CS520

Knowledge Graphs

What should AI Know ?

What is a Knowledge Graph?

1. Introduction

Knowledge graphs have emerged as a compelling abstraction for organizing world's structured knowledge over the internet, and a way to integrate information extracted from multiple data sources. Knowledge graphs have also started to play a central role in machine learning as a method to incorporate world knowledge, as a target knowledge representation for extracted knowledge, and for explaining what is learned.

Our goal here is to explain the basic terminology, concepts and usage of knowledge graphs in a simple to understand manner. We do not intend to give here an exhaustive survey of the past and current work on the topic of knowledge graphs.

We will begin by defining knowledge graphs, some applications that have contributed to the recent surge in the popularity of knowledge graphs, and then use of knowledge graphs in machine learning. We will conclude this note by summarizing what is new and different about the recent use of knowledge graphs.

2. Knowledge Graph Definition

A knowledge graph is a directed labeled graph in which the labels have well-defined meanings. A directed labeled graph consists of nodes, edges, and labels. Anything can act as a node, for example, people, company, computer, etc. An edge connects a pair of nodes and captures the relationship of interest between them, for example, friendship relationship between two people, customer relationship between a company and person, or a network connection between two computers. The labels capture the meaning of the relationship, for example, the friendship relationship between two people.

More formally, given a set of nodes N, and a set of labels L, a knowledge graph is a subset of the cross product $N \times L \times N$. Each member of this set is referred to as a triple, and can be visualized as shown below.

The directed graph representation is used in a variety of ways depending on the needs of an application. A directed graphs such as the one in which the nodes are people, and the edges capture

friendship relationship is also known as a data graph. A directed graph in which the nodes are classes of objects (e.g., Book, Textbook, etc.), and the edges capture the *subclass* relationship, is also known as a taxonomy. In some data models, *A* is referred to as *subject*, *B* is referred to as *predicate*, and *C* is referred to as object.

Many interesting computations over graphs can be reduced to navigating the graph. For example, in a friendship knowledge graph, to calculate the friends of a friends of a person *A*, we can navigate the knowledge graph from *A* to all nodes *B* connected to it by a relation labeled as friend, and then recursively to all nodes *C* connected by the friend relation to each *B*.

A ***path*** in a graph *G* is a series of nodes $(v_1, v_2,..., v_n)$ where for any $i \in N$ with $1 \leq i < n$, there is an edge from v_i to v_{i+1}. A ***simple path*** is a path with no repeated nodes. A ***cycle*** is a path in which the first and the last nodes are the same. Usually, we are interested in only those paths in which the edge label is the same for every pair of nodes. It is possible to define numerous additional properties over the graphs (e.g., connected components, strongly connected components), and provide different ways to traverse the graphs (e.g., shortest path, Hamiltonian path, etc.).

3. Recent Applications of Graphs

There are numerous applications of knowledge graphs both in research and industry. Within computer science, there are many uses of a directed graph representation, for example, data flow graphs, binary decision diagrams, state charts, etc. For our discussion here, we have chosen to focus on two concrete applications that have led to recent surge in popularity of knowledge graphs: organizing information over internet and data integration.

3.1 Graphs for organizing Knowledge over the Internet

We will explain the use of a knowledge graph over the web by taking the concrete example of Wikidata. Wikidata acts as the central storage for the structured data for Wikipedia. To show the interplay between the two, and to motivate the use of Wikidata knowledge graph, consider the city of Winterthur in Switzerland which has a page in Wikipedia. The Wikipedia page for Winterthur lists its twin towns: two are in Switerzland, one in Czech Republic, and one in Austria. The city of Ontario in California that has a Wikipedia page titled *Ontario, California*, lists Winterthur as its sister city. The sister city and twin city relationships are identical as well as reciprocal. Thus, if a city A is a sister city of another city B, then B must be a sister city of A. This inference should be automatic, but because this information is stated in English in Wikipedia, it is not easy to detect this discrepancy. In contrast, in the Wikidata representation of Winterthur, there is a relationship called *twinned administrative body* that lists the city of Ontario. As this relationship is symmetric, the Wikidata page for the city of Ontario automatically includes Winterthur. Thus, when Wikidata knowledge graph will be fully integrated into Wikipedia, such discrepancies will naturally disappear.

Wikidata includes data from several independent providers, for example, the Library of Congress that publishes data containing information about Winterthur. By using the Wikidata identifier for Winterthur, the information released by Library of Congress can be easily linked with information available from other sources. Wikidata makes it easy to establish such links by publishing the definitions of relationships used in it in *Schema.Org*.

The vocabulary of relations in *Schema.Org* gives us, at least, three advantages. First, it is possible to write queries that span across multiple datasets that would not have been possible otherwise. One example of such a query is: Display on a map the birth cities of people who died in Winterthour? Second, with such a query capability, it is possible to easily generate structured

information boxes within Wikipedia. Third, structured information returned by queries also can appear in the search results which is now a standard feature for the leading search engines.

A recent version of Wikidata had over 80 million objects, with over one billion relationships among those objects. Wikidata makes connections across over 4872 different catalogs in 414 different languages published by independent data providers. As per the recent estimate, 31% of the websites, and over 12 million data providers publish Schema.Org annotations are currently using the vocabulary of *Schema.Org*.

Let us observe several key features of the Wikidata knowledge graph. First, it is a graph of unprecedented scale, and is the largest knowledge graph available today. Second, it is being jointly created by a community of contributors. Third, some of the data in Wikidata may come from automatically extracted information, but it must be easily understood and verified as per the Wikidata editorial policies. Fourth, there is an explicit effort to provide semantic definitions of different relation names through the vocabulary in *Schema.Org*. Finally, the primary driving use case for Wikidata is to improve the web search. Even though Wikidata has several applications using it for analytical and visualization tasks, but its use over the web continues to be the most compelling and easily understood application.

3.2 Graphs for Data Integration in Enterprises

Data integration is the process of combining data from different sources, and providing the user with a unified view of data. A large fraction of data in the enterprises resides in the relational databases. One approach to data integration relies on a global schema that captures the interrelationships between the data items represented across these databases. Creating a global schema is an extremely difficult process because there are many tables and attributes; the experts who created these databases are usually not available; and because of lack of documentation, it is difficult to understand the meaning of the data. Because of the challenges in creating a global schema, it is convenient to sidestep this issue, and convert the relational data into a database with the generic schema of triples, ie, a knowledge graph. The mappings between the attributes are created on as needed basis, for example, in response to addressing specific business questions, and can themselves be represented within a knowledge graph. We illustrate this process using a concrete example.

Many financial institutions are interested in creating a company knowledge graph that combines the internal customer data with the data licensed from third parties. Some examples of such third party datasets include Dunn & Bradstreet, S&P 500, etc. An example usage of a company graph is to assess the risk while making loan decisions. The external data contain information such as the suppliers of a company. If a company is going through financial difficulty, it increases the risk of awarding loan to the suppliers of that company. To combine this external data with the internal data, one has to relate the external schemas with the internal company schema. Furthermore, the company names used in the external sources have to be related to the corresponding customer identifiers used by the financial institutions. While using a knowledge graph approach to data integration, determining such relationships can be delayed until they are actually required.

4. Graphs in Artificial Intelligence

Knowledge graphs, known as semantic networks, have been used as a representation for Artificial Intelligence since the early days of the field. Over the years, semantic networks were evolved into different representations such as conceptual graphs and description logics. To capture uncertain knowledge, probabilistic graphical models were invented.

Orthogonal to the representation of knowledge, a central challenge in AI is knowledge acquisition bottleneck, ie, how to capture knowledge into the chosen representation in an economically scalable manner. Early approaches relied on knowledge engineering. Efforts to automate portions of knowledge engineering led to techniques such as inductive learning, and the current generation of machine learning.

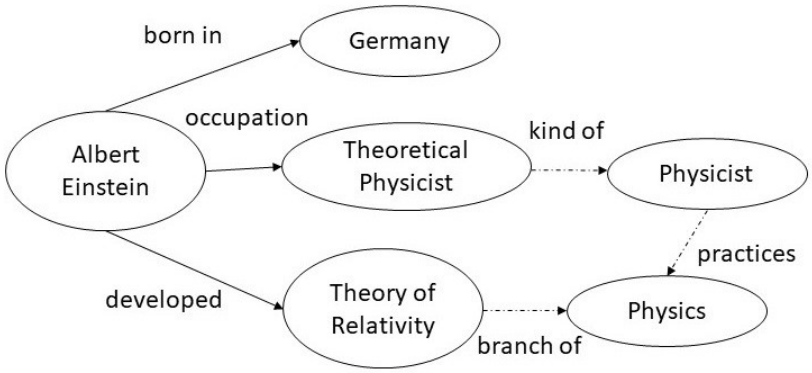
Therefore, it is natural that the knowledge graphs are being used as a representation of choice for storing the knowledge automatically learned. There is also an increasing interest to leverage domain knowledge that is expressed in knowledge graphs to improve machine learning.

4.1 Graphs as the output of Machine Learning

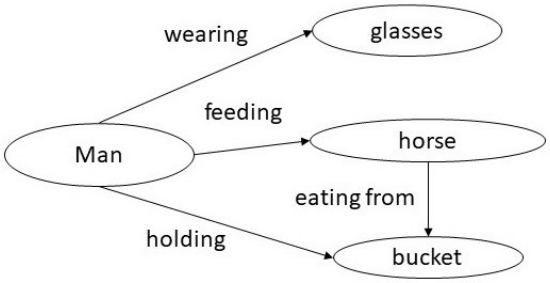
We will consider how graphs are being used as a target output representation for natural language processing and computer vision algorithms.

Entity extraction and relation extraction from text are two fundamental tasks in natural language processing. The extracted information from multiple portions of the text needs be correlated, and graphs provide a natural medium to accomplish such a goal. For example, from the sentence shown on the left, we can extract the entities *Albert Einstein*, *Germany*, *Theoretical Physicist*, and *Theory of Relativity*; and the relations *born in*, *occupation* and *developed*. Once this snippet of the graph is incorporated into a larger graph, we get additional links (shown by dotted edges) such as a Theoretical Physicist is a *kind of* Physicist who *practices* Physics, and that Theory of Relativity is a *branch of* Physics.

Albert Einstein was a **German-born theoretical physicist** who developed the **theory of relativity**.



A holy grail of computer vision is the complete understanding of an image, that is, creating a model that can name and detect objects, describe their attributes, and recognize their relationships. Understanding scenes would enable important applications such as image search, question answering, and robotic interactions. Much progress has been made in recent years towards this goal, including image classification and object detection.



For example, from the image shown above, an image understanding system should produce a graph shown to the right. The nodes in the graph are the outputs of an object detector. Current research in computer vision focuses on developing techniques that can correctly infer the relationships between the objects, such as, man *holding* a bucket, and horse *feeding* from the bucket, etc. The graph shown to the right is an example of a knowledge graph, and conceptually, it like the data graphs that we introduced earlier.

4.2 Graphs as input to Machine Learning

Popular deep machine learning models rely on a numerical input which requires that any symbolic or discrete structures should first be converted into a numerical representation. *Embeddings* that transform a symbolic input into a vector of numbers have emerged as a representation of choice for input to machine learning models. We will explain this concept and its relationship to knowledge graphs by taking the example of *word embeddings* and *graph embeddings*.

Word embeddings were developed for calculating similarity between words. To understand the word embeddings, we consider the following set of sentences.

I like knowledge graphs.
I like databases.
I enjoy running.

In the above set of sentences, we will count how often a word appears next to another word, and record the counts in a matrix shown below. For example, the word *I* appears next to the word *like* twice, and next to word *enjoy* once, and therefore, its counts for these two words are 2 and 1 respectively, and 0 for every other word. We can calculate the counts for other words in a similar manner to fill out the table.

counts	I	like	enjoy	knowledge	graphs	databases	running	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
knowledge	0	1	0	0	1	0	0	0
graphs	0	0	0	1	0	0	0	1
databases	0	1	0	0	0	0	0	1
running	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Above table constitutes a matrix which is often referred to as *word cooccurrence* counts. We say that the meaning of each word is captured by the vector in the row corresponding to that word. To calculate similarity between words, we can simply calculate the similarity between the vectors corresponding to them. In practice, we are interested in text that may contain millions of words, and a more compact representation is desired. As the above matrix is sparse, we can use techniques from Linear Algebra (e.g., singular value decomposition) to reduce its dimensions. The resulting vector corresponding to a word is known as *word embedding*. Typical word embeddings in use today rely on vectors of length 200. There are numerous variations and extensions of the basic idea presented here. Techniques exist for automatically learning word embeddings for any given text.

Use of word embeddings has been found to improve the performance of many natural language processing tasks including entity extraction, relation extraction, parsing, passage retrieval, etc. One of the most common applications of word embeddings is in auto completion of search queries. Word embeddings give us a straightforward way to predict the words that are likely to follow the partial query that a user has already typed.

As a text is a sequence of words, and word embeddings calculate co-occurrences of words in it, we can view the text as a graph in which every word is a node, and there is a directed edge between each word and another word that immediately follows it. Graph embeddings generalize this notion for general network structure. The goal and approach, however, remains the same: represent each node in a graph by a vector, so that the similarity between the nodes can be calculated as a difference between their corresponding vectors. The vectors for each node are also referred to as graph embeddings.

To calculate graph embeddings, we define a method for encoding each node in the graph into a vector, a function to calculate similarity between the nodes, and then optimize the encoding function. One possible encoding function is to use a *random walk* of the graph and calculate co-occurrence counts of nodes on the graph yielding a matrix similar to co-occurrence counts of words in text. There are numerous variations of this basic method to calculate graph embeddings.

We have chosen to explain graph embeddings by first explaining word embeddings because as it is easy to understand them, and their use is common place. Graph embeddings are a generalization of the word embeddings. They are a way to input domain knowledge expressed in a knowledge graph into a machine learning algorithm. Graph embeddings do not induce a knowledge representation, but are a way to turn symbolic representation into a numeric representation for consumption by a machine learning algorithm.

Once we have calculated graph embeddings, they can be used for a variety of applications. One obvious use for the graph embeddings calculated from a friendship graph is to recommend new friends. A more advanced tasks involve link prediction (ie, the likelihood of a link between two nodes), etc. Link prediction in a company graph could be used to identify potential new customers.

5. Summary

Graphs are a fundamental construct in discrete mathematics, and have applications in all areas of computer science. Most notable uses of graphs in knowledge representation and databases have taken the form of data graphs, taxonomies and ontologies. Traditionally, such applications have been driven by a top down design. As a knowledge graph is a directed labeled graphs, we are able to leverage theory, algorithms and implementations from more general graph-based systems in computer science.

Recent surge in the use of knowledge graphs for organizing information on the internet, data integration, and as an output target for machine learning, is primarily driven in a bottom up manner. For organizing information on the web, and in many data integration applications, it is extremely difficult to come up with a top down design of a schema. The machine learning applications are driven by the availability of the data, and what can be usefully inferred from it. Bottom up uses of knowledge graphs do not diminish the value of a top down design of the schema or an ontology. Indeed, the Wikidata project leverages ontologies for ensuring data quality, and most enterprise data integration projects advocate defining the schema on as needed basis. Machine learning applications also benefit significantly with the use of rich ontology for making inferences from the information that is learned even though a global ontology or a schema is not required at the outset.

Word-embeddings and graph-embeddings both leverage a graph structure in the input data, but they are necessarily more general than *knowledge graphs* in that there is no implicit or explicit need for a schema or an ontology. For example, graph embeddings can be used over the network defined by exchange of messages between nodes on the internet, and then used in machine learning algorithms to predict rogue nodes. In contrast, for the Wikidata graph, knowledge graphs in the enterprises, and in the output representation of machine learning algorithms, a schema or ontology can play a central role.

We conclude by observing that the recent surge in interest in knowledge graphs is primarily driven by the bottom up requirements of several compelling business applications. Knowledge graphs in these applications can certainly benefit from the classical work on the top down representation design techniques, and in fact, we envision that eventually the two will converge.

Exercises

Exercise 1.1. Identify which of the following satisfies the definition of a knowledge graph introduced in this chapter.

(*a*) A *data graph* defined among data items representing real-world entities.

(*b*) A *schema graph* defined among classes in a schema.

(*c*) A *process graph* representing the steps of a process, their order, and branching conditions.

(*d*) A *parse tree* is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.

(*e*) An *entity relationship diagram* shows the relationships of entity sets stored in a database.

Exercise 1.2. Which of the following counts as a well-defined meaning of the labels in a knowledge graph?

(*a*) Names of the labels in a human understandable language.

(*b*) Everything in (a) plus a documentation string that explains the label in sufficient detail.

(*c*) Embeddings calculated for the relation names over a large corpus of text.

(*d*) Everything in (a) plus a specification in a formal language.

(*e*) Everything in (b) plus a specification in a formal language.

Exercise 1.3. Identify which of the following statements about knowledge graphs are true.

(*a*) Knowledge graphs are the only way to achieve data integration in enterprises.

(*b*) Edges in a knowledge graph are like the links between web documents except that the edges have semantically defined labels.

(*c*) A knowledge graph is the best representation for recording the output of NLP and vision algorithms.

(*d*) Semantic networks were the earliest knowledge graphs in AI.

(*e*) Understanding is to brain as a knowledge graph is to AI.

Exercise 1.4. Identify if the following statements are true or true or false.

(*a*) If word embeddings of two words show high similarity, they are likely to be synonyms.

(*b*) A word embedding calculation views the text as a knowledge graph.

(*c*) A sentence is to a word embedding as a path is to a graph embedding.

(*d*) Edge detection is to computer vision as relation extraction is to NLP.

(*e*) Calculating similarity using word embeddings is always better than using hand curated sources.

Exercise 1.5. What is not new about the knowledge graphs in their recent usage?

(*a*) Directed labeled graph representation

(*b*) The large sizes of the knowledge graphs

(*c*) Creating a knowledge graph using automated methods

(*d*) Publicly available curated graph data sets

(*e*) Ability to make high value predictions using graph data

CS520

Knowledge Graphs

What should AI Know ?

What are some Graph Data Models?

1. Introduction

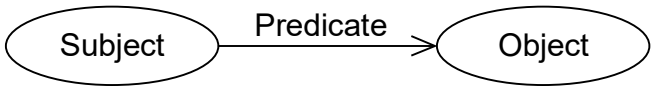
Two popular graph data models are Resource Description Framework (RDF), and the property graph (PG) model. The query language for RDF is SPARQL, and the query language for the property graph model is Cypher. In this chapter, we present an informal overview of both of these data models and give example queries for them. This chapter introduces the two models without giving a comprehensive technical overview. We consider translation of data represented using one of the models to data represented in the other, and also compare these graph data models to using a conventional relational data model.

2. Resource Description Framework

RDF is a framework for representing information on the web. The RDF data model and its query language SPARQL have been standardized by the World Wide Web Consortium.

2.1 RDF Data Model

An RDF triple, the basic unit of representation in this model, consists of a subject, a predicate, and an object. A set of such triples is called an RDF graph. We can visualize an RDF triple as a node and a directed edge diagram in which each triple is represented as a node-edge-node graph as shown below.



There can be three kinds of nodes: IRIs, literals, and blank nodes. An IRI is an Internationalized Resource Identifier used to uniquely identify resources on the web. A literal is a value of a certain data type, for example, string, integer, etc. A blank node is a node that does not have an identifier, and is similar to an anonmyous or an existential variable.

As an example illustration of the information expressed using RDF, let us take the example of representing *knows* relationship between people. In this example, a person with name *art* is denoted by the IRI http://example.org/art. In the notation used below the IRIs can be abbreviated by defining a prefix. For example, we have defined *foaf* as a prefix for <*http://xmlns.com /foaf/0.1/*>. The relation *knows* is defined by the IRI http://xmlns.com/foaf/0.1/knows

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix ex: <http://example.org/>

ex:art foaf:knows ex:bob
ex:art foaf:knows ex:bea
ex:bob foaf:knows ex:cal
```

```
ex:bob foaf:knows ex:cam
ex:bea foaf:knows ex:coe
ex:bea foaf:knows ex:cory
ex:bea foaf:age 23
ex:bea foaf:based_near _:o1
```

The last two triples illustrate a literal node and a blank node. The value of *foaf:age* is the integer 23 which is an example of a literal. A string is another common literal data type. The value of *foaf:based_near* is an anonymous spatial thing which is indicated by an underscore as a node identifier. Here *o1* is an internal data identifier which has no significance outside the context of the present graph.

An RDF vocabulary is a collection of IRIs intended for use in RDF graphs. The IRIs in an RDF vocabulary often begin with a common substring known as a namespace IRI. In the example above, *<http://xmlns.com/foaf/0.1/>* is a name space prefix. Some namespace IRIs are associated by convention with a short name known as a namespace prefix. In the example above, we defined *foaf* and *ex* as name space prefixes.

RDF graphs are atemporal in the sense that they provide a static snapshot of data. With suitable vocabulary extension, they can express information about events, or other dynamic properties of entities.

An RDF dataset is a collection of RDF graphs and comprises exactly one default graph that can be empty and does not need to have a name, and one or more named graphs. Each named graph consists of an IRI or a blank node that represents its name and the RDF graph.

2.2 SPARQL Query Language

SPARQL (pronounced "sparkle", a recursive acronym for Simple Protocol and RDF Query Language) is a query language to retrieve and manipulate data stored in the Resource Description Framework (RDF). SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The results of SPARQL queries can be sets or RDF graphs.

Most forms of a SPARQL query contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables in the graph pattern and the result is the RDF graph equivalent to the sub graph.

The example below shows a SPARQL query against the data graph shown above and queries for the people known by a person. The query consists of two parts: the SELECT clause identifies the variables to appear in the query results, and the WHERE clause provides the graph pattern to match against the data graph. The graph pattern in this example consists of a single triple with a single variable (?person) in the object position.

```
SELECT ?person
WHERE
<http://example.org/art> <http://xmlns.com/foaf/0.1/knows> ?person
```

Above query returns the following result set on our data graph.

?person1
<http://example.org/bob>
<http://example.org/bea>

The graph pattern can contain multiple triples. For example, in the query below, we ask for *art*'s friends of friends.

```
SELECT ?person ?person1
WHERE
<http://example.org/art> <http://xmlns.com/foaf/0.1/knows> ?person
?person <http://xmlns.com/foaf/0.1/knows> ?person1
```

Above query returns the following result set on our data graph.

?person1	?person2
<http://example.org/bob>	<http://example.org/cal>
<http://example.org/bob>	<http://example.org/cam>
<http://example.org/bea>	<http://example.org/coe>
<http://example.org/bea>	<http://example.org/cory>

Each solution gives one way in which the selected variables can be bound to RDF terms so that the query pattern matches the data. The result set gives all the possible solutions. In the above example, two different subsets of the data provided the matches that resulted in the answers. Above examples illustrate a basic graph pattern match; all the variables used in the query pattern must be bound in every solution.

The SPARQL queries can return blank nodes in the result. The identifiers for blank nodes used in the result may not be the same as the one used in the original RDF graph. The WHERE class in a SPARQL query provides a way to match against specific literal types and to filter the results based on numerical constraints.

The SPARQL queries have various forms. The SELECT form that we have considered until now returns the variable bindings. The CONSTRUCT form can be used to create results that define an RDF graph. The queries can also specify more than one graph patterns such that all of them or some of them have to match against the RDF data. The query results can also be further processed by providing directives to order them, eliminate duplicate results, or to limit the total number results returned.

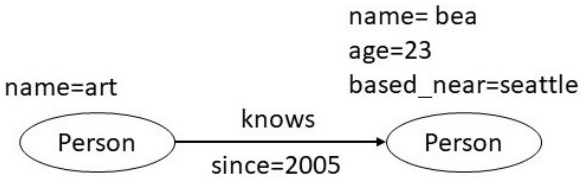
3. Property Graphs

Property graphs data model is used by many popular graph database systems. Unlike RDF that was explicitly motivated by a need to model information on the web, the graph database systems handle general graph data. Graph database systems distinguish themselves from traditional relational databases with little reliance on a predefined schema, and the optimization of operations that involve graph traversals. In this section, we will consider the property graph data model and the Cypher language that is used to query it.

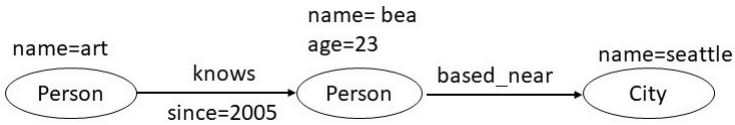
3.1 Property Graph Data Model

A property graph data model consists of nodes, relationships and properties. Each node has a label, and a set of properties in the form of arbitrary key-value pairs. The keys are strings and the values are arbitrary data types. A relationship is a directed edge between two nodes, has a label, and can have a set of properties.

In the property graph shown below *art* and *bea* are two nodes. The node for *bea* has two properties: *age* and *based_near*. These two nodes are connected by an edge labeled as *knows*. This edge has the property *since* that indicates the year from which *art* and *bea* have known each other.



While defining a property graph data model, one has to decide the nodes, the edges, and the properties. For example, one can question why not to represent a city also as a node, and create an edge labeled as *based_near* between a person and the city instead of making them a property of the node representing a person. In general, any value that may be related to multiple other nodes in the graph such that there is either an application requirement to traverse those relationships efficiently, or we need to associate addiational properties with how it is related to other nodes, should itself be represented as a node. In this example, if we intend to traverse the *based_near* relationships, the following design would be more appropriate. Furthermore, it allows us to associate any additional properties with the *based_near* relationship.



3.2 Cypher Query Language

Cypher is a language for querying data represented in a property graph data model. The design concepts from Cypher are being considered for adoption into an ISO standard for a graph query language. In addition to querying, Cypher can also be used to create, update and delete data from a graph database. In this section, we will introduce only the query capabilities of Cypher.

The example below shows the Cypher query against the data graph considered earlier and queries for the the persons known by *art*. The query consists of two parts: the MATCH clause specifies a graph pattern that should match against the data graph and the RETURN clause specifies what should the query return. The graph pattern is specified in an ASCII notation for graphs: each node is written in parentheses, and each edge is written as an arrow. Both node and relation specifications include their respective types, and any additional properties that should be matched.

```
MATCH (p1:Person {name: art}) -[:knows]-> (p2: Person)
RETURN p2
```

In the example below, we show the Cypher query that asks for all the friends of a person that have existed since 2010.

```
MATCH (p1:Person {name:art}) -[:knows {since: 2010}]-> (p2: Person)
```

```
RETURN p2
```

From the above query, we can see that it is equally easy to associate properties with relations as it is with nodes. A person may have friends from years before 2010, and if we wanted the query to include those friends as well, it can be done by adding a WHERE clause.

```
MATCH (p1:Person {name:art}) -[:knows {since: Y}]-> (p2: Person)
WHERE Y <= 2010
RETURN p2
```

Through the WHERE clause, it is possible to specify a variety of filtering constraints as well as patterns that can be used to restrict the query results. In addition, Cypher provides language constructs for counting results, grouping data by values, and finding minimum/maximum values, and other mathematical and aggregation operations.

4. Comparison of Data Models

In this section, we will start off by comparing the RDF and the property graph data models. We will then compare both of them to relational data model.

4.1 Comparison of RDF and Property Graph Data Models

Beyond the features of RDF considered in the previous section, it has several additional layers, for example, RDF schema, Web Ontology Language (OWL), etc. Our discussion here will not consider those advanced features. The primary differences between the basic RDF model and the property graph model are that: (a) the property graph model allows edges to have properties (b) the property graph model does not require IRIs and does not support blank nodes. To support the edge properties, the RDF model supports an extension known as *reification*. We will consider this extension, and then describe different ways in which the data represented in either data model can be converted into the other format.

To understand reification in RDF, consider a situation in which we need to represent the provenance of the triple shown below. This triple asserts the weight of an item. The literal "2.4"^^xsd:decimal denotes the number 2.4 which is of type xsd:decimal. We are interested in specifying the person who took this measurement.

```
exproducts:item10245 exterms:weight "2.4"^^xsd:decimal
```

We can associate provenance information with the above triple using the RDF reification vocabulary. The RDF reification vocabulary consists of the type *rdf:Statement*, and the properties *rdf:subject*, *rdf:predicate*, and *rdf:object*. Using the reification vocabulary, a reification of the statement about the weight of the item would be given by assigning the statement an IRI such as *exproducts:triple12345* (so statements can be written describing it), and then describing the statement as shown below. The last triple in the list below specifies the desired provenance information by asserting the identifier for the person who created the original triple.

```
exproducts:triple12345 rdf:type rdf:Statement .
exproducts:triple12345 rdf:subject exproducts:item10245 .
exproducts:triple12345 rdf:predicate exterms:weight .
exproducts:triple12345 rdf:object "2.4"^^xsd:decimal .
exproducts:triple12345 dc:creator exstaff:85740 .
```

These statements say that the resource identified by the IRI *exproducts:triple12345* is an RDF statement, that the subject of the statement refers to the resource identified by *exproducts:item10245*, the predicate of the statement refers to the resource identified by *exterms:weight*, and the object of the statement refers to the decimal value identified by the typed literal *"2.4"^^xsd:decimal*. The final statement asserts that *exproducts:triple12345* was provided by the person with the IRI *exstaff:8574*.

With the above reification vocabulary, it becomes possible to mechanically translate the data in the property graph model to RDF. Each node and its property value in the property graph data becomes a triple. Each edge in property graph data also becomes an RDF triple. Every edge in the property graph data that has a property is reified, and the properties of the edge become the triples of the reified edge that use the reification vocabulary as explained above.

To translate data expressed in the RDF model to the property graph model, a straightforward approach is to map each node and an edge to the corresponding node and an edge in the property graph. A possible refinement is that we create new property nodes only for those nodes that are either IRIs or blanks nodes. For any triple in RDF in which the target is a literal, we make it a property of the node in the property graph data.

In addition to converting data between RDF and property graph models, we are also interested in converting the syntactic form of data and the queries. For property graph model, there is no syntatic standard for their expression, and therefore, a custom translator needs to be written for the format one is working with. Once a translation scheme is fixed between the two data models, the corresponding translation between SPARQL and Cypher is straightforward.

4.2 Comparison of Graph Models and Relational Data Model

We can define a translation to and from the data expressed using relational model to data expressed using the RDF model and the property graph model. Some argue that the graph models are easier for humans to understand and that the graph query languages are more compact for certain queries. In principle, we can implement a user interface to visualize the relational schemas, and implement a query compiler that can map a query written in a graph query language into an equivalent form that operates on the relational tables. If an application requires navigating relationships, a graph database has an edge as it is optimized for graph traversals. For the rest of the section, we will consider an example to illustrate how the graph queries can be more compact than the corresponding relational queries, and conclude by mentioning the systems that attempt to support graph processing on a relational system.

To understand the contrast between graph queries and relational queries, we will consider a simple example in which we have three tables: an *Employee* table, a *Department* table, and an *Employee-Department* join table. An employee can be associated with multiple departments because of which they are stored in separate tables. Two tables are related with a join table that contains their foreign keys employee id and department id. We show these tables below.

Employee		
id	**name**	**ssn**
e01	alice	...
e02	bob	...
e03	charlie	...
e04	dana	...

Employee_Department	
employee id	**department id**
e01	d01
e01	d02
e02	d01
e03	d02

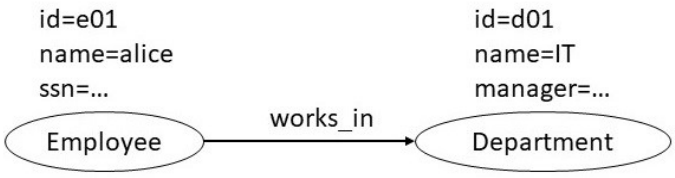
Department		
id	**name**	**manager**
d01	IT	...
d02	Finance	...
d03	HR	...

e04	d03

Given the tables as shown above, suppose we wish to list the employees in the IT department. The SQL query to perform this task will first need to join the *employee* and the *department* tables, and then filter the results on the *department* name. The required query is shown below.

```
SELECT name FROM Employee
LEFT JOIN Employee_Department
   ON Employee.Id = Employee_Department.EmployeeId
LEFT JOIN Department
   ON Department.Id = Employee_Department.DepartmentId
WHERE Department.name = "IT"
```

If we were to represent the same information using a property graph data model, we will have a node for *department* and *employee*. The employee *ssn* and department *name* will be the node properties. The *Employee_Department* table will be captured using a relationship in the property graph representation. If the *Employee_Department* table had additional attributes, they will be represented as edge properties in the property graph data model. A sample node in such a property graph is shown below.



We can query this data using the following Cypher query:

```
MATCH (p:Employee) -[:works_in]-> (d:Department)
WHERE d = "IT"
RETURN p
```

The Cypher query above is much more compact than its SQL counterpart. This compactness stems from the fact that the joins are naturally captured using graph patterns.

There have been some recent systems that represent the relational data in a schema free manner by representing each node property as a triple in one table, and each edge property as a four tuple in a second table. Such systems provide a query planner that accepts queries in a language like Cypher that computes an efficient execution plan over the two relational tables. Such systems are able to leverage the existing relational technology, and are also able to perform optimizations when some of the legacy data is in a traditional relational table.

5. Limitations of a Graph Data Model

A graph data model is not the most appropriate choice when the application contains primarily numeric data, and the reliance on only binary relationships is limiting. For example, the relational model is more effective in capturing timeseries data such as evolution of the population of a country. Even though we can represent such data using a graph, but it results in a huge number of triples without necessarily giving us advantages of better conceptual understanding and/or faster query performance through graph traversals. There are many relationships that cannot be naturally

represented using binary relations. For example, *between* relation that captures that an object *A* is between two other objects *B* and *C* is inherently a ternary relationship. A ternary relationship can be transformed into a set of binary relationships using the reification technique, but by doing so, we lose the advantage of better conceptual understanding that we get from the graph data model. Graphs are also not the most natural representation for mathematical equations and chemical reactions where easy to understand domain specific representations exist.

6. Summary

In this chapter, we reviewed two popular graph data models: RDF and property graphs. RDF was devised for representing information over the web, and makes an extensive use of IRIs. The property graph model is a popular choice in many graph database systems, and provides a direct support for associating properties with both nodes and edges. Even though there are small differences between the two models, it is possible to inter-translate the data represented in one to the other. SPARQL is the query language for accessing data in RDF, and Cypher is the corresponding language for the data represented in property graphs. In a graph query language, queries requiring traversals are much more compact in comparison to an equivalent formulation in a relational data model. A graph data model can also provide a better user understanding of the knowledge in the subject domain. There are some systems that use a relational database as the storage for graph data and provide query optimizers to still allow queries to be expressed in a graph query language. Finally, a graph data model offers significant advantages for application that have rich relationships between objects, and require extensive traversal of those relationships.

Exercises

Exercise 2.1. For the following triple, identify which of the elements is subject, predicate or object.

```
<http://example.org/john> <http://xmlns.com/foaf/0.1/name> "John
Smith"
```

Exercise 2.2. Which of the following statements is true?

(*a*) An anonymous node is the same as a blank node.

(*b*) Every IRI is also a URI.

(*c*) Blank nodes can never be used outside the RDF document in which they were originally defined.

(*d*) An RDF document can refer to identifiers defined in only one namespace.

(*e*) An RDF dataset must contain exactly one dataset.

Exercise 2.3. The following SPARQL query illustrates the use of OPTIONAL graph patterns. Within each result returned by this query, what is the minimum and the maximum possible data items it must much:

```
SELECT ?foafName ?mbox ?gname ?fname
WHERE
{ ?x foaf:name ?foafName .
    OPTIONAL { ?x foaf:mbox ?mbox } .
    OPTIONAL { ?x vcard:N ?vc .
        ?vc vcard:Given ?gname .
        OPTIONAL { ?vc vcard:Family ?fname }
              }
```

```
}
```

(*a*) minimum 4 / maximum 4

(*b*) minimum 0 / maximum 4

(*c*) minimum 1 / maximum 4

(*d*) minimum 2 / maximum 4

(*e*) minimum 1 / maximum 3

Exercise 2.4. Which of the following statements is true about the property graph data model?

(*a*) Nodes and relationships define the graph while properties add context by storing relevant information in the nodes and relationships.

(*b*) Property graph defines a graph meta-structure that acts as a model or schema for the data as it is entered.

(*c*) The Property graph is a model like RDF which describes how Neo4j stores resources in the database.

(*d*) The Property graph allows for configuration properties to define schema and structure of the graph.

(*e*) All of the above.

Exercise 2.5. Which of the following Cypher queries will return the actors who directed the movies they acted in?

(*a*)
```
MATCH (actor)-[a:ACTED_IN]->(movie)<-[a:DIRECTED]-(actor)
RETURN a
```

(*b*)
```
MATCH (actor)-[:ACTED_IN]->(movie)
JOIN (movie)<-[:DIRECTED]-(actor)
RETURN actor
```

(*c*)
```
MATCH (actor)-[:ACTED_IN]->(movie)
CONNECT (movie)<-[:DIRECTED]-(actor)
RETURN actor
```

(*d*)
```
MATCH (actor)-[:ACTED_IN]->(movie)<-[:DIRECTED]-(actor)
RETURN actor
```

(*e*) None of the above.

Exercise 2.6 . Suppose we wish to associate a confidence of 0.9 with an automatically extracted statement *John works for ABC Corporation*. Which of the following is true about relative approaches for representing this statement in RDF and property graph data models.

(*a*) To represent confidence of a statement, we must reify the statement in both property graph and RDF data models.

(*b*) In a property graph data model, we can represent the confidence of a statement as a relationship property.

(*c*) We can represent confidence of a statement by reifying it in both property graph and RDF data models.

(*d*) The only way to represent the confidence of a statement in an RDF data model is through reification.

(*e*) The confidence of the statement cannot be captured in an RDF data model.

Exercise 2.7 . Which of the following statements is true regarding a relational data model and a

graph data model?

(*a*) A relational database system can be used as a storage mechanism for a graph database.

(*b*) It is easier to change the schema in a graph database than in a relational database.

(*c*) Queries always run faster in a graph database.

(*d*) A graph database is an ideal choice for storing timeseries data.

(*e*) Relational data model has a definite advantage if we need to capture relations of arity higher than 2.

CS520

Knowledge Graphs

What should AI Know ?

How to Create a Knowledge Graph?

1. Introduction

It is possible to get started with a knowledge graph with no upfront design of its schema, and populate both its schema and instances during the creation process. To the degree an upfront design of a knowledge graph is practical, it can significantly improve its usefulness. Such a design involves making a suitable choice of the nodes, node labels, node properties, relations and relation properties.

The input to knowledge graph population can come from one or more sources consisting of structured data, semi structured data, free text or images, or direct authoring by human input. When we are working with structured and semi structured data sources, we have to perform schema mapping task (i.e., relating the schema in the input source with the schema of the knowledge graph) and record linkage task (i.e., relating new instances with the pre-existing instances in the knowledge graph). These exact same tasks are also faced during data integration with the only difference that the integrated data is expressed in a graph data model. When we are working with the unstructured sources, we have to solve the information extraction problems of entity extraction and relation extraction.

The choice of methods used in knowledge graph population depend on the scale of the problem and the desired accuracy. If a knowledge graph is to be used on the web scale for information retrieval, the accuracy need not be perfect, and it is infeasible to use human verification for every triple of the graph. If a knowledge graph is to be used within an enterprise where the accuracy needs to be nearly perfect, the human verification is essential even if it is performed just before the information is to be used. As accuracy is always desired regardless of the enterprise or the WWW settings, to ensure cost effectiveness and scalability, there is an empahsis on crowdsourcing and other low-cost methods of obtaining human input.

In this chapter, we will focus on knowledge graph schema design. In the next two chapters we will discuss the problems that arise in populating a knowledge graph from structured data, i.e., the problems of record linkage and schema mapping, and the problems that arise while populating from text, i.e., entity extraction and relation extraction.

2. Knowledge Graph Design

Both property graph and RDF data models have a set of design issues some of which are common across the two, while others are unique. For example, both models need to use reification for situations that cannot be directly modeled using triples. An RDF model needs to adopt a scheme for IRIs which is not necessary for property graphs. In a property graph model, we need to decide whether a value should be represented as a property or as a node, while this distinction is unnecesary in an RDF model. In this section, we will present an overview of such design issues that are faced in each of these two models.

2.1 Design of an RDF Graph

The knowledge graph authoring guidelines for RDF data on the WWW are known as the linked data principles as outlined below.

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

We will consider each of these guidelines in greater detail.

2.1.1 Use URIs as names for things

To publish a knowledge graph on the WWW, we first have to identify the items of interest in our domain. They are the things whose properties and relationships, we want to describe in the graph. In WWW terminology, all items of interest are called resources. The resources are of two kinds: information resources and non-information resources. All the resources we find on the traditional WWW, such as documents, images, and other media files, are information resources. But many of the things we want in our knowledge graph are not: People, physical products, places, proteins, scientific concepts, etc. As a rule of thumb, all "real-world objects" that exist outside of the WWW are non-information resources.

The publishers of knowledge graphs should construct the URIs to be shared in a way that they are simple, stable and manageable. Short, mnemonic URIs will not break as easily when sent in emails and are in general easier to remember. Once we setup a URI to identify a certain resource, it should remain this way as long as possible. To ensure long-term persistence, it is best to keep implementation-specific bits and pieces such as ".php" and ".asp" out of the URIs. Finally, the URIs should be defined in a way that they can be fully managed by the publisher.

2.1.2 Use HTTP URIs so that people can look up those names

We identify resources using Uniform Resource Identifiers (URIs). We restrict ourselves to using HTTP URIs only and avoid other URI schemes such as Uniform Resource Names (URNs) and Digital Object Identifiers (DOIs).

The process of looking up names is referred to as URI dereferencing. When we dereference a URI for an information object, we expect to get the representation of its current state (e.g., a text document, an image, a video, etc.) But, when we dereference a non-information resource, we can obtain its description in RDF expressed in an XML notation.

2.1.3 When someone looks up a URI, provide useful information using RDF and SPARQL

When someone looks up a URI, the provider should return a knowledge graph in RDF. The data should reuse standardized vocabularies to name the IRIs used in describing the RDF data. Several useful vocabularies are available for describing data catalogs, organizations, and multidimensional data, such as statistics on the Web. An open source effort called Schema.Org publishes community created open source vocabularies for open use over the web. We consider a few examples of such vocabularies.

The following RDF data describes a snippet of the organizational structure of the UK Cabinet office.

```
@prefix uk_cabinet: <http://reference.data.gov.uk/id/department/>
uk_cabinet:co rdf:type org:Organization
uk_cabinet:co skos:prefLabel "Cabinet Office"
uk_cabinet:co org:hasUnit uk_cabinet:cabinet-office-communications
uk_cabinet:cabinet-office-communications rdf:type org:OrganizationUnit
uk_cabinet:cabinet-office-communications skos:prefLabel "Cabinet Office
Communications"
uk_cabinet:cabinet-office-communications org:hasPost uk_cabinet:post_246
uk_cabinet:post_246 skos:prefLabel "Deputy Director, Deputy Prime
Minister's Spokesperson"
```

In the data above, the first triple uses the class *org:Organization* from the Organization ontology. The second triple uses the relation *skos:prefLabel* drawn from the SKOS ontology. SKOS stands for a Simple Knowledge Organization System, and provides a few commonly useful relations such as *skos:prefLabel* for describing data. In this case, *skos:prefLabel* simply allows us to associate a text label with *uk_cabinet:co*. The third triple uses the relation *org:hasUnit* from the Organization ontology to describe a unit within the UK Cabinet office. Next two triples make additional assertions about this unit. The sixth triple uses the *org:hasPost* relation to describe a position with in a department, and the final two triples give additional information about that position.

It may not be always possible to find pre-existing vocabularies that can be used in creating an RDF data set. If creating a new vocabulary becomes necessary, one should ensure that it is documented, self-describing, has a versioning policy, is defined in multiple languages, and is published by a trusted source so that the URIs used in it persist for a long period of time. We say that a vocabulary is self-describing if each property or term has a label, definition and comment defined.

2.1.4 Include links to other URIs, so that they can discover more things

While publishing data using RDF one should provide links to other objects so that its usefulness increases. There can be three kinds of links: *relationship links*, *identity links*, and *vocabulary links*. We will consider an example of each of these kinds of links.

Relationship links point at related things in other data sources such as other people, places or genes. For example, relationship links enable people to point to background information about the place they live, or to bibliographic data about the publications they have written. In the triple below, we show a link in which a person in one data set is asserted to be based near a geographical location that is specified using a URI in another data set.

```
@prefix big: <http://biglynx.co.uk/people/>
@prefix dbpedia: <http://dbpedia.org/resource/>
big:dave-smith foaf:based_near dbpedia:Birmingham
```

Identity Links point at URI aliases used by other data sources to identify the same real-world object or abstract concept. Identity links enable clients to retrieve further descriptions about an entity, and serve an important social function as they enable different views of the world to be expressed on the WWW of Data. It is a standard practice to use the link type *http://www.w3.org/2002/07 /owl#sameAs* to state that two URI aliases refer to the same resource. For example, if *Dave Smith* would also maintain a private data homepage besides the data that *Big Lynx* publishes about him,

he could add a *http://www.w3.org/2002/07/owl#sameAs* link to his private data homepage, stating that the URI used to refer to him in this document and the URI used by *Big Lynx* both refer to the same real-world entity. A triple capturing this information is shown below.

```
@prefix ds: <http://www.dave-smith.eg.uk>
@prefix owl: <http://www.w3.org/2002/07/owl>
@prefix big: <http://biglynx.co.uk/people/>
ds:me owl:sameAs big:dave-smith
```

Vocabulary links point from data to the definitions of the vocabulary terms that are used to represent the data, as well as from these definitions to the definitions of related terms in other vocabularies. Vocabulary links make data self-descriptive and enable Linked Data applications to understand and integrate data across vocabularies. In the vocabulary link shown below, the class *SmallMediumEnterprise* defined by BigLynx is defined to be a subclass of the class *Company* in the DBpedia ontology. By making such a link, it is possible to retrieve various assertions about the class Company from the DBPedia, and use them with the class *SmallMediumEnterprise*.

```
@prefix dbpedia: <http://dbpedia.org/ontology/>
big:sme#SmallMediumEnterprise rdfs:subClassOf dbpedia:Company
```

2.2 Design of a Property Graph

The design of a property graph involves choosing nodes, node labels, node properties, edges and edge properties. The basic design questions are whether to model a piece of information as a property, label or as a separate object; when to introduce relation properties; and how to to handle higher arity relationships. We will illustrate the process of making these choices using examples.

2.2.1 Choosing Nodes, Labels and Properties

In a property graph model, the nodes usually represent entities in the domain. If we were interested in representing information about people, we will create a node for each individual person (e.g., John), and associate the label Person with that node.

There are several considerations in making further choices of node labels, node properties and edges. These considerations include: naturalness of labels, whether the labels might change over a period of time, runtime query performance, and the cardinality of values.

To illustrate the choice of whether to model a piece of information as a label, property, or as a separate object, consider the task of representing the gender of a person. We have three potential ways to capture this information: we can create *:Male* and *:Female* as labels and associate them with the *Person* nodes; (2) we can create a property called "gender", and associate it with *Person* nodes and allow it to have the values "male" and "female"; (3) we can create a *Gender* object, associate it with *Person* using a *has_gender* relationship, and give it a property called "name" that can take "male" and "female" as values.

The labels in a property graph model are used to group nodes into sets. All nodes labeled with the same label belong to the same set. Queries can work with these sets instead of the whole graph, making queries easier to write and more efficient. A node may be labeled with any number of labels, including none, making labels an optional addition to the graph. As a label groups nodes into a set, it can be viewed as a class. The question of whether to introduce a new label can be restated as whether to introduce a new class?

Creating new classes *Male* and *Female* vs introducing a node property "gender" that can take two values of "male" and "female" captures the same information. In general, whenever a phrase naturally occurring in language is frequently used in a domain, it is a candidate to be introduced as a class as long as the membership in the class does not change with time. As some implementations optimize the retrieval based on the use of labels, the use of labels can result in fast performance on queries that need to filter the results based on the membership in the class. If class membership changes with time, neither a label nor a node property value is an appropriate choice, and we need to use a relation. We will consider this in the next section.

2.2.2 When to introduce Relationships between Objects

For situations that could be modeled either by using a node property or by introducing a separate object and relationship, there are, at least, two different considerations. First of those considerations was introduced in the previous section: the membership in the class changes with time. The second consideration arises when we wish to achieve better query performance. We will consider these situations in greater detail next.

Continuing the example from the previous section, when the gender of a person could change over a period of time, then our only choice is to capture the information as a separate *Gender* object that is related to *Person* using the *has_gender* relationship. We can then associate a relationship property with the *has_gender* relationship that indicates the time duration for which that particular value of gender holds. Creating a separate *Gender* node would, however, lead to a huge number of edges which is wasteful as for most people the gender does not change. In such a situation, a combination of the two solutions might be desired where for most people the gender is represented as a node property value, but for a small fraction of people, it is represented as a relation property value on a relation to a separate *Gender* node.

Let us consider a situation where better query performance is a key consideration. Suppose we wish to model movies, and their genres. In one design, for a node of type *Movie*, we can introduce a property "genre" that can take values such as "Action", "SciFci", etc. In another design, we can introduce a new node type *Genre* that has a node a property "name" that can take values such as s "Action", "SciFci". We will then relate a node of type *Movie* with a node of type *Genre* using the *has_genre* relationship. In general, we can associate more than one genre with a movie. Suppose we wish to query for those movies that have at least one common genre. In the first solution in which we use the node property "genre", this query would be stated in Cypher as follows:

```
MATCH (m1:Movie), (m2:Movie)
WHERE any(x IN m1.genre WHERE x IN m2.genre)
AND m1 <> m2
RETURN m1, m2
```

When we model genre as a separate object, the same query can be stated as follows:

```
MATCH (m1:Movie)-[:has_genre]->(g:Genre),
      (m2:Movie)-[:has_genre]->(g)
WHERE m1 <> m2
RETURN m1, m2
```

In the second query above, we are able to more directly make use of graph patterns, and in some graph engines, this query has a faster runtime performance because of indexing on relations. Hence, in this case, one has to choose between the two designs depending on the kind of queries

that will be expected.

2.2.3 When to introduce Relationship Properties

We have already seen an example of a property associated with a relationship to deal with situations when the relationship changes with time. Other situations in which it makes sense to introduce properties with relationship include associating weights or confidence with a relationship or to associate provenance or other meta data with a relationship.

Some graph engines do not index based on relationship properties. If the use case is such that much of the query evaluation can be done without using the relationship properties, and they are required only for final filtering of the results, one may not pay significant performance panelty because of lack of indexing. If access to relationship properties is central to query performance, it is better to reify the relation as we will discuss in the next section.

2.2.4 Handling non-binary Relationships

We often need to model relationships that are not binary. A common example of such a relationship is the *between* relationship that given objects A, B and C captures that C is between A and B. A standard approach to capturing such higher arity relationships in a graph is *reification*. We have previously discussed *reification* in the context of RDF, but this technique is equally useful and desirable for property graphs. To capture the *between* relationship we introduce a new node type, *Between_Relationship* that has two properties: *has_object* (with values A and B) and *has_between_object* (with value C). We can use reification for relationships with any arity by creating a new node type for the relation, and by introducing node properties for the different arguments of that relation.

3. Summary

In this chapter, we considered the design of the graph data model for both RDF and property graphs. The data model design concerns such as whether to reify a relationship, handling non-binary relationships, etc., are common across the RDF and the property graph data models. The choice of whether to use a property vs a relation is unique to the property graph data model. The RDF model provides explicit guidelines on the use of IRIs, reuse of existing vocabularies, and making links across vocabularies. Even though the data linking considerations are not integral to the property graph model, but their use can make a property graph system more useful in data integration.

Exercises

Exercise 3.1.Which of the following statements about knowledge graph design are true?

- (a) As knowledge graphs are schema free, no design of the schema is required.
- (b) Knowledge graphs are always created using automatic techniques.
- (c) For many knowledge graph applications, a perfect accuracy is not a hard requirement.
- (d) Knowledge graphs can contain undirected relationships.
- (e) Knowledge graphs do not use keys and foreign keys as defined for the relational database systems.

Exercise 3.2.Which of the following is a good choice of an IRI for an RDF knowledge graph?

(*a*) ISBN-13 : 978-1681737225
(*b*) http://fcvcz.abt.co/mckz/
(*c*) https://www.wikidata.org/wiki/Q6135847
(*d*) http://worksheets.stanford.edu/homepage/index.php
(*e*) http://dbpedia.org/resource/Frederick_Loewe

Exercise 3.3.What type of link is captured by each of the following RDF statements? (Assume the following prefixes have been defined.)

@prefix dbpedia: http://dbpedia.org/resource/
@prefix bbc: http://www.bbc.co.uk/nature/species/
@prefix umbel-rc: https://umbel.org/umbel/rc/Person
@prefix foaf: http://foaf.org/

(*a*) dbpedia:Aardvark owl:sameAs bbc:Aardvark
(*b*) dbpedia:Lady_Gaga skos:broader_of dbpedia:Lady_Gaga_audio_samples
(*c*) dbpedia:Tetris foaf:isPrimaryTopicOf wikipedia-en:Tetris
(*d*) dbpedia:Person rdf:subClassOf umbel-rc:Person
(*e*) dbpedia:Sky_Bank foaf:homepage http://www.skyebankng.com/

Exercise 3.4.Which of the following are good class labels in a knowledge graph?
(*a*) Customers with overdue accounts
(*b*) Australian Customers
(*c*) Customers with revenues between 5 to 10 million
(*d*) Customers who supply to recently funded startups
(*e*) High Networth Value Customers

Exercise 3.5.Which of the following requires reification for representing in a knowledge graph?
(*a*) John believes that life is good.
(*b*) John was referred to Peter by Mary.
(*c*) The effectiveness of a vaccine is 95%.
(*d*) Earth revolves around the Sun.
(*e*) On LinkedIN John rated Peter for being an expert in AI.

CS520

Knowledge Graphs

What should AI Know ?

How to Create a Knowledge Graph from Data?

1. Introduction

Large organizations generate lot of internal data, and also consume data produced by third party providers. Many data providers obtain their data by processing unstructured sources, and invest significant effort in providing it in a structured form for use by others. To make an effective use of such external data, it must be related to company's internal data. Such data integration enables many popular use cases such as 360 view of a customer, fraud detection, risk assessment, loan approval etc. For this chapter, we will discuss the problem of creating a knowledge graph by integrating the data available from structured sources. We will consider the problem of extracting data from unstructured data sources in the next chapter.

When combining data from multiple sources into a knowledge graph, we can undertake some upfront design of schema as we discussed in the previous chapter. We can also begin with no schema as it is straightforward to load external data as triples into a knowledge graph. The design of the initial schema is typically driven by the specific business use case that one wishes to address. To the degree such an initial schema exists, we must determine how the data elements from a new data source should be added to the knowledge graph. This is usually known as the *schema mapping* problem. In addition to relating the schemas of the two sources, we also must deal with the possibility that an entity in the incoming data (e.g., a Company) may already exist in our knowledge graph. The problem of inferring if the two entities in the data may be the same real world entity is known as the *record linkage* problem. The record linkage problem also arises when third party data providers send new data feeds, and our knowledge graph must be kept up-to-date with the new data feed.

In this chapter, we will discuss the current approaches for addressing the schema mapping and the record linkage problems. We will outline the state-of-the-art algorithms, and discuss their effectiveness on the current industry problems.

2. Schema Mapping

Schema mapping assumes the existence of a schema which will be used for storing new data coming from another source. Schema mapping then defines which relations and attributes in the input database corresponds to which properties and relations in the knowledge graph. There exist techniques for bootstrapping schema mappings. The bootstrapped schema mappings can be post corrected through human intervention.

We will begin our discussion on schema mapping by outlining some of the challenges and arguing that one should be prepared for the eventuality that this process will be largely manual and labor-intensive. We then describe an approach for specifying mappings between the schema of the input source and the schema of the knowledge graph. We will conclude the section by considering some of the techniques that can be used to bootstrap schema mapping.

2.1 Challenges in Schema Mapping

The main challenges in automating schema mapping are: (1) difficult to understand schema (2) complexity of mappings, and (3) lack of training data available. We next discuss these challenges in more detail.

Commercial relational database schemas can be huge consisting of thousands of relations and tens of thousands of attributes. Sometimes, the relation and attribute names do not have semantics (e.g., segment1, segment2) which do not lend themseves to any realistic automated prediction of the mappings.

The mappings between the input schema and the schema in the knowledge graph is not always simple one-to-one mapping. Mappings may involve calculations, applying business logic, and taking into account special rules for handling situations such as missing values. It becomes a tall order to expect any automatic process to infer such complex mappings.

Finally, many automated mapping solutions rely on machine learning techniques which require large amount of training data to function effectively. As the schema information, by definition, is much smaller than the data itself, it is unrealistic to expect that we will ever have large number of schema mappings available against which a mapping algorithm could be trained.

2.2 Specifying Schema Mapping

In this section, we will consider one possible approach to specify mappings between input data sources and a target knowledge graph schema. We will take an example in the domain of cookware. We can imagine different vendors providing products which an E-commerce site may wish to aggregate and advertise to its customers. We will consider two example sources, and then introduce the schema of the knowledge graph to which we will define mappings.

We show below some sample data from the first data source in a relational table called *cookware*. It has four attributes: *name*, *type*, *material*, and *price*.

cookware			
name	**type**	**material**	**price**
c01	skillet	cast iron	50
c02	saucepan	steel	40
c03	skillet	steel	30
c04	saucepan	aluminium	20

The second database shown below lists the products of a manufacturer. In this case, there are multiple tables, one for each product attribute. The `kind` table specifies the type of each product. The `base` table specifies whether each product is made from a corrosible metal (aluminum or stainless), a noncorrosible metal (iron or steel), or something other than metal (glass or ceramic). The `coated` table specifies those products that have nonstick coatings. The `price` table gives the selling price. There is no material information. The company has chosen not to provide information about the metal used in each product. Note that the `coated` table has only positive values; products without nonstick coatings are left unmentioned.

kind	**base**	**coated**	**price**

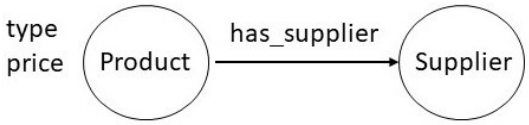
id	value
m01	skillet
m02	skillet
m03	saucepan
m04	saucepan

id	value
m01	corrosible
m02	noncorrosible
m03	noncorrosible
m04	nonmetallic

id	value
m01	yes
m02	yes

id	value
m01	60
m02	50
m03	40
m04	20

Suppose the desired schema for the knowledge graph expressed as a property graph is as shown below. We have two different node types: one for products, and the other for suppliers. These two nodes are connected by a relationship called *has_supplier*. Each product node has properties "type" and "price".



To specify the mappings, and to illustrate the process, we will use a triple notation so that a similar process is applicable regardless of whether we use an RDF or property graph data model for the knowledge graph. For an RDF knowledge graph, we will need to create IRIs which is a process orthogonal to relating the two schemas, and hence omitted from here. The desired triples in the target knowledge graph are listed below.

knowledge graph		
subject	**predicate**	**object**
c01	type	skillet
c01	price	50
c01	has_supplier	vendor_1
c02	type	saucepan
c02	price	40
c02	has_supplier	vendor_1
c03	type	skillet
c03	price	30
c03	has_supplier	vendor_1
c04	type	saucepan
c04	price	20
c04	has_supplier	vendor_1
m01	type	skillet
m01	price	60
m01	has_supplier	vendor_2
m02	type	skillet
m02	price	50
m02	has_supplier	vendor_2
m03	type	saucepan
m03	price	40
m03	has_supplier	vendor_2
m04	type	saucepan
m04	price	20
m04	has_supplier	vendor_2

Any programming language of choice could be used to express the mappings. Here, we have chosen to use Datalog to express the mappings. The rules below are straightforward. Variables are indicated by using upper case letters. The third rule introduces the constant *vendor_1* to indicate the source of the data.

```
knowledge_graph(ID,type,Type) :- cookware(ID,TYPE,MATERIAL,PRICE)
knowledge_graph(ID,price,PRICE) :- cookware(ID,TYPE,MATERIAL,PRICE)
knowledge_graph(ID,has_supplier,vendor_1) :- cookware(ID,TYPE,MATERIAL,PRICE)
```

Next, we consider the rules for mapping the second source. These rules are very similar to the mapping rules for the first source except that now the information is coming from two different tables in the source data.

```
knowledge_graph(ID,type,Type) :- kind(ID,TYPE)
knowledge_graph(ID,price,PRICE) :- price(ID,PRICE)
knowledge_graph(ID,has_supplier,vendor_2) :- kind(ID,TYPE)
```

In general, it may not make sense to reuse the identifiers from the source databases, and one may wish to create new identifiers for use in the knowledge graph. In some cases, the knowledge graph may already contain objects equivalent to the ones in the data being imported. We will consider this issue in the section on record linkage.

2.3 Bootstrapping Schema Mapping

As noted in Section 2.1, a fully automated approach to schema mapping faces numerous practical difficulties. There is a considerable work on bootrapping the schema mappings based on a variety of techniques and validating them using human input. Bootstrapping techniques for schema mapping can be classified into the following categories: linguistic matching, matching based on instances, and matching based on constraints. We will consider examples of these techniques next.

Linguistic techniques can be used on the name of an attribute or on the text description of an attribute. First and most obvious approach is to check if the names of the two attributes are equal. One can have greater confidence in such equality if the names were IRIs. Second, one can canonicalize the names by processing them through techniques such as stemming and then checking for equality. For example, through such processing, we may be able to conclude the mapping of *CName* to *Customer Name*. Third, one could check for the mapping based on synonyms (e.g., car and automibile) or hypernyms (e.g., book and publication). Fourth, we can check for mapping based on common substrings, pronunciation, and how the words sound. Finally, we can match the descriptions of the attributes through semantic similarity techniques. For example, one approach is to extract keywords from the description, and then check for similarity between them using the techniques we have already listed.

In matching based on the instances, one examines the kind of data that exists. For example, if a particular attribute value contains dates, it can only match against those attributes that contain date values. Many such standard data types can be inferred by examining the data.

In some cases, the schema may provide information about constraints. For example, if the schema specifies that a particular attribute must be unique for an individual, and must be a number, it is a potential match for identification attributes such as an employee number or social security number.

The techniques considered here are inexact, and hence, can be only used to bootstrap the schema mapping process. Any mappings must be verified, and validated by human experts.

3. Record Linkage

We will begin our discussion by illustrating the record linkage problem with a concrete example. We will then give an overview of a typical approach to solving the record linkage problem.

3.1 A Sample Record Linkage Problem

Suppose we have data in the following two tables. The record linkage problem then involves inferring that record a_1 is the same as the record b_1, and that record a_3 is the same as the record b_2. Just like in schema mapping, these are inexact inferences, and need to undergo human validation.

	Table A		
	Company	**City**	**State**
a_1	AB Corporation	New York	NY
a_2	Broadway Associates	Washington	WA
a_3	Prolific Consulting Inc.	California	CA

	Table B		
	Company	**City**	**State**
b_1	ABC	New York	NY
b_2	Prolific Consulting	California	CA

A large knowledge graph may contain information about over 10 million companies. It may receive a data feed, that was extracted from natural language text. Such data feeds can contain over 100,000 companies. Even if the knowledge graph had a standardized way to refer to companies, but this new data feed that was extracted from text will not have those standardized identifiers. The task of the record linkage is to relate the companies contained in this new data feed with the companies that already exist in the knowledge graph. As the data volumes are large, performing this task efficiently is of paramount importance.

3.2 An Approach to Solve the Record Linkage Problem

Efficient record linkage involves two steps: *blocking* and *matching*. The blocking step involves a fast computation to select a subset of records from the source and the target that will be considered during a more expensive and precise matching step. In the matching step, we pairwise match the subset of records that were selected during blocking. In the example considered above, we could use a blocking strategy that considers matching only those records that match on the state. With that strategy, we need to consider matching only a_1 with b_1, and a_3 with b_2, thus significantly reducing the comparisons that must be performed.

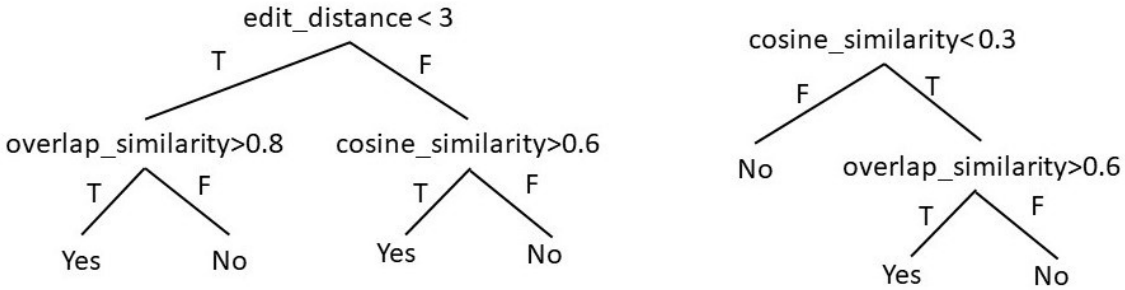
Both blocking and matching steps work by learning a random forest through an active learning process. A random forest is a set of decision rules that gives its final prediction through a majority vote returned by individual rules. Active learning is a learning process that constructs the random forest by actively monitoring its performance on the test data, and selectively choosing new training examples to iteratively improve its performance. We will next explain each of these two steps in a greater detail.

3.3 Random Forests

Blocking rules rely on standard similarity checking functions, such as, exact match, Jaccard similarity, overlap similarity, cosine similarity, etc. For example, if we were to check the overlap similarity between "Prolific Consulting" and "Prolific Consulting Inc.", we will first gather the

tokens in each of them, and then check which tokens are in common, giving us a similarity score of 2/3.

We show below a snippet of a random forest for blocking. A random forest can also be viewed as a set of set of rules. The random forest shown below is a set of two sets of rules. The arguments of each predicate are two values to be compared.



r_1: (edit_distance ≥ 3) and (cosine_similarity > 0.6) $\rightarrow$ match

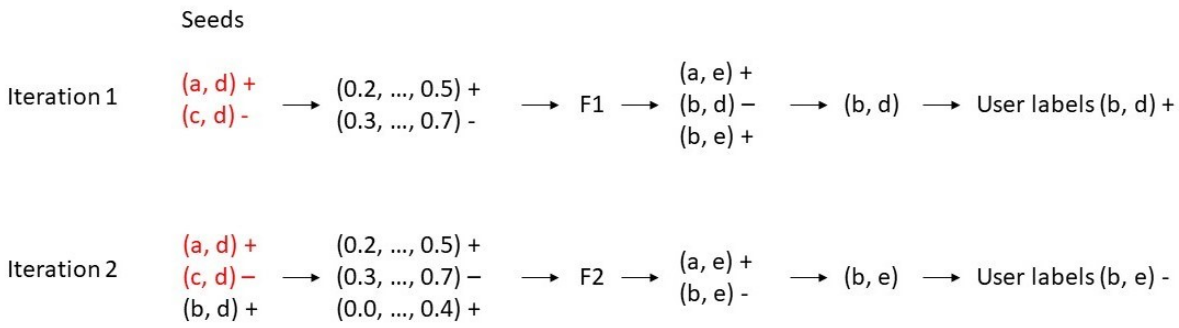
r_2: (edit_distance < 3) and (overlap_similarity > 0.9) $\rightarrow$ match

r_3: (cosine_similarity > 0.6) and (overlap_similarity < 5) $\rightarrow$ match

Several general principles exist for automatically choosing the similarity functions for blocking rules. For example, for a numerical valued attributes such as age, weight, price, etc., candidate similarity functions are: exact match, absolute difference, relative difference, and Levenstein distance. For string valued attributes, it is common to use edit distance, cosine similarity, Jaccard similarity, and TF/IDF functions.

3.4 Active Learning of Random Forests

We can learn the random forest for the blocking rules through the following active learning process. We randomly select a set of pairs from the two datasets. By applying the similarity functions on each pair, we obtain a set of feautres for each pair. Using these features, we learn a random forest. We apply the resulting rules to new pairs selected from the data set, and evaluate their performance. If the performance is below a certain threshold, we repeat the cycle, by providing additional labeled examples until an acceptable performance is obtained. We illustrate this process using the following example.



We assume that our first dataset contains three items: a, b, and c, and our second dataset contains two items, d and e. From this dataset, we pick two pairs (a,d) and (c,d) which are labeled by the

user as similar and not similar respectively. On this pair, we apply the similarity functions each of which results in a feature of the pair. We then use those features to learn a random forest. We apply the learned rules to the tuples which were not in the training set, and ask the user to verify the result. The user informs us that the result for the pair (b,d) is incorrect. We add (b,d) to our training set, and we repeat the process for another iteration. Over several iterations, we anticipate the process to converge, and give us a random forest that we can effectively use in the blocking step.

Once a radom forest has been learned, we present each of the learned rules to the user. Based on the user verification, we choose the rules that will be used in the subsequent steps.

3.5 Applying the Rules

Once we have learned the rules, the next step is to apply them on the actual data. When the data size is huge, it is still not possible to apply the blocking rules to all the pairs of objects. Therefore, we resort to indexing. Suppose, one of the rules requires that the Jaccard index should be greater than 0.7, and we are looking to match the movie *Sound of Music*. As the length of the movie name is 3, we need consider only those movies in our data whose length is between 3*0.7 and 3/0.7, ie, between 2 and 4. If we have indexed the dataset on the size of the movies, it is efficient to choose only those movies whose sizes are between 2 and 4, and filter the set even further through the application of the blocking rule.

3.6 Performing the Matching

The blocking step produces a much reduced set of pairs which must be tested for whether they match. The general structure of the matching process is very similar in that we first define a set of features, learn a random forest, and through the active learning process refine it. The primary difference between the blocking and the matching step is that the matching process needs to be more exact and can require more computation. That is because this is the final step in the record linkage, and we need to have high confidence that the two entities indeed match.

4. Summary

In this chapter, we considered the problem of creating a knowledge graph by integrating the data coming from structured sources. The integrated schema of the knowledge graph can be refined and evolved as per the business requirements. The mappings between the schemas of different sources can be bootstrapped through automated techniques, but they do require verification through human input. Record linkage is the integration of the data at the instance level where we must infer the equality between two instances in the absence of unique identifiers. The general approach for record linkage is to learn a random forest through active learning process. For applications requiring a high accuracy, automatically computed record linkage may eventually need to undergo human verification.

Exercises

Exercise 4.1.Two popular methods to calculate similarity between two strings are edit distance (also known as Levenshtein distance) and the Jaccard measure.

We can define the Levenshtein distance between two strings a, b (of length |a| and |b| respectively), given by lev(a,b) as follows:

- lev(a,b) = a if |b| = 0

- lev(a,b) = b if |a| = 0
- lev(tail(a),tail(b)), if a[0] = b[0]
- 1 + min{lev(tail(a),b), lev(a,tail(b)), lev(tail(a),tail(b))} otherwise.

where the tail of some string x is a string of all but the first character of x, and x[n] is the nth character of the string x, starting with character 0.

We can define the Jaccard measure between two strings a, b as the size of the intersection divided by the size of the union between the two.

$$J(a,b) = |a \cap b| / |a \cup b|$$

(*a*) Given the strings "JP Morgan Chase" and "JPMC Corporation", what is the edit distance between the two?

(*b*) Given the strings "JP Morgan Chase" and "JPMC Corporation", what is the Jaccard measure between the two?

(*c*) Given three strings: x = Apple Corporation, CA, y = IBM Corporation, CA, and z = Apple Corp, which of these strings would be equated by the edit distance methods?

(*d*) Given three strings: x = Apple Corporation, CA, y = IBM Corporation, CA, and z = Apple Corp, which of these strings will be equated by the Jaccard measure?

(*e*) Given three strings: x = Apple Corporation, CA, y = IBM Corporation, CA, and z = Apple Corp, what intuition you would use to ensure that x is equated to z?

Exercise 4.2. A commonly used approach to account for the importance of words is a measure known as TF/IDF. Term frequency (TF) denotes the number of times a term occurs in a document. Inverse Document Frequency (IDF) denotes the number of documents containing a term. The TF/IDF score is calculated by taking the product of TF and IDF. Use your intuition to answer whether the following is true or false.

(*a*) Higher the TF/IDF score of a word, the rarer it is.

(*b*) In a general news corpus, TF/IDF for the word *Apple* is likely to be higher than the TF/IDF for the word *Corporation*

(*c*) Common words such as stop words will have a high TF/IDF.

(*d*) If a document contains words with high TF/IDF, it is likely to be ranked higher by the search engines.

(*e*) The concept of TF/IDF is not limited to words, and can also be applied to sequence of characters, for example, bigrams.

Exercise 4.3. To check if two names might refer to the same real-world organization, one strategy is to check the similarity between two documents that describe them. Cosine similarity is a measure of similarity between two vectors, and is defined as the cosine of the angle between them. Highly similar documents will have a cosine score closer to 1. Which of the following might be a viable approach to convert a document into a vector for calculating cosine similarity?

(*a*) Word embeddings of the words used in a document.

(*b*) TF/IDF scores of the words used in a document.

(*c*) TF/IDF of bigrams used in a document.

(*d*) None of the above.

(*e*) Any of (a), (b) or (c).

Exercise 4.4. Which of the following is true about schema mappings?

(*a*) Schema mapping is largely an automated process.
(*b*) It is usually straightforward to learn simple schema mappings.
(*c*) Complete schema documentation is almost never available.
(*d*) Examining the actual data in the two sources can provide important clues for schema mapping.
(*e*) Database constraints play no role in schema mapping.

Exercise 4.5. Which of the following is true about record linkage?

(*a*) Blocking can be an unnecessary and expensive step in the record linkage pipeline.
(*b*) A random forest is a set of set of rules.
(*c*) Blocking rules must always be authored manually.
(*d*) Active learning of blocking rules minimizes the training data we must provide.
(*e*) Matching rules are as expensive to apply as the blocking rules.

CS520

Knowledge Graphs

What should AI Know ?

How to Create a Knowledge Graph from Text?

1. Introduction

Textual sources such as business and financial news, SEC filings, and websites such as Wall Street Journal, contain information that is of great value for numerous business tasks such as market research, business intelligence, etc. We can use Natural language processing (NLP) to process textual sources, and create knowledge graphs which can support a variety of analytics. NLP, however, is a specialized and sophisticated technology, and our goal here is not to provide a comprehensive and detailed coverage of it. Our goal here is to introduce some basic concepts of NLP that can be useful to those whose primary interest and goal is to create a knowledge graph. Several vendors have created businesses around processing natural language text and delivering structured data to others for consumption.

There are three NLP tasks that are directly relevant to knowledge graph construction: entity extraction, relation extraction, and entity resolution. Entity extraction is the task of identifying key entities of interest (e.g., Organizations, People, Places, etc.) from the text. These entities typically constitute the nodes in a knowledge graph. The relation extraction is the task in which given two entities of interest, and some text, we extract the relations between them (e.g., net sales, management team information, etc.) from the text. Sometimes, the relation extraction is also used to extract properties of a given entity. The extracted relations and properties will typically become relations or node properties in our knowledge graph. Entity resolution is the task of identifying whether multiple mentions in a text refer to the same entity. For example, in a paragraph of text, "John Smith", "He", and "Her father", may all refer to the same entity.

In this chapter, we will give an overview of the techniques used for entity and relation extraction. We omit the entity resolution from our discussion here because it is an advanced topic for the scope of the present volume. Most techniques for entity and relation extraction, that are popular these days, rely on adapting a pre-trained language model for the task at hand. For our purpose, we treat the pre-trained language model and the machine learning techniques as black boxes as both are now available for use as off-the-shelf commodities. This progress in NLP and machine learning allows the knowledge graph creators to focus on the end-product, and on providing suitable training and evaluation data that is required for the adaptation of the language models. We will begin with an overview of the language models, and then describe the entity and relation extraction tasks in greater detail.

2. Overview of Language Models

Language modeling is the task of predicting what word comes next in a text. For example, given a sentence fragment: "students opened their", a language model will predict the next word that can complete this sentence. In this case, the next words might be book, exam, laptop, etc. More formally, given a set of words $x_1, \ldots, x_{n-1}$, a language model predicts the probability $P(x_n \mid x_1, \ldots, x_{n-1})$, where x_n is any word in the vocabulary. Language models are used extensively in auto

completing search requests on the web, in auto correcting words in word processing, etc.

Modern language models are created by training a deep learning model, such as a Recurring Neural Network, on a large corpus of text. Numerous variations of pre-trained language models are available as open source products that can be adapted for the purpose of the specific task at hand. As we discuss the techniques for entity and relation extraction, we will also describe how the language models can be adapted for these tasks.

3. Entity Extraction

We will begin by considering a concrete example of entity extraction, and then give an overview of different approaches to entity extraction, and conclude the section by discussing some challenges in performing well at this task.

3.1 An Example of Entity Extraction

Let us consider the following piece of text from a news story:

> Cecilia Love, 52, a retired police investigator who lives in Massachusetts, said she paid around $370 a ticket with tax for nonstop United Airlines flights to Sacramento from Boston for her niece's high school graduation in June, 2020.

We will consider the *named* entities in the above text. A named named entity is anything that can be referred to with a proper name: a person, a location, an organization, etc. The definition of a named entity is commonly extended to include things that are not entities per se, including dates, times, and other kinds of expressions involving time, and even numerical expressions, for example, prices. Here is the above text with the named entities marked:

> [$_{\text{PER}}$ Cecilia Love], 52, a retired police investigator who lives in [$_{\text{LOC}}$ New Jersey], said she paid around [$_{\text{MONEY}}$ $370] a ticket with tax for nonstop [$_{\text{ORG}}$ United Airlines] flight to [$_{\text{LOC}}$ Sacramento] from [$_{\text{LOC}}$ Boston] for her niece's high school graduation in [$_{\text{TIME}}$ June, 2020].

The paragraph contains seven named entities, one of which is a person (indicated by PER), three are locations (indicated by LOC), one is money (indicated by MONEY), one is an organization (indicated by ORG), and one is a time (indicated by TIME). Depending on the domain of application, we may introduce more or less named entity types. For example, in the task of identifying key terms in a text, there is only one entity type that captures a key term.

Entity extraction task is useful in many different applications. For example, in question answering, it may help pull out answers from a retrieved passage of text. In a word processing application, it could help automatically connect entities appearing in the text with additional information (e.g., definition, facts, etc.) about those entities.

3.2 Approaches to Entity Extraction

First and foremost, we can view entity extraction as a labeling problem. We associate a label with each word, and the task becomes to predict the label. We can perform entity extraction by three broad approaches: sequence labeling, deep learning models, and rule-based approaches. We will briefly introduce each of these approaches.

To facilitate the labeling, we introduce a labeling scheme that is known as BIOES in which the meaning of different tags is as follows: B stands for the beginning of an entity, I stands for the interior of an entity, O stands for a word that is not part of an entity, E stands for the end of an entity, and S stands for a single word entity. As an example, the words in the text snippet shown above will be labeled as shown below.

Cecilia	B	Love	E	,	O	52	O	,	O
a	O	retired	O	police	O	investigator	O	who	O
lives	O	in	O	Massachusetts	S	,	O	said	O
she	O	paid	O	around	O	$370	S	a	O
ticket	O	with	O	tax	O	for	O	nonstop	O
United	B	Airlines	E	flights	O	to	O	Sacramento	S
from	O	Boston	S	for	O	her	O	niece's	O
high	O	school	O	graduation	O	in	O	June	B
,	I	2020	E						

In the sequence labeling approach, we train one of the machine learning algorithm (for example, Conditional Random Fields), using features such as: part of speech, presence of the word in a list of standard words, word embeddings, word base form, whether word contains a prefix or suffix, whether it is in all caps, etc. Significant effort is needed in feature engineering, because the performance of a particular choice of features and the machine learning algorithm can vary by the domain.

In a deep learning approach, there is no feature engineering, and we simply input word embeddings to a language model. Instead of predicting the next word, the language model now predicts one of the five tags (B, I, O, E, S) tags that are required for entity recognition. To adapt the language model to this new task, we first pre-train it using the corpus for that domain, and then train it for the task at hand. In the task-specific training of the language model, we provide the training by adding a distinguished token [CLS] that denotes the beginning of an entity, and a second distinguished token [SEP] that denotes the end of an entity. This training allows the model to predict these distinguished tags in response to a text input. Such predictions are enough for us to produce one of the five required tags for each word.

Finally, in a rule-based approach, one specifies labeling rules in a formal query language. The rules can include regular expressions, references to dictionaries, semantic constraints, and may also invoke automated extractors and reference table structures. The rules may also invoke machine learning modules for specific tasks. Rule application can be sequenced in a way that we first use high precision rules, followed by lookup in standard name list, followed by language-based heuristics, and when all else fails, resort to probabilistic machine learning techniques.

3.3 Challenges in Entity Extraction

Even though for specific tasks, entity extractors may exhibit precision and recall above 90%, but obtaining good performance across all the domains can be challenging. In this section, we consider a few challenges that are faced during entity extraction.

While labeling entities with their classes, there are numerous cases of ambiguity. For example, given the entity *Louis Vuitton*, it can refer to either a person, or an organization, or a commercial product. Resolving such ambiguities is not possible unless considerable context is taken into

account.

For using a machine learning model, we require tremendous amount of data. In practice, either the data are not available, or they are largely incomplete. When we train our model using incomplete data, it seriously affects the performance.

One variation of the entity extraction task is to identify key phrases in the text. As the key phrases are not limited to a few classes, the task of identifying the specific class corresponding to a key phrase can become even more challenging. Sometimes, the key phrases are overly complex (e.g., duplication of a cell by fission), and sometimes, too general (e.g., Attach) making it very challenging to apply a uniform technique across the board.

Entities can appear in many different surface forms, for example, synonyms, acronyms, plurals, and morphological variations. In general, entity extraction should be aware of the lexical knowledge which usually does not exist when entering a new domain. As a result, for improving the performance of entity extraction, lexicon extraction becomes an essential related task.

4. Relation Extraction

We will begin by considering several concrete examples of relation extraction, then give an overview of different approaches to relation extraction, and conclude the section by discussing some challenges in performing well at this task.

4.1 Examples of Relation Extraction

Considering the text snippet from the previous section, we can extract relations such as Cecilia Love *lives in* Massachusetts, United Airlines *flies from* Boston, and United Airlines *flies to* Sacramento, etc. In a typical relation extraction task, entities have been previously identified, and in this sense, it extends the entity extraction task. The relations to be extracted are also specified ahead of time.

A popular instance of relation extraction task is to extract information from Wikipedia Infoboxes. The obvious application of this task is to improve the search results over the internet. Wikipedia infoboxes define relationships such as *preceded by*, *succeeded by*, *children*, *spouse*, etc. Achieving a high accuracy on this task can be challenging because of numerous corner cases. For example, Larry King has been married multiple times, and therefore, the extractor must be able to take that into account the time duration for which the marriage existed.

There also exist domain-specific relationships. For example, the Unified Medical Language Systems supports relationships such as *causes*, *treats*, *disrupts*, etc. Apart from standard relationships such as *subclass-of*, and *has_part*, the relationships to be extracted are domain-specific and usually require some up front design work. There are some approaches to relation extraction that do not require choosing relationships ahead of time, but the usefulness of such approaches in practice has been found to be limited.

4.2 Approaches to Relation Extraction

There are three broad approaches to relation extraction: syntactic patterns, various forms of supervised machine learning, and unsupervised machine learning. As noted in the previous section, the unsupervised machine learning has limited use in practice. Therefore, we will primarily consider the use of syntactic patterns and supervised machine learning for relation extraction.

A classical approach to extract relations relies on syntactic patterns known as Hearst Patterns. For example, consider the following sentence.:

> The bow lute, such as the Bambara ndang, is plucked and has an individual curved neck for each string.

Even though we may have never heard of Bambara ndang, we can still infer that it is a kind of bow lute. More generally, we can identify syntactic patterns, that are strong indicators of the *subclass of* relationship. The following five syntactic patterns have been around for quite some time, and have been found extremely effective in practice.

Pattern Name	Example
such as	... works by authors *such as* Herric, Goldsmith, and Shakespear ...
or other	Bruises, wounds, broken bones, *or other* injuries ...
and other	... temples, treasuries, *and other* Civic Buildings, ...
including	All common law countries *including* Canada and England ...
especially	Most European countries *especially* France, England, and Spain, ...

We can discover new syntactic patterns for any relationship of choice as follows. We first collect examples for which the relationship is known to hold true. We then look for sentences where the relationship holds true. By identifying commonalities across such sentences, we can define a new pattern. We can then test such patterns against the corpus. One such algorithm is known as Dual Iterative Pattern Relation Expansion (DIPRE). We illustrate its working on the problem of extracting the (author, title) pairs from a corpus. We begin with a small set set of known (author, title) pairs, and we find all their occurrences in the corpus, and from those we generate more patterns. The algorithm continues recursively by using the new patterns to discover more books, and from their discovering new patterns. As a concrete example, given the seed pair of (William Shakespear, The Comedy of Errors), and the following sentences,

- The Comedy of Errors, by William Shakespeare, was ...
- The Comedy of Errors, by William Shakespeare, is ...
- The Comedy of Errors, one of William Shakespeare's earliest attempts ...
- The Comedy of Errors, one of William Shakespeare's most ...

we can derive the following patterns:

- ?x , by ?y,
- ?x , one of ?y‘s

Using the newly derived patterns, the extraction process continues recursively.

The supervised approaches to relation extraction require extensive training data. Whenever such training data is available, many of the off-the-shelf learning algorithms could be trained. But, in the absence of adequate data, weak supervision approaches are becoming popular to arrive at the required data. The basic idea in weak supervision is to write several approximate labeling functions that can generate the training data automatically. These weak labels are then combined using a probabilistic function.

As an example of a weak labeling function, consider the *has part* relation. For this relation, it has not been possible to develop the syntactic patterns of the sort suggested above. A possible weak

labeling function for this relation is to first produce a dependency parse of the sentence, and then look for two nodes in the parse that have a path of length one that contains the verbs has or have. For taxonomic relationships, an additional weak labeling function is that if two entities end with the same base word but one has an additional modifier in front of it, this suggests a taxonomic relation (e.g., eukaryotic cell SUBCLASS cell).

To adapt a language model to the task of relation extraction, we change the input representation of a sentence so that each of the individual terms are explicity marked. For example, we input a sentence such as ["All", "[TERM1-START]", "cells", "[TERM1-END]", "have", "a", "[TERM2-START]", "cell" "membrane", "[TERM2-END]", "."], and expect the model to output the probability distribution over different relationships that might exist between the two terms. The predicted relationship depends on the input training data.

4.3 Challenges in Relation Extraction

The primary challenge in relation extraction is to obtain the necessary training data. The weak supervision approach is quite promising because the training data need not be perfect. One can resort to sources such as Wikidata and Wordnet as a source of data for defining weak labeling functions. Developing new approaches for weak labeling functions is a topic of current and ongoing research.

We also need a good workflow to validate the results of the extractors. Quite often the validation can be done through crowdsourcing. One can also prioritize the validation of those extracted relations where the confidence is low. Developing such active learning loops is another important and current research challenge.

5. Summary

In this chapter, we considered the problem of creating a knowledge graph by processing text. We addressed two fundamental problems of entity extraction and relation extraction. For both of these tasks, the early work focused on defining manual and syntactic rules for extraction. More recent popular approaches rely on adapting pre-trained language models, and customizing them to the specific corpus at hand.

For both entity and relation extraction, the most prevalent current approach is to adapt a language models that has been previously created using deep learning. Existing approaches that are syntactic or rule-based play an important role in bootstrapping the training data required for the deep learning models. Validating the output of extraction remains an ongoing challenge.

The problem of entity linking or entity resolution is an equally important problem for creating knowledge graphs, but we have chosen to omit it from the discussion here for two reasons. First, we believe that the challenge of getting good performance on entity and relation extraction, by itself, is a significant hurdle. Second, a prerequisite to a good performance on entity linking is the availability of a good lexicon. For these reasons, entity resolution is an advanced technique, and it may or may not be the primary bottleneck in solving the business problem for which we are creating the knowledge graph.

Exercises

Exercise 5.1. Using the concept of a language model on the following sentence corpus, answer the questions below:

- I love running.
- Good health can be achieved by running.
- I love good health

(*a*) What is P(health|good)?
(*b*) What is P(running|love)?
(*c*) What is P(love|I)?
(*d*) What is (good|love)?
(*e*) What is P(love|running)?

Exercise 5.2. An important feature used for entity extraction is **Word shape**: it represents the abstract letter pattern of the word by mapping lower-case letters to 'x', upper-case to 'X', numbers to 'd', and retaining punctuation. Thus for example C.I.A. would map to X.X.X. and IRS-1040 would map to XXX-dddd. In a shorter-version of word shape, consecutive character types are removed. For example, C.I.A. would still map to X.X.X, but IRS-1040 would map to X-d. With these definitions, address the following questions.

(*a*) What is the shape of the word: Googenheim?
(*b*) What is the short-shape of the word: Googenheim?
(*c*) What is the regular expression for the shape of the word Googenheim?
(*d*) What is the regular expression for the short-shape of the word Googenheim?
(*e*) Is it true that the short-shape is always strictly smaller than the regular shape of a word?

Exercise 5.3. Which of the following may not be a good feature for learning entity extraction?

(*a*) word shape
(*b*) part of speech
(*c*) presence in Gazeteer
(*d*) presence in Wikipedia
(*e*) number of characters

Exercise 5.4. Given the following sentence corpus, and the seed (Sacramento,California), what patterns will be extracted by the DIPRE algorithm?

- The bill was signed in Sacramento, California.
- Sacramento is the capital of California.
- Sacramento is the capital of California, and its sixth largest city.
- California's Sacramento is home to the state legislature, but not the state supreme court.
- California Governor Jerry Brown signed the bill in Sacramento.

(*a*) in ?x, ?y
(*b*) ?x is the capitol of ?y
(*c*) ?y's ?x
(*d*) ?x's ?y
(*e*) ?y Governor * in ?x

Exercise 5.5. Which of the following would be a candidate for a weak labeling function to extract

the parthood relationship between entities, i.e., an entity A has part entity B.

(*a*) ?xs have ?y
(*b*) ?x includes ?y
(*c*) ?x contains ?y
(*d*) ?y surrounds ?x
(*e*) ?x causes ?y

CS520

Knowledge Graphs

What should AI Know ?

6. What are some Knowledge Graph Inference Algorithms?

1. Introduction

Once we have created the knowledge graph, we are interested in retrieving information, and using that information to make new conclusions. We have already introduced query languages using which we can perform retrieval operations on the graph. In this chapter, we will focus on inference algorithms, that go beyond retrievals, i.e., conclude new facts from the knowledge graph that are not explicitly present in it. Inference algorithms can be invoked through the declarative query interface.

We will consider two broad classes of inference algorithms: graph algorithms, and ontology-based algorithms. The graph algorithms are applicable to any graph-structured data, and support operations such as finding minimum paths between nodes in a graph, identifying salient nodes in a graph, etc. The ontology-based algorithms operate on the structure of the graph but take its semantics into account, for example, traversing specific paths, or concluding new connections based on background domain knowledge. In this chapter, we will consider both classes of these algorithms.

In principle, we could invoke graph algorithms, and the ontology-based inference through a declarative query interface of the sorts we have previously considered. Ontology-based inference may leverage graph-based algorithms. For example, checking if an object A is an instance of a class B, could be done by checking whether a path exists in the class graph between C and B, where C is the immediate type of A.

2. Graph-based Inference Algorithms

We will consider three broad classes of graph algorithms: path finding, centrality detection, and community detection. Path finding involves finding a path between two or more nodes in a graph that satisfies certain properties. Centrality detection is about understanding which nodes are important in a graph. Different methods used to define the meaning of importance lead to many variations in the centrality detection algorithms. Finally, community detection is about identifying a group of nodes in a graph that satisfies some criteria of being in a community. Community detection is useful for studying emergent behaviors in graphs that may otherwise not be noticed.

We will consider each of these categories of graph algorithms in more detail. For each category, we will present an overview, discuss different ways it is useful, and consider some sample algorithms.

2.1 Path Finding Algorithms

There are several variations of the problem of finding paths in a graph: finding the shortest path between any given two nodes, or finding the shortest paths between all pairs of nodes, finding a minimum spanning tree, etc. The shortest path between any given two nodes can be used in

planning an optimal route in a traffic navigation system. A minimum spanning tree calculates the least cost for visiting all the nodes in a set of nodes, and can be useful in problems such as trip planning.

As an example of a specific shortest path algorithm, we will consider the A* algorithm which is a generalization of the classical Dijkastra's algorithm. A* algorithm is also widely used as a search algorithm for solving AI Planning problems.

The A* algorithm operates by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each step, it determines which of the paths to extend based on the cost of the path until now and the estimate of the cost required to extend the path all the way to the goal. If *n* is the next node to visit, *g(n)* the cost until now, and *h(n)* the estimate of the cost required to extend the path all the way to the goal, then it choose the node that minimizes *f(n)*, where *f(n)=g(n)+h(n)*.

We must choose an *admissible* heuristic such that it never over-estimates the cost of arriving at the goal. In one possible variation, known as the best first search, the heuristic chooses the path with the least overall cost until now, i.e., it sets *h(n)=0*. There exists an extensive literature on the different heuristics that can be used in the A* search.

2.2 Centrality Detection Algorithms

The centrality detection algorithms are used to better understand the roles played by different nodes in the overall graph. This analysis can help us understand the most important nodes in a graph, the dynamics of a group and possible bridges between groups.

There are several variations of the centrality detection algorithm: degree centrality, closeness centrality, between-ness centrality, and page rank. Degree centrality simply measures the number of incoming and/or outgoing edges. A node with a very high outgoing degree in a supply chain network suggests a supplier monopoly. A closeness centrality identifies nodes that have shortest paths to all other nodes. Such information can be useful in identifying the location of a new service so that it is most accessible to the widest range of customers. Between-ness centrality identifies a node based on the number of shortest paths between nodes that pass through it. Between-ness centrality is a measure of the sphere of influence exercised by the nodes in the network. Finally, the PageRank algorithm measures the importance of a node based on other nodes it is recursively connected to.

For our discussion here, we will consider the PageRank algorithm in more detail. The Page rank was originally developed for ranking pages for WWW search. It is able to measure the transitive influence on a node. For example, a node connected to a few very important nodes could be more important than the node connected to a large number of unimportant nodes. We can define the PageRank of a node as follows.

$$PR(u) = (1-d) + d * (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))$$

In the above formula, we assume that the node *u* has incoming edges from nodes *T1,...,Tn*. We use *d* as a damping factor which is usually set at 0.85. *C(T1),...,C(Tn)* is the number of outgoing edges of nodes *T1,...,Tn*. The algorithm operates iteratively by first setting the PageRank for all the nodes to the same value, and then iteratively improving it for a fixed number of iterations, or until the values converge.

Beyond its original use in ranking search results for WWW queries, the PageRank has found many

other interesting uses. For example, it is used on social media sites to recommend who should a particular user follow. It has also been used in fraud analysis to identify highly unusual activity associated with nodes in a graph.

2.3 Community Detection Algorithms

The general principle underlying the community detection algorithms is that the nodes in a community have more relationships within the community than to nodes outside the community. Sometimes, the community analysis can be the first step in analyzing a graph so that a more in-depth analysis could be undertaken for nodes within the community.

There are several flavors of community detection algorithms: connected components, strongly connected components, label propagation and fast unfolding. The first two of these algorithms, connected components, and strongly connected components are frequently used in the initial analysis of a graph. Connected components algorithm and strongly connected components algorithm are standard techniques of graph theory. A connected component is a set of nodes such that there is a path between any two nodes in the underlying undirected graph. A strongly connected component is a set of nodes such that for any given nodes A and B in the set, there is a directed path from node A to node B, and path from node B to node A. Both label propagation and fast unfolding are bottom up algorithms for identifying communities in large graphs. We consider both of these algorithms in more detail.

Label propagation begins by assigning each node in the graph to a different community. We then arrange the nodes in a random order to update their community as follows. We examine the nodes in the assigned order, and for each node, we examine its neighbors, and set its community to the community shared by a majority of its neighbors. The ties are broken in a unform random manner. The algorithm terminates when each node is assigned to a community that is shared by a majority of its neighbors.

In the fast unfolding algorithm, there are two phases. We initialize each node to be in a separate community. In the first phase, we examine each node and each of its neighbors and evaluate if there would be any overall gain in modularity in placing this node in the same community as a neighbor. A suitable measure to calculate modularity is defined. If there will be no gain, the node is left in its original community. In the second phase of the algorithm, we create a new network in which there is a node corresponding to each community from Phase 1, and an edge between the two nodes if there was an edge between some nodes in their corresponding phase 1 communities. Links between the nodes of the same community in phase 1 lead to self-loops for the node corresponding to their community in Phase 2. Once Phase 2 is completed, the algorithm repeats by applying phase 1 to the resulting graph.

One example of a modularity function used in the above algorithm is shown below.

$$Q = \sum_{i=1}^{k} \left[\frac{e_i}{m} - \left(\frac{d_i}{2m} \right)^2 \right]$$

In the above formula, we calculate the overall modularity score Q of a network that has been divided into k communities where e_i and d_i are respectively the number of nodes, and total degree of nodes in community i, and m is the total number of edges in the network.

Both label propagation and fast unfolding algorithms reveal emergent and potentially unanticipated communities. Different executions may also lead to identification of different communities.

3. Ontology-based Inference Algorithms

Ontology-based inference distinguishes a knowledge graph system from a general graph-based system. We will categorize the ontology-based inference into two categories: taxonomic inference and rule-based inference. Taxonomic inference primarily relies on the hierarchy of classes and instances and inheritance of values across the hierarchy. Rule-based inference can involve general logical rules. We can access ontological inference through a declarative query interface, and thus, it can be used as a specialized reasoning service for a certain class of queries.

3.1 Taxonomic Reasoning

Taxonomic reasoning is applicable in situations where it is useful to organize knowledge into classes. Classes are nothing but unary relations. We will consider the concepts of class membership, class specialization, disjoint classes, value restriction, inheritance and various inferences that can be drawn using them.

Both property graph and RDF data models support classes. For property graphs, the node types are equivalent to classes. For RDF, there is an extension called RDF schema that supports the definition of classes. In more advanced extensions of RDF such as Web Ontology Language and Semantic Web Rule Language, a full-fledged support is available for defining classes and rules.

To discuss taxonomic reasoning, we have chosen to abstract away from property graph and RDF data models. We will introduce the basic concepts of taxonomies such as class membership, disjointness, constraints and inheritance using Datalog as a specification language.

3.1.1 Class Membership

Suppose we wish to model data about kinship. We can define the unary relations of `male` and `female` as classes, that have members `art`, `bob`, `bea`, `coe`, etc. The member of a class is referred to as an instance of that class. For example, `art` is an instance of the class `male`.

For every unary predicate that we also wish to refer to as a class, we introduce an object constant with the same name as the name of the relation constant as follows.

```
class(male)                    class(female)
```

Thus, `male` is both a relation constant, and an object constant. This is an example use of *metaknowledge*, and is also sometimes known as *punning*.

To represent that `art` is an instance of the class `male`, we introduce a relation called `instance_of` and use it as shown below.

```
instance_of(art,male)          instance_of(bea,female)
instance_of(bob,male)          instance_of(coe,female)
instance_of(cal,male)          instance_of(cory,female)
instance_of(cam,male)
```

3.1.2 Class Specialization

Classes can be organized into a hierarchy. For example, we can introduce a class `person`. Both `male` and `female` are then *subclasses* of `person`.

```
subclass_of(male,person)          subclass_of(female,person)
```

The `subclass_of` relationship is transitive, ie, if `A` is a subclass of `B`, and `B` is a subclass of `C`, then `A` is a subclass of `C`. For example, if `mother` is a subclass of `female`, then `mother` is also a subclass of `person`

```
subclass_of(A,C) :- subclass_of(A,B) & subclass_of(B,C)
```

The `subclass_of` and `instance_of` relationships are related in that if `A` is a subclass of `B`, then all instances of `A` are also the instances of `B`. In our example, all instances of `male` are also all the instances of `person`

```
subclass_of(I,B) :- subclass_of(A,B) & instance_of(I,A)
```

A class hierarchy must not contain cycles, because that would imply that a class is a subclass of itself, which is semantically incorrect.

3.1.3 Class Disjointness

We say that a class `A` is *disjoint* from another class `B` if no instance of one can be an instance of another. We can declare two classes to be disjoint from each other or a set of classes to be a partition such that each class in the set is pairwise disjoint from every other class. In our kinship example, the classes `male` and `female` are disjoint from each other.

```
~instance_of(I,B) :- disjoint(A,B) & instance_of(I,A)
~instance_of(I,A) :- disjoint(A,B) & instance_of(I,B)
disjoint(A1,A2) :- partition(A1,...,An)
disjoint(A2,A3) :- partition(A1,...,An)
disjoint(An-1,An) :- partition(A1,...,An)
```

3.1.4 Class Definition

Classes are defined using *necessary* and *sufficient* relation values. For example, `age` is a necessary relation value for a `person`. If we were to define the class of a brown-haired person, it is necessary and sufficient for a person to have brown hair to be an instance of this class.

```
instance_of(X,brown_haired_person) :-
          instance_of(X,person) & has_hair_color(X,brown)
```

The classes that have only necessary relation values are known as *primitive* classes and the classes for which we know both necessary and sufficient relation values are known as *defined* classes. The sufficient definition of a class has `instance_of` literal in its head.

3.1.5 Value Restriction

We can restrict the arguments of a relation to be instances of specific classes. In the kinship

example, we can restrict the `parent` relationship so that its arguments are always instances of the class `person`. Thus, if the reasoner is ever asked to prove `parent(table,chair)`, it can conclude that it is not true simply by noticing that neither `table` nor `chair` is an instance of `person`. The restriction on the first argument of a relation is usually referred as a `domain` restriction and the restriction on the second argument of a relation is referred to as a `range` restriction. Similar restrictions can be defined for higher arity relationships.

```
illegal :- domain(parent,person) & parent(X,Y) &
~instance_of(X,person)
illegal :- range(parent,person) & parent(X,Y) & ~instance_of(Y,person)
```

3.1.6 Cardinality and Number Consraints

We can further restrict the values of relations by specifying cardinality and number constraints. A cardinality constraint restricts the number of values of a relation, and the numeric constraint specifies the range of numeric values that a relation may take. For example, we may state that a person has exactly two parents, and that the age of a person is between 0 and 100 years.

```
                                        >>
illegal :- instance_of(X,person) & ~countofall(P,parent(P,X),2)
illegal :- instance_of(X,person) & age(X,Y) & min(0,Y,Y)
illegal :- instance_of(X,person) & age(X,Y)& min(100,Y,100)
```

3.1.6 Inheritance

The relation values of a class are said to inherit to its instances. For example, if we assert that `art` is an instance of `brown_haired_person`, we can conclude that `has_hair_color(art,brown)`. In general, an object can be an instance of multiple classes. In case of multiple inheritance, the values inherited from different superclasses can conflict and cause constraint violation. For example, if `art` is an instance of the class `brown_haired_person`, and a class `bald_person` with a constraint that the person has no hair, we will have constraint violation. In case of constraint violation, either the value causing the violation must be rejected, or techniques for para-consistent reasoning must be used to manage such inconsistency.

3.1.7 Reasoning with Classes

There are four broad classes of inference that are interesting with classes.

1. Given two classes `A` and `B`, whether `A` is a subclass of `B`?
2. Given a class `A` and an instance `I`, whether `I` is an instance of `I`?
3. Given a ground relation atom determine whether it is true or false.
4. Given a relation atom, determine different values of variables for which it is true.

The first two inferences are equivalent to computing the views on `subclass_of` and `instance_of` relations. They can also be implemented as path finding algorithms on the graph defined by the classes and their instances. The last two inferences are equivalent to the view on the relation atom of interest.

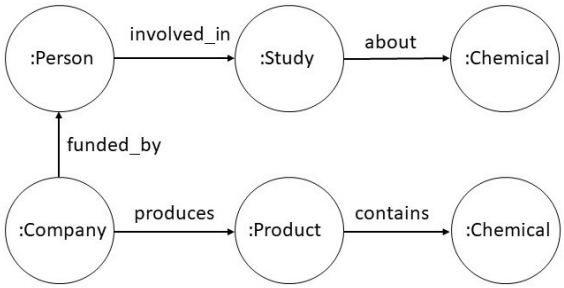
3.2 Rule-based Reasoning

It is not possible to draw a strict line between rule-based reasoning and taxonomic reasoning. Even

though we used Datalog as a specification language for taxonomic reasoning, but it is possible to implement many of the desired inferences in a rule engine. In this section, we will consider an example of rule-based reasoning that leverages an advanced form of rules known as existential rules. We will begin with an example knowledge graph that requires such reasoning, and then we will consider rule-based reasoning algorithms to perform the required reasoning.

3.2.1 Example Scenario Requiring Rule-based Reasoning

Consider a property graph with the schema shown below. Companies produce products that contain chemicals. People are involved in studies about chemicals, and they can be funded by companies.



Given the above property graph, we are interested in determining if a person might have a conflict of interest in being involved in a study. We can define the conflict relationship using the following Datalog rule.

```
coi(X,Y,Z) :- involved_in(X,Y) & about(Y,P) & funded_by(X,Z) &
has_interest(Y,P)
has_interest(X,Z) :- produces(X,Y) & contains(Y,Z)
```

The relation *has_interest* was not in the property graph schema introduced above. But, with the help of its definition using a rule, a rule engine can calculate the conflict of interest relation *coi*. In some cases, we may be interested in adding the computed values of the *coi* relationship to our knowledge graph. As *coi* is a ternary relation, we will need to reify it. As reification requires adding new objects in the graph, we can specify it using an existential rule as shown below.

```
∃c conflict_of(c,X) & conflict_reason(c,Y) & conflict_with(c,Z) :-
involved_in(X,Y) & about(Y,P) & funded_by(X,Z) & has_interest(Y,P)
```

In general, existential rules are needed whenever we need to create new objects in our knowledge graph. Relationship reification is an obvious such situation. Sometimes, we may need to create new objects to satisfy certain constraints. For example, consider the constraint: every person must have two parents. For a given person, the parents may not be known, and if we want our knowledge graph to remain consistent with this constraint, we must introduce two new objects representing the parents of a person. As this can lead to infinite number of new objects, it is typical to set a limit on how the new objects are created.

3.2.2 Approach for Rule-based Reasoning

To support rule-based reasoning on a knowledge graph, one usually interfaces a rule engine with the data in the knowledge graph. We consider here a few different reasoning strategies used by the rule engines.

In a bottom up reasoning strategy, also known as Chase, we apply all the rules against the knowledge graph, and add new facts to it until we can no longer derive new facts. As noted in the previous section, we need to put in place aggressive termination strategies to deal with situations where additional reasoning offers no additional insight. Once we have computed the Chase, the reasoning can proceed using traditional query processing methods.

In top down query processing, we begin from the query to be answered, and apply the rules on as needed basis. A top down strategy requires a tighter interaction between the query engine of the knowledge graph with the rule evaluation. This approach, however, can use lot less space as compared to the bottom up reasoning strategy.

Highly efficient and scalable rule engines use query optimization and rewriting techniques. They also rely on caching strategies to achieve efficient execution.

4. Summary

In this chapter, we considered different inference algorithms for knowledge graphs. Graph algorithms such as path finding, community detection, etc. are supported by most practical graph engines. Graph engines often provide limited support for ontology and rule-based reasoning. Knowledge graph engines are now starting to become available that support both general-purpsose graph algorithms as well as ontology and rule-based reasoning.

Exercises

Exercise 6.1. For the tiled puzzle problem shown below, we can define two different admissible heuristics: Hamming distance and Manhanttan distance. The Hamming distance is the total number of misplaced tiles. The Manhattan distance is the sum of the distance of each tile from its desired position. Answer the questions below assuming that in the goal state, the bottom right corner will be empty.

1	2	3
6	5	7
4	8	

(*a*) What is the Manhattan distance for the configuration shown above?

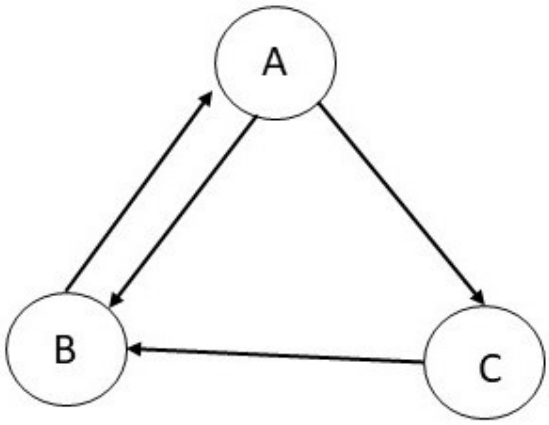
(*b*) What is the Hamming distance for the configuration shown above?

(*c*) If your algorithm was using the Manhattan distance as a heuristic, what would be its next move?

(*d*) If your algorithm was using the Hamming distance as a heuristic, what would be its next move?

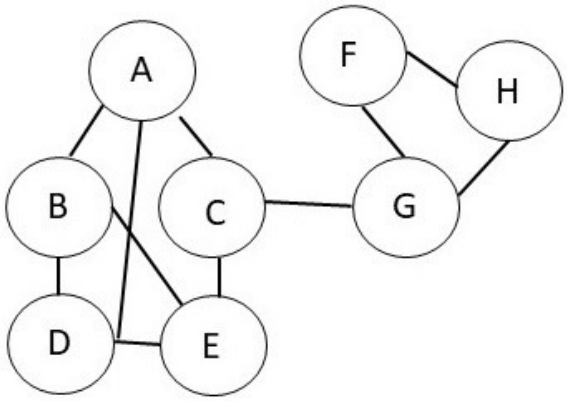
Exercise 6.2. Consider a graph with three nodes and the edges as shown in the table below for which we will go through first few steps of calculating the Page rank. We assume that the damping factor is set to 1.0. We initialize the process by setting the Page rank of each node to 0.33. In the first iteration, as A has only one incoming edge from B with a score of 0.33, and B has only one outgoing edge, the score of A remains 0.33. B has two incoming edges --- the first incoming edge from A (with a score of 0.33 which will be divided into 0.17 for each of the outgoing edge from

A), and the second incoming edge from C (with a weight of 0.33 as C has only one outgoing edge). Therefore, B now has a Page rank of 0.5. C has one incoming edge from A (with a score of 0.33 which will be divided into 0.17 for each of the outgoing edge from A), and therefore, the Page rank of C is 0.17. Follow this process to calculate their ranks at the end of iteration 2.



(*a*) What is the Page rank of A at the end of second iteration?
(*b*) What is the Page rank of B at the end of second iteration?
(*c*) What is the Page rank of C at the end of second iteration?

Exercise 6.3. For the graph shown below, calculate the overall modularity score for different choices of communities.



(*a*) What is the overall modularity score if all the nodes were in the same community?
(*b*) What is the overall modularity score if we have two communities as follows. The first community contains A, B and D. The second community contains the rest of the nodes.
(*c*) What is the overall modularity score if we have two communities as follows. The first community contains A, B, C, D and E. The second community contains the rest of the nodes.
(*d*) What is the overall modularity score if we have two communities as follows. The first community contains C, E, G and H. The second community contains the rest of the nodes.

Exercise 6.4. Considering the statements below, state whether each of the following sentence is true or false.

```
subclass_of(B,A)        subclass_of(C,A)        subclass_of(D,B)
```

```
subclass_of(E,B)        subclass_of(F,C)        subclass_of(G,C)
subclass_of(H,C)        disjoint(B,C)           partition(F,G,H)
```

(*a*) `disjoint(D,E)`

(*b*) `disjoint(B,G)`

(*c*) `disjoint(F,G)`

(*d*) `disjoint(E,C)`

(*e*) `disjoint(A,F)`

Exercise 6.5. Consider the following statements.

1. George is a Marine.
2. George is a chaplain.
3. Marine is a beer drinker.
4. A chaplain is not a beer drinker.
5. A beer drinker is overweight.
6. A Marine is not overweight.

(*a*) Which statement must you disallow to maintain consistency?

(*b*) What are the alternative set of statements that might be true?

(*c*) What conclusions you can always draw regardless of which statement you disallow?

(*d*) What conclusion you can draw only some of the times?

CS520

Knowledge Graphs

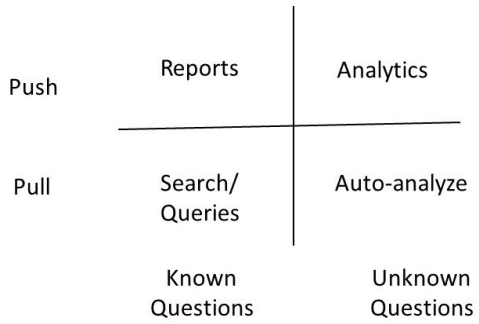
What should AI Know ?

7. How do Users Interact with a Knowledge Graph?

1. Introduction

In previous chapters, we have discussed the design of a knowledge graph, different methods for creating it, and techniques for doing inference with it. We will now turn our attention to how users interact with a knowledge graph. To some degree, the design of interaction begins with the design of a knowledge graph schema as the schema is meant to be easily understood. In fact, a key advantage of knowledge graphs is that the conceptual view of a domain expressed in the schema is also the basis for its implementation. In this chapter, we will consider interaction techniques once the knowledge graph schema has been populated with instances.

The purpose of a knowledge graph is to answer a user's questions. Some of the questions may be known upfront, while some questions users may never think of themselves. The interaction of a user with the knowledge graph could be real-time, or a batch process might be run to produce certain reports to answer some predefined questions and to produce certain analytics. This results in a matrix of four different modes of interaction interactions with the knowledge graph along the dimensions of whether the interaction is initiated by the user (ie, Pull), or in response to information presented to the user (ie, Push), and whether the questions are known in advance vs questions are not known in advance.



Above modes of interaction are usually supported through a combination of search, query answering, and graphical interfaces. A search interface is like the interface of a search engine where the user may simply type keywords. The query interfaces range from a formal query language to a natural language interface. A graphical interface may be used for composing a query, for viewing the results of a query or for browsing the graph defined by the instances in the knowledge graph.

The actual interface to a knowledge graph will typically use a combination of methods. For example, a query might be composed through a combination of search and structured query interface. Similarly, the results may be partly graphical, and partly textual. In this section, we will consider graphical visualization of knowledge, structured query interface, and a natural language query interface. Formal query languages such as Cypher and SPARQL have already been covered

in the previous chapters.

2. Visualization of a Knowledge Graph

It is too common for us to see graphical visualizations of knowledge graphs containing thousands of nodes and edges on a screen. Many times, the graphical visualization is simply a backdrop for the points to be made in contrast to driving and contributing to the insights that help us identify what points to make. Just because we are working with a knowledge graph, it should not automatically imply that a graphical visualization is the best way to interact with it. One should turn to the best principles for visualization design, and choose the most effective medium for presenting the information. Consequently, we begin this section by summarizing the key principles for visualization design, and then outline a few best practices for graphically visualizing knowledge graphs.

2.1 General Principles for Visualization Design

The overall purpose for visualizing a knowledge graph is to come up with a representation of the data to significantly amplify its understanding by the end-user. The improvement in user understanding results from the effective use of the following elements. First, the visualization presents more information in a display than the user might be able to remember at one time. Second, it takes away the burden from the user for having to look for important pieces of information. Third, by placing relevant data next to each other, it enhances the ability to make comparisons. Fourth, it keeps track of user's attention as they are navigating the information. Fifth, it provides a more abstract view of a situation through omission and recoding of information. Finally, by letting the user interact and manipulate the visualization, it helps the user in deeply engaging and immersing in the information.

A visualization is an adjustable mapping from the data to a visual form for a human perceiver. A Knowledge graphs is powerful because its schema can be visualized directly in the same form in which it is stored as a data, i.e., without requiring any transformation. We should not assume this approach to always carry over when we are visualizing the data stored in the knowledge graph, because, the stored data size is much larger than the size of the schema. We, therefore, need to explicitly undertake a design of visual structures for best presenting knowledge graph data.

The design of a visual structure involves mapping the desired information into a combination of ways for visual encoding: spatial substrate, marks, connection and enclosure, retinal properties, and temporal encoding. Choosing an appropriate spatial encoding for our information is the first and the most important step. For example, if we want to display geographical data, usually its representation on a map is most intuitive. For non-graphical data, we may choose suitable axes and coordinates along which to display information. Marks are visible things that occur in space, and help a user in distinguishing different data values (e.g., points, lines, area and volume). Connections include graphs, trees, and other hierarchical organizations. Enclosing lines can be drawn around certain objects. Retinal properties include elements such as color, crispness, resolution, transparency, hue, saturation, etc. to visually highlight certain aspects of the data. Finally, a temporal encoding as an animation can enhance a visualization and help us see the dynamics of how the data change over a period of time.

We can group the visualizations into four categories: simple, composed, interactive, and attentive reactive. In a simple visualization, we show up to three different dimensions/variables which is considered the barrier for human understandability. In a composed visualization, we combine one or more simple visualizations so that we can capture more variables in the same display. In an interactive visualization, the user can selectively explore, expand and navigate through the

information. Finally, in an attentive/reactive visualization, the system responds to user actions, and the system potentially anticipates the most useful things to display next.

2.2 Best Practices for Knowledge Graph Visualization

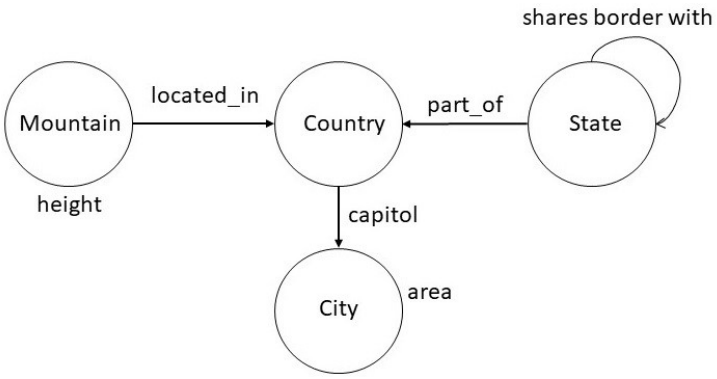
In the previous section, we reviewed some of the principles for designing visualizations, and argued that we should never assume that displaying all the data in a graph in a cluttered display is the most useful presentation. In this section, we consider a recipe for the design of a knowledge graph visualization that leverages some of the principles considered in the previous section.

The design of visualization for a knowledge graph could be divided into the following steps. First, determine which variables of the problem domain to map into spatial position in the visual structure. Second, combine these mappings to increase dimensionality. Third, use retinal properties as an overlay to add more dimensions. Fourth, add controls for user interaction so that selective navigation is possible. Finally, consider attention-reactive features to expand space and manage attention.

A possible visualization for a knowledge graph would have the following elements: overview, dynamic queries, zooming in, details on demand, and retrieval by example. As the amount of data in a knowledge graph is huge, we could begin by presenting an aggregated summary of the data. The user can then filter the data and focus on an area of interest by either posing queries or zooming into certain parts of the data. While viewing a segment of knowledge graph, the user could ask for details on a particular element. Finally, as a use of attentive-reactive feature, the system could proactively make suggestions of the kind of data elements that already exist in the knowledge graph, and offer a choice to the user to select retrieving them.

3. Structured Query Interfaces

A structured query interface can be an important ingredient to interacting with a knowledge graph. In such an interface, the user starts typing expressions, with system suggesting completions in a way that the resulting expression can be mapped into an underlying query language such as Cypher or SPARQL. To illustrate structured queries, consider the following snippet of a knowledge graph schema from Wikidata.



For the instance data corresponding to the above schema, we can pose the following queries.

```
city with largest area
top five cities by area
countries whose capitals have area at least 500 squared kilometers
states bordering Oregon and Washington
```

```
second tallest mountain in France
country with the most number of rivers
```

One way to specify a structured query interface is to specify rules of grammar in the Backus Naur Form (BNF). The rules shown below illustrate this approach for the set of examples considered above.

```
<np> ::= <noun> "and" <noun>
<np> ::= <geographical-region> |
        <geographical-region> <spatial-relation> <geographical-region>
<geographical-region> ::= "capital of country" | "city" | "country" |
                          "mountain" | "river" | "state"
<property> ::= "area" | "height"
<aggregate-relator> ::= "with the most number of" | "with largest" |"with"
<aggregate-modifier> ::= "top" <number> | "second tallest"
<spatial-relation> ::= "bordering" | "inside"
<number-constaint> ::= "atleast" <quantity>
<quantity> ::= <number> <unit>
<unit> ::= "Square Kilometer"
<ranking> ::= "by"
```

We can reduce the queries above to expressions that conform to the grammar above. For example, the first query, "city with the largest area", conforms to the following expression:

```
<geographical-region> <aggregate-relator> <property>
```

Next consider the query "top five cities by area". The following expression that captures this query is equivalent to "top five city by area". For our structured query interface to be faithful to original English, we will need to incoporate the lexical knowledge about plurals.

```
<aggregate-modifier> <geographical-region> <ranking> <property>
```

Finally, we show below the expression for the third query: "countries whose capitals have area at least 500 squared kilometers". The expression show below has the following version in English: "country with capital with area at least 500 squared kilometers". In addition to the incorrect pluralization, it uses different words, for example, "with" instead of "whose", and "with" instead of "have". Most reasonable speakers of English would consider both the queries to be equivalent. This example highlights the tradeoffs in designing structured query interfaces: they may not be faithful to all the different ways of posing the query in English, but they can handle a large number of practical useful cases. For example, the above grammar will correctly handle the query "state with largest area", and a numerous other variations.

```
<geographical-region> <aggregate-relator> <geographical-region>
     <aggregate-relator> <property> <number-constraint> <quantity>
```

Once we have developed a BNF grammar for a structured query interface that specifies the range of queries of interest, it is straightforward to check whether an input query is legal, and also to generate a set of legal queries which could be suggested to the user proactively to autocomplete what they have already typed. We show below a logic programming translation of the above BNF grammar.

```
np(X) :- np(Y) & np(Z) & append(X,Y,Z)
```

```
np(X) :- geographical_region(X)
np(A) :- geographical_region(X) & spatial_relation(Y) &
geographical_region(Z) & append(A,X,Y,Z)
geographical_region(capital)
geographical_region(city)
geographical_region(country)
geographical_region(mountain)
geographical_region(river)
geographical_region(state)
property(area)
property(height)
aggregate_relator(with_the_most_number_of)
aggregate_relator(with_largest)
aggregate_relator(with)
aggregate_modifier(second_tallest)
aggregate_modifier(X) :- number(N) & append(X,top,N)
spatial_relation(bordering)
aggregate_relation(insider)
number_constraint(X) :- quantity(Q) & append(X,atleast,Q)
quantity(Q) :- number(N) & unit(U) & append(Q,N,U)
unit(square_kilomters)
ranking(by)
```

Improvements in structured queries accepted by the system require improving the grammar. This approach can be very cost-effective for situations where the entities of interest and their desired relationships can be easily captured in a grammar. In the next section, we consider approaches that aim to accept queries in English, and try to overcome the problem of required engineering of the grammar through a machine learning approach.

4. Natural Language Query Interfaces

A possible approach to go beyond the structured queries, and to get around the problem of massive engineering of the grammar, is to use a semantic parsing framework. In this framework, we begin with a minimal grammar and use it with a natural language parser that is trained to choose the most likely interpretation of a question. A semantic parsing system for understanding natural language questions has five components: executor, grammar, model, parser and learner. We briefly describe each of these components and then illustrate them using examples.

Unlike traditional natural language parsers that produce a parse tree, a semantic parsing system produces a representation that can be directly executed on a suitable platform. For example, it could produce a SPARQL or a Cypher query. The engine that evaluates the resulting query then becomes the executor for the semantic parsing system.

The grammar in a semantic parsing system specifies a set of rules for processing the input sentences. It connects utterances to possible derivations of logical forms. Formally, a grammar is set of rules of the form $\alpha \Rightarrow \beta$. We show below a snippet of a grammar used by a semantic parsing system.

```
NP(x) with the largest RelNP(r) ⇒ argmax(1,1,x,r)
top NP(n) NP(X) by the RelNP(r) ⇒ argmax(1,n,x,r)
```

```
NP(x) has RelNP(r) at least NP(n) ⇒ (< arg(x,r) n)
```

The first rule in the above grammar can be used to process "city with the largest area". The right-hand side of each rule specifies an executable form. For the first rule, *argmax(M,N,X,R)* is a relation that consists of a ranked list of X (ranked between M and N) such that the ranking is defined on the *R* values of *X*. Similarly, the third rule handles the query "countries whose capital has area at least 15000 square kilomter". In the right hand side of the rule, *(< arg(X,R), N)* specifies the computation to determine those *X* such that their *R* value is less than *N*.

Given an input question, the application of grammar may result in multiple alternative interpretations. A model defines a probability distribution over those interpretations to specify which of them is most likely. A common approach is to use the log-linear machine learning model that takes as input the features of each interpretation. We define the features of an interpretation by maintaining a vector in which each position is a count of how many times a particular rule of the grammar was applied in arriving at that interpretation. The model also contains another vector that specifies the weight of each feature for capturing the importance of each feature. One goal of the learning then is to determine an optimal weight vector.

Given a trained model, and an input, the parser computes its high probability interpretation. Assuming that the utterance is given as a sequence of tokens, the parser (usually a chart parser), recrusively builds interpretations for each span of text. As the total number of interpretations can be exponential, we typically limit the number of interpretations at each step to a pre-specified number (e.g., 20). As a result, the final interpretation is not guaranteed to be optimal, but it very often turns out to be an effective heuristic.

The learner computes the parameters of the model, and in some cases, additions to the grammar, by processing the training examples. The learner optimizes the parameters to maximize the likelihood of observing the examples in the training data. Even though we may have the training data, but we do not typically have the correct interpretation for each instance in the data. Therefore, the training process will consider any interpretation that can reproduce an instance in the training data to be correct. We typically use a stochastic gradient descent to optimize the model.

The components of a semantic parsing system are relatively loosely coupled. The executor is concerned purely with what we want to express independent of how it would be expressed in natural language. The grammar describes how candidate logical forms are constructed from the utterance but does not provide algorithmic guidance nor specifies a way to score the candidates. The model focuses on a interpretation and defines features that could be helpful for predicting accurately. The parser and the learner provide algorithms largely independent of semantic representations. This modularity allows us to improve each component in isolation.

Understanding natural language queries accurately is an extremely difficult problem. Most semantic parsing systems report an accuracy in the range of 50-60%. Improving the performance further requires amassing training data or engineering the grammar. Overall success of the system depends on the tight scope of the desired set of queries and availability of training data and computation power.

5. Summary

In this chapter, we considered different ways end-users interact with knowledge graphs. Perhaps, the most important take away from this chapter should be that just because we are working with knowledge graphs, it does not imply that displaying many nodes and edges on a computer screen is

the most appropriate way to interact with it. Graphical views are often appropriate for the schema information because of its limited size, but not always effective for interacting with the instance data. We argued that the best user interface needs to be designed by considering the business problem to be solved, the kind of data we are working with, and the resources that can be invested into the design and engineering. Best interfaces, often, will use a combination of methods that leverage search, structured queries, and use natural language question answering only to a limited degree.

Exercises

Exercise 7.1. How does visualization amplify the user understanding of a knowledge graph?

(*a*) By presenting impressive graphs
(*b*) By using beautiful colors
(*c*) By presenting a more abstract view of a situation by omission and recoding of information
(*d*) By using optimal graph layout algorithms
(*e*) By leveraging machine learning to better understand a user

Exercise 7.2. Which of the following is not a step in the process of designing a visual structure?

(*a*) Choice of retinal properties
(*b*) Identifying and presenting time varying information
(*c*) Fine tune the features for machine learning
(*d*) Choosing a spatial substrate
(*e*) Choosing suitable axes and coordinates

Exercise 7.3. What is an attentive reactive visualization?

(*a*) It is a simple visualization in which we show upto three different pieces of information.
(*b*) It is a visualization in which we combine one or more simple visualizations with the goal of capturing more variables in the same display.
(*c*) It is a visualization in which the response of the interface depends on how a user interacts with it.
(*d*) It is a visualization in which the user can selectively explore, expand and navigate information.
(*e*) It is a visualization that uses heptic devices.

Exercise 7.4. Which of the following queries does not follow from the BNF grammar considered in Section 3?

(*a*) capitol of country with second tallest mountain
(*b*) mountain bordering river
(*c*) city inside mountain with the most number of river
(*d*) top 100 cities by Square Kilometer
(*e*) country with the most number of state

Exercise 7.5. Which of the following statements is false?

(*a*) Given an input English sentence, it often leads to multiple alternative interpretations.
(*b*) Semantic parser can be trained to choose the interpretation that has the highest probability of being correct.

(*c*) The machine learner of a semantic QA system is capable of learning arbitrary extensions to the grammar.

(*d*) A semantic parsing system requires a manually engineered grammar.

(*e*) We still require a large number of training examples to get a good performance on a semantic parsing system.

CS520

Knowledge Graphs

What should AI Know ?

8. How to Evolve a Knowledge Graph?

1. Introduction

Once a knowledge graph has been built, it will need to evolve in response to changes in the real-world, and changes in the business requirements. Such changes can be either at the schema level or at the level of individual facts in the knowledge graph. Updating a knowledge graph at the level of individual facts is usually much easier than updating the schema. This is because some changes to schema can have far reaching consequences in the data that is already stored, and sometimes, even in the software that makes assumptions about the schema.

Approaches to handle the evolution of a knowledge graph must address both social and technical challenges. Social challenges need to be addressed for, at least, two reasons. First, design changes inherently have an element of subjectivity, and must have a buy in from the users. Second, as the changes can have impact on the work of multiple stakeholders, suitable workflow processes need to be in place for any change to be rolled out across the user community. Unfortunately, there are not many standards and best practices for how best to handle such social processes.

The technical problems involved in evolving a knowledge graph have been fundamental to database and knowledge base management, and have been researched under the topics of schema evolution, view maintenance, and truth maintenance. Each of these techniques is meant for a different category of updates, and is backed by significant theory that can be adapted for the context of knowledge graphs.

The choice of different techniques for evolving a knowledge graph is also strongly influenced by the business requirements and the cost of making a change. Most large-scale knoweldge graphs have numerous inconsistencies that persist over a period of time, and may be left unaddressed, simply because they do not affect a functionality that is critical for business.

We will begin this chapter by considering several concrete examples of changes that are required in knowledge graph. We will then briefly introduce schema evolution, view maintenance, and truth maintenance. Our goal is not to provide a comprehensive coverage on each of these three topics, but to suggest how these methods are relevant to knowledge graphs, and should be adapted.

2. Examples of Changes to a Knowledge Graph

We will consider examples of the following categories of changes to the knowledge graph: changing world, changing requirements, changing sources, changes affecting previous inferences, and changes requiring redesign. We recognize that there may be more than one way to classify the examples considered here, but we hope that this categorization introduces some structure to the series of examples considered below.

Consider the Amazon product knowledge graph and how it must respond to the changes in the real world. This knowledge graph is constructed by combining information provided by multiple

vendors. The product quantities need to be kept up to date with their sales, and as new shipments arrive. Suppliers are constantly providing new products, and some of them may require introducing new properties in the knowledge graph. Some products or product lines may be dropped, and may no longer be necessary.

As an example of changing requirements, consider the concept of *Artist* in Google knowledge graph. The original knowledge graph assumed that an *Artist* can only be a *Person*. Significant checks had been put in the code to ensure that the incoming data satisfied this constraint. But, over a period of time, they started to see data that contained a *Vocaloid* as an artist. As a *Vocaloid* is not a *Person*, and there were a large number of users who cared about listing it as an *Artist*, this assumption needed to be changed.

As an example of the changing sources, consider the problem of creating a knowledge graph for music albums. Many times, the complete information about an album is not available from a single source, and it must be created by combining data from multiple sources. Frequently, new sources need to be added to get a complete picture of an album, and these sources, themselves, change over a period of time. We need to keep the knowledge graph in sync with these changing sources.

As an example of changes that impact inference, consider the constraint that a movie theater shows only movies. A knowledge graph may infer that any program that is being shown in a movie theater is a movie. But, more recently, sporting events, and sometimes, operas are shown in a movie theater. Based on the constraint that a movie theater shows only movies, the inference algorithm will incorrectly conclude that a sporting event or an opera is also a movie. Fixing this requires updating the relevant inference rules that draw these conclusions.

As an example of a change requiring redesign, consider the situation where a knowledge graph initially represents the *CEO* of a company as a relation that has a *Person* as a value. The designers may choose to redesign this representation so that the value of the *CEO* relation is another object with properties such as the time period for which that person held that position.

3. Schema Evolution Techniques

Schema evolution for a relational database is referred to as database reogranization, and addresses problems that arise when adding/removing a column. Schema evolution for a knowledge graph is more complex, and has operations such as adding/removing a class from the class hierarchy, adding/removing a superclass to an existing class, adding/removing type of an individual, adding/removing a relation or a property from a class. The additional complexity arises because the class hierarchy needs to satisfy some constraints, and some of the information inherits and propagates through the knowledge graph. We will next discuss some of these intricacies in greater detail.

When we remove or rename a property, this change must be propagated throughout the knowledge graph. It is typical to generate a summary of all affected places where such an update will impact the knowledge graph.

When we add a new class to the class hierarchy without specifying its immediate superclass, we can assume it to be the class of a system-defined root class. This assumption is based on the constraint that we will not like to have orphan classes in the class hierarchy.

When we delete a class from the class hierarchy, or remove the superclass of a class, we need to make decisions about what to do about its subclasses, and its instances. If its subclasses have another superclass, this does not pose a problem. But, if the class being deleted is the only

superclass of a class, we need to either delete all the subclasses, and their instances or we need to assign a new superclass. To assign a new superclass to class *A* whose only superclass *B* is being deleted, the new superclass could be one or all immediate superclasses of *B*. Furthermore, if there was any property associated with the class that is being deleted, we need to either delete those properties from all the classes and instances where it was inherited, or ensure that the property is associated with another class that will remain in the knowledge graph after the deletion of the class for which this property was originally defined.

Some updates to a class hierarchy can create subtle situations. Consider a situation in which *A* is a superclass of *B*, and *B* is a superclass of *C*. Suppose we assert *A* to be a direct superclass of *C*. As this relationship can also be inferred through transitivity, should such an update even be allowed? Many knowledge graph systems will reject such an update to the class hierarchy.

When adding a new superclass relationship to a class hierarchy, we need to make sure that we maintain the acylicity in the class hierarchy. If adding a new class will create cycles, support should be provided to detect and address the source of the problem.

When we change the constraint on a relation, we can run into at least two different situations. If we are relaxing the constraint, then it does not affect the existing data. But if we make the constraint on a relation tighter, it may invalidate some existing data, and suitable repair actions will be required.

4. View Maintenance Techniques

Views are a mechanism in relational database management systems to name a query so that it can be executed simply by referencing the name. The query is defined with respect to a set of one or more tables, referred to as the *base* tables. If we choose to store the results of the query corresponding to a view, it is referred to as a *materialized* view. If a view is materialized, and there is a change to data in the base tables, the view must be updated. Even though the simplest approach is to recompute the view from scratch, several efficient algorithms, known as *incremental* view maintenance algorithms are available that do not compute the view from scratch.

In the context of knowledge graphs, the use of view computation and maintenance techniques is currently not very common. There are situations where data in the knowledge graph is processed, and the results are stored. For such situations, view definition and maintenance techniques are directly applicable. Leveraging view update methods for such situations is open for future work.

5. Truth Maintenance Techniques

Truth maintenance techniques were originally developed in the context of rule-based systems to keep track of the derived conclusions which can be updated in response to any changes in data or rules. A popular implementation of a truth maintenance system is known as a justification system. In a justification-based system, every time a new piece of information is derived, the system records a justification for the derivation. The justification often includes other facts, and rules that were used during the derivation. At a future time, whenever there is a change in a fact or a rule, one can examine various derivations where that fact or rule appears, and update those derivations by taking the new information into account.

In the context of knowledge graphs, the inferences are usually derived by the application code. In the current state of knowledge graph practice, the derived inferences are not explicitly tracked. As the modern graph engines mature, we anticipate there to be need for explicitly tracking the source of an inference to enable efficient update mechanisms.

6. Summary

Knowledge graphs are created in response to specific business needs, and have a life cycle. Many knowledge graphs persist over a long period of times, and must evolve in response to the changes in the real-world, and the changes in the business requirements. The evolution of a knowledge graph needs to be sensitive to its usage by the user community, needs to follow well-defined engineering processes, and can benefit by using design principles and algorithms. The design principles and algorithms can leverage from the past work on schema evolution, view maintenance, and truth maintenance system which will be increasingly important in the future generation of knowledge graph system.

Exercises

Exercise 8.1. Assess the qualitative ease (ie, easy, moderately involved, highly involved) of making the following changes in a knowledge graph.

(*a*) Incorporating release of a new iPhone into Amazon product graph

(*b*) Incorporating the effect of Brexit

(*c*) Launch of a new vendor for distributing face masks

(*d*) Repurposing a hotel as a hospital for COVID patients

(*e*) Changing Wikipedia knowledge graph to model corporate mergers and acqusitions

Exercise 8.2. For each of the following change, which of the knowledge graph change management technique would be most directly applicable?

(*a*) Eliminating a product category

(*b*) Wiki Data integrates data from different museums, city governments, and bibliographic databases

(*c*) Evolving Microsoft academic publications knowledge graph

(*d*) Updates in the schema mapping rules

(*e*) Splitting a category into two different categories

CS520

Knowledge Graphs

What should AI Know ?

What are some High Value Use Cases of Knowledge Graphs?

1. Introduction

Knowledge graphs are being used for a wide range of applications from space, journalism, biomedicine to entertainment, network security, and pharmaceuticals. We cannot do justice to discussing the full range of knowledge graph applications in this short chapter. Therefore, we have chosen the financial industry vertical for which we will describe three different flavors of knowledge graphs: analytics, tax calculations and financial reporting. The use of knowledge graphs for analytics is probably the most common usage. The use of knowledge graphs in tax calculations (or, more generally, for financial calculations), is similar to their usage in compilers and programming languages. Use of knowledge graphs for exchanging reporting data is an emerging area of application that will likely become increasingly important in the future.

2. Knowledge Graphs for Financial Analytics

Consider a large financial organization such as a bank that must deal with millions of customers some which are companies, and some are individuals. Such organizations routinely face questions that can be instantiated from the following templates.

1. If a company goes into financial trouble, which of our clients are its suppliers and vendors? Are any of those applying for a loan? How much of their business depends on that company?
2. In a supply chain network, is there a single company that connects a group of companies?
3. Which startups have attracted the most influential investors?
4. Which group of investors tend to co-invest?
5. Which companies are most similar to a given company ?
6. Which company might be a good future client for us?

To answer the first question above, we need data about suppliers and vendors of a company. Such data are usually available through third party data providers and must be purchased. The external data about the suppliers and the vendors must be combined with a company's internal data. Doing so leverages the techniques of schema mapping and entity linking that we considered in Chapter 4 for creating a knowledge graph from structured data. Increasingly, institutions are starting to leverage data from the daily news for market intelligence. To leverage the financial news, we will need to extract information from the text using the entity extraction and relation extraction techniques described in Chapter 5. Once a knowledge graph is built, we can use path finding algorithms considered in Chapter 6 to answer this query. Assuming our knowledge graph represents the supplier and vendor relationships between the companies, the traversal algorithms need traverse those relationships to answer the query. The query results may benefit if we use a simple visualization of the supply chain. To answer the second question, we can leverage the centrality detection techniques. In particular, the betweenness centrality is one possible approach for identifying the company that plays a central role in a supply chain.

The third question above is clearly relevant to the valuation of a startup. Just as in the first question, the answer requires getting data about investments from a third-party provider, and integrating it with the internal customer data. Answering the query requires using centrality detection techniques that we considered in Chapter 6. In particular, the graph adaptation of the page rank algorithm is an appropriate technique for answering this question. The answer presentation will benefit by showing a graphical visualization of how the influential investors are connected to various startups they are involved in. The fourth question is an example of community detection. In a knowledge graph that captures the investment relationships, the investors who co-invest in a company often will form a community.

The fifth and sixth questions are examples of reasoning techniques based on graph embeddings that were briefly introduced in chapter 1. Fifth question can be answered using techniques for calculating similarity based on the embedding vectors for the nodes of interest. Sixth question is an instance of the link prediction problem in a graph which is very similar to the problem addressed in language models. Here, instead of predicting the next word, we are interested in predicting the most likely links from a given node.

3. Knowledge Graphs for Income Tax Calculations

The income tax law in United States consists of more than 80,000 pages of text. Every year, more than 150 million income tax returns are filed. The US tax law includes thousands of forms and instructions that can appear in an income tax return. The requirements change every year, and sometimes, can even change in the middle of a tax filing year. With the advent of the income tax preparation software, this difficult to understand body of law has been made accessible to end-users so that they can prepare and file their income tax return on their own.

Some income tax preparation tools represent the income tax law using a set of rules. Once the law is represented as rules, it can not only be used for calculations, but also for generating user dialogs, providing explanations, checking completeness of input, etc. While the calculation of income tax requires rule-based reasoning similar to what we considered in Chapter 6, but many of the supporting operations such as generating user dialogs, determining the effect of changing a rule to the rest of the system, are achieved by modeling the rules as graphs, and using graph-based algorithms to perform those computations. To illustrate such use of the knowledge graphs, consider the following rule from the income tax law.

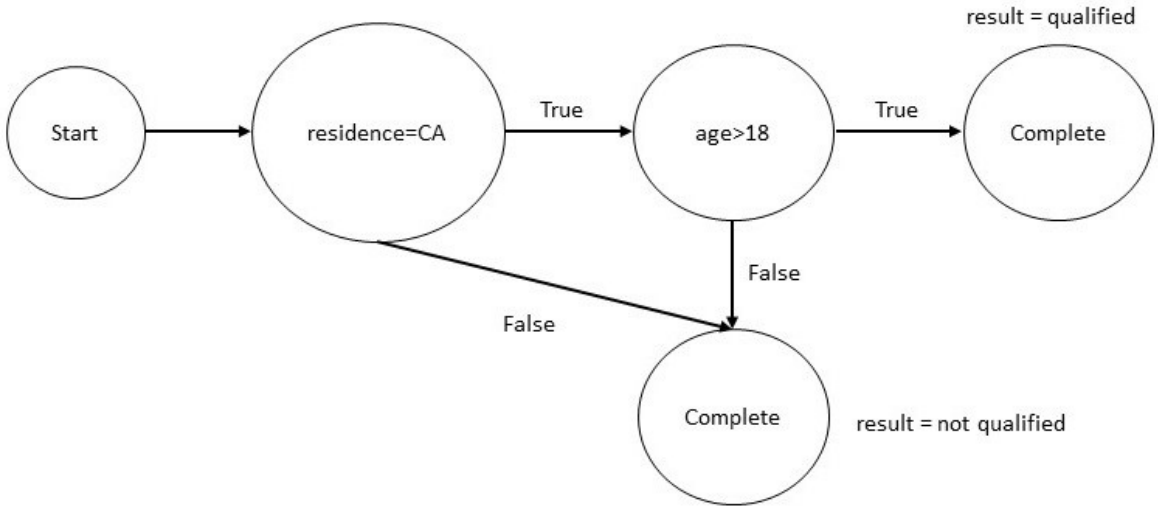
A person is qualified for a tax benefit if:

- the person is a resident of California, and
- the age of the person is greater than 18 years.

We can express this rule as a Datalog rule as shown below.

```
qualified_for_tax_benefit(P) :-
          resident_of(P,CA) & age(P,N) & min(N,18,18)
```

Given the rule above, we can construct a knowledge graph shown below.



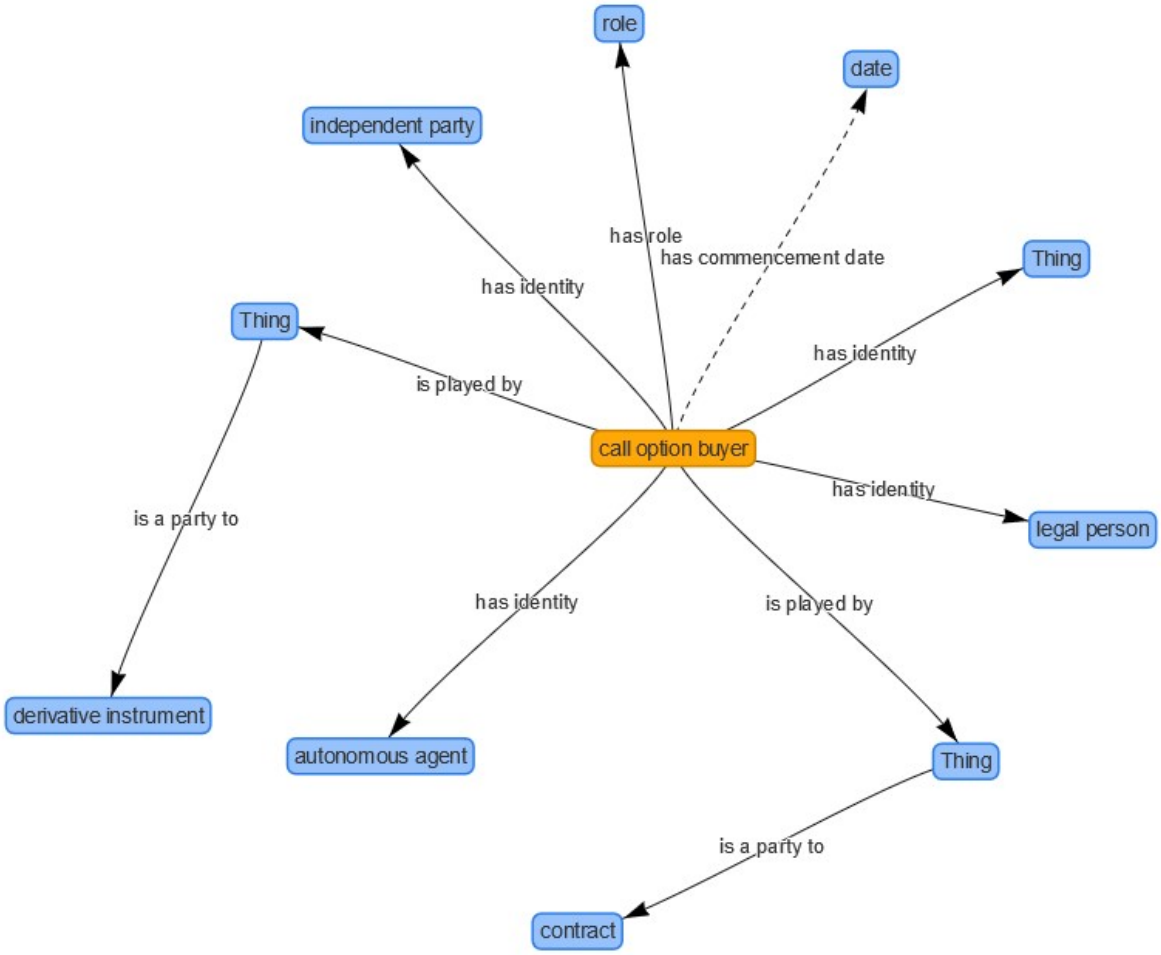
If we are preparing the tax return for a person whose age is 17 years, through a reachability analysis on the above graph, we can determine that in all cases, the person will not be qualified, and hence it is not even necessary to determine their residence. The above example is small as it involves only a single rule. But, as stated earlier, thousands of rules are applicable to a person, and in practice, such analysis needs to be performed on a very large and complex graph.

3. Knowledge Graphs for Financial Reporting

The financial institutions are required to report the derivative contracts that they currently hold. The examples of such contracts include interest rate and commodity swaps, options, futures and forwards, and various asset-backed securities. Such reports are of interest to the compliance teams within an organization, brokers and dealers who need to understand and manage such portfolios, and regulators who need to analyze and oversee these markets. If each financial institution provides such reports in a different format, it becomes challenging for these diverse set of stakeholders to process, aggregate and make sense of these reports. This problem is a classic instance of the data integration problem that knowledge graphs are meant to address.

An industry-wide initiative to address this problem has taken the approach of defining a common semantic model, called, Financial Industry Business Ontology (FIBO). FIBO is defined using a formal language called Web Ontology Language (OWL). FIBO defines things that are of interest in financial business applications and the ways those things can relate to one another. In this way, FIBO can give meaning to any data (e.g., spreadsheets, relational databases, XML documents) that describe the business of finance. FIBO concepts have been developed by a community of users coming from multiple financial institutions and represent a consensus of the common concepts as understood in the industry and as reflected in industry data models and message standards.

FIBO provides the concepts and terminology required for reporting on derivatives. To help users in getting started with FIBO, the developers have provided examples to get started with modeling basic concepts such as a Company, its global legal identifier, the derivatives, etc. For example, FIBO defines that a *Derivatives Contract* could have *has part* one or more *Options*. An *Option* can have a *call option buyer*. We show below a graphical view of connections from a *call option buyer* to related entities in the diagram below.



The graph shown above is an ontology graph in the sense that it does not capture the relationship between data values but the relationships that exist at the schema level. For example, it captures that a call option buyer has to be an independent party, a legal person and an autonomous agent. When a financial institution uses the terms and definitions from FIBO for the financial reports it generates, it provides the foundation for integrating its reports with reports coming from other providers. Use of FIBO can lead to significant streamlining and reduction of costs in aggregating and understanding the information about derivatives contract.

The development of FIBO is being driven by a set of motivating use cases. Derivatives contracts are only one of the several use cases under consideration. Other use cases include counter party exposure, index analysis for ETF development, and exchange instrument data offering.

4. Summary

Knowledge graphs have wide-ranging applications across multiple industries. In this chapter, we chose to focus on three different uses of knowledge graphs in financial industry: analytics, calculations and reporting. Use of knowledge graph for analytics is the most mainstream and wide-spread use of knowledge graphs as it has the potential to offer novel insights into data that an organization may already have. Use of knowledge graph for computations has been around for quite some time as it is similar to the use of graphs in compilers and rule engines for various reasoning and analysis tasks. Finally, the use of ontologies in data exchange is an emerging area for knowledge graphs with tremendous potential that will be increasingly important and mainstream in the future.

Exercises

Exercise 9.1. Suppose, you are facing the problem of finding alternative routes in the face of a traffic jam. Which of the following graph algorithms might be a good choice for this use case?

(*a*) A* Search
(*b*) Minimal Spanning Tree
(*c*) Random walk
(*d*) All pairs shortest path
(*e*) Depth-first Search

Exercise 9.2. Suppose, you are facing the problem of deciding the location of a branch office in a city, and you need to choose the most accessible location. Which graph algorithm might you use to help you make this decision?

(*a*) Degree centrality
(*b*) Closeness centrality
(*c*) Betweenness centrality
(*d*) Page rank
(*e*) Public opinion survey

Exercise 9.3. Suppose, you are working on fraud analysis, and you need to check if a group has a few discrete bad behaviors or is acting as a systematic collection of entities, which of the following graph algorithms might be ideally suited?

(*a*) Triangle count
(*b*) Connected components
(*c*) Strongly connected components
(*d*) Page Rank
(*e*) Fast unfolding algorithm

Exercise 9.4. An income tax application can use as knowledge graph in which of the following ways?

(*a*) Analyze the graph of rules to determine what inputs are required
(*b*) Create a taxonomy of classes to better organize the tax calculation rules
(*c*) Connect customer data with different investment opportunities
(*d*) All of the above
(*e*) None of the above

Exercise 9.5. Which of the following could not be achieved by financial ontologies such as FIBO?

(*a*) Stock recommendations that will outperform the market
(*b*) A standard vocabulary to exchange information
(*c*) Index analysis for ETF development
(*d*) Analysis of counter party exposure
(*e*) Exchange instrument data offering

CS520

Knowledge Graphs

What should AI Know ?

How do Knowledge Graphs Relate to AI?

1. Introduction

In this concluding chapter, we will discuss different ways in which the knowledge graphs intersect with Artificial Intelligence (AI). As we noted in the opening chapter, use of labeled directed graphs for knowledge representation has been around since the early days of AI. Our focus for discussion in this chapter is on the use of knowledge graphs in the recent developments. Consequently, we have chosen three themes for further elaboration: knowledge graphs as a test bed for AI algorithms, emerging new specialty area of graph data science, and knowledge graphs in the broader context of achieving the ultimate vision of AI.

2. Knowledge Graphs as a Test-Bed for Current Generation AI Algorithms

Knowledge graphs have a two way relationship with AI algorithms. On one hand, knowledge graphs enable many of the current AI applications, and on the other, many of the current AI algorithms are used in creating the knowledge graphs. We will consider this symbiotic synergy in both directions.

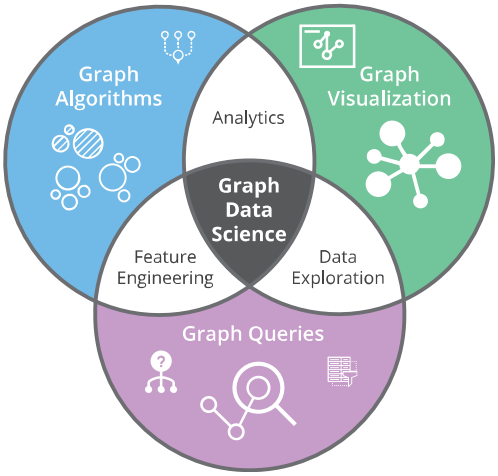
Personal assistants, recommender systems, and search engines are applications that exhibit intelligent behavior and have billions of users. It is now widely accepted that these applications behave better if they can leverage knoweldge graphs. A personal assistant using a knowledge graph can get more things done. A recommender system with a knowledge graph can make better recommendations. Similarly, a search engine can return better results when it has access to a knowledge graph. Thus, these applications provide a compelling context and a set of requirements for knowledge graphs to have an impact on immediate product offerings.

To create a knowledge graph, we must absorb knowledge from multiple information sources, align that information, distill key pieces of knowledge from the sea of information, and mine that knowledge to extract the wisdom that would influence the intelligent behavior. The AI techniques play an important role at each step of knowledge graph creation and exploitation. For extracting information from sources, we considered entity and relation extraction techniques. For aligning information across multiple sources, we used techniques such as schema mapping and entity linking. To distill the extracted information, we can use techniques such as data cleaning and anamaly detection. Finally, to extract the wisdom from the graph we used inference algorithms, natural language question answering, etc.

Hence, knowledge graphs enable AI systems, which provide motivation and a set of requirements for them. AI techniques are also fueling our ability to create the knowledge graph economically and at scale.

3. Knowledge Graphs and Graph Data Science

Graph data science is an emerging discipline that aims to derive knowledge by leveraging structure in the data. Organizations typically have access to huge amount of data, but their ability to leverage this data has been limited by a collection of preset reports that are generated using that data. The discipline of graph data science is transforming that experience by combining graph algorithms, graph queries and visualizations into products that significantly speedup the process of gaining insights.



As we saw in the analytics-oriented use cases for financial industry, businesses are keen to exploit the relational structure in their data to make predictions about risk, new market opportunities, etc. For making predictions, it is common to use machine learning algorithms that rely on careful feature engineering. As machine learning algorithms are now becoming a commodity and can be used as off-the-shelf products, there is emerging a distinct skill of feature engineering. Feature engineering requires a deep understanding of the domain as well as understanding of the workings of machine learning algorithms.

It is this synergy among the traditional graph-based system and the availability of machine learning to identify and predict relational properties in data, that has catalyzed the creation of the sub-discipline of graph data science. Because of the high impact use cases possible through graph data science, it is becoming an increasingly sought after software skill in the industry today.

4. Knowledge Graphs and Longer-Term Objectives of AI

Early work in AI focused on explicit representation of knowledge and initiated the field of knowledge graphs through representations such as semantic networks. As the field evolved, semantic networks were formalized, and led to several generations of representation languages such as description logics, logic programs, and graphical models. Along with the development of these languages, an equally important challenge of authoring the knowledge in the chosen formalism was addressed. The techniques for authoring knowledge have ranged from knowledge engineering, inductive learning, and more recently deep learning methods.

To realize the vision of AI, an explicit knowledge representation of a domain that matches human understanding and enables reasoning with it is essential. While in some performance tasks such as search, recommendation, translation, etc., human understanding and precision are not hard requirements, but there are numerous domains where these requirements are essential. Examples of such domains include knowledge of law for income tax calculations, knowledge of a subject domain for teaching it to a student, knowledge of a contract so that a computer can automatically execute it, etc. Furthermore, it is being increasingly recognized that for many situations where we

can achieve intelligent behavior without explicit knowledge representation, the behavior still needs to be explainable so that humans can understand the rationale for it. For this reason, we believe, that an explicit representation is essential.

There is narrative in the research community that knowledge engineering does not scale, and that the natural language processing, and machine learning methods scale. Such claims are based on an incorrect characterization of the tasks addressed by natural language processing. For example, using language models, one may be able to calculate word similarity, but the language model gives us no information on the reason for that similarity. In contrast, when we use a resource such as Wordnet for calculating word similarity, we know exactly the basis for that similarity. A language model might have achieved scale, but at the cost of human understandability of its conclusions. The success of web-scale methods is crucially dependent on the human input in the form of hyperlinks, click data, or explicit user feedback. Leveraging these scalable and automated methods to create human understandable knowledge graphs, and using them to achieve intelligent behavior truly addresses how an AI system should function.

It is well-known that a simple labeled graph representation is insufficient for many of the performance tasks desired from AI. That was precisely the reason for developing more expressive representation formalisms. Due to a need to address the economics and the scale of creating such representation, expressive formalisms are less commonly used, but it does not imply that the problems such formalisms address have been solved by the newer deep learning and NLP methods. Some examples of such problems include self-awareness, commonsense reasoning, model-based reasoning, experimental design, etc. A self-aware system can recognize and express the limits of its own knowledge. Commonsense understanding of the world gives a system ability to recognize obviously nonsensical situations, e.g., a coin that has a date stamp of 1800 B.C., could not be a real coin. Current language models can generate sentences that make sense only up to a certain length, but they lack an overall model of the narrative to generate longer coherent texts. Creating AI programs that can master a domain, formulate a hypothesis, design an experiment, and analyze its results is a challenge that is out of reach of any of the current generation systems.

5. Summary

We considered three different ways the work on knowledge graphs intersect with AI: as a test-bed for evaluating machine learning and NLP algorithms, as an enabler of the emerging discipline of graph data science, and as a core ingredient to realizing the long-term vision of AI. We hope that this volume will inspire many to leverage what is possible through the scalable creation of knowledge graphs and their exploitation. And yet, we should not let go a longer-term vision of creating expressive, and human understandable representations that can also be created scalably.

Exercises

Exercise 10.1. Which of the following statements is false?

(*a*) Knowledge graphs have been an essential ingredient to the success of personal assistants.
(*b*) Machine learning is indispensable for creating large-scale modern knowledge graphs.
(*c*) Knowledge graphs will eventually be unnecessary for the success of AI applications.
(*d*) Knowledge graphs significantly expand the inferences possible using natural language.
(*e*) Technology is now available to create rudimentry knowledge graphs from images.

Exercise 10.2. Which of the following is out of scope of graph data science?

(*a*) Transaction management

(*b*) Feature engineering
(*c*) Visual Analytics
(*d*) Block Chain
(*e*) Semantic transformation of logical expressions

Exercise 10.3. Which of the following is true about how knowledge graphs might relate to the future of AI?

(*a*) Property graphs provide a sufficient representation for us to build future intelligent applications.
(*b*) Expressive logic-based representations are what we need for future intelligent applications.
(*c*) Some explicit representation similar to knowledge graphs is required, but exactly what is needed, is open for future research.
(*d*) Future techniques of AI will have reducing reliance on explicit representation.
(*e*) The goal of AI should be to eliminate the need for any representation.