Hans Sandbu

Spatiotemporal 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 RDF 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.

Experiments using breadth first traversal on different data sets, and different conditions for exploration is used to gather data. The data sets are divided into spatial, temporal, and spatiotemporal data, taken from the YAGO data set. Using three different conditions for traversal, data is gathered to find what aspects of a search is needed for fast and accurate searching.

From the results, it is clear that the speed and accuracy both depend on the amount of vertices visited during traversal. By reducing the amount of edges followed through predicate pruning, and reducing the amount of vertices visited with filtering, the speed of a search is increased.

This thesis have found that existing methods for RDF graph search can be extended to introduce spatiotemporal keyword search. More sophisticated methods can be used to increase speed and accuracy, and such methods can be implemented on existing data sets and structures.

Sammendrag

Preface

Spatiotemporal keyword search on RDF graphs.

Hans Sandbu

Trondheim, 25th May 2020

Contents

List of Figures

List of Tables

1 Introduction

In computer science, graphs are structures used for representing relationships between objects. Most information is related to other pieces of information and by using these inherent relations a graph can be created. By ordering information in a graph structure it can considered a "knowledge graph". When creating knowledge graphs a standard called "Resource Description Framework" (RDF) is commonly used. RDF was created to be a common machine-readable standard for use on the web. In RDF data is stored as triples, where a triple has the structure "subject-predicate-object". Comparing this structure to other graphs, the subject and object is the same as vertices in the graph, and the predicate is a directed 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 a few projects using RDF as an information storage system. Some of the more well-known are DBPedia[5], YAGO[18], and Creative Commons. These projects often consist of a large linked data set, or in the case of Creative Commons, it is used for embedding licenses. Both YAGO and DBPedia containing spatial and temporal data, making both suited for applications related to time and space. Because RDF is machine readable this data is suited for use in many different aspects of computer science, but it requires knowledge of the underlying structure of the data. By creating methods that allows for more human accessible information from RDF data, the format can be used without a deep understanding of the graph and makes it possible to create applications with utility.

Storing large graph structures requires specialized storage methods. With RDF a commonly used storage method is the triple storage. There have also been created special databases for graphs. Comparing such graph storage to relational storage, graph storage generally outperforms relational storage on structural queries, but not on data queries [21]. Structural queries reference the graph structure, or the relationships in the data, while data queries look for specific data. When creating a graph database, an indexing system can also be added. Such a system is "Neo4j"[1] that use Apache Lucene[2] for indexing.

When using graphs as a form of storage, the vertices hold data, and the edges can be given an extra property describing the relation. Combining such data storage with common

[1] https://neo4j.com/
[2] https://lucene.apache.org/

traversal methods for graphs results in an efficient extraction model for relational data. Some common traversal methods include breadth first search, where all neighboring vertices are discovered before moving on to the children of the currently discovered vertices. This contrasts to depth first search, where the children of a vertex are visited as they are discovered. In this thesis methods for efficient keyword search on spatial and temporal data will be explored.

1.1 Background and motivation

Most of the information on the web is unstructured, or semi-structured data. Using metadata and automated processes to create structures can help both humans and computers to find new uses for this information. Making the information in RDF more accessible can creating new applications and makes it possible to create more utility from existing data sets. To be able to get more utility out of RDF, it needs to be more accessible for humans. 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 can be created.

1.2 Goals and research questions

Goal *This thesis has the goal of researching methods for spatiotemporal keyword searches on large RDF graphs, and finding aspects needs to be considered for speed and accuracy.*

Accomplishing this goal proves that RDF graphs can be used as a tool for structuring spatial and temporal data. 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 data, and how different pieces data can be related to real places. This is also extended to time, meaning that RDF graphs can model how time relates to other data.

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

Extending existing methods for searching RDF data can make it simpler to search spatiotemporal data. By combining methods for temporal and spatial search, methods for spatiotemporal search should be possible to research.

Research question 2 *What methods can be used to achieve greater speed and accuracy for searches on RDF data?*

An important aspect of any search is speed and accuracy. Since searching in graphs usually entails some form of traversal, the methods of traversal, as well as factors that

affect speed and accuracy will be researched.

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

Comparing spatial, temporal and spatiotemporal searches can tell how spatiotemporal data differs in RDF graphs. This is important so that optimized methods can be developed.

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. Comparing spatial temporal, and spatiotemporal searches using time and accuracy as metrics, will give insight into how the data differs, and how searching can be optimized.

2 Related work

Keyword search on RDF graphs, and methods for traversal have been researched before. This research forms the basis when extending search to include spatiotemporal data.

2.1 RDF graphs

In 1999 the World Wide Web Consortium (W3C) introduced the "Resource Description Framework Model and Syntax Specification" [2]. Here the first definitions of RDF were described. This was an XML based syntax designed to provide interoperability between applications on the web, in a machine-readable format. By providing information in a machine-readable fashion, the creation of automated processes should be easier to create, and by using a common standard, the same automated processes could read any page containing RDF data.

The data model of RDF can be compared to object-oriented data. RDF consists of objects, literals, and the connection between them [6]. The objects can have literals connected to them, indicating some form of data, and the objects can be connected to each other. Connections are called predicates and form the relation between an object and literal data or form the relation between two objects. This forms a graph structure, where the objects and literals are vertices, and the predicates are edges.

Modeling RDF as a graph creates a directed graph [14]. In this model the predicates define a direction between the two objects. Such a model is called a triple, consisting of a "Subject predicate object" structure[6]. Here the subject is the start vertex of the direction, and the object denotes the end vertex.

2.2 Keyword search on graphs

Keyword search on RDF graphs often follows a set of common strategies, that usually involves graph traversal. One method is to find vertices containing one or more keyword, then following the edges from the vertices to explore the graph and find subgraphs where the combined vertices contain as many keywords as possible while also spreading as little as possible. BLINKS [11] propose such a method, in combination with indexing and cost balancing for expanding clusters of accessed vertices. A similar approach is used by the authors in [8] where each vertex has an associated document containing terms from the

triple. When querying the keywords are matched with these documents using an inverted index, 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 [12]. 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 [20]. Here the query is first used to find vertices containing some part of the query, then the graph is explored to find a connection between the vertices. The result is a series of subgraphs with vertices that each contain part of the query. Each of the subgraphs are in turn used to create a conjunctive query with edges mapped to predicates, and vertices 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 vertices, the more accurate a result should be. Of the above mentioned papers, three [11, 8, 20] 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 add a scoring system where shared vertices are counted multiple time, once for each vertex connected to it. This is done to score trees with vertices close to the root higher than vertices further away, even if the further vertices have many shared edges. The contents of the vertices are also scored based on an IR style TF/IDF method. In paper [8] 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 and patterns formed from triples. 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, [20], adds popularity, and keyword matching to the minimum spanning graph. Popularity is calculated based on how many edges a vertex has, so that the more connected a vertex is, the less cost a path through that vertex has. The keyword matching score is based on keyword matches in a vertex but is also weighted based on syntactic and semantic similarity, which is in turn done by using WordNet data.

2.3 Spatial search on RDF graphs

Keyword searching and ranking spatial RDF graphs is quite similar to regular RDF keyword searching. Paper [17] outlines methods that incorporates spatial data into the search. These methods is similar to some of those previously mentioned here [20, 8]. 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 vertices in the graph close to the real world point. The root is used as a starting point when traversing the graph, and like the previous methods, the goal is to find a minimum subgraph. Subgraphs are ranked based on how close the root vertex is to the selected point, the size of the tree, and how well the result tree fits the query words.

2.4 Temporal search on RDF graphs

Using the syntax specification for RDF [1] it is possible to declare dates as literals. Using such literals, dates and times can be connected to any vertex using a user defined predicate. A user defined predicate is any predicate created for a graph that is not part of any standard. This makes it easy to add time and date data to any graph, but some problems arise [19]. When adding time with user defined predicates, semantics can be lost. A predicate such as "borneOnDate" denotes a start date for a human but cannot be used as a start date for other concepts. It is possible to remedy this by attaching a concept such as "startDate" to the predicate, but this has the drawback of complicating the structure. Adding time as literals also means that the specific time is only connected to one vertex. This is a problem if a subgraph is temporal. It is a possibility to add predicates from all vertices in the subgraph to the date, but this has the drawback of creating many extra predicates.

Using literals for temporal data can be called "time labeling". Another method for adding time to RDF graphs is using snapshots, called "versioning". This method maintains the state of the graph at a given time. Such a model will create multiple versions of the graph to be able to model time. A versioned graph will be more difficult to traverse and query than one using literals [10].

Temporal data comes in two different forms, time points, and time intervals. Time points is modeled using a single literal. Combining two literal time points, "$[a, b]$" makes it possible to create a time interval, where "$a \leq b$". Here the vertices a and b is usually defined by the xsd datatype *date* [19]

Using query languages such as SPARQL it is possible to query time labels. In temporal queries, time labels can be queried directly, and filtered for use in a time interval query.

3 Background theory

In this chapter, the theory for building search methods are explored. General graph theory is introduced, with algorithms for traversal of graphs. Different structures used in RDF graphs are discussed, and the technology used for traversal and search are explored.

3.1 Graph theory

Graphs are mathematical structures modeling relationships between objects, and the study of these structures is called graph theory. A graph consists of vertices, sometime called nodes, and connected by edges, *G(V, E)*. 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. In computer science the relationships in graphs make them suited for applications in many different fields, such as data mining, image segmentation, networking and more[16].

A graph where the edges have a direction is called a directed graph as seen in figure 3.1. Here the vertices are connected to each other, with an arrow displaying the direction. An example of an undirected graph can be seen in figure 3.2. In both types of graphs, a vertex can have multiple edges. Graphs can contain cycles, meaning that vertices are connected to each other so that it is possible to follow edges in such a way that it leads back to the initial vertex. An example of such a cyclic graph can be seen in figure 3.3. Acyclic graphs have an inherent topological ordering. This ordering makes it easy to find properties in the graph, such as the shortest path between two vertices.

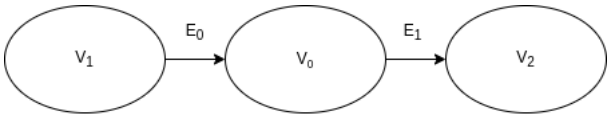


Figure 3.1: Simple Directed graph.

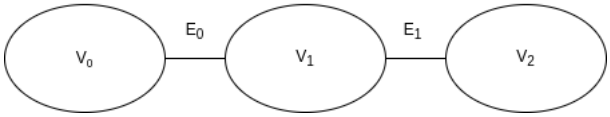


Figure 3.2: Example of non-directed graph.

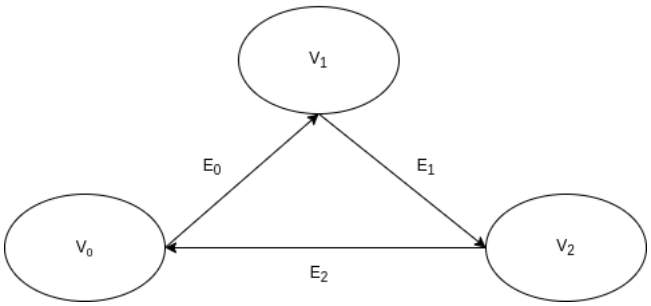


Figure 3.3: Directed graph with a cycle.

3.1.1 Graph traversal

When traversing a graph, two methods are commonly used, breadth first and depth first. Both traversal methods need a starting point. If there is no natural starting point or root, any vertex in the graph can be selected. Breadth first traversal use a queue containing vertices in the order they are discovered. From the start vertex, all connected vertices are added do the queue, then the first vertex in the queue is visited, and explored. Vertices connected to the current node is added to the back of the queue, so that the oldest discovered node is the next to be visited. In contrast a depth first search will add newly discovered vertices to the front, so that the newly discovered vertex will be visited next.

Both traversal methods can be used on directed and undirected graphs. On an undirected graph, any vertex discovered can be added to the list of vertices to visit. In a directed graph, only vertices connected by an edge with a direction going from the current vertex, and to another vertex will be followed. Both traversal types can have a termination condition, so that the traversal will stop before traversing the entire graph.

Algorithm 1: Breadth first traversal exploring entire graph

Result: List of vertices in the order traversed
Start vertex R;
Queue **Q** Add(R);
List l;
while **Q** $\neq \emptyset$ **do**
    v = **Q** GetFirstElement();
    if *v not discovered* **then**
        v.setDiscovered;
        n = GetAllConnectedVertices(e);
        Q Add(n);
        l Add(v);
    end
end
return *l*;

Algorithm 2: Depth First traversal exploring entire graph

```
Result: List of vertices in the order traversed
Start vertex R;
Stack S ADD(R);
List l;
while S ≠ ∅ do
    v = Q GetLastElement();
    if v not discovered then
        v.setDiscovered;
        n = GetAllConnectedVertices(e);
        Q Add(n);
        l Add(v);
    end
end
return l;
```

Algorithm 1 and 2 are very similar. Both algorithms explore an entire graph from a root vertex r and returns the graph as a tree structure. The main difference between breadth first and depth first is how the next vertex to visit is selected. If we use the graph in figure 3.4 as an example, we can see that the vertices are numbered 0 to 6, and the graph have one cycle. Using breadth first the result is a list with the vertices in order from 0 to 1, the paths selected is seen in figure 3.5. Using depth first traversal, the order and paths are different, as seen in figure 3.6. The order in depth first would be 0, 2, 6, 5, 1, 4, 3, assuming the left most vertex is the first to be added.

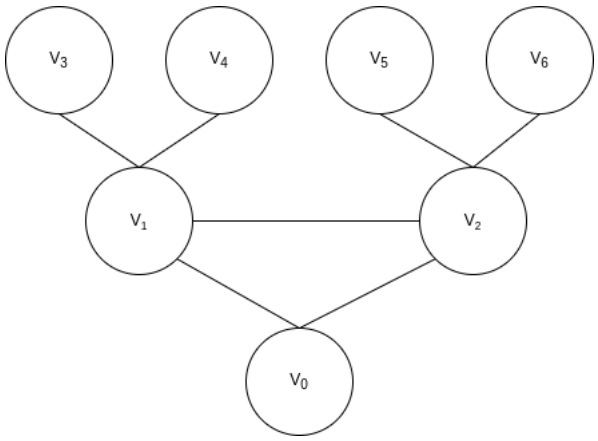


Figure 3.4: Graph with one cycle

Building on the breadth first traversal, it is possible to create an algorithm for finding a minimum spanning tree. In algorithm 3 this is done, while also adding a termination condition. The algorithm terminates when a set of terms have been discovered in the graph and returns the minimum spanning tree contains all the terms. This algorithm

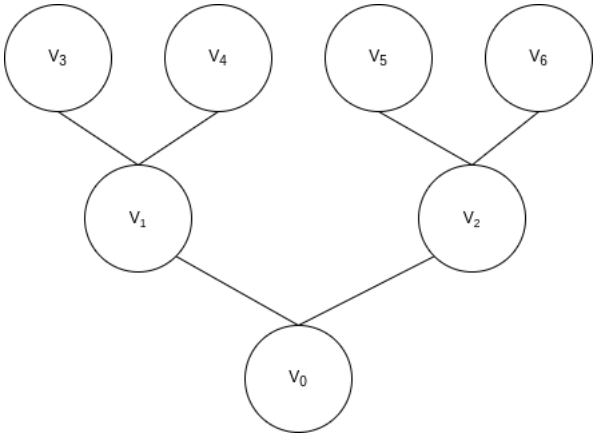


Figure 3.5: Breadth first traversal of graph 3.4

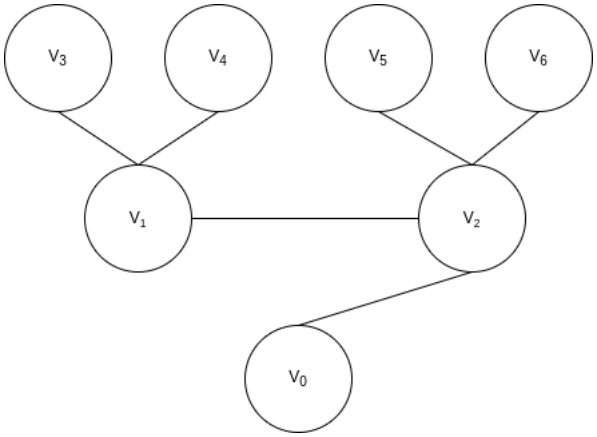


Figure 3.6: Depth first traversal of graph 3.4

assumes the vertices are objects containing a list of connected nodes.

Algorithm 3: Minimum spanning tree with breadth first search

Result: Minimum tree containgin terms Q_t

```
Root vertex R;
Terms Q_t;
Queue Q Add(R);
Tree t;
MatchedWords M_w;
while Q ≠ ∅ do
    e = POP(Q);
    if e.tokens ⊆ Q_t then
        t Add(e);
        M_w.Add(e.token ∩Q_t);
    end
    if Q_t = M_t then
        Break;
    end
    n = GetNeighbors(e);
    Q Add(n);
    t Add(e)
end
return t
```

3.2 RDF, ontologies, and knowledge graphs

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 thesis and is not a definitive definition.

3.2.1 Ontology

Ontologies contain representations, conceptualization, relations, categorization, and formal naming of data [4]. Ontologies allows for semantic modeling of knowledge. Here knowledge is data that is not ordered in a strict structure. Using ontologies allows data to be better structured, and reason about the semi-structured data. Ontologies also have its own language, called the "web ontology language" or OWL. This language was created to standardize ontologies.

Using the structure of ontologies makes it possible interpret natural language [3]. Here the ontology gives meaning to the words, based on the classes and relations in the ontology. Building on the meaning given to words from the ontology, it is possible for a computer to infer meaning from a sentence, or set of words.

3.2.2 RDF as knowledge graphs

RDF often have its own form of storage, a "triplestore" based on the structure of the data.

Structuring an ontology using a framework like RDF creates a knowledge graph. There is no clear definition on exactly what constitutes a knowledge graph. 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." [7] This definition encompasses multiple technologies. Another definition using RDF graphs as a bases is "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" [9]. This latter definition is how knowledge is structured in the data used in this thesis.

Using RDF as a format when modeling knowledge makes it possible to create structured content from semi-structured knowledge. This structure makes knowledge usable for computer, where it previously would have been difficult to extract accurate and exact knowledge. This information can then be presented in a human readable fashion, or it can be used in other fields, like artificial intelligence.

When structuring knowledge from other content it is divided into small packets of information, where one such packet represents a single fact. In RDF facts are represented as triples. Each triple has a subject that the fact is describing. These subjects are called entities, and a single entity can have any number of facts. Entities are linked together through facts, forming the graph. When linking entities, both vertices must be URIs.

In the RDF graph each triple consists of a subject, object, and a predicate. Some entities can be present in multiple languages, or there can be different names for a single entity. This creates a special case, where a single entity can be described by multiple subjects. In such cases, the subject identifies an entity, alias for the entity, or other variation of an entity, and the subject can then have relations to main vertex describing the entity.

Predicates are always a URI. URIs used for predicates differ from the ones used for subjects and objects in that predicates are connected to specific types. A type is an identifier used to describe what the relation between the subject and object represents. Predicates can also be treated as a subject in a triple, which is used to give special properties, types, and classes to predicates.

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 vertices are vertices without any data or connections other than connections pointing to the blank vertex. Literals are vertices containing an atomic value. Atomic values can be defined by the user, so that literals can hold virtually any type of data. A literal can only have edges from other vertices, just like blank vertices.

3.2.3 Existing knowledge graphs and ontologies

Currently two of the largest open knowledge graphs are YAGO and DBPedia. Both these projects use information 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.

YAGO

YAGO is an acronym for Yet Another Great Ontology and is the main data source in this thesis. The project describes itself as a knowledge base[18] and an ontology [13], but is often described as a knowledge graph by others. In this paper Yago is described as a knowledge graph. YAGO use automated information extraction from Wikipedia to create its knowledge graph. This graph is supplemented with data from WordNet, and GeoNames.

An example of YAGO entities can be seen in figure 3.7. From this example we can see that Elvis acted in Flaming Star on the 20th of December 1960. We can also see that Bobby Darin and Elvis is connected through the lifetime achievement award, and that Bobby Darin was borne in New York. New York also have coordinates connected to the node, rooting the vertex to a real place, but this is not present in the figure. In this example, only a small portion of the edges and vertices connected to the entities are displayed.

Yago uses vertices for storing both spatial and temporal data. For temporal vertices unique predicates[18] are introduced. Temporal vertices use the "xsd:date" and "xsd:dateTime" data types. Both types follow the ISO 8601 format, YYYY-MM-DD, and introduces # as a wildcard symbol. A fact can only hold a single time point and uses yagoDate as an extension of the xsd types[18]. In a date fact, the object holds the date information, and the predicate describes a connection between subject and date. In figure 3.8 we can see the triple "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 an 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[18].

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. Using this transitive relation to the spatial fact, many entities can be related to a single spatial fact. All spatial facts must have a predicate that fall

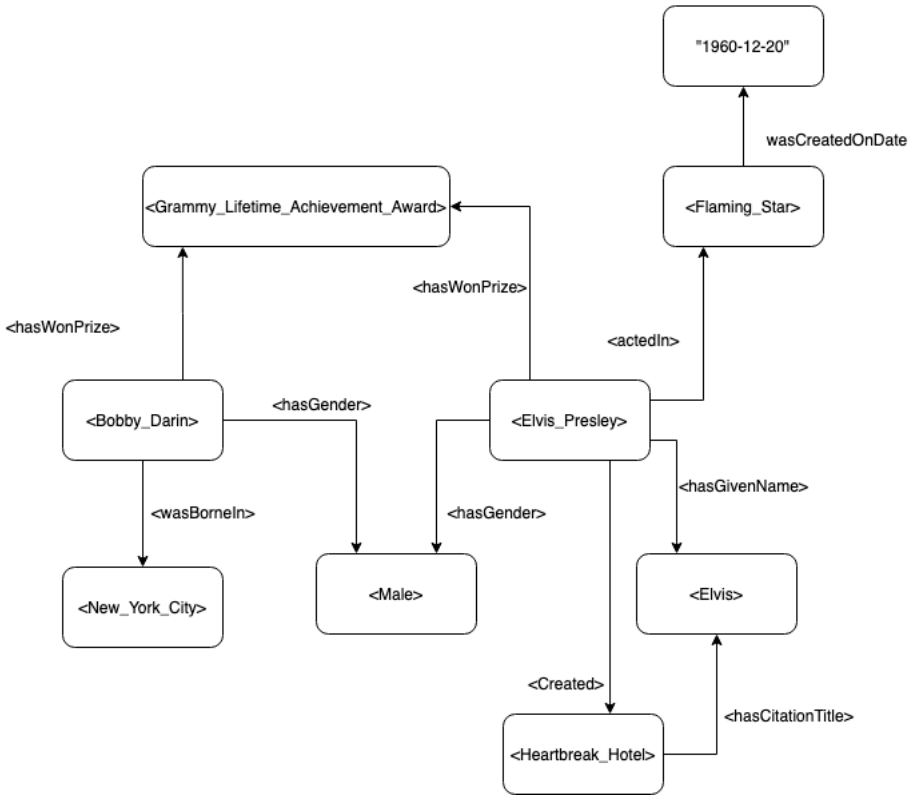


Figure 3.7: Connections in Yago around Elvis

under the yagoGeoEntity class, and all objects used in a fact with a yagoGeoEntity must have a relation containing both “hasLatitude” and “hasLongitude”.

DBPedia

DBPedia is a crowd sourced knowledge graph using Wikimedia data[2]. This differs from YAGO, which is automated. Like YAGO, DBPedia use RDF for linking facts and entities, and both contain facts in multiple languages, and have spatial and temporal data. With open data, and quarriable using SPARQL the data is easily available for anyone. DBPedia have multiple domains for spatial data, based on what type of spatial data it is. This includes coordinates, addresses, elevation, and many more, all creating a rich taxonomy.

[1]https://gate.d5.mpi-inf.mpg.de/webyago3spotlx/SvgBrowser?entityIn=%3CNidaros_Cathedral%3E&codeIn=eng
[2]https://wiki.dbpedia.org/about

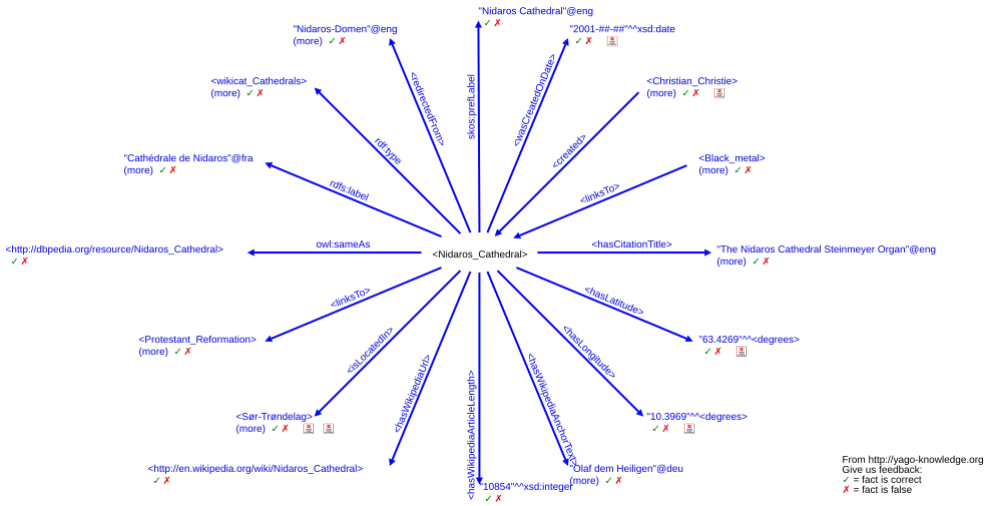


Figure 3.8: Nidaros cathedral as a subject in YAGO[1]

3.2.4 Uses for knowledge graphs

One common use of knowledge graphs is to create an info box in search engines with data related to the search input. 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.

3.2.5 Jena

Apache Jena is a framework for working with semantic web and linked data like RDF. The framework contains tools for SPARQL querying, 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.

Jena provides a persistent storage solution called TDB[3]. This storage contains tables for vertices, indexing of triples, and a prefix table. The node table stores the representation of RDF "terms", where terms are any vertex, excluding some literals. Excluded literals are the xsd:decimal, xsd:integer, xsd:dateTime, xsd:date, and xsd:boolean types. Each vertex is stored with an ID used when data is loaded, and when querying constant terms. Node to ID mappings are stored as a B+ tree, while the ID to node mappings are stored as a sequential access file. Triples are stored in the triple table as a tuple containing three node IDs. The prefix table is used for supporting mappings that are used mainly

[3]https://jena.apache.org/documentation/tdb/architecture.html

for serialization of triples. Queries in Jena are run using ARQ, a SPARQL supported query engine.

4 Architecture

In this chapter the implementation of search and traversal is described, along with a method for ranking results and methods for optimizing a search. When building a system for keyword search, the first implementations created for this thesis was based on previously created systems. This early method was based on [17] and the method was also used as the base for adding a temporal dimension to the search.

4.1 Search

Each search will return a subgraph, based in a root. This forms a tree structure. This result can be described as follows:

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

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 vertex, and each tree contains at least one query word.

A root vertex can be described as:

Root *Starting point of a BFS traversal, and the vertex all other vertices are connect to in a result tree*

A spatiotemporal root is like any other root, but with an addition:

spatiotemporal root *Any root R where the spatial input I_s and temporal input I_t overlaps, so that $R \in I_s \cap I_t$*

One basic method of finding a match for keywords is using a breadth first search (BFS) [11, 17]. 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. When this vertex is found, the whole tree is returned as a result tree.

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, place 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 are geographically located within the selected place. This is done to bound the search close to the selected place. From the root vertices 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.

The first part of the search is to find a set of root vertices used as the starting point for traversal. 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 selecting specific predicates can greatly increase speed by limiting the number of vertices traversed. The BFS algorithm described in 4 is used on each of the possible roots, stopping when all keywords are matched, or the distance from the root of the search exceeds a threshold, or exceeds the currently shallowest subgraph that has matched all keywords. The reason for limiting the distance from root of subgraphs is to ensure at least a partial hit within a reasonable time. Limiting the distance from root to the current shallowest subgraph is done because a subgraph with a greater distance from root will be scored lower.

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

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

After traversing the main graph, we have found all subgraphs containing at least one query term. These subgraphs contain many vertices 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 vertices in the subgraphs, then keeping the vertices 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 vertex, and the graph will be as shallow as possible.

In the graph traversal implementation all vertices within a set distance threshold is explored. This thresholds updated if all query words are found within the tree. The traversal will still check the rest of the queued items, even if all are found. This is done because some might be a better match than what is already added. The nodes found during traversal is added to a list of vertices, and each vertex have a link to the parent, where the

parent is the current vertex and the children are the vertices discoverable from the parent.

Algorithm 4: GetFullResultTree(p, t, T_q)

Result: Set containing all vertices with at least one keyword within threshold

```
Queue Q ADD(p);
Threshold t;
Set n;
while Q ≠ ∅ do
    e = GetFirstElement(Q);
    if GetDistance(e) > t then
        continue;
    end
    foreach Term t ∈ T_q do
        if t ∈ e then
            p.AddMatchChild(e);
        end
    end
    if Qt ⊆ e then
        t = GetDistance(e);
    end
    v = GetConnectedVertices(e);
    foreach Child c ∈ v do
        c.AddParentNode(v);
    end
    Q Add(v);
    n add(v);
end
```

4.2 Ranking

All result trees are given a score based on how well they fit the query. This score is called accuracy and is made up of two parts. The first part is how well a vertex fits the query words, and the other is how far it is removed from the root. In algorithm 5 a minimum tree is created by discarding all nodes except those with the most query word hits, at the shortest distance from root. If multiple vertices are found at the same distance and with the same number of hits, the first to be added is used in the tree.

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

To find the accuracy, a minimum spanning tree is first extracted from the result tree. To extract this tree, the algorithm 5 is used.

Algorithm 5: FindMinimumTree(R_t)

Result: A minimum spanning tree based on keyword vertices found during traversal
ResultTree $\mathbf{R_t}$;
MinimumTree M_t Add(R_t[0]);
foreach *Vertex* $r \in R_t$ **do**
  foreach *Vertex* $m \in M_t$ **do**
    if $r.hits = m.hits \wedge r.distance > m.distance$ **then**
      Break;
    end
    if $r.hits = m.hits \wedge r.distance = m.distance$ **then**
      Continue;
    end
    if $r.hits \neq m.hits$ **then**
      m.matchedWords Remove_common_Words(r);
      M_t ADD(r);
    end
    if $m.hits = \varnothing$ **then**
      M_t Remove(m);
    end
  end
end
return M_t

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 selection

When traversing the graph, a lot of unnecessary predicates are followed, resulting in many extra vertices added to the search. Specifying a set of predicates that contain the relevant information can greatly increase the speed of the algorithm, and 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 vertices are highly connected. Many of the links in the graph are from predicates such as “linksTo” or “redirectedFrom”. These predicates create a highly connected graph and ensures a hit within a few vertices of the start. The same predicates will also often add the same vertices 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 method will remove the predicates that contain little or no new information, such as “linksTo” or “redirectedFrom” mentioned above. This will keep the graph connected, and keeps the predicates containing more useful information. Another 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 predicates to be explored.

4.3.2 Place hierarchy

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 allowed to be queried, e.g. a query of Boston can return the entirety of 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 the input.

5 Experiments and results

In this chapter the planning, setup, results from experiments are presented. The results use the time and accuracy as metrics, and is used to evaluate how different traversal criteria, and datasets affect a search.

5.1 Experimental plan

The goal of the experiments is to gather data for comparison of data types searched and criteria for vertices to follow. when comparing, speed and accuracy will be the metrics used. In total, nine different searches will be conducted, one using spatial vertices as a starting point, one for temporal vertices as start, and one combining the two, searching spatiotemporal vertices. For each of these vertex types, there will also be different criteria for how the graph will be explored. The first method follows all predicates, excluding connections to types and classes, and will have a max distance of two. The second method follows the same predicates as the first but has the max distance set to one. Finally, a criterion for following only predicates from nodes with at least one keyword match is used.

When comparing the methods, the goal is to see how reducing the amount of data search will affect speed and accuracy of the search, and how differences in distance and edges followed differs based on the starting data.

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 if 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 subgraph 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.

5.2 Experimental setup

All experiments was run on a single laptop with the specifications listed in table 5.1. The code is written in Java, using openJDK 11.0.7 and building with gradle 6.0. For triple store, Jena tdb storage was used, using version 3.14.0 of Jena.

Table 5.1: Platform used for experiments

OS	Ubuntu 19.10 x86_64
Host	20KGS0N400 ThinkPad X1 Carbon 6th
Kernel	5.3.0-46-generic
CPU	Intel i5-8350U (8) @ 3.600GHz
Memory	15760MiB

5.2.1 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 vertices 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 vertices. These vertices 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 some 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.

5.2.2 Queries

When selecting words for the queries, all nodes in the graph was scanned to count word frequency. This list was sorted based on number of occurrences, stop words and numbers were removed. From this shortened list, the top 150 words were chosen, based on vocabulary size needed to meet more than half of nodes [15, 22]. From this list of 150 words, a final set of 10 words was chosen at random. It is worth noting that all words are from the name of a vertex, so the list of words does not reflect natural language.

When selecting places, a set of 7 places were manually chosen. This choice was made to ensure places from different parts of the world, and difference in population and language. With these differences, root vertices found should have variation in edges, differences in keyword vertices discovered, and paths discovered for shared keyword vertices.

Date selection for temporal search was done manually. This selection included a combination of non-significant dates, and significant dates. The selection was made to have a high probability of finding hits and ensure that the vertices have variation in the amount of edges.

5.2.3 Combining data and queries

When running the experiments, the same set of query words was used for all runs. For each type of data, spatial, temporal, and spatiotemporal, the query words were combined in five groups containing two words, and four groups containing four words. variation in the amount of words used would affect both the chance of hitting keyword vertices, and the score of the subgraphs discovered.

5.2.4 Max. distance and following edges

For each of the data types a total of three different exploration methods was used. One with a max. distance of 1 from the root node, one with a max. distance of 2, and one where only edges leading to a keyword vertex would be explored further. Using a max. depth of 2 would serve as a base line for the comparison. The two other methods was used to see how much distance from the root would affect subgraphs accuracy, and if optimal graphs could be found even with a severe limitation put on the search.

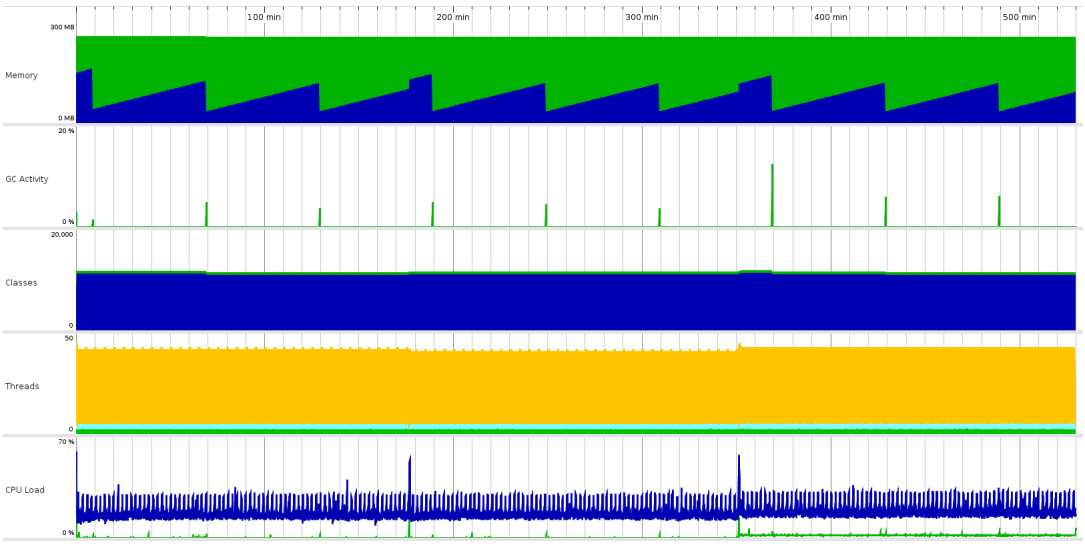


Figure 5.1: Code profile of spatial data using distance 2.

5.3 Experimental results

5.3.1 Code profiling and external impact

Because the platform used to run the experiments was a laptop, there are external factors that affect the result. One such factor is the CPU of the laptop overheating and throttling, meaning it does not run at top speed, and instead moves between a high and low speed in intervals. This can clearly be seen in figure 5.1, where the CPU throttles between ≈20% and ≈32% with the high value lasting for about 20 seconds, and the lower values for just under a minute. The figure displays three runs of the algorithm, the first two being warmup runs, and the third is a profile run. During the initiation of each run there is a spike in CPU usage almost reaching 70%. CPU usage is also split in to two categories, where blue is system load, and green is process load. During the warmup, the process load is low, and is slightly increased during the profiling.

Memory usage and garbage collection will also affect the performance. In figure 5.2 and 5.3 the memory usage is low, and there is less garbage collection. Compare this to the runs with a distance of two, where garbage collection is done often. It is also worth noting that the algorithm runs on two threads, with a further four used for IO. Most IO operations are from reading vertices from the triple store.

5.3.2 Effect of distance from root

When traversing a graph, the number of vertices visited will heavily impact the time used. When the max. distance of the search increase, the number of vertices visited will grow with the number of roots discovered times the distance from the root. Looking at table

Figure 5.2: Code profile spatial data and distance 1

Figure 5.3: Code profile of spatial data following vertices with hits

5.2 we see that the amount of roots found for a query is the most significant factor for the speed and amount of vertices visited. Since there only is three data points, no broad conclusion can be drawn, but in 5.4 a trend can be seen, and all points line up well. This indicates a linear increase of vertices visited, where the slope is based on the amount of edges the vertices have and the depth from the root vertex. Comparing this to the more detailed graph 5.5 we can see that the time used for a query still follows a linear increase. Using a distance of 2, the time varies more than the other two methods, but the timings appear to increase linearly based on the number of roots discovered. In graph 5.5 all the different data types are added, and since both temporal, and spatiotemporal queries have fewer roots per query, most of the data points are clustered in the lower end of the graph. All three data sets contain one point for each query with at least one root, totaling 388 points for each traversal method.

Table 5.2: Follow all edges, max. distance 2

Traversal following all edges max. distance 2			
Result type	Temporal data	Spatial data	Spatial and temporal data
Avg. time per query (ms)	27 679	166 479	1 842
Avg. roots per query	192	1 527	10
Avg. vertices per root	3 097	3 684	4 843
Avg. vertices visited per query	594 683	5 626 989	49 719
Avg. accuracy per result	0.075	0.077	0.066
Avg. top accuracy per query	0.231	0.280	0.108
Query miss (no keywords found)	24/63 (38.1%)	6/63 (10.5%)	357/441 (81.0%)

5.3.3 Effect of following specific edges

In table 5.4 we see that very few vertices with a keyword are directly connected to other vertices with a keyword. This makes the search results similar to the results in table 5.3. The most noticeable difference is the number of vertices visited. Even with a greater max. depth, the search only following edges from vertices with keyword hits still visits fewer vertices. Visiting fewer vertices increase the speed of the search, something that is evident from graph 5.6.

Table 5.3: Follow all edges, max. depth 1

Traversal following all edges max. distance 1			
Result type	Temporal data	Spatial data	Spatial and temporal data
Avg. time per query (ms)	4 111	14 618	467
Avg. roots per query	192	1 527	10
Avg. vertices per root	450	257	335
Avg. vertices visited per query	86 479	391 914	3 446
Avg. accuracy per result	0.231	0.232	0.174
Avg. top accuracy per query	0.313	0.349	0.215
Query miss (no keywords found)	35/63 (55.5%)	20/63 (31.7%)	415/441 (94.1%)

5.3.4 Differences between data types

In table 5.3 we can see how many connections the average root have. Because the max. distance is set to 1, the average vertices per root is the same as average connections per root. Temporal data have the highest number of edges, followed by spatiotemporal vertices. This indicates that vertices that contain temporal data generally are more connected that spatial vertices. The difference in roots discovered can be attributed to the input data.

5.3.5 Implications

From the results it is clear that increasing the distance traveled from the root have the most significant impact on both speed and accuracy. When the distance is increased many more vertices are visited, so that the chance of finding a keyword match is increased. Visiting more vertices will also take significantly more time. From figure 5.5 it is also clear that the amount of roots discovered for a query will make a significant difference. This difference is further amplified when the distance is higher. Even though the increase in time is linear, the slope is much steeper for traversals with higher distance, resulting in significant time increases.

Figure 5.4: Linear regression for time and roots

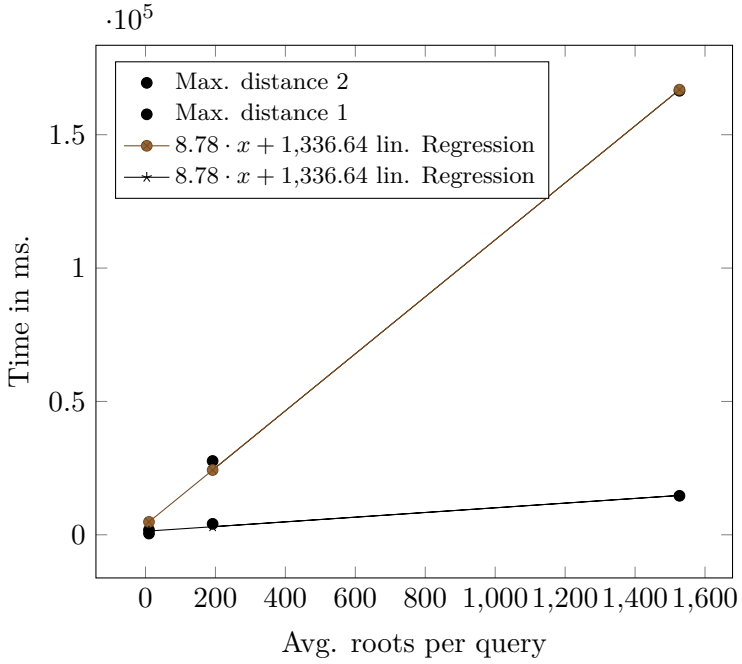


Table 5.4: Follow only edges from node hits

Traversal following edges with keyword match vertex			
Result type	Temporal data	Spatial data	Spatial and temporal data
Avg. time per query (ms)	3033	466	367
Avg. roots per query	192	1527	10
Avg. vertices per root	1.02	1.03	1.02
Avg. vertices visited per query	197	1569	11
Avg. accuracy per result	0.231	0.232	0.174
Avg. top accuracy per query	0.313	0.349	0.215
Query miss (no keywords found)	35/63 (55.5%)	20/63 (31.7%)	415/441 (94.1%)

Figure 5.5: Linear regression for time and roots

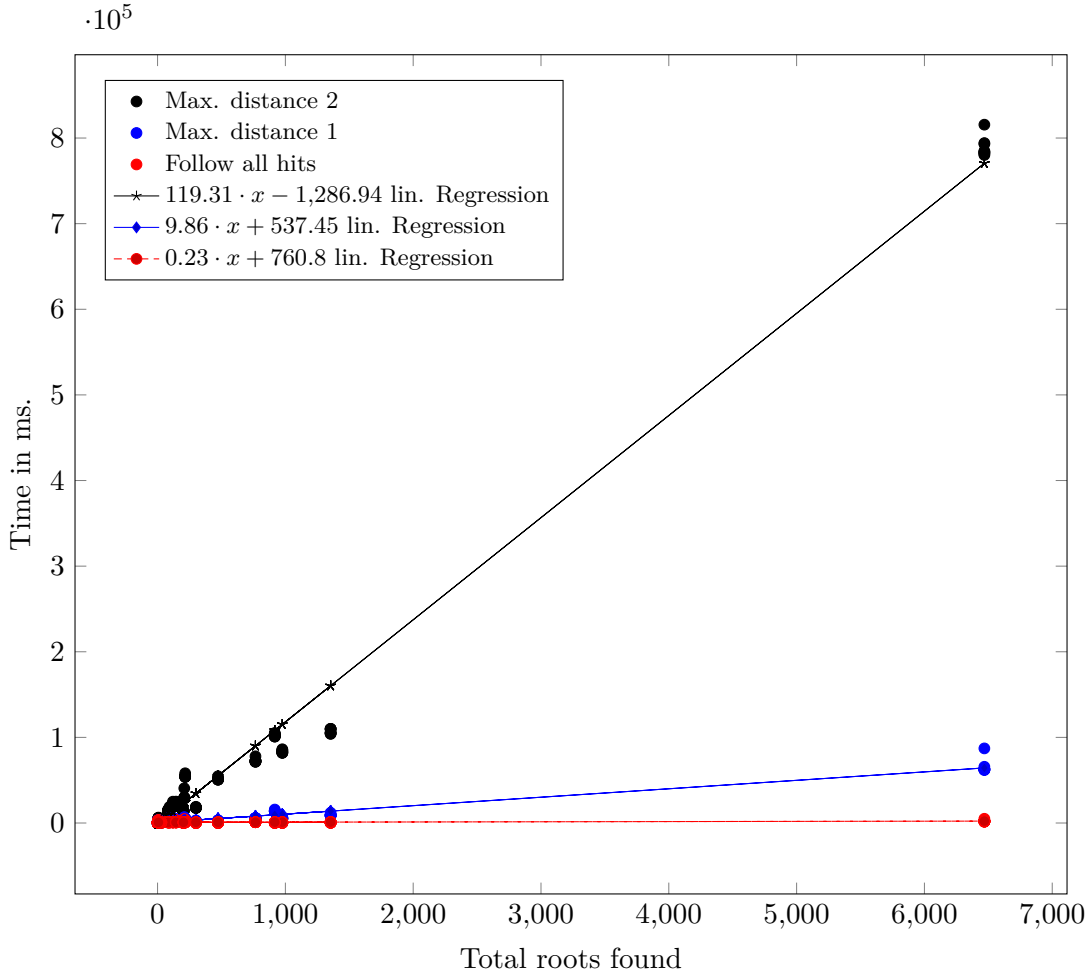
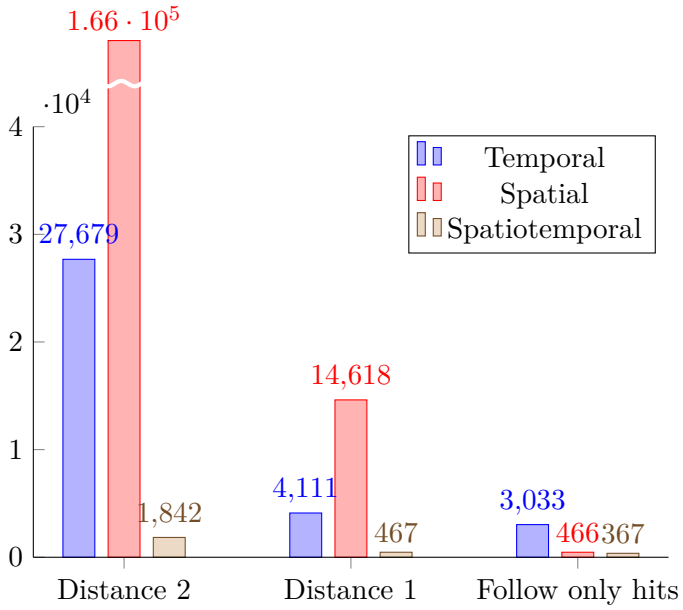


Figure 5.6: Avg. time used for each data type and traversal criteria per query in ms.



6 Discussion and limitations

In this chapter traversal methods are explored further. By exploring the methods implemented, and how these methods can be changed or extended to improve speed and accuracy.

6.1 Discussion

Using traditional methods for graph traversal it is possible to implement keyword search on RDF graphs. One such method is the use of BFS to find a minimum spanning subgraph containing the query words. Using spatial nodes as a root for the subgraph, creating a minimum spanning tree, it is possible to retrieve spatial data from such a keyword search. The same method is used when searching for spatial data as with other keyword searches, with the key difference being to root the BFS in a spatial vertex.

This BFS method can be implemented with a temporal dimension instead of spatial. When changing the data type of the root from spatial to temporal, the traversal method stays the same, but the root used is matched with temporal input instead of spatial input. The result tree for a temporal search can be ranked using the same methods as used for spatial search.

Combining spatial and temporal data makes spatiotemporal search possible. These searches will be rooted in a vertex containing spatial and temporal data matching the input, as defined in 4.1. Because the root needs to contain both the spatial and the temporal input, the subset of vertices containing this wil be small. This makes spatiotemporal searches quick to execute, but the accuracy and hit rate will be low. A method that could be implemented to increase the hit rate of these searches would be to change the criteria for one of the two dimensions. The search could be rooted in either space or time, and then look for the other dimension as a special case while traversing.

6.1.1 Spatial and temporal vertices as special cases

Spatiotemporal searches rooted only in time could be accomplished by discarding all result trees that are does not contain a vertex inside the queried location. To discover vertices inside a location the “isLocatedIn” predicate would be followed. Using this predicate to move upwards in the hierarchy the predicate creates while looking for the queried location will determine if the discovered vertex is located inside the queried

location. A maximum distance of three should be used when following this predicate, as this will traverse the hierarchy up to at least country level, and possibly up to the earth vertex. Since we are looking for a specific vertex, and only looking at the vertices above in the hierarchy this will not add more than three extra visited vertices for each vertex connected with the isLocatedIn predicate. This will not increase the time used by any significant margin compared to the regular temporal search, but will increase the hit rate of spatiotemporal search.

Rooting spatiotemporal search in a spatial vertex and treating the temporal dimension as a special case can be done with three different methods. Two of the methods is accomplished by looking at the predicate, and the third checks the type of literals. When using the predicate to determine if the connected vertex is temporal, the predicate can either be checked against a predetermined list of temporal predicates, or the range of valid values can be checked. The range of valid values is determined by a special vertex connected to the predicate, called “rdfs:range”. Checking if the range is “xsd:date” or “xsd:dateTime” will determine if the predicate is temporal. The final method, checks if the type of the literal is “xsd:date” or “xsd:dateTime”. Since all literals have a type as part of the vertex, this is done by reading directly from the literal. Checking the predicate would add an extra traversal to each predicate connected to a vertex since the rdfs:range is treated as any other vertex. Using a predetermined list requires more knowledge of the data set, and adds some small overhead. Checking the type of the objects is the fastest since this would not require any extra traversal, and carries little overhead. This would not require any extra knowledge about the data set, assuming that no extra user added date types are created for the graph.

6.1.2 Reduction of vertices and edges

Since the amount of vertices visited is the factor most correlated to time used for a search, reducing the amount of vertices visited will also reduce the time used for a search. To reduce the amount of vertices, a possibility is to reduce the predicates followed. Following predicates based on type can be used to create search methods that will only follow specific predicates. This can be accomplished using natural language processing, similar to what is done in [20, 12]. A less sophisticated method would be to allow for the selection of categories to be search, then using the selected categories to extract all predicates that are connected. Because the predicates have types and properties, running a query to find all connected predicates to a category would not add more than two extra traversals, one for type, and one for property. Such a category would contain a set of types and properties that would be used to find predicates. This would make it easy to add new predicates, without having to rework the search. The same categories can be used for natural language processing, but then the query have to be preprocessed to find the categories that should be included. A downside of removing predicates, is the possibility of missing some results, and decreasing the accuracy.

Reduction of root nodes is also a method that can be implemented to increase the

speed of a search. To be able to reduce the roots while maintaining accuracy, using more sophisticated natural language processing is an option. Implementing a system for inferring structure from the keywords, like that used in [8] would make it possible to find connections where the object and predicate is related to the query. Using the taxonomy from YAGO, this is possible to implement.

6.1.3 Indexing and natural language processing

It is possible to index the vertices in the graph, so that they can be search using full text search. If all vertices are tokenized and indexed so that all nodes containing a keyword can be retrieved, this can be combined with the roots for temporal, spatial or spatiotemporal searches. Using SPARQL it is possible to find paths between two vertices using "property paths". From this, the shortest paths between a root and keyword vertex can be determined, and finally the roots can be ranked. This will guarantee hits for all roots, but it could be painfully slow, depending on the input keywords. Using the keyword "south" as an example, this keyword have 2 798 973 hits in the YAGO data used for the experiments in this thesis. Combining this with the average number of temporal roots from section 5.3, 192, this search would have to query and rank 537 395 904 paths, just for one keyword. Using such an index would require more than just the shortest path.

Using an index for keywords to find shortest paths without exploring the entire graph will find the best result for a given keyword. Depending on how it is implemented, it might not find the best solution for vertices containing multiple keywords. This is because a vertex containing more than one keyword at a greater distance may score higher than multiple vertices containing one keyword each at a shorter distance. The score would depend on the amount of keywords in a single vertex, and the difference in depth. If even a single vertex is found at the same depth as the multi-keyword vertex, the result tree with the fewest vertices will score better.

6.2 limitations

Some temporal vertices in YAGO use a so called "wildcard date". These dates contain a "#" symbol for parts of the date, making it impossible to directly query such dates when selecting roots. This means that the temporal search have some possible vertices missing from the set of root nodes used when traversing the graph. This could have been remedied by creating two new temporal vertices for such dates, one start date indicating the lowest possible value a wildcard date could have, and an end date indicating the highest possible value a wildcard could have.

When searching spatiotemporal data the wildcards have been used. This was done by adding some logic comparing the wildcards found connected to spatial vertices to the date range in the search. By doing this, the spatiotemporal search should be more accurate,

and should retrieve some nodes that a regular temporal search would not retrieve.

Because of the large amount of vertices discovered during a search, memory would run out on traversal with high distance from root. This could be solved by writing to disk, bout would bbe slow, and since the search already reads from disk to load vertices, this would have a significant impact on search time.

BFS search can take up 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.

7 Conclusion and future work

This thesis have explored search methods for RDF graphs, finding methods for spatial, temporal, and spatiotemporal search. By building on existing keyword search methods, a new method for spatiotemporal search have been introduced and evaluated. From the evaluation further improvements have been proposed.

7.1 Contributions

This thesis had the goal to research methods for keyword searches on large spatiotemporal RDF graphs. This was broken down into three research questions.

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

In the chapter 2 previous methods used for keyword searches on RDF graphs was investigated. Using methods from other research spatiotemporal keyword search have been implemented. The effectiveness of the search can be further improved, and implementing methods such as natural language processing for categorizing the keywords could further increase speed, without being detrimental to accuracy.

Research question 2 *What methods can be used to achieve greater speed and accuracy for searches on RDF data.*

This thesis looked at the parameters of a search that could increased speed, and decreased accuracy. When increasing speed, the main goal should be to reduce the amount of root vertices used as starting points for a search. Reducing the edges followed will also have a great effect on the speed. To be able to do this, processing the query, so that a structure can be created from the keywords is a possibility.

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

Using a BFS for traversing the graph, a spatiotemporal search does not need to differ from spatial or temporal search. When selecting the roots used as starting points for the search, a spatiotemporal search needs to find roots that fall into both the spatial dimensions of the search, and the temporal. This makes the total set of roots used for a spatiotemporal search less than if the search was done just on one dimension.

7.2 Future work

Further research into spatial, temporal, and spatiotemporal keyword search on RDF graphs can make the technology more accessible. Exploring search methods implementing natural language processing for pre-processing of queries, similar to [12] could improve accuracy of search. Relying more on the ontology to infer meaning from a keyword search can also make searching RDF graphs faster and more accurate.

Because each search is rooted in one specific root, and traverse the graph from that root, parallelizing the search should be researched. Running multiple traversals at the same time would not affect any of the other traversals result tree but would make it possible to execute a search faster. This would not affect the accuracy, and can be implemented on many of the existing search methods for RDF graphs.

Finally, treating spatial or temporal data as a special case while traversing the graph, as discussed in chapter 6 could be considered a continuation of this thesis. Testing these search methods can give insight into how spatiotemporal searches on RDF graphs can be further improved.

Bibliography

[1] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10(2.3), 2004.

[2] Dan Brickley, Ramanathan V Guha, and Andrew Layman. Resource description framework (rdf) schema specification. 1999.

[3] Philipp Cimiano, Christina Unger, and John McCrae. *Ontology-based interpretation of natural language*. Morgan & Claypool Publishers, 2014.

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

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

[6] Stefan Decker, Prasenjit Mitra, and Sergey Melnik. Framework for the semantic web: an rdf tutorial. *IEEE Internet Computing*, 4(6):68–73, 2000.

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

[8] 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.

[9] 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.

[10] Claudio Gutierrez, Carlos A Hurtado, and Alejandro Vaisman. Introducing time into rdf. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2006.

[11] 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.

[12] 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.

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

[14] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet computing*, 6(6):55–59, 2002.

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

[16] Ferozuddin Riaz and Khidir M Ali. Applications of graph theory in computer science. In *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*, pages 142–145. IEEE, 2011.

[17] 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.

[18] 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.

[19] Jonas Tappolet and Abraham Bernstein. Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In *European Semantic Web Conference*, pages 308–322, 2009.

[20] 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.

[21] 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.

[22] Denice Worthington and Paul Nation. Using texts to sequence the introduction of new vocabulary in an eap course, 1996.

Appendices