L. An RDF term is either a URI $u \in U$, a blank node $b \in B$, or a literal $l \in L$” [5]

Subject

In a RDF triple the subject is a URI or a blank node. The URI when used in the subject identifies an entity, or is an alias, different language or other variation of an entity. This subject can have relations to objects describing the same entity.

Predicate

Predicates are always a URI. URIs used for predicates differ from the ones used for subjects and objects in that predicates are of a type. A type is an identifier used to describe the relation between the subject and object.

Object

Objects have the widest range of possible entries. Like subjects and predicates, objects can be URIs. Object URIs can be entity identifiers like subjects, they can be class identifiers, or they can contain some data and a data type. Blank nodes are just that, blank, and literals are an atomic value.

3.1.4 Facts and entities

A fact is a term often used when describing knowledge bases, ontologies and knowledge graphs. Usually a fact is the smallest piece of information in such a system. In a knowledge graph this is a single RFD triple. Another term often used is entity. An entity is a collection of facts, usually from the same article. Entities can be linked together through facts. In such a fact, one entity is used as the subject, the predicate describes the relation, and the object is the other entity. In such a relation both entities will be URIs.

3.2 Existing knowledge graphs and ontologies

Currently two of the largest open technologies for knowledge graphs are Yago and DBPedia. Both these projects use automatic extraction from Wikipedia to create the graph, but differ in the ontology used to build the graphs. Yago also includes data from WordNet and GeoNames to accurately assign entities to classes. Both projects use RDF triples to create a knowledge graph.

3.2.1 Uses for Knowledge graphs

One of the uses of knowledge graphs today is to find and display a info box in search engines. This information is a compact set of facts that tries to fit the search query. Because of the graph structure of knowledge graphs the information in the info box can be adapted to the query by choosing the predicates and related facts closest related to the query. This makes it possible to create a set of information that can give the user a quick overview of the information retrieved by the query.

3.2.2 Yago

Yago is an acronym for Yet Another Great Ontology, and is main data source in this paper. The project describes it self as a knowledge base[12] and an ontology [9], but is often described as a knowledge graph by others. In this paper Yago is described as a knowledge graph.

Temporal data

Yago have many entities with facts describing date and spatial data.[12] Date facts follows the ISO 8601 format, YYYY-MM-DD, and introduces # as a wildcard symbol. A fact can only hold information on a single point in time, and uses yagoDate as a data type in addition to information on the date.[12] In a date fact, the object holds the date information, and the predicate describes a connection between subject and date. “Nidaros_Cathedral wasCreatedOnDate 1300-##-## .” In this example the predicate wasCreatedOn is used to describe a relation between the subject and a date.

To describe a time span two facts are required. One of the facts describes a start date, and the second en end date. Since an entity can have multiple date facts connected, all date predicates are also assigned to a class. Start dates are assigned to a predicate with a type that has a “creation” class, such as “StartedOnDate”. End dates are assigned to “destruction” type predicates. This makes it possible to deduce a time span for a given subject and predicate combination.[12]

Spatial data

Yago only contains permanent spatial data for entities on earth. This means that entities like cities, buildings, rivers and mountains are given a spatial dimension. In addition, events, people, groups and artifacts can be given a spatial dimension by relating the entity to a specific place. All spatial facts must have a predicate that fall under the yagoGeoEntity class, and all objects used in a fact with a yagoGeoEntity must have a relation containing both “hasLatitude” and “hasLongitude”.

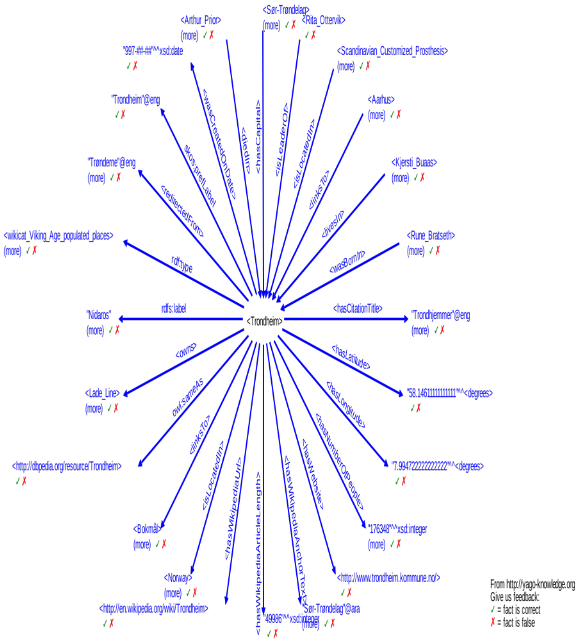


Figure 3.1: Trondheim as a subject in YAGO

3.2.3 DBPedia

3.2.4 Jena

Apache Jena is a set of tools for working with RDF and other semantic web, and linked data. The framework contains tools for querying with SPARQL, a query language made specifically for RDF graphs. Jena also contains REST-style SPARQL endpoints making the RDF data easily accessible. Using the existing standards makes it easy to use existing data sets, such as Yago or DBPedia, and build utility on top of that data using some of the tools Jena provides.

3.3 Introducing Figures

3.4 Introducing Tables in the Report

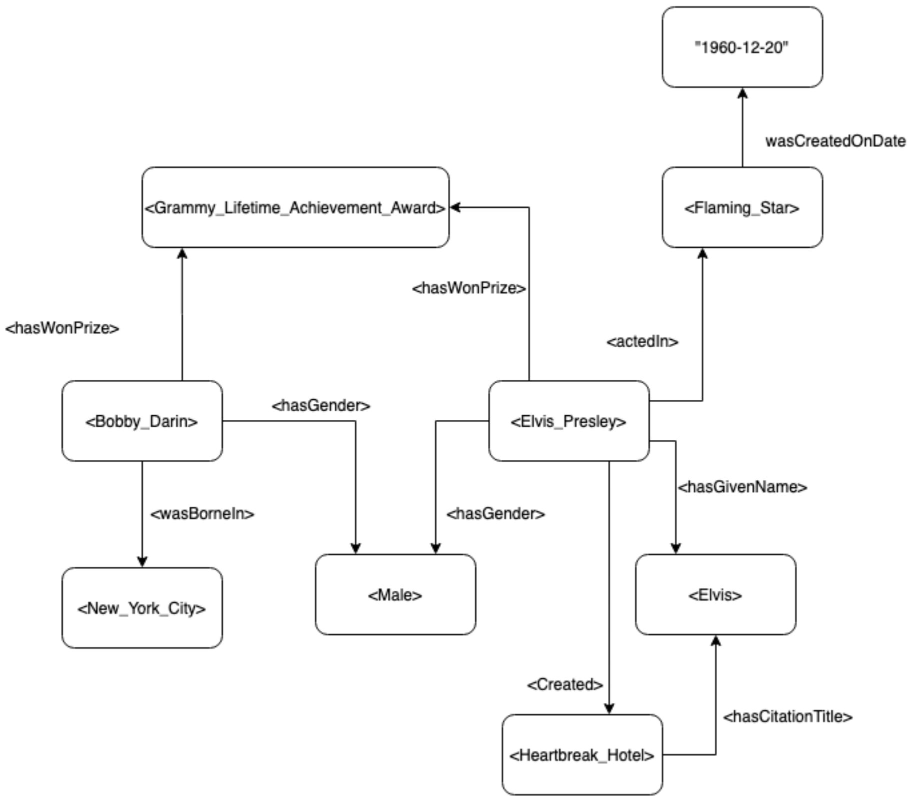


Figure 3.2: Connections in Yago around Elvis

4 Architecture

When building a system for keyword search, the first implementations was based on previously created systems. This early method was based on [11] and the method was also used as the base for adding a temporal dimension to the search.

4.1 Search

One basic method of finding a match for keywords is using a breadth first search (BFS) [7, 11]. For each keyword in the query the algorithm will find all vertices that contain the keyword. From that set of vertices the BFS search finds the first vertex that can connect all the vertices in the set. The vertex that connects the the rest of the set is the one that best fits the keywords in the search.

Using BFS starting on keyword vertices works for keyword search, but not necessarily when searching for spatial or temporal data. When adding spatial or temporal vertices to the search, these can be used as a root, and a starting point for traversal. When a search initiates, a place and/ or time vertex existing in the graph needs to be selected. From this vertex or vertices, all connected places will be used as roots when searching. For the selected place vertex, the roots is geographically located within the selected place. This is done to bound the search close to the selected place. From the root nodes the search will traverse the graph looking for vertices matching the query words, and when all words are found, that subgraph is the best match for the start root. After all roots are searched, the subgraphs are given a score based on how well they fit the query.

Algorithm 1: GetFullResultTree(p, t, Qt)

```
Result: Set containing all nodes with at least one keyword
Queue Q ADD(p);
Threshold t;
Set n ;
while Q ≠ ∅ do
    Clear(n) e=POP(Q) if GetDistance(e) > t then
        continue;
    end
    foreach Term t ∈ Qt do
        if t ∈ e then
            p.AddMatchChild(e);
        end
    end
    if Qt ⊆ e then
        t = GetDistance(e);
    end
    n = GetNeighbors(e) ;
    Q ADD(n);
end
```

The first part of the search is to find a set of root nodes for sub-graphs. This is done by collecting all neighbors from the place queried. The neighbors can be connected by any predicate in the first developed algorithm, but as explain in 4.3 this selecting specific predicates can greatly increase speed by limiting the amount of nodes traversed. The BFS algorithm described above is used for each of the possible roots of subgraphs, stopping when all keywords are matched, or the depth of the graph exceeds the three or exceeds the currently shallowest subgraph that has matched all keywords. The reason for limiting the depth of subgraphs is to ensure at least a partial hit within a reasonable time. Limiting the depth of subgraphs to the current shallowest subgraph is done because no a deeper graph cannot be a more accurate hit.

When traversing the graph and finding a match, a node object is created. This object is used to keep track of the pseudo hierarchy in the graph. All objects contain information on the depth of the node, matched query terms, and relation to parent and children, if any. In addition root objects contains a list of all query terms hit in the sub graph. If a child node is found within multiple subgraphs, a new object will be created representing the node for each subgraph. This creates some objects that are nearly identical, but with different relations and possible different depth.

When a node object is created, a document with tokens is created. This document is used to calculate score, and to match a node with the query words. A document is created from the last part of a Yago URI, after the last slash. Each URI si further split on each underscore, creating a list of words.

After traversing the main graph we have found all subgraphs containing at least one

Algorithm 2: minimum spanning graph

```
procedure MINIMUM For each node in hits:
for each minNode in min list:
if node has same hits and higher depth than minNode:
Break
if node and minNode is equal in depth and hits:
continue
if node contains other words than minNode:
add node
remove potential common words from minNode
```

query term. These subgraphs contain many nodes which hit the same terms. Before ranking the subgraphs the minimum spanning graph needs to be found. The minimum spanning tree is found using a greedy algorithm that iterates through all nodes in the subgraphs, then keeping the nodes containing the most terms, and lowest depth. The minimum tree will contain as many terms as possible, a term will only be found in one node, and the graph will be as shallow as possible.

4.2 Ranking

- Find the minimum spanning tree for each of the root nodes containing the maximum amount of the query words.
- Rank based on 1. Query words hit 2. Nodes in the tree 3. depth of the tree

The final piece is to rank the minimum subgraphs. This is done using the nr. 2 formula found in [11].

4.3 Pruning

When using the BFS search method, all possible spatial vertices close to the queried place will be explored. This is expensive and many of the vertices will be irrelevant. Pruning the potential place vertices will reduce the amount of subgraphs traversed, and will in turn reduce the overall time used to find results for the query.

4.3.1 Predicate pruning

When traversing the graph a lot of unnecessary predicates are followed, resulting in many extra nodes added to the search. Specifying a set of predicates that contain the relevant information can greatly increase the speed of the algorithm, and also keep the memory requirements a lot lower. When selecting predicates for traversal the information expected from the search should be the top priority. Because of this, all predicates that may contain spatial or temporal data should be kept.

When using the entirety of the Yago data set, most nodes are highly connected. Many of the links in the graph are from predicates such as “linksTo” or “redirectedFrom”. These predicates creates a highly connected graph, and ensures a hit within a few nodes of the start. The same predicates will also often add the same nodes multiple times, create circular graphs, and take up unnecessary CPU power and memory.

When pruning predicates there are two methods that are possible to implement. The first will remove the predicates that contain little or no new information, such as “linksTo” or “redirectedFrom” mentioned above. This will still keep the graph connected, and keeps the predicates containing more useful information.

An other method of pruning is to create a list of predicates to be followed. This can drastically reduce the connections in the graph, but the results will only contain information relevant to the query. When preselecting predicates there is a much greater chance of not finding a match for a query. In addition a lot of metadata could be lost, if the metadata predicates are not added to the list of predicate to be explored.

4.3.2 Unqualified place pruning

For any spatial search, a lot of places will be found. The graph also contains information on the relationship between places. To find the best possible matches for a spatial query, only places of a higher resolution should be chosen. This means that places of similar or smaller expanse should be allows to be queried, e.g. a query of Boston can return the entirety of Boston, close to Boston, or within Boston. Massachusetts has multiple predicates linking it to Boston, but should not be queried because the information returned would be less detailed than what is queried.

4.3.3 Bounding

When selecting places or times for a query, the boundaries should be set so that they encompass the entirety of the queried time and or place.

5 Experiments and Results

5.1 Experimental Plan

The goal of the experiments is to gather data for comparison of search methods. when comparing search methods, speed and accuracy will be the metrics used. Three methods for searching is used, one following all gathering all objects connected to an input subject, regardless of the predicate. One method following Yago predicates leading directly to a Yago URI. And the final method will follow the same predicates as the second, and will also add a condition where all nodes must match at least one query word.

5.1.1 Time

When timing the search, two different times will be measured, time for each query, and time for each root node. In addition to these, avg. nodes visited for each root will also be used. Taking the average of these over a large input set should generate an appropriate result. Both time metrics measure how fast results are found, so the results should be similar as long as the input data does not contain a high number of highly connected nodes.

The timing of the algorithm should be the same as any breath first search, $\Theta(\mathsf{V} + \mathsf{E})$ so that the best case is visiting only the first vertex, and the worst case is traversing the entire graph.

5.1.2 Scoring and ranking

Each query is given a score. This score will be used to see how pruning, and difference in predicates can lead to differences in the result tree. Two metrics for accuracy will be used, avg. accuracy for each result and avg. highest accuracy for each query.

BFS

BFS with pruning

5.2 Experimental Setup

5.3 Data set

All of the data used is from YAGO. YAGO was selected for the large open data set, rich taxonomy, and for the spatial and temporal parts of the ontology. There are several data sets available for download, divided into categories. For running experiments data from taxonomy, core, and geonames were selected. From the taxonomy category, all data sets where used. This data describes class structure, entities, and defines relationships.

From the core category all data was also used. The core contains is most of the data used in the graph. This includes dates, relationships between nodes, literals, and labels. Most of the vertexes and edges used in the experiment comes from this category, but this data can be further structured using some of the data from other categories.

Geonames contains data and structure for geographical vertexes. These vertexes have a hierarchical structure based on what places are located within others. In addition, the data contains literals for coordinates, alternative names and links for neighbors. In addition the category contains additional classes and types specific for the geographical vertex.

Before the data could be loaded into a store, some preprocessing was necessary. This includes replacing non-unicode character, and replacing spaces with underscore in URIs. In addition the data from YAGO contained som illegal characters for Jena, such as double quoted URIs and illegal escape sequences. There were also some unterminated ttl lines. All the data was run through a sed script to ensure correctly formatted data for Jena. After formatting the data correctly a persistent TDB storage was created using tdbloader from Jena.

5.4 Queries

When selecting the keywords used in queries, a semi random selection method was used at first. This method created three pools of words, rare words (less than 1000 occurrences) uncommon words (less than 100 000 occurrences) and common words (more than 100 000 occurrences). The three pools where created by stemming all words, then counting occurrences. From each pool of words, a set of 100 where randomly selected, and from those 100, 10 where selected manually to ensure a range of occurrences, and to ensure no stop words, misspelled words or other errors were in the final set of keywords.

A second set of words were also chosen to ensure a larger hit rate. This set was a random selection of 20 words from the top 150 words. From the 20 random words 10 where manually selected, the 150 word number was selected based on Zipfs law [10].

When selecting places for spatial queries, a set of 20 random places were selected from the YagoGeonamesOnlyData set, and from that set, 5 were chosen manually for the final set. In the final set, three places where added manually, Oslo, London, and New York City were added to ensure variation in placement and node connections.

Generating random queries can create results that will not have any hits.
Find a good method of creating queries.

5.5 Pruning

A simple method to improve performance when searching in large data sets is to remove data that is unnecessary to search.

5.5.1 Predicate pruning

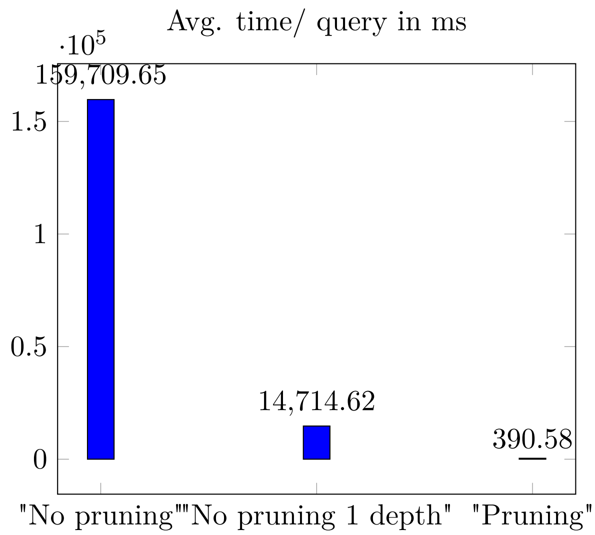
Predicate pruning is the process of eliminating predicates from the search. The taxonomy of Yago creates assigns predicates to one or more group or class. By selecting specific groups or classes it is possible to follow the types of data searched for, while eliminating a lot of other data. When pruning based on classes or groups the type of data returned must be known beforehand, to ensure that no possible hits is overlooked.

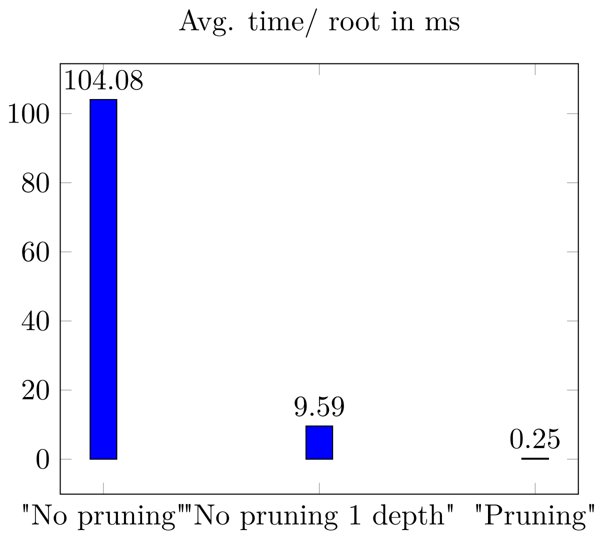
Another possible method for pruning is to specify the exact predicates to follow. This will remove a lot of data, but is possibility when searching for something specific. This will also greatly increase the speed of the search, and reduce the memory usage for the search.

5.6 Experimental Results

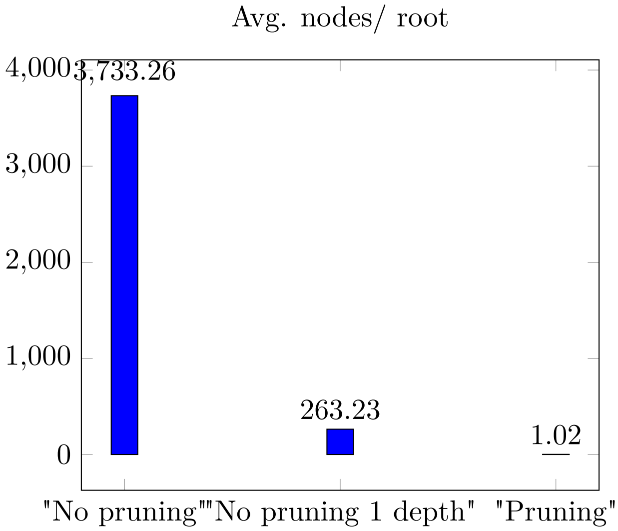
- Most roots dont have a result for the query
- Some form of pruning, or reduction in the amount of visited nodes is needed.
- Most hits that significantly increase accuracy occurs in the first set of neighboring nodes?

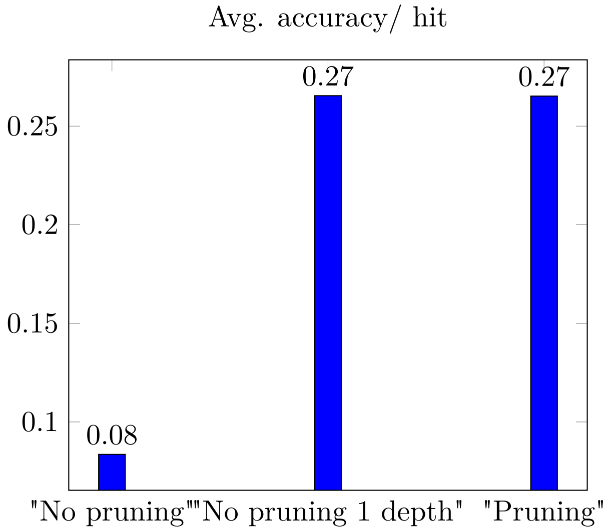
5.6.1 Time

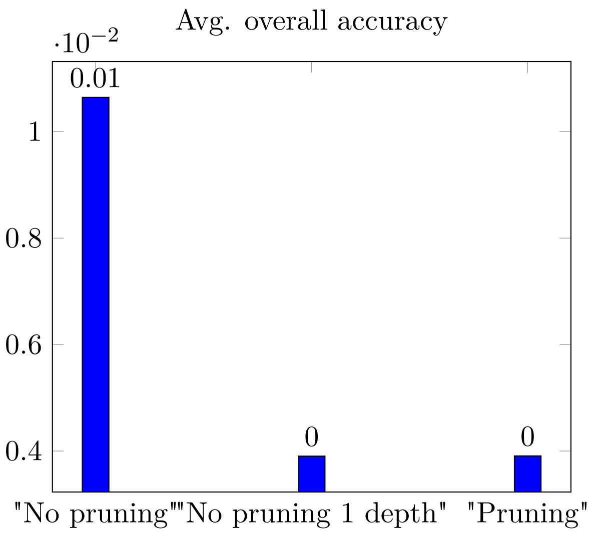


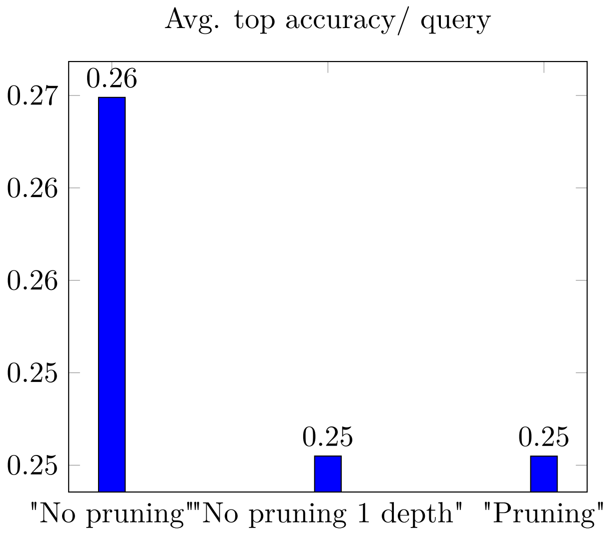


5.6.2 Accuracy









6 Evaluation and Discussion

6.1 Evaluation

6.2 Discussion

When traversing an rdf graph, the predicates chosen will have a great effect on the time used. Knowing the data searching through will help when choosing the predicate to follow. When choosing what predicates to use in the search, the goal should be to minimize the amount of unnecessary nodes found and followed. When removing predicates, the obvious downside is the possibility of missing some results, and decreasing the accuracy.

BFS search can take ua a lot of memory. This is because all nodes have to be stored while searching. It is possible to limit the maximum distance a node can be from the root node, which in turn will limit the amount of nodes visited, and reduce the memory needed. When testing without any form of pruning, the computer would run out of memory. Because of this, no results are gathered from a method following all edges of each node. Following all edges on each node would also lead to highly connected category nodes being discovered. With more than 120 million nodes, and 350.000 classes, each node would on average be connected to more than 340 other nodes, from the classes alone. Following this with a depth of 3, the average search would visit more than 40 million nodes.

- limitations
- Effectiveness of pruning
- choice of query terms
- Root nodes
- other methods for increasing speed

7 Conclusion and Future Work

7.1 Contributions

7.2 Future Work

Bibliography

[1] John Davies, Rudi Studer, and Paul Warren. *Semantic Web technologies: trends and research in ontology-based systems*. John Wiley & Sons, 2006.

[2] DBpedia. Dbpedia. URL `https://wiki.dbpedia.org`.

[3] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. 09 2016.

[4] Shady Elbassuoni and Roi Blanco. Keyword search over rdf graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 237–242. ACM, 2011.

[5] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, 9(1): 77–129, 2018.

[6] Christina Feilmayr and Wolfram Wöß. An analysis of ontologies and their success factors for application to business. *Data & Knowledge Engineering*, 101:1 – 23, 2016.

[7] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: Ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 305–316. ACM, 2007.

[8] Vanessa Lopez, Victoria Uren, Enrico Motta, and Michele Pasin. Aqualog: An ontology-driven question answering system for organizational semantic intranets. *Journal of Web Semantics*, 5(2):72 – 105, 2007.

[9] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *CIDR*, January 2013.

[10] Steven T Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychon Bull Rev*, 21:1112–1130, 2014.

[11] Jieming Shi, Dingming Wu, and Nikos Mamoulis. Top-k relevant semantic place retrieval on spatial rdf data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1977–1990. ACM, 2016.

[12] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A Core of Semantic Knowledge. In *16th International Conference on the World Wide Web*, pages 697–706, 2007.

[13] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 790–801. VLDB Endowment, 2003.

[14] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *2009 IEEE 25th International Conference on Data Engineering*, pages 405–416, March 2009.

[15] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, pages 1–6, 2010.

[16] Simonas Šaltenis, Christian Jensen, Christian (codirector, Michael Böhlen, Curtis Dyreson, Heidi Gregersen, Dieter Simonas, Janne Skyt, Giedrius Slivinskas, Kristian Torp, Richard (codirector, Bongki Moon, Michael Soo, Amazon Com, and Andreas Timeconsult. R-tree based indexing of general spatio-temporal data. 01 2000.

Appendices