Foundations of Knowledge Graphs

Prof. Dr. Axel Ngonga

Student lecture notes by
Tanja Tornede & Alexander Hetzer
Status: August 24, 2018

Contents

CHAPTER 1

Introduction

1.1 The 4 V's

Volume Amount of data grows continuously

- Exponential hardware improvement is over
- **Knowledge extraction:** Need better digital processing
- **Knowledge representation:** Machines need to "understand"
- **Knowledge access:** Information must be easy to use

Velocity We produce and consume data exponentially

- **Knowledge access:** Increasing number of internet users
- **Knowledge storage:** Exponential increase of data generated
- **Knowledge access:** Improved retrieval of knowledge

Variety Most of the data is unstructured

- **Knowledge extraction:** Growing amount of unstructured data
- **Knowledge storage:** Exponential increase of data generated
- **Knowledge access:** Need for improved retrieval of knowledge

Value Need to gather valuable information out of flood of data

- Need ways to gain valuable knowledge out of flood of information

1.2 Guiding Principles

1.2.1 The Knowledge Principle

"If a program is to perform a complex task well, it must know a great deal about the world in which it operates." (Lenat and Feigenbaum, 1991)

1.2.2 The Breadth Principle

"To behave intelligently in unexpected situations, an agent must be capable of falling back on increasingly general knowledge." (Lenat and Feigenbaum, 1991)

1.3 Building Blocks of a Solution

- **Knowledge Representation**
  - Different models ranging from triples to property graphs
- **Knowledge Extraction**
  - Knowledge graphs need to be filled
  - Will study **knowledge extraction** from tables, text and XML
- **Knowledge Storage & Querying**
  - Knowledge graphs need to be queried and updated
  - Will study alternatives to knowledge storage from relational to graph databases
- **Knowledge Manipulation**
  - Knowledge graphs need processed efficiently
  - Will study approaches for graph management including link prediction
- **Knowledge Access**
  - Knowledge graphs must be available to lay users
  - Will study approaches for search and question answering

1.4 Knowledge Graphs

1.4.1 Definition

Graph structures which encode domain-specific knowledge.

- Vague definition as no generally accepted definition available at the moment
- Three main components: Graph, Knowledge, Domain

1.4.2 Graphs / Networks

- Basic structure: $G = (V, E)$, where V is the set of vertices and E is the set of edges
  - Often $E \subseteq 2(V)$ or $E \subseteq V^2$
  - Often directed graphs, i.e. $E \subseteq V^2$

1.4.3 Knowledge

- Common definition: "Knowledge is justified true Belief." (Stanford Dictionary of Philosophy, 2017)

  Belief Anz proposition p

  Truth Assertion of belief in p

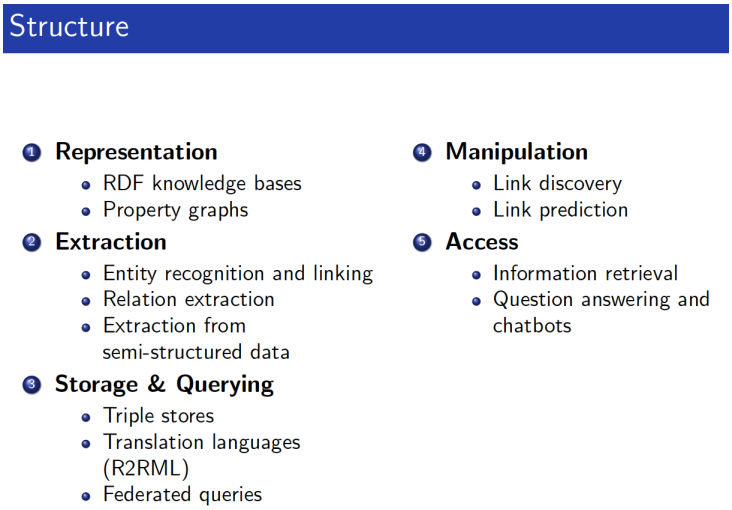
  Justification Deontological (no obligation to refrain from belief) or non-deontological (proper probabili

  cation of belief)

- Working definition: Set of statements which **model a particular domain**

1.4.4 Domain

- **Domain of Discourse:** Set of entities over which certain variables of interest in some formal treatment may range.



1.5 Semantic Networks

1.5.1 Definition

- **semantic network**: graph structure for representing knowledge in patterns of interconnected nodes and arcs

1.5.2 Kinds of Semantic Networks

Definitional Network : generalization/subsumption hierarchies

- model information assumed to be necessarily true
- emphasize the subtype or is-a relation

Assertional Network : designed to assert propositions

- designed to assert propositions
- model information is assumed to be contingently true (partly)

Implicational Networks : use implication as the primary relation for connecting nodes

- used to represent patterns of belief

Executable Networks : allow to perform operations on data

Learning Networks : build or extend their representations by acquiring knowledge from examples

- allow for structural modifications
- can perform inference or search

Hybrid Networks : combine two or more of the previous techniques

CHAPTER 2

RDF Graphs

- Relational graphs ⇒ Assertional Networks
- It is special because of its explicit semantics
- Why not XML?
  - Structure must be a tree
  - Merging is difficult/impossible as all XML documents could have different structures

2.1 RDF Triples

- Each triple describes a fact
  - Subject: IRI or blank node
  - Predicate: IRI
  - Object: IRI, blank node or literal
- Triples are joined into graphs by conjunctions

IRI = Internationalized Resource Identifier

- Extension of URIs (must be US ASCII)
- IRI can contain Unicode characters

Literals Used to model data values

- Interpretation using data type

Blank Nodes Resources without URIs

- Only have a local scope
- Interpretation: existential quantification

2.2 RDF Schema

Express schema knowledge (also called terminological knowledge).

Class A class is a set of things (i.e. entities)

- Every class is a resource and has a URI
- Every URI which stands for a class must be an instance of `rdfs:Class`
- A resource can belong to several classes with the predicate `rdf:type`
- Classes can be organized in hierarchies with the predicate `rdfs:subClassOf`
  - `rdfs:subClassOf` is **transitive**
- Equivalence of classes through double inclusion
  (`A subClassOf B` and `B subClassOf A`) $\Rightarrow$ `A subClassOf A`
  - `rdfs:subClassOf` is **reflexive**

Predicates Describe relations between "proper" resources or individuals (subject, object)

- `rdf:Property` denotes the class of all properties
- `rdfs:subPropertyOf` can be used to create hierarchies of properties
  (`ex:drives rdfs:subPropertyOf ex:controls`)
- Every predicate can be assigned domain and range which constitute the "semantic link" between classes and properties because they provide the only way of describing the desired terminological interdependencies between those distinct kinds of ontology elements.

  Domain Classify subjects (`rdfs:domain`)

  Range Type objects that co-occur with certain predicate (`rdfs:range`)

Reification Proposition referring to other propositions, therefore many-valued relation

- Connecting the auxiliary node via RDF-triple with the respective triple constituents
- Mark the "central" node of a reified triple as `rdf:Statement`
  - Represent it as blank node if it is to be referenced only locally
- Example: "The detective supposes that the butler killed the gardener."

```
ex:detective  ex:supposes     ex:theory .
ex:theory     rdf:subject     ex:butler .
ex:theory     rdf:predicate   ex:hasKilled .
ex:theory     rdf:object      ex:gardener .

ex:theory     rdf:type      rdf:Statement .
```

2.3 Semantics of RDF

- Why Semantics?
  - Knowledge graphs = distributed, declarative knowledge
    Rightarrow Need for computable semantics, e.g. for merged knowledge
  - Example-based semantics insufficient (cornercases)
    Rightarrow Need for formal model of meaning
  - Semantics defined through one relation: **logical entailment**

2.3.1 Model-Theoretic Semantics

- Interpretation: Potential realities (also called worlds)
- Define when an interpretation I *satisfies* a proposition p

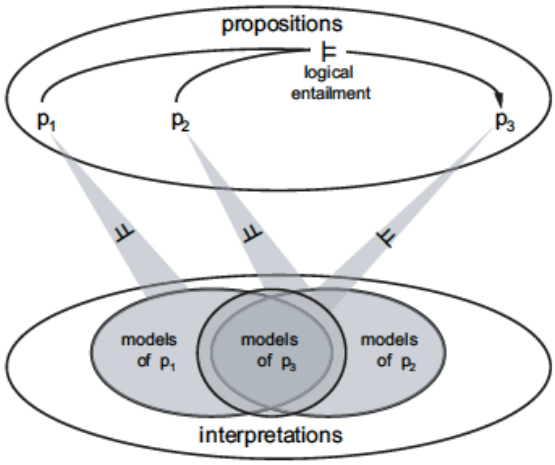


FIGURE 3.1: Definition of the entailment relation via models

- Interpretation I satisfies a specific proposition $p \in \mathbb{P}$, i.e. I is a model of p: $I \models p$
- I is a model of a set $P \in \mathbb{P}$ of propositions if it is a model for every $p \in P$: $I \models P$
- **Entailment:** $\alpha, \beta \subseteq \mathbb{P}$. $\alpha \models \beta$ iff $\forall I$: $I \models \alpha \rightarrow I \models \beta$
  - For set of propositions K: $I \models K$ iff $\forall \beta \in K : I \models \beta$
  - For set of interpretations J: $J \models K$ iff $\forall I \in J : I \models K$
- Interpretations
  - Begin with simple interpretations (most basic)
  - Add set of rules for RDF interpretations
  - Strictest are RDFS interpretations

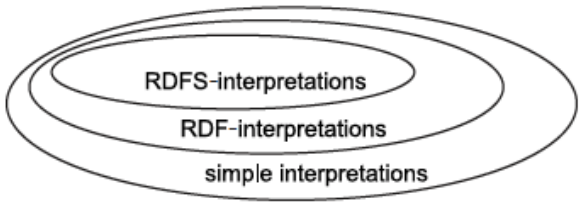


FIGURE 3.2: Correspondence of interpretations

Syntax of RDF(S)

- Model-Theoretic semantics: We define interpretations and define when an interpretation models a graph
- Basic elements (vocabulary V): URIs, blank nodes and literals (are not propositions)
- **Propositions**
  - Triples are propositions: $(s, p, o) \in (URI \cup bNode) \times URI \times (URI \cup bNode \cup Literal)$
  - Finite set of triples (i.e. graphs) are propositions [**ToDo**: *Why?*]

Simple Interpretation

- Vocabulary: arbitrary set containing URIs and literals
  - (Syntactic) URIs or *literals stand* for or *represent* (semantic) resources
  - This kind of representation is encoded by further functions that assign a semantic counter part to every URI and literal
  - Here URIs are treated equallz as there is no "semantic special treatment" for the RDF and RDFS vocabulary
- Elements of a simple interpretation $\mathcal{I}$ of a vocabulary V

  IR**:** Non-empty set of **resources**, also called domain of discourse or universe of $\mathcal{I}$

  IP**:** Set of **properties** of $\mathcal{I}$ ($IR \cap IP$ can be non-empty; they may overlap)

  $I_{EXT} : IP \to 2^{IR \times IR}$ **Extension function**, maps each property to a set of pairs from IR

  $I_{EXT}(p)$**:** The **extension** of the property p

  $I_S : V \to IR \cup IP$**: Interpretation function** maps URIs from V into $IR \cup IP$

  I_L**:** Maps **typed literals** from V to IR

  $LV \subseteq IR$**:** Set of literal values, contains (at least) all **untyped literals** from V

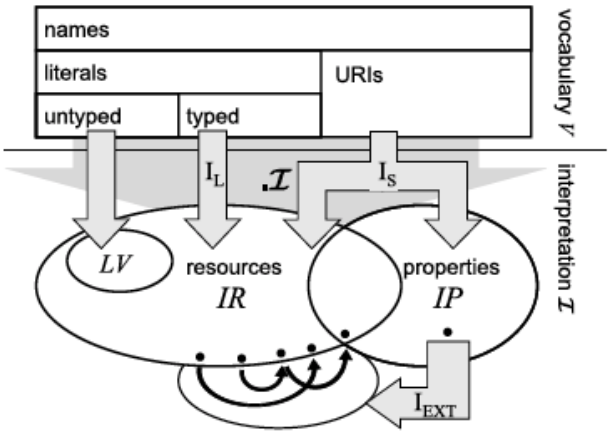


FIGURE 3.3: Schematic representation of a simple interpretation

- Interpretation Function $\cdot^{\mathcal{I}}$ maps all literals and URIs from V to resources and properties
  - Grounded triple: every triple not containing blank nodes
    - Interpretation $((s,p,o))^{\mathcal{I}}$ of a grounded triple (s,p,o) is true iff
      - $s,p,o \in V$ and $\langle s^{\mathcal{I}}, o^{\mathcal{I}} \rangle \in I_{EXT}(p^{\mathcal{I}})$
  - Grounded graph: only contains grounded triples
    - Interpretation $G^{\mathcal{I}}$ of a grounded graph is true iff
      - Every triple $T \in G$: $T^{\mathcal{I}}$ is true
  - Graph with blank nodes
    - Interpretation of graphs with blank nodes:
      - $I \models G$ iff $I \models G'$ for at least one grounding G' of G
    - Generate grounded graph by replacing every blank node by a URI not used so far
      - Define mapping A that assigns a resource from IR to every blank node
      - Combine $\mathcal{I} + A$ that behaves exactly like $\mathcal{I}$ and additionally like A (can be extended to triples and further to graphs)

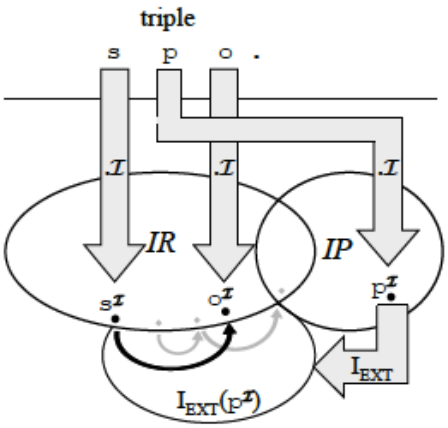


FIGURE 3.4: Criterion for the validity of a triple with respect to an interpretation

RDF Interpretation

- V_{RDF} consists of the following URIs plus an infinite number of URIs `rdf_`i for every positive integer i

```
rdf:type rdf:Property rdf:XMLLiteral rdf:nil rdf:List rdf:Statement
rdf:subject rdf:predicate rdf:object rdf:first rdf:rest rdf:Seq
rdf:Bag rdf:Alt rdf:value
```

- Simple interpretation of vocabulary $V \cup V_{RDF}$ that additionally satisfies come conditions
- **Axiomatic triples**: subsequent triples values as true (infinite many)
  - Mostly serve the purpose of making resources that are assigned to particular RDF URIs as properties
- **RDF Entailment**
  - G_1 RDF-entails G_2 iff every RDF-interpretation that is a model of G_1 is also a model of G_2

RDFS Interpretation

- V_{RDFS} consists of the following names

```
rdfs:domain rdfs:range rdfs:Resource rdfs:Literal rdfs:Datatype
rdfs:Class rdfs:subClassOf rdfs:subPropertyOf rdfs:member
rdfs:Container rdfs:ContainerMembershipProperty rdfs:comment
rdfs:seeAlso rdfs:isDefinedBy rdfs:label
```

- Simple interpretation of vocabulary $V \cup V_{RDFS}$ that additionally satisfies some conditions
- **Axiomatic triples**: subsequent triples values as true (infinite many)
  - Triples with predicate `rdfs:domain`, enforces class membership for every URI occurring as a subject
  - Triples having the predicate `rdfs:range`, enforces class membership for every URI occurring as object
  - Some triples defining containers and lists as properties
- **RDFS Entailment**
  - G_1 RDFS-entails G_2 iff every RDFS-interpretation that is a model of G_1 is also a model of G_2

2.3.2 Syntactic Reasoning

- So far we can decide whether a model is valid w.r.t. RDFS Semantics
- Problems
  - Need to deal with all interpretations of a graph to state whether it is entailed by a set of graphs
  - Computationally not viable
  - Infinitely many interpretations (blank nodes)
- Define **deduction calculus** (algorithm) to decide the validity of conclusions syntactically

2.3.3 Deduction Calculus

- Decide the validity of conclusions syntactically
- Operate only on the given propositions of a logic without directly recurring to interpretations
- Deduction rules in general have the following form, given the validity of proposition p_i deduct that p must also be valid

$$\frac{p_1 \cdot \ldots \cdot p_n}{p} \tag{2.1}$$

- **Deduction Calculus** is the set of deduction rules given for a logic (usually there are several)
- Propositions of set P' can be **proven** by using proposition set P and some sequence of applications of the deduction calculus

$$P \vdash P' \tag{2.2}$$

- **Soundness:** Every proposition set P' that can be derived from P is also entailed by P

$$P \vdash P' \rightarrow P \models P' \tag{2.3}$$

- **Completeness:** Every entailment can be derived by using the calculus

$$P \models P' \rightarrow P \vdash P' \tag{2.4}$$

- Existence of a (sound and complete) deduction calculus does not guarantee the existence of a decision procedure
- **Complexity:** Simple, RDF- and RDFS-entailment are NP-complete
  - Removing blank nodes leads to polynomial complexity [**ToDo**: *Intuitive explanation?*]

[**ToDo**: *Zu diesem Thema im Buch die folgenden Seiten lesen:*

- *74 - 85 Entailment Rules (3.2 - 3.2.3)*
- *90-101 Deduction Rules (3.3 - 3.3.3)*

]

2.4 Query Language

- Only a rather restricted set of questions can be asked, especially in simple formalisms such as RDF
- Retrieving information from a knowledge base is not just a question of query expressivity, but must also address particular requirements such as post-processing and formatting of results (e.g.: language information)
- Requirements
  - High expressiveness
  - Operators for formatting, constraining and manipulating result sets

2.5 SPARQL: Query Language for RDF

- Stands for: SPARQL Protocol And RDF Query Language
- Similar to SQL
- Uses Turtle syntax ⇒ not all triples need to be finished with a full stop, it is more like a separator between triples
- Queries based on query patterns
- SPARQL 1.1 recommended by W3C (since 2013)

2.5.1 SPARQL Queries

`PREFIX` Abbreviations can be used for URIs

`?variable` Query can contain variables as s, p or o, beginning with `?` or `$` (`?var` = `$var`)

Blank Node If ID was used it may not be used again in another graph pattern of the same query

- Query Pattern: Can be used as s or o, behaves like variable that cannot be selected
- Result: Placeholder for unknown elements where the IDs are arbitrary (differ to the ones in the KG)

Datatypes Must match exactly the query

- Implicitly via syntax: String, Integer, Decimal, Double

Graph Pattern

Basic Graph Pattern (BGP) Use the Turtle syntax for RDF, `WHERE` enclosed in curly braces

```
PREFIX ex:  <http://example.org/>
SELECT ?title ?author
WHERE  { ?book    ex:publishedBy  <http://crc-press.com/uri> .
         ?book    ex:title        ?title .
         ?book    ex:author       ?author }
```

Complex Graph Pattern Can be grouped using {...}

```
PREFIX ex:  <http://example.org/>
SELECT ?title ?autor
WHERE
  { { ?buch  ex:hatVerlag  <http://springer.com/Verlag> .
      ?buch  ex:titel       ?title . }
    { }
    ?buch  ex:autor        ?autor .
  }
```

`OPTIONAL` Allows specification of optional parts of pattern (results can be unbounded)

```
{ ?buch    ex:hatVerlag  <http://springer.com/Verlag> .
  OPTIONAL { ?buch    ex:titel       ?title . }
  OPTIONAL { ?buch    ex:autor       ?autor . }
}
```

`UNION` Specifies alternative graph patterns (variables do not influence each other)

```
{ ?buch    ex:hatVerlag  <http://springer.com/Verlag> .
  { ?buch    ex:autor       ?autor . } UNION
  { ?buch    ex:Verfasser   ?autor . }
}
```

`FILTER` Keyword followed by an expression in brackets

- Comparison operators: `<`, `=`, `>`, `<=`, `>=`, `!=` (No comparison of incompatible types)
- Arithmetic operators: `+`, `-`, `*`, `/` (only for numerical datatypes)
- **Function** RDF-specific filter functions

`BOUND(A)`	true if `A` is a bound variable
`isURI(A)`	true if `A` is a URI
`isBLANK(A)`	true if `A` is an empty node
`isLITERAL(A)`	true if `A` is an RDF literal
`STR(A)`	lexical representation (`xsd:string`) of RDF literals or URIs
`LANG(A)`	Language code of an RDF literal (`xsd:string`) or an empty code if no language code is given

`sameTERM(A,B)`	true, if `A` and `B` are the same RDF term.
`langMATCHES(A,B)`	true if the language tag of `A` matches `B`
`REGEX(A,B)`	true, if the string `A` contains the regular expression `B`
`DATATYPE(A)`	URI of the datatype of an RDF literal (`xsd:string` for untyped literals without language tag)

- **Boolean Operator** Connects filter conditions: `&&`, `||`, `!`
  - Conjunctions are equivalent to using several filters
  - Disjunctions are equivalent to the use of filters in `UNION`s

- Combination of options and alternatives
  - `Optional` influences exactly the one graph pattern on its right
  - `Optional` and `Union` bind with the same strength and are left-associated (like the subtraction)

Example:

```
{ {s1 p1 o1} OPTIONAL {s2 p2 o2} UNION {s3 p3 o3}
             OPTIONAL {s4 p4 o4} OPTIONAL {s5 p5 o5}
}
```

means

```
{ { { { {s1 p1 o1} OPTIONAL {s2 p2 o2}
      } UNION {s3 p3 o3}
    } OPTIONAL {s4 p4 o4}
  } OPTIONAL {s5 p5 o5}
}
```

Output Format

`SELECT: SELECT <ListOfVariables>` or `SELECT *`

- Determines result format: Result selection using selection of variables
- Structure/Relations between objects is unclear in result

`CONSTRUCT: CONSTRUCT <RDF Template in Turtle>`

- Results are encoded in RDF graph
- Shorthand if template and query are identical: `CONSTRUCT WHERE`
- Sequential processing is more difficult
- No processing of unbound variables possible

```
PREFIX ex: <http://example.org/>
CONSTRUCT { ?person  ex:mailbox  ?email .
            ?person  ex:telefon  ?telefon . }
WHERE { ?person  ex:email   ?email .
        ?person  ex:tel     ?telefon . }
```

`ASK` Checks whether a result exists and does not require any parameter

`DESCRIBE` Returns an RDF description of a given URI (implementation depends on storage solution used)

Modifier

Modifies solutions so that results are shown in a defined order

`ORDER BY` Order results

- Ordering of different datatypes and types of elements:
  Unbound variable < blank nodes < URIs < RDF literals
- `ORDER BY DESC(?var)` or `ORDER BY ASC(?var)`
- `ORDER BY a, b` Hierarchical ordering (first a, then b)

`LIMIT` Maximal number of results (only useful with `ORDER BY`)

`OFFSET` Position of the first result returned by the query (only useful with `ORDER BY`)

`SELECT DISTINCT` Removal of duplicates

Processing order of modifiers

1) `ORDER BY`
2) Removal of unselected variables
3) Removal of duplicates (`DISTINCT`)
4) Removal of first `OFFSET` results
5) Removal of all results after `LIMIT`

2.5.2 SPARQL 1.1

Extensions

- SPARQL Update (write to triple stores)
- SPARQL Entailment (reasoning)
- SPARQL Service (Allows endpoints to describe themselves)
- SPARQL Federated Queries (Access several endpoints in one query)
- Quite a few changes in SPARQL query (see next slide)

Projection Assign new values to variables, Data errors returned as unbound variables

```
PREFIX ex: <http://example.org/>
SELECT ?book (?price * 1.1 AS ?newPrice )
WHERE { ?book ex:price ?price}
```

Aggregation x

- `COUNT(.)` for counting values, variable bindings
- `SUM(.)`, `AVG(.)`, `MIN(.)`, `MAX(.)` as known from SQL
- `HAVING()` to filter groups
- `SAMPLE` to select data in a non-deterministic manner
- `GROUP_CONCAT` for the concatenation of values using string operators

Subqueries Allow the formulation of complex queries (as known from SQL)

Negation

- `MINUS` for set difference at the level of variables
- `NOT EXISTS` in `FILTER`

Property Paths For graph-based queries

- For graph-based queries
- Basic elements
  - `?s ^p1 ?t`: Inverse path, i.e., `?t p1 ?s`
  - `?s p1/p2 ?t`: Sequence of paths
    $\exists$ `?x`: (`?s p1 ?x`) $\wedge$ (`?x p2 ?t`)
  - `?s p1|p2 ?t`: Alternative paths, i.e., `?s p1 ?t` $\vee$ `?s p2 ?t`
  - `?s p1*` `?t` (Kleene star)
  - `?s p1+` `?t`: $\exists$ `?x`: `?s p1 ?x` $\wedge$ `?x p1*` `?t`
  - `?s !p1 ?t`: Negation

2.5.3 Semantics

- **Formal Logic**
  - Model-theoretic semantics: Which interpretations fulfill a knowledge base?
  - Proof-theoretical semantics: Which derivations from a knowledge base are permitted?
- **Programming Language**
  - Axiomatic semantics: Which logical statements apply to a program?
  - Operational semantics: How does the execution of a program affect?
  - Denotational semantics: How can a program be displayed as an input/output function abstract?
- **Query Language**
  - Query Entailment
    - Request as a description of allowed query results
    - Database as set of logical assumptions (theory)
    - Result as a logical conclusion
  - Query Algebra
    - Request as calculation rule for the determination of results
    - Database as input
    - Result as output

SPARQL-Algebra

Semantics of a SPARQL-query

1) Transformation of the querz into an algebraic expression
2) Calculation of the result of this expression

1) Replacement of simple graph patterns
   - Operator `BGP`
   - Resolution of abbreviated URIs at the same time
2) Summary of alternative graph patterns
   - Operator `UNION`
   - Reference to patterns adjacent to `UNION` (binds more strongly than conjunction)
   - Left-associative parenthesis
3) Remaining translation gradually from inside to outside
   - Special rules how to do that, and what to do with modifier operations

```
SELECT ?y SAMPLE(?name)
WHERE {
   ?x foaf:name ?name .
}
ORDER BY ?x
GROUP BY ?name
```

```
{ BGP(?book <http://eg.org/price> ?price.)
  FILTER (?price < 15)
  OPTIONAL
   {BGP(?book <http://eg.org/title> ?title.)}
  {BGP(?book <http://eg.org/author>
                    <http://eg.org/Shakespeare>.)}
  UNION
  {BGP(?book <http://eg.org/author>
                    <http://eg.org/Marlowe>.)}
}
```

```
{ BGP(?book <http://eg.org/price> ?price.)
  FILTER (?price < 15)
  OPTIONAL
   {BGP(?book <http://eg.org/title> ?title.)}
  Union({BGP(?book <http://eg.org/author>
                    <http://eg.org/Shakespeare>.)},
        {BGP(?book <http://eg.org/author>
                    <http://eg.org/Marlowe>.)})
}
```

```
Filter((?price < 15),
 Join(
  LeftJoin(
   BGP(?book <http://eg.org/price> ?price.),
   BGP(?book <http://eg.org/title> ?title.),
   true
  ), Union(BGP(?book <http://eg.org/author>
                    <http://eg.org/Shakespeare>.),
           BGP(?book <http://eg.org/author>
                    <http://eg.org/Marlowe>.))
 )
)
```

Computation

Idea:

- Every operator of the SPARQL algebra returns a query result which might be further processed by other operators in case of nestings
  - Query result: table where each row represents a variable binding, where some of the variables might be unbound
    - Row encoded as partial Function that maps variables to RDF terms
    - *Solution* for a given query: result row

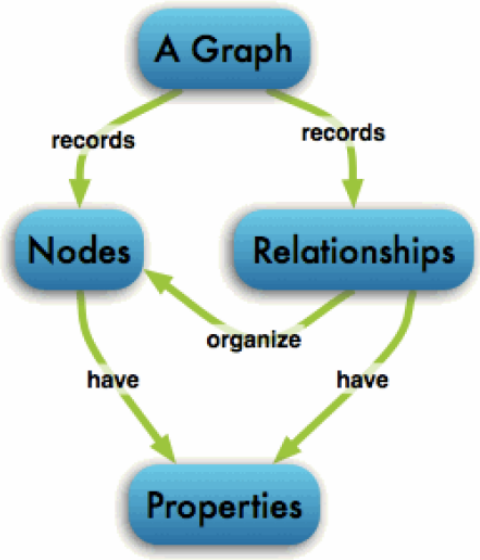
  - *Domain* of solution: set of variables that is mapped by the solution to an RDF term

- Somehow merge the solutions of two operators, until all operators are covered

CHAPTER 3

Property Graphs

- Need to capture large amounts of data represented as graphs
- **Guiding Principles**
  - Uniqueness and identity
  - Data types
  - Properties on relationships
  - Relations are lose w.r.t. types
  - Linked lists



3.1 Formal Specification

- A property graph is a (weighted) directed, vertex-labelled, edge-labelled multigraph with self-edges, where edges have their own identity
- Let $G = (V, E, P, \lambda, \lambda_E, \omega)$ be a property graph

V Set of all nodes

E Multiset of directed edges (relationships)

P Set of all properties

$\lambda_V : V \to 2^P$ returns the property of a node

$\lambda_E : E \to 2^P$ returns the property of an edge

$\omega : E \to [0, 1]$ returns the weight of an edge

3.2 Components

Note: Different incarnations. Here we use the openCypher definition.

- **Entity**
  - Has unique identity (index)
  - Identity used to check equality
  - Can be assigned a set of properties
- **Node**
  - Basic entity of the graph
  - Can exist in and of itself
  - Can have a set of unique labels (token)
  - Can have zero or more incoming/outgoing relationships
- **Relationship**
  - Encodes the relation between a source and a target node
  - Has exactly one **relationship type**
- **Property**
  - Key-value pair
  - Key uniquely identifies an entity's property
  - Value is instantiation of some value type

3.2.1 Comparison

	RDF Graphs	Property Graphs
Data Model	Graphs	Graphs
Syntax	Standardized	Varied
Semantics	Explicit	Partly implicit
Inference	Supported	Not supported
Design purpose	General and domain-specific knowledge	Graph algorithms
Query language	SPARQL	Several (e.g., Cypher)

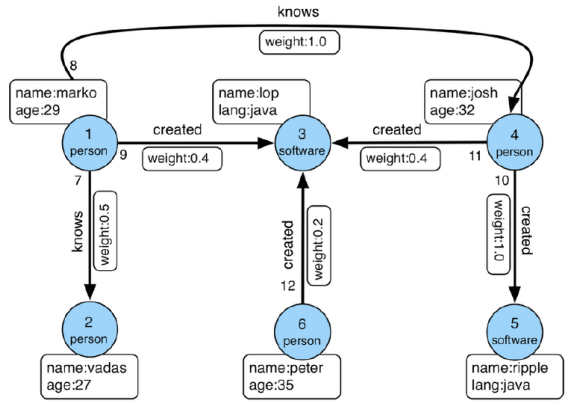
3.3 Querying Property Graphs

3.3.1 Paradigms

Imperative Describe which data is to be collected by describing how data is to be collected (unique semantics) ⇒ Gemlin

Declarative Describe what data is to be collected, How is decided by query engine ⇒ Cypher (most popular), SPARQL, GraphQL

3.3.2 Gremlin



Example

```
g.V(1).out().map(values('name'))
⇒ lop, vadas, john
```

- Basic concept: *Traverse* input graph using *chains of atomic operations* (steps) to gather relevant information
- Basic structure: **Traversal**
  - Fluent
    - Built using method chaining
    - Defined trough return value of called method
    - Self-referential
  - Monadic
    - Associative operations [**ToDo**: *why is this good?*]
    - Existence of zero element
- Basic components:
  - **Step**: $S \to E$: Individual function applied to S yield E
    - Returns traversal
    - Executes traversal and returns a result, if step is value check or transformations for further processing is needed

- **TraversalStrategy**: Alters the execution of the traversal [**ToDo**: *eg?*]
- **TraversalSideEffects**: Key/value pairs to store global information
- **Traverser**: The object propagating trough the Traversal, stores intermediary travel information
  - Current traversed object
  - Current path traversed by the traverser
  - Number of times the traverser has gone through the current loop
  - Number of objects represented by this traverser
  - Local data structure associated with this traverser
  - Side-effects associated with the traversal
- Steps are chained within a traversal

- **Traversal Strategy** analyzes a traversal and, if the traversal meets its criteria, can mutate it accordingly

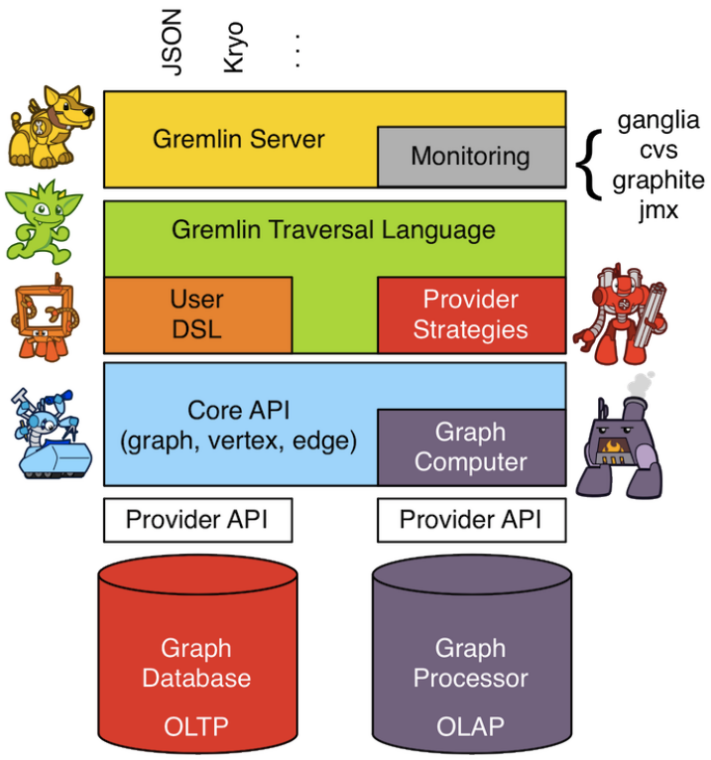
  Decoration There is an application-level feature that can be embedded into the traversal logic

  Optimization There is a more efficient way to express the traversal at the TinkerPop3 level

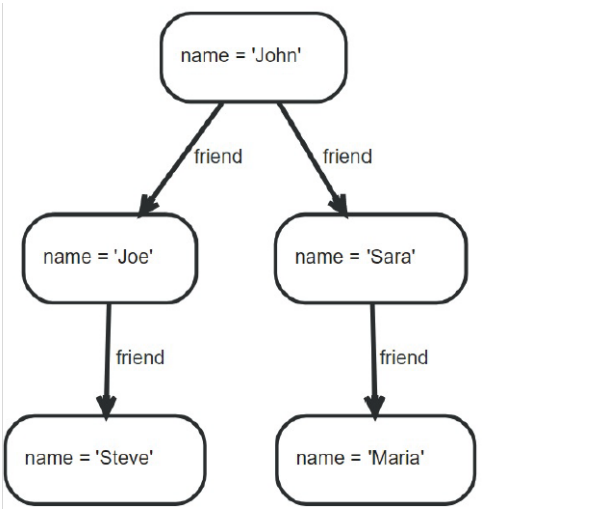
  Provider Optimization There is a more efficient way to express the traversal at the graph system/language/driver level

  Finalization There are some final adjustments/cleanups/analyses required before executing the traversal

  Verification There are certain traversals that are not legal for the application or traversal engine



3.3.3 Cypher



Example

```
MATCH (user)-[:friend]->(follower)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve']
    AND follower.name =~ 'S.*'
RETURN user.name, follower.name
```

- Describe *what* we want but not *how* to compute it
- Related to SQL ⇒ Clauses chained together
- Basic clauses to **read from graphs**

  `MATCH` Find graph pattern

  `WHERE` Conditions that must be fulfilled

`RETURN` Output specification

- Basic clauses to **write graphs**

  `CREATE` **and** `DELETE` For nodes and relationships

  `SET` **and** `REMOVE` Set/remove values to/from properties and labels to/from nodes

  `RETURN` Match existing or create new nodes and patterns

- **Query Paradigms**
  - Querying and Updating
    - Queries consist of part marked by `WITH` (horizon)
    - A query part *cannot* both math and update the graph
    - Every part can either read and match, or make updates on the graph
    - Matching is *lazy*
    - `RETURN` is not a part of a query, as it does not read or write
      - Can be combined with other clauses such as `SKIP`, `LIMIT`, `ORDER By`
  - Transactions
    - Cypher is **transactional** on updates
      - An updating query will always either fully succeed or not succeed at all
      - Automatic creation of transaction and commit if no transaction exists
      - New queries run in existing transaction if one exists
      - Several queries can run one transaction, commit occurs once all queries in the transaction are completed
- Uniqueness
  - Matches where the same graph relationship is found multiple times in a single pattern are not included by default
- Syntax
  - Values and Types
    - Property Types (Number, String, Boolean)
      - Can be returned from Cypher queries
      - Can be used as parameters
      - Can be stored as properties
      - Can be constructed with Cypher literals
    - Structural Types (Nodes, Relationships, Paths (nodes+relationships))
      - Can be returned from Cypher queries
      - Cannot be used as parameters
      - Cannot be stored as properties
      - Cannot be constructed with Cypher literals

  - Composite Types (Lists, Maps (Key = string, value = any type)
    - Can be returned from Cypher queries
    - Can be used as parameters
    - Cannot be stored as properties
    - Can be constructed with Cypher literals
- Expressions

Summary

- Datatypes (integer, string, boolean), e.g., "Hello", 2
- Variables, e.g., x
- Properties, e.g., x.prop, n.[n.zip + n.city]
- Parameters, e.g., `$param0`
- Lists of expressions, e.g., `['a', 'b']`
- Functions and aggregates, e.g., `nodes`(p), `count`(*)
- Paths, e.g., `(a) --> () <--(b)`
- Regular expressions, e.g., `a.name = ~ 'Tob.*'`
- Case expressions (coming up)

- Parameters
  - Can be used for
    - Literals and expressions
    - Node and relationship IDs
    - Explicit indexes
  - Cannot be used for the following, as they influence the query plan heavily
    - Property keys
    - Relationship types
    - Labels
- Operators

General	`DISTINCT`, `.` for property access, `[]` for dynamic property access
Mathematical	`+`, `-`, `*`, `/`, `%`, `^`
Comparisons	`=`, `<>`, `<`, `>`, `<=`, `>=`, `IS NULL`, `IS NOT NULL`
String-specific comparisons	`STARTS WITH`, `ENDS WITH`, `CONTAINS`
Booleans	`AND`, `OR`, `XOR`, `NOT`
Strings	`+` for concatenation, `=~` for regex matching
Lists	`+` for concatenation, `IN` to check existence of an element in a list, `[]` for accessing element(s)

- Pattern
  - Node variables in brackets
  - Typed nodes with multiple types allowed
  - Directed and undirected edges also allowed

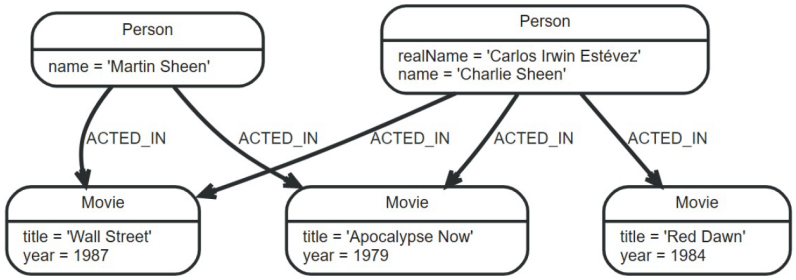
```
(a:User:Admin)-->(b)--(c)
```

- Expectations on relations can be defined
- Types can be given for relations
- Relation names can be omitted
- Number of repetitions can also be defined

```
(a)-[r:TYPE1|TYPE2*2..5]->(b)
(a)-[{blocked: false}]->(b)
```

- Extensive support for lists and maps
- Output of pattern comprehension

```
MATCH (a:Person { name: 'Charlie Sheen' })
RETURN [(a)-->(b) WHERE b:Movie | b.year] AS years
```



- Reading clauses

  `MATCH` Specify the patterns to search for in the database

  `OPTIONAL MATCH` Specify the patterns to search for in the database while using nulls for missing parts of the pattern

- Projection clause

  `RETURN .. {AS}` Defines what to include in the query result set

  `WITH .. {AS}` Allows query parts to be chained together

  `UNWIND .. {AS}` Expands a list into a sequence of rows

  `WHERE` Adds constraints to the patterns in a `MATCH` or `OPTIONAL MATCH` clause or filters the results of a `WITH` clause

  `ORDER BY {ASC {ENDING} | DES {Ending}}` A sub-clause following `RETURN` or `WITH`, specifying that the output should be sorted in either ascending (the default) or descending order

  `SKIP` Defines from which row to start including the rows in the output

  `LIMIT` Constrains the number of rows in the output

- Writing clauses

  `CREATE` Create nodes and relationships

  `DELETE` Delete graph elements (nodes, relationships or paths). Any node to be deleted must also have all associated relationships explicitly deleted

`DETACH DELETE` Delete a node or set of nodes. All associated relationships will automatically be deleted

`SET` Update labels on nodes and properties on nodes and relationships

`REMOVE` Remove properties and labels from nodes and relationships

`FOREACH` Update data within a list, whether components of a path, or the result of aggregation

`MERGE` Ensures that a pattern exists in the graph. Either the pattern already exists or it needs to be created

`.. ON CREATE` Specifies the actions to take if a pattern needs to be created (used with `MERGE`)

`.. ON MATCH` Specifies the actions to take if the pattern already exists (used with `MERGE`)

`CALL {..YIELD}` Invoke a procedure deployed in the database and return any results

- Other Clauses

`UNION (ALL)` Combines the result of multiple queries into a single result set. Duplicates are removed (kept)

`LOAD CSV` Loads CSV files

`.. USING PERIODIC COMMIT` Prevents out-of-memory errors when using `LOAD CSV`

`CREATE | DROP CONSTRAINT` Create or drop an index on all nodes with a particular label and property

`CREATE | DROP INDEX {.. YIELD}` Create or drop a constraint pertaining to either a node label or relationship type and property

- Functions
  - Predicate functions: `all(), any(), exists(), ..`
  - Scalar functions: `id(), timestamp(), ..`
  - Aggregating functions: `avg(), collect(), ..`
  - List functions: `tail(), nodes() (for paths), ..`
  - Mathematical functions: `rand(), sqrt(), ..`
  - String functions: `reverse(), toLower(), ..`
  - Spatial functions: `point(), distance(), ..`

- Query Planning
  - Based on indexes and selectivity approximations
  - Can be modified by enforcing use of indexes

3.3.4 GraphQL

- Basics
  - Supports data fetching and manipulation
  - Data model: Graph of objects
  - Query resembles expected data
  - Strong typing system guarantees shape and nature of responses
  - Introspective, i.e. type system can be queried using GraphQL
  - Transport-independent
- Subqueries: Fields can be matched on objects in an answer ⇒ no need for several API calls
- Attribute matching
- Fragments allow to compact query code
- Variables: Explicit function name for code (names including default definition)
- Directives for code inclusion (include skip)
- **Type System**
  - Basic constructs
    - Object type
    - Fields (scalar, arrays, other object types)
    - Named arguments for fields
    - `!` stands for non-nullable
    - Scalars (Int, Float, String, Boolean, ID)
    - Enumerations
    - Lists
  - Interfaces and implementation thereof

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}
```

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}
```

- Write queries
  - Mutations for writing

```
mutation CreateReviewForEpisode($ep: Episode!,
  $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}


{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

3.3.5 Comparison

- Gremlin
  - Relies on traversal paradigm
  - Commonly rather verbose
  - Allows for extension with DSL and user-specific optimizations
- Cypher
  - High support
  - Easy to combine with DB technologies (optimizers, etc.)
- GraphQL
  - Implements graph matching
  - Developed to simplify API-like access to data

CHAPTER 4

OWL

language for **describing an ontology**

4.1 Ontology

"An ontology is a **formal specification** of a shared conceptualization **of a domain of interest**." (Gruber, 1993)

- Machine-readable
- Describes concepts agreed upon
- Within a particular domain

4.2 Why RDFS is not suitable for complex models

- Cannot express that `:ancestor` is transitive or `:spouse` is a 1-1 relation
- No negation

4.3 Knowledge Representation

- two kinds of knowledge representations
  1) logic based (e.g. description logics (DL))
     - based on first order logic (FOL) ⇒ consistent semantics ⇒ entailment and reasoning
  2) non logic based
     - much more intuitive for humans
     - BUT: no consistent semantics ⇒ no entailment and reasoning
- DLs are fragments of FOL
- OWL depends on a description logic DL

- FOL are very expressive
- BUT: too bulky for modeling and only **semi-decidable** (entailment algorithms might not terminate)

4.4 Description Logics (DL)

- idea: use **simple descriptions** to create **more complex descriptions** using **constructors**
- (most) DLs are **decidable** while still having enough expression power
  - they are usually also **feasible**, i.e. decidable in a reasonable amount of time
- most simple DL: ALC := attribute language with complement

4.4.1 ALC: Attribute Language with Complement

Basic Components

- three main building blocks:
  1) **Classes** (Concepts), e.g. `Student`, `Book`
  2) **Roles** (Properties), e.g. `bornIn`, `worksFor`
     - describe relations between classes
  3) **Individuals** (Objects), e.g. `Steven`, `Harry Potter 1`
- **Signature/Vocabulary**: Sets containing classes, roles and individuals
- examples:
  - `Student(Steven)`: Steven is a student
  - `Book(Harry Potter 1)`: Harry Potter 1 is a book
  - `reads(Steven, Harry Potter 1)`: Steven reads Harry Potter 1

Formal (Syntax) Definition

- ALC consists of
  1) **Atomic Types**
     - Concept Names: $A, B, C, \ldots$
     - Two special concepts:
       - Top concept: $\top$ (each other concept is a subclass of this)
       - Bottom concept: $\bot$ (is a subclass of each other class and no concept is subclass of bottom concept)
     - Role Names: $r, s, t, \ldots$
  2) **Constructors** (allow to combine atomic types to complex types) (let C, D be concepts)

- $\neg C$ (Negation / Complement)
- $C \sqcap D$ (Conjunction)
- $C \sqcup D$ (Disjunction / Union)
- $\exists r.C$ (existential restriction)
- $\forall r.C$ (universal restriction)

3) **Terminological Knowledge Box (TBox)**: finite set of terminological axioms of the form
   - $C \sqsubseteq D$
   - $C \equiv D$

   where C and D are complex classes

4) ABox $\mathcal{A}$: Description of individuals
   - **Assertion Box**: Finite set of assertional axioms (facts) of the form
     - $C(a)$ (conceptional facts)
     - $R(a, b)$ (relational facts)

     where C is a complex class, R is a role and a, b are individuals

- ALC knowledge base $\mathcal{K} = (TBox, ABox)$ composed of TBox and ABox

Examples

- complex classes can be built according to

  •C,D::= A | ⊤ | ⊥ | ¬C | C⊓D | C⊔D | ∃R.C | ∀R.C

- bindings work as follows

  •**Strict Binding** of the Range of a Role to a Class
  •Examination ⊑ ∀hasSupervisor.Professor
  •*An Examination must be supervised by a Professor*
  •(∀x)(Examination(x) → (∀y)(hasSupervisor(x,y) → Professor(y)))

  •**Open Binding** of the Range of a Role to a Class
  •Examination ⊑ ∃hasSupervisor.Person
  •*Every Examination has at least one supervisor (who is a person)*
  •(∀x)(Examination(x) → (∃y)(hasSupervisor(x,y) ∧ Person(y)))

Semantics

- semantics (i.e. entailment) defined using interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where
  - **Domain** $\Delta^{\mathcal{I}}$: Non-empty set of individuals
  - **Interpretation Mapping** $\cdot^{\mathcal{I}}$ which maps
    - every individual a to an element of the domain $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
    - every concept name A to a subset of the domain $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
    - every role r to a set of pairs of elements of the domain $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
- Interpretation is extended to **complex expressions**
  - $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
  - $\bot^{\mathcal{I}} = \emptyset$
  - $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
  - $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
  - $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
  - $(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \exists y \in C^{\mathcal{I}} \text{ with } (x, y) \in r^{\mathcal{I}}\}$
  - $(\forall r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | (x, y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
- and **axioms**
  - $\mathcal{I} \models C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ (we say I entails $C \sqsubseteq D$
  - $\mathcal{I} \models C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$
  - $\mathcal{I} \models C(a)$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$
  - $\mathcal{I} \models r(a, b)$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$
- Interpretations which fulfil a set of axioms are called **models**

Complete Knowledge Base Example

TBox $\mathcal{T}$

- $Male \equiv \neg Female \sqcap Person$
- $Female \sqsubseteq Person$
- $Mother \equiv Female \sqcap \exists hasChild.\top$

ABox $\mathcal{A}$

- $Male(STEPHEN)$
- $\neg Male(MONICA)$
- $Female(JESSICA)$
- $hasChild(STEPHEN, JESSICA)$

Let $\mathcal{I}$ be an interpretation with

- $Male^{\mathcal{I}} = \{STEPHEN, JESSICA\}$
- $Female^{\mathcal{I}} = \{JESSICA, MONICA\}$
- $Mother^{\mathcal{I}} = \{\}$
- $Person^{\mathcal{I}} = \{JESSICA, MONICA, STEPHEN\}$
- $hasChild^{\mathcal{I}} = \{\langle STEPHEN, JESSICA\rangle\}$

Does $\mathcal{I} \models \mathcal{T}$ und $\mathcal{I} \models \mathcal{A}$?

Let $\mathcal{I}$ be an interpretation with

- $Male^{\mathcal{I}} = \{STEPHEN\}$
- $Female^{\mathcal{I}} = \{JESSICA, MONICA\}$
- $Mother^{\mathcal{I}} = \{\}$
- $Person^{\mathcal{I}} = \{JESSICA, MONICA, STEPHEN\}$
- $hasChild^{\mathcal{I}} = \{\}$

Does $\mathcal{I} \models \mathcal{T}$ und $\mathcal{I} \models \mathcal{A}$?

Let $\mathcal{I}$ be an interpretation with

- $Male^{\mathcal{I}} = \{STEPHEN\}$
- $Female^{\mathcal{I}} = \{JESSICA, MONICA\}$
- $Mother^{\mathcal{I}} = \{MONICA\}$
- $Person^{\mathcal{I}} = \{JESSICA, MONICA, STEPHEN\}$
- $hasChild^{\mathcal{I}} = \{\langle MONICA, STEPHEN\rangle, \langle STEPHEN, JESSICA\rangle\}$

Does $\mathcal{I} \models \mathcal{T}$ und $\mathcal{I} \models \mathcal{A}$?

Correspondence to OWL

The $\mathcal{ALC}$ operators are equivalent to the following OWL expressions

- $\top$: `owl:Thing`
- $\bot$: `owl:Nothing`
- $\neg$: `owl:complementOf`
- $\sqcup$: `owl:unionOf`
- $\sqcap$: `owl:intersectionOf`
- $\exists$: `owl:someValuesFrom`
- $\forall$: `owl:allValuesFrom`

4.4.2 Extension of $\mathcal{ALC}$ to $\mathcal{SHOIN}(\mathcal{D})$

- $\mathcal{ALC}$ can be extended in the following sense

- $\mathcal{ALC}$: Attribute Language with Complement
- $\mathcal{S}$: $\mathcal{ALC}$ + Transitivity of Roles
- $\mathcal{H}$: Role Hierarchies
- $\mathcal{O}$: Nominals
- $\mathcal{I}$: Inverse Roles
- $\mathcal{N}$: Number restrictions ≤n R etc.
- $\mathcal{Q}$: Qualified number restrictions ≤n R.C etc.
- $(\mathcal{D})$: Datatypes
- $\mathcal{F}$: Functional Roles
- $\mathcal{R}$: Role Constructors

- $\mathcal{S}$: **Transitivity** (OWL: `owl:TransitiveProperty`)
  - Transitivity of role r expressed as $Trans(r)$
  - $Trans(r)$ holds in an interpretation $\mathcal{I}$ iff $(x,y) \in r^{\mathcal{I}} \wedge (y,z) \in r^{\mathcal{I}} \rightarrow (x,y) \in r^{\mathcal{I}}$
- $\mathcal{H}$: **Role Hierarchies** (OWL: `rdfs:subPropertyOf`)
  - For roles r, s:
    - Role inclusion axioms (RIA): $r \sqsubseteq s$
    - $r \equiv s$ is short hand for $r \sqsubseteq s$ and $s \sqsubseteq r$
    - $r \sqsubseteq s$ holds in an interpretation $\mathcal{I}$ iff $r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
- $\mathcal{O}$: **Nominals** (OWL: `owl:oneOf`)
  - Let $a_1, \ldots, a_n$ be individuals
  - A nominal $\{a_1, \ldots, a_n\}$ is a concept with the following semantics:

$$(\{a_1, \ldots, a_n\})^{\mathcal{I}} = \{a_1^{\mathcal{I}}, \ldots, a_n^{\mathcal{I}}\} \tag{4.1}$$

- $\mathcal{I}$: **Inverse Roles** (OWL: `owl:inverseOf`)
  - A role is either a role name r or an inverse role r^-
  - Semantics of inverse roles: $(r^-)^{\mathcal{I}} = \{(y,x) | (x,y) \in r^{\mathcal{I}}\}$
- $\mathcal{N}$: **Unqualified Cardinality Restrictions**
  - Cardinality restrictions on r have the following semantics
    - $(\geq nr)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} : |\{y \in \Delta^{\mathcal{I}} | (x,y) \in r^{\mathcal{I}}\}| \geq n\}$
    - $(\leq nr)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} : |\{y \in \Delta^{\mathcal{I}} | (x,y) \in r^{\mathcal{I}}\}| \leq n\}$
    - $(= nr)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} : |\{y \in \Delta^{\mathcal{I}} | (x,y) \in r^{\mathcal{I}}\}| = n\}$
  - OWL constructs
    - $\geq nr$ = `owl:minCardinality`
    - $\leq nr$ = `owl:maxCardinality`
    - $= nr$ = `owl:exactCardinality`
- $\mathcal{F}$: **Functionality** (OWL: `owl:FunctionalProperty`)
  - Functionality of axiom for role r expressed as $Func(r)$
  - $Func(r)$ holds in interpretation $\mathcal{I}$ iff $(x,y) \in r^{\mathcal{I}} \wedge (x,z) \in r^{\mathcal{I}} \rightarrow y = y$

4.4.3 Inference and Reasoning

Open- vs. Closed-World-Assumption

Open World Assumption (OWA)

- Existence of other individuals is possible as long as their non-existence is not made explicit
- **Basic assumption in DL/OWL**

Closed World Assumption (CWA)

- The knowledge base contains all individuals and facts
- Used in databases and in Prolog

Reasoning Types

1) **deduction**
   - begins with a set of rules and generates inferences that must hold
   - example: When it rains, things outside get wet. The grass is outside. ⇒ When it rains, the grass gets wet.
2) **induction**
   - begins with observations and generates general rules (rules are **probable** but not **certain**)
   - example: The grass got wet numerous times when it rained ⇒ the grass always gets wet when it rains.
3) **abduction**
   - starts with a limited number of observations
   - generates plausible explanations for these facts
   - example When it rains, the grass gets wet. The grass is wet ⇒ it might have rained.

Notation

- Let $\mathcal{I}$ be an interpretation, $\mathcal{T}$ a TBox, $\mathcal{A}$ be an ABox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a knowledge base. The following holds:
- $\mathcal{I}$ is a model of $\mathcal{T}$, iff $\mathcal{I} \models \alpha$ for every axiom $\alpha \in \mathcal{T}$, denoted $\mathcal{I} \models \mathcal{T}$.
- $\mathcal{I}$ is a model of $\mathcal{A}$, iff $\mathcal{I} \models \alpha$ for every axiom $\alpha \in \mathcal{A}$, denoted $\mathcal{I} \models \mathcal{A}$.
- $\mathcal{I}$ is a model of $\mathcal{K}$, iff $\mathcal{I} \models \mathcal{T}$ and $\mathcal{I} \models \mathcal{A}$.
- An axiomα is a consequence of K, denoted $K \models \alpha$, iff every model $\mathcal{I}$ of K is a model of α.

Inference Problems

- several questions we can ask, i.e. problems we can formulate

Satisfiability of a concept
- Notation: $\mathcal{K} \not\models C \equiv \bot$
- Tests whether a concept C is satisfiable in $\mathcal{K}$, i.e., whether a model $\mathcal{I}$ of $\mathcal{K}$ exists, such that $C^{\mathcal{I}} \neq \emptyset$.

Subsumption
- Notation: $\mathcal{K} \models C \sqsubseteq D$
- Tests whether a concept C is subsumed by a concept $\mathcal{D}$ w.r.t $\mathcal{K}$, i.e., whether $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{K}$.

Satisfiability (Consistency)
- Notation: $\mathcal{K} \not\models \top \sqsubseteq \bot$
- Tests whether $\mathcal{K}$ is consistent, i.e., whether a model of $\mathcal{K}$ exsits.

Instance Test
- Notation: $\mathcal{K} \models C(a)$
- Tests whether the class assignment $C(a)$ is fulfilled in $\mathcal{K}$, i.e., whether $a^{\mathcal{I}} \in C^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{K}$.

Retrieval
- Notation: $\{a|\mathcal{K} \models C(a)\}$
- Finds all individuals a, which belong to the concept C in $\mathcal{K}$, i.e., find all a for a given C that are such that $a^{\mathcal{I}} \in C^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{K}$.

Realisation
- Notation: $\{C|\mathcal{K} \models C(a)\}$
- Find all atomic concepts C, which contain an individual a w.r.t $\mathcal{K}$, i.e., find all C for a given a with $a^{\mathcal{I}} \in C^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{K}$.

- solve problems by mapping them to **satisfiability tests**
- more precisely: we try to detect **inconsistencies** in our knowledge base by combining it with instantiations to proof its unsatisfiability

  - All reasoning services can be mapped to satisfiability tests:
  - Concept satisfiability
    - $K \not\models C \equiv \bot$
    - Equivalent to "Does an x exist which satisfies $K \cup \{C(x)\}$?"
  - Subsumption
    - $K \models C \sqsubseteq D \longleftrightarrow K \cup \{C \sqcap \neg D(x)\}$ is unsatisfiable.
  - Instance test
    - $K \models C(a) \longleftrightarrow K \cup \{\neg C(a)\}$ is unsatisfiable

- algorithm to perform satisfiability checks: **tableau algorithm**

4.4.4 Tableau Algorithm

[**ToDo**: *optimize*]

- Recall: A concept is satisfiable if a model $\mathcal{I}$ for that concept exists
- Need a constructive decision algorithm for the generation of models $\Rightarrow$ Tableau Algorithm

- Stepwise approach for proving satisfiability; Steps:
  1) Transform the concept into **negation normal form** (NNF)
     - A concept is in NNF when negations in the concept can only be found next to atoms
     - Every $\mathcal{ALC}$ concept can be transformed into NNF using some rules
  2) **Use rules** in an arbitrary order as long as possible
  3) The concept is **satisfiable** iff a tableau free of contradiction was derived once no further rule could be employed
- A **tableau** (completion graph) for $\mathcal{ALC}$ concept is a directed label graph $G = \langle V, E \rangle$
  - Every node $x \in V$ is a set of concepts $L(x)$
  - Every edge $\langle x, y \rangle \in E$ is labelled with a set of roles $L(\langle x, y \rangle)$
- A tableau G contains a **contradiction** if
  - $\{A, \neg A\} \in L(x)$ for an atomic concept A
  - $\bot \in L(x)$
  - $\neg\top \in L(x)$
- A tableau is **completed** if no further rules can be used

$\sqcap$ Rule
- If $C \sqcap D \in L(v)$, for any $v \in V$ und $\{C, D\} \not\subseteq L(v)$,
- Then $L(v) := L(v) \cup \{C, D\}$

$\sqcup$ Rule
- If $C \sqcup D \in L(v)$, for any $v \in V$ and $\{C, D\} \cap L(v) = \emptyset$,
- Then choose $X \in \{C, D\}$ and set $L(v) := L(v) \cup \{X\}$
- Results into two different tableaus

$\exists$ Rule
- If $\exists r.C \in L(v)$ for any $v \in V$ and there is no r-child v' of v such that $C \in L(v')$,
- Then $V := V \cup \{v'\}, E := E \cup \{\langle v, v' \rangle\}, L(v') := \{C\}$ and $L(\langle v, v' \rangle) := \{r\}$ for a new node v'

$\forall$ Rule
- If $v, v' \in V, v'$ is a r-child of $v, \forall r.C \in L(v)$ and $C \notin L(v')$,
- Then $L(v') := L(v') \cup \{C\}$.

- Extended version to check satisfiability of $\mathcal{ALC}$ TBoxes
  - An $\mathcal{ALC}$ TBox only contains axioms of the form $C \sqsubseteq D$
  - Each of these axioms is equivalent to $\neg C \sqcup D$
  - **Internalisation:** compression of a whole TBox into an axiom

$$\mathcal{T} = \{C_i \sqsubseteq D_i | 1 \leq i \leq n\} \text{ is the same as } C_\mathcal{T} = \prod_{1 \leq i \leq n} (\neg C_i \sqcup D_i) \tag{4.2}$$

  - A supplementary rule is as follows:
    - If $C_\mathcal{T} \in L(v)$ for any $v \in V \Rightarrow L(v) := L(v) \cup \{C_\mathcal{T}\}$
- **Blocking:** Check for cycles
  - **Goal:** Ensure that the Tableau algorithm terminates
  - **Approach:** Check for cycles created by the $\mathcal{T}$ rule
  - **Result:** Graph is finite
  - A node $v' \in V$ is **blocked directly** by a node $v \in V$ iff
    1) v is a predecessor of v'

2) $L(v') \subseteq L(v)$

3) There is no directly blocked $v'' \in V$ such that v'' is a predecessor of v'

- A node $v' \in V$ is **blocked** if
  - v' is blocked directly, or
  - There is a directly blocked node v which is a predecessor of v'
- The application of the $\exists$ rule is limited to nodes which are not blocked
- Example:

- Assume the TBox $T = \{A \sqsubseteq \exists r.A\}$
- We aim to check whether the concept A is satisfiable
- We get the following conflict-free Tableau

$$L(x) = \{A, \neg A \sqcup \exists r.A, \exists r.A\}$$
$$L(y) = \{A, \neg A \sqcup \exists r.A, \exists r.A\}$$
$$L(\langle x, y \rangle) = \{r\}$$

  where y is blocked directly by x
- We have a finite model given that
  - blocked nodes are not elements of the model
  - an edge from a node v to a directly blocked node v' is replaced by an "edge" from v to the node which blocks v' directly
- For our example, we get $\Delta^{\mathcal{I}} = \{x\}, A^{\mathcal{I}} = \{x\}, r^{\mathcal{I}} = \{\langle x, x \rangle\}$

4.5 Overview: OWL

- OWL depends on a description logic (DL)
  - DLs are fragments of First-Order logics (FOL)
  - we do not use FOLs here as they are only semi-decidable and extremely complex
- two versions of OWL (1,2)
  - OWL (1) depends on DL $\mathcal{SHOIN}(\mathcal{D})$
    - three variations: OWL Lite $\subseteq$ OWL DL $\subseteq$ OWL Full
      - **OWL DL**: no reification (i.e. not RDFS compatible), but **decidable**
      - OWL Full: superset of RDFS, but only **semi-decidable**
  - OWL 2 depends on DL $\mathcal{SHROIQ}(\mathcal{D})$
    - all new special OWL 2 variations are subsets of OWL (2) DL
- **Decidability**: question of the existence of an effective method for determining membership in a set of formulas
  - existence of an algorithm that will return a boolean true or false value wrt. to decision problem that is correct (instead of looping indefinitely, crashing, returning "don't know" or returning a wrong answer). (Wikipedia)

4.6 OWL Documents

- Consist of a set of axioms
- Axioms can be expressed as set of RDF triples
- Dedicated formats commonly easier to read and even process
- Syntax
  - Turtle: Not developed for OWL, therefore complex expressions are difficult to read
  - Manchester Syntax: Designed for OWL (W3C standard since OWL 2)
- Building blocks of ontology axioms

  Individuals/objects Concrete elements of the model

  Classes/concepts Sets of objects

  Roles/properties Links between two individuals

4.6.1 Role

- Two types of roles

  Concrete role (datatype properties) are connected to concrete values, i.e. literals

  Abstract roles (object properties) are linked to URIs, i.e. they connect resources
- [**ToDo**: *Roles are generally not functional?? Persons can be men or women?!*]
- Characteristics
  - Domain
  - Range
  - Transitivity, i.e. $r(a,b) \wedge r(b,c) \rightarrow r(a,c)$
  - Symmetry, i.e. $r(a,b) \rightarrow r(b,a)$
  - Functionality, i.e. $r(a,b) \wedge r(a,c) \rightarrow b = c$
  - Inverse functionality, i.e. $r(a,b) \wedge r(c,b) \rightarrow a = c$
- Relations (Turtle/Manchester Syntax)
  - `rdfs:subClassOf` / `SubClassOf`
    - `A rdfs:subClassOf B` iff all instances of `A` are instances of `B`
  - `owl:disjointWith` / `DisjointWith`
    - `A owl:disjointWith B` iff `A` have no common instances `B`
    - Pairwise disjoint: `owl:AllDisjointClasses` / `DisjointClasses`
  - For classes `owl:equivalentClass` / `EquivalentTo` or for properties `owl:equivalentProperty` or for individuals `owl:sameAs`
    - Class `A` is equivalent to class `B` iff all instances of `A` are also instances of `B` and vice-versa

- `owl:differentFrom` / `DifferentIndividuals`
- `[..,..]` / `{..,..}` (List of all individuals in a class: Nominals)
- Boolean combinations
  - Conjunction: `owl:intersectionOf` / `A and B`
  - Disjunction: `owl:unionOf` / `A or B`
  - Disjoint Union: `owl:disjointUnionOf` / `DisjointUnionOf`
  - Negation: `owl:complementOf` / `not A`
- `owl:allValuesFrom` / `only A`
- `owl:someValuesFrom` / `some A`
- `owl:maxCardinality` / `exactly/min/max X` (Cardinality)
- `owl:hasValue` / `value A`

4.6.2 Semantics

- Semantics of OWL are based on Description Logics (DL)

CHAPTER 5

Knowledge Extraction

5.1 Situation

- Most data generated is unstructured
- Structured sources are hard to integrate

⇒ **Lift this data** for knowledge-driven applications

- we do **knowledge extraction from text** ⇒ transform text into structured knowledge

5.1.1 Challenges

- Coreference resolution
- Emerging entities
- Entity disambiguation, open relation extraction, time scoping

5.1.2 Goal and Approach

Given document d and knowledge graph G: find all facts in d which reference resources in G

- 3 main steps
  1) **Named Entity Recognition**: **Recognize** mentions of class instances in d
  2) **Named Entity Disambiguation and Linking**: **Disambiguate** these mentions and **link** them to resources in G
  3) **Relation Extraction**: Compute all **relations** which link these resources
- In RDF
  - Detect all resource mentions m
  - Map mentions m to resources $r \in V(G)$
  - Detect triples (r, p, r') and add them to G
- Two major paradigms

- Closed Knowledge Extraction
- Open Knowledge Extraction

5.2 Named Entity Recognition (NER)

Recognize mentions of class instances in d

5.2.1 Definition

Given a set of classes $C \in K$, Detect all mentions $m \in d$ with

1) r `rdfs:label` m
2) r `rdf:type` c
3) $c \in C$

5.2.2 Approach: Dictionaries

- Build extensive dictionary of domain
- Detection = string matching
- Example: DBpedia Spotlight
  - Wikipedia as corpus
  - Labels from page title, redirects, Wikilinks, disambiguation pages
- Pros: time-efficient and fast learning of priors
- Cons: Can only spot known substrings and do not support unknown entities

5.2.3 Approach: Rules

- Use context to detect entities
- Create rules which model context
- Pros: Easier to maintain, can be mined from data and is understandable for humans
- Cons: Rules can get very complex and the model can become large
- Machine Learning Approaches
  - Generally supervised machine learning
  - Classification: Reduce task to classification problem
    - Use part of speech and context information as features

    $\Rightarrow$ Problem: Make limited use of sequence information

5.2.4 Approach: Sequence Learning

- NER is modelled as Sequence Learning problem via Hidden Markov Model
  - Hidden states: named entity class labels
  - Output symbols: words of sentence
- Given a sentence, output the most probable sequence of NER tags

⇒ State Sequence Decoding problem solved via **Viterbi Algorithm**

5.2.5 Improvement: Ensemble Learning

- Trains different NER taggers
- Aggregates results to output final result
- E.g. FOX (dice group)

5.3 Entity Linking

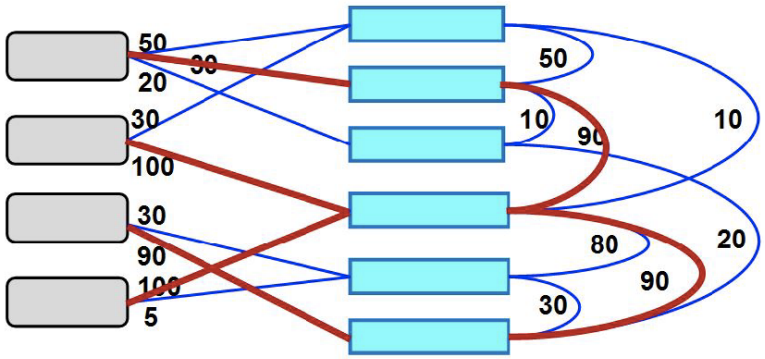
Disambiguate these mentions and link them to resources in G

5.3.1 Definition

Given a knowledge graph $G = (V, E)$, document d and set of mentions M: compute a mapping $l : M \to V$

5.3.2 General Approach Idea

- reduce joint mapping to a **graph problem**:
  - nodes: entities (from knowledge base) and mentions (from text)
  - weighted edges between mentions and entities, capturing context similarities
    - in addition usually also a-priori similarities between mentions and entities are considered
  - weighted edges among entities, capturing coherence
- goal: identify **dense subgraph** containing **exactly one** (or at most one) **(mention-entity) edge** for each mention, (giving the most likely disambiguation)
- note: problem is NP-hard, so we only have approximate solutions



5.3.3 Entity Linking Approach: AIDA

Idea

- compute dense subgraph $G' = (E' = E'_{m,e} \cup E'_{e,e}, V')$ which maximizes a weighted objective function

$$f(G') = \alpha \cdot \sum_{(m,e) \in E'_{m,e}} prior(m,e) + \beta \cdot \sum_{(m,e) \in E'_{m,e}} contextSim(m,e) + \gamma \cdot \sum_{(e,e) \in E'_{e,e}} coherence(e,e) \tag{5.1}$$

  where

  - $E'_{e,e}$: sets of edges connecting entities
  - $E'_{m,e}$: sets of edges connecting mentions and entities

Computation of Prior (A-Priori)

- idea: prominence of a name wrt. to an entity is a good prior estimation
- implementation: $prio(e,m)$: check how often mention m occurs in wikipedia links pointing to entity e

Computation of Context Similarity

- idea: compare the contexts of a given mention and entity
- **context of a mention**: based on all words in the input text
- **context of an entity**: based on keyphrases with high pointwise mutual information (precomputed from Wikipedia)
  - e.g. something like "related words to the entity": Bill Gates := Microsoft, Windows, Philantroph,...
  - use weights for words in keyphrases as contexts:

$$p(e,w) = \frac{|\left\{w \in \left(KP(e) \cup \bigcup_{e' \in IN(e)} KP(e')\right)\right\}|}{N} \tag{5.2}$$

- **context similarity**: some similarity measure comparing the contexts

Computation of Coherence

- Use number of shared incoming links in Wikipedia articles

$$coherence(e_1, e_2) = 1 - \frac{\log\left(\frac{\max(|IN(e_1)|,|IN(e_2)|)}{|IN(e_1)\cap IN(e_2)|}\right)}{\log(N) - \log(\min(|IN(e_1)|, |IN(e_2)|))} \tag{5.3}$$

Algorithm: Coherence Graph

- approximation algorithm
- steps
  1) While we have **non-taboo** entities
     a) Remove **non-taboo** entity with minimum weighted degree
        - taboo entity: only this entity is connected to a mention (and no other entity)
  2) Select solution with maximal total score out of all possible solutions remaining
- after removing all non-taboo entities, we might still have ≥ 1 entities linked for some mentions
- but now all possible solutions where we have a true one-to-one mapping can easily be enumerated to the select the best

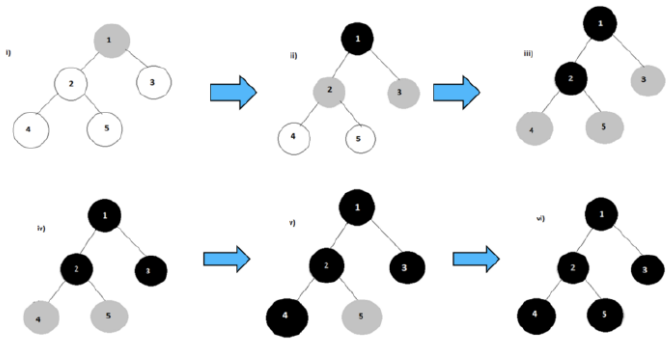
5.3.4 Algorithm: Random Walks

- For each mention run random walks with restart
  - i.e. PageRank on entity graph with jump edges to initial mention from all entities
  - (ignore edges to mentions except for current mention)
- Rank candidate entities by page rank values (stationary probabilities)
- Very efficient, decent accuracy

5.3.5 AGDISTIS

- (mention, entity)-Pair Candidate Generation
  - Given a set of entity labels (mentions)
  - Output: Set of candidate resources for each label
  - Greedy Approach:
    1) for each resource, merge objects of all label-based properties into a **label set**
    2) for a mention m: select all resources where a label from the label set has a similarity $\geq \theta$ with m
- Selection of Best Candidate: Breadth-First Search and HITS
  - for a given label:

1) Build graph of candidate resources (nodes) which are connected if they are connected in the KG
2) Expand graph by BFS by running BFS on the KG from each candidate resource(depth = 2)
3) Run HITS algorithm on this graph: $a = (A^T/cdotA)a$
4) Choose resource with highest authority



5.4 Relation Extraction

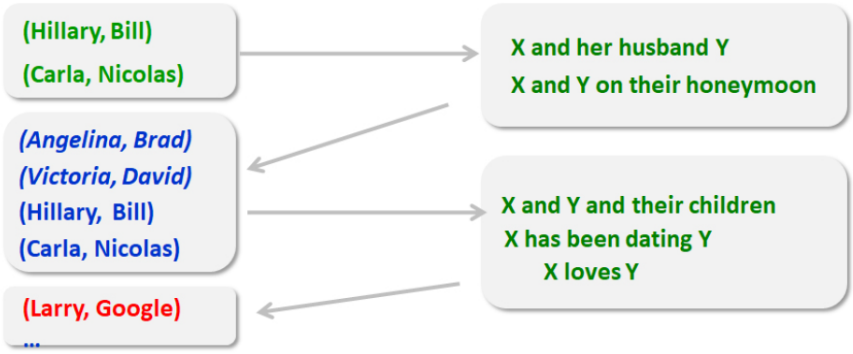
compute all **relations** contained in a text (referring to our extracted resources)

5.4.1 Rule-Based Approach

- Regex made up of POS tags and text (e.g. X, such as Z → Z rdf:type X)
- Pros: Time-efficient
- Cons: Low recall and manual pattern creation

5.4.2 DIPRE - Dual Iterative Pattern Relation Extraction

- Iterative pattern extraction approach
- Supervised approach: Input is a set of positive and negative examples
- idea:
  - assume we start with a relation (e.g. marriedTo)
  - assume we have some positive examples (e.g. Merkel marriedTo Sauer) and negative ones (e.g. Page marriedTo Google)
  - find all occurrences of the positive examples and analyze their context to create new patterns
  - find all occurrences of these new patterns and do the same again
  - use negative examples to discard wrong patterns



5.4.3 NELL - Never-Ending Language Learner

Approach

- Idea: Extract new classes, relations and instances of such from given examples
- Open KG with ontological backbone
  - Set of classes and relations
  - Open set of arguments (instances)
- Periodic human checks
- Main concept: **coupled learning**
  - concurrent learning of patterns and instances across several learners
  - define constraints to ensure **consistency** among learners
    - Coupled output constraints
    - Compositional constraints
    - Multi-view agreement

Learners/Extractors

- **Coupled Pattern Learner** (CPL)
  - For each $i = 1, \ldots, \infty$ do
    - For each predicate p do
      - **Extract** new candidates instances/contextual patterns of p using recently promoted instances
      - **Filter** candidates that violate constraints
      - **Rank** candidate instances/patterns
      - **Promote** top candidates for next round
- **Meta-Boostrap Learner** (MBL)
  - For each predicate p and for each extractor e do
    - Extract new candidates for p using e with recently promoted instances

- For each predicate p
  - **Filter** candidates that violate mutual-exclusion or type constraints
  - **Promote** candidates that were extracted by all extractors

5.4.4 BOA - Bootstrapping Linked Data

- Uses **distance supervision** to find patterns for predicates (found in base knowledge graph)
  - For a given relation p from our knowledge graph
  - Retrieve all pairs (s, o) with $(s, p, o) \in K$
  - Find all sentences which contain label(s) and label(o)
  - Replace labels with variables ?D? and ?R?
- **Support:** Patterns θ should be used across several triples (the higher the support the better)

$$support(\theta, p) = \log(\max_{s,o} I(s, o, \theta, p)) \cdot \log(|I(\theta, p)|)) \tag{5.4}$$

  - Assume $(s, p, o) \in K$
  - $I(s, o, \theta, p) = \#$ sentences which contain s, o and θ
  - $I(\theta, p) =$ number of pairs (s, o) which instantiate θ
- **Specificity**: Patterns should be used for the given relation only
- **Typicity**: Resource types connected by the pattern should match
- generate new KG triples by
  - search full text web for patterns
  - create triples for subject and object matching the pattern
- (Features are based on assumptions on patterns)
- (Combination via neural networks)

5.4.5 Summary

- Pros: Tons of instance knowledge, scalable implementations and a large number of new facts
- Cons: Semantic drift and longtail distributions

5.5 XML to RDF

5.5.1 Lifting and Lowering

- semantic data in RDF is on a higher level of abstraction than semi-structured XML data
- **lifting**: XML to RDF
- **lowering**: RDF to XML

5.5.2 Method: GRDDL - Gleaning Resource Descriptions from Dialects of Languages

- assumes document has a head with meta-information (providing information based on some **profile**)
- fetches transformation (from external resource) giving information on how data in profile and page can be transformed to RDF
  - usually done via templates
- advantages
  - effective
  - helps to lift documents
- disadvantages
  - writing new transformations is cumbersome
  - not much used
  - introduced a zoo of profiles and transformations

5.5.3 Method: XSPARQL

- combines XQuery (XML query language) with SPARQL
- can be used for lifting and lowering
- advantages
  - helps to lift documents
- disadvantages
  - not far spread
  - little tool support

5.6 Table to RDF

5.6.1 Table Types

- database tables
  - complex formal definition
  - interdependences between tables
  - strongly typed
- web tables
  - no defined schema
  - each table is separate entity
- open data tables
  - usually excerpts from spreadsheets
  - tables can have a relation

5.6.2 Relational Model (Databases)

- table is called a **relation** and has a name
- columns are called **attributes**, have a name and a domain
- each line in a table is **tuple** or **record**
- **database schema**: specifies the structure of the database
- **database instance**: specifies the actual content of the database

5.6.3 Web Table Model

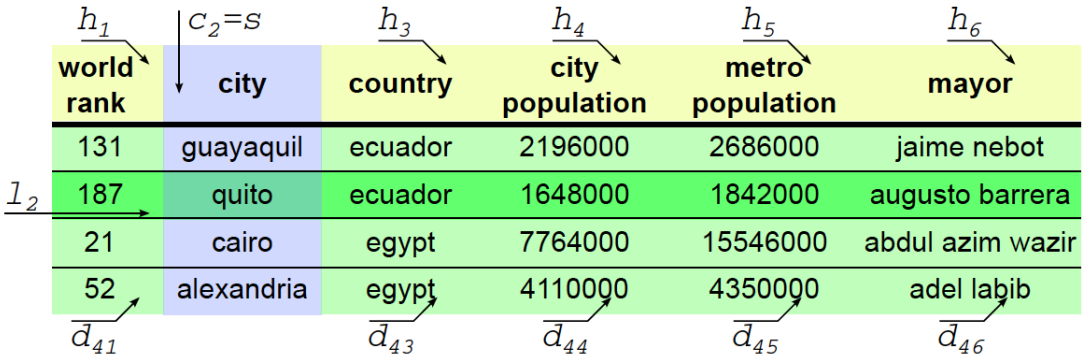
Types of Tables

- layout tables
- real tables
  - **relational tables**
    - main concept: each row provides data about specific objects, called entities, and the columns represent attributes that describe the entities
  - entity tables
  - ... many more

5.6.4 Relational Web Table Model

- relational web table $T = (H, D)$ consists of
  - Header $H = \{h_1, h_2, \ldots, h_n\}$ is an n-tuple of *header elements* h_i.
  - Data $D = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,n} \end{pmatrix}$ is a (m, n) matrix consisting of n columns and m rows.
- **subject column** s: a column into $(n-1)$ two-column tables (**atomic table**) where the binary relation between s and each other column c_i corresponds to a property in a reference knowledge graph

Illustration Relation Web Tables Formal Model



5.7 From Database Tables to RDF

5.7.1 Method: Sparqlify

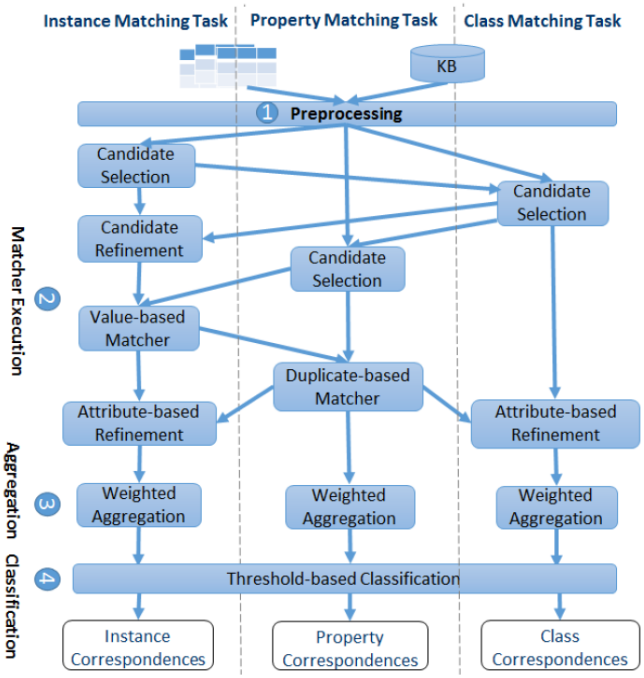
- SPARQL->SQL rewriter which allows to define **RDF views on SQL databases** and **query them via RDF**
- RDF views essentially define how SPARQL queries have to be rewritten to be run against the SQL database
- using the views, we can transform a whole databse into RDF
- advantages
  - helps to lift huge amount of relational knowledge
  - easy to implement mappings
- disadvantages
  - sometimes ineffective
  - manual mappings required
  - not standardized

5.8 From Web Tables to RDF

5.8.1 Web Tables Extraction Process

1) collect web pages
   - crawl the web
2) extract tables
   - parse HTML table markup and pick inner most table
   - extract tables from files (e.g. PDF) in website
3) detect relational tables
   - can be done by ML classifiers
4) recover table semantics (metadata)
   - attribute label row detection (head detection)
   - column data type detection (string, int, etc.)
   - entity label column detection (subject column)
   - language detection
5) table matching (3 matching operations
   a) match table to a class
   b) match rows to instances
   c) match columns to properties

5.8.2 Approach: T2K++



- advantages
  - helps to lift tables
  - can fill missing properties
- disadvantages
  - requires existing knowledge base

5.8.3 Approach: TAIPAN

1) identify subject column
   - use binary ML classifier to detect a subject column
   - features: column support and connectivity
   - support: **ratio** between cells with **disambiguated entities** inside and total number of cells for a column
   - connectivity: ratio of number of connections (i.e., properties) of the column to other columns inside the same table and the total number of columns
     - can be computed based on relations found between columns using patterns and a KB
2) atomize table
   - split table s.t. we have only atomic tables
3) identify property of each table
   - assume we have a knowledge base
   - for each atomic table try to find a corresponding property in the knowledge base
   - measure being used to check if a property corresponds to a table: **relation-property probability**
   - **relation-property probability**: relative number of pairs from table found in KB associated by relation
4) return mappings

CHAPTER 6

Link Discovery

6.1 Concept

- idea: discover links between data (elements)
- required for
  - decentralized dataset creation
  - complex queries across multiple knowledge bases
- links are central for
  - cross-ontology Q/A
  - reasoning
  - data integration

6.1.1 Definition (Informal)

- given two sets of resources S,T and a relation $\mathcal{R}$, find links between S and T of type $\mathcal{R}$

6.1.2 Definition: Declarative Link Discovery

- given: sets of resources S and T and a relation $\mathcal{R}$
- find

$$M = \{(s,t) \in S \times T | \mathcal{R}(s,t)\} \tag{6.1}$$

Common Implementation: Similarity / Distance Based

- $\mathcal{R}$ is usually quantified using a similarity $\sigma : S \times T \to \mathbb{R}$ or distance function $\delta : S \times T \to \mathbb{R}$ and according acceptance thresholds θ / τ
  - **similarity based** link discovery:

$$M' = \{(s,t) \in S \times T | \sigma(s,t) \geq \theta\} \tag{6.2}$$

  - **distance based** link discovery:

$$M' = \{(s,t) \in S \times T | \delta(s,t) \leq \tau\} \tag{6.3}$$

6.1.3 Most Common Relation: *owl:sameAs*

- known as **deduplication**, i.e. find similar or near duplicate resources
- similarity/distance functions based on resource properties

6.1.4 Challenges in Link Discovery

1) Time Complexity (Efficiency)
   - large number of triples ⇒ comparing all elements is infeasible
   - solutions are usually designed to fit in-memory ⇒ infeasible
2) Accuracy (Effectiveness)
   - usually combination of several attributes required for high precision
   - dataset dependent similarity functions

6.2 Time Complexity Related Solutions

- assume we have distance/similarity functions
- assume we represent resources somehow (usually as vector in some space)

6.2.1 LIMES (Distance Based)

Idea

- some distance measures are **mathematical distance** measures (e.g. Levenshtein or Minkowski)
- mathematical distance measures fulfill the **triangle inequality**: for all elements x, y, z:

$$\delta(x,z) - \delta(z,y) \leq \delta(x,y) \leq \delta(x,z) + \delta(z,y) \tag{6.4}$$

- main idea: do not compute distances between all pairs explicitly, BUT
  - compute n **exemplars** $E(T)$ (exemplatory resources) for T

- approximate real distance of (s, t) by triangle inequality
- approximation involves computations between elements from S and $E(S)$ and elements from $E(T)$ and T
- useful as $S \cdot T < n \cdot (S + T) \rightarrow$ we do less comparisons

Computing Exemplars

1) choose exemplar e_1 uniformly at random from T
2) until we have enough exemplars
   a) choose exemplar with maximum cumulative distance to existing exemplars

$$e_{i+1} = \arg\max_{x \in T} \sum_{k=1}^{i} \delta(x, e_i) \tag{6.5}$$

- element $t \in T$ is represented by its nearest exemplar $e(t)$

$$e(t) = \arg\min_{e_i \in E(T)} \delta(t, e_i) \tag{6.6}$$

- optimal number of exemplars $n \approx \sqrt{T}$

Computing Distances

- approximate distance $\delta(s, t)$ by

$$\delta(s, t) \geq \delta(s, e(t)) - \delta(e(t), t) \tag{6.7}$$

Remarks

- time-efficient approach
- no guarantees wrt. reduction of comparisons

6.2.2 Multi-Block (Similarity Based)

Idea

- create multi-dimensional index of data
- partition space s.t. $\sigma(s, t) < \theta \Rightarrow index(s) \neq index(t)$
- decrease number of comparisons by only comparing pairs (s, t) with $index(s) = index(t)$

Approach

- assume we have multiple similarity measures
- three main phases
  1) index generation
     - for each similarity measure, generate an index
     - similarity measures *sim* consist of two functions
       - actual similarity measure $sim_s : S \times T \to [0, 1]$
       - index function: $sim_i : (S \cup T) \times [0, 1] \to \mathcal{P}(\mathbb{N}^n)$ which assigns a resource to one or multiple blocks inside the index s.t. only elements with a similarity $\geq$ threshold are in the same block (n: number of blocks)
  2) index aggregation
     - aggregate indexes into one multi-dimensional index
     - aggregation must define
       - similarity aggregation function
       - block-aggregation function, aggregating two sets of blocks
       - threshold aggregation function
     - aggregation functions are used to map setup to single index and adapt values / functions without loosing main property
  3) candidate pair generation
     - generate a pair for each two entities sharing a block
- compare only candidate pairs

6.2.3 Reduction-Ratio-Optimal Link Discovery

Reduction Ratio

- idea: reduce number of comparisons $\mathcal{C}(\mathcal{A})$ ($\mathcal{A}$: algorithm)
- always: $\mathcal{C}(\mathcal{A}) \geq |M'|$
- principle: maximize **reduction ratio**

$$RR(\mathcal{A}) = 1 - \frac{\mathcal{C}(\mathcal{A})}{|S| \cdot |T|} \tag{6.8}$$

- advantages of a guaranteed reduction ratio
  - space management
  - runtime predictions

Reduction Ratio Guarantee

- best possible reduction ratio:

$$RR_{\max} = 1 - \frac{|M'|}{|S| \cdot |T|} \tag{6.9}$$

- approach $\mathcal{H}(\alpha)$ fulfills RR guarantee, if and only if

$$\forall r < RR_{\max} \exists \alpha : RR(\mathcal{H}(\alpha)) \geq r \tag{6.10}$$

- more general measure: **relative reduction ratio**

$$RRR(\mathcal{A}) = \frac{RR_{\max}}{RR(\mathcal{A})} \tag{6.11}$$

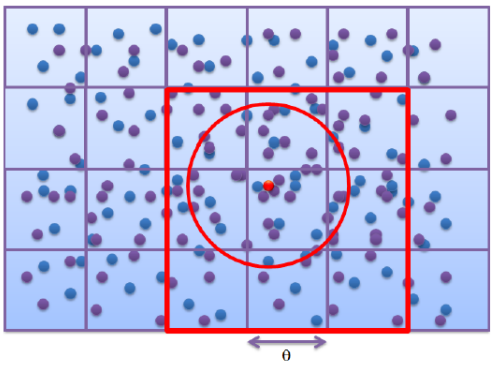
- approach $\mathcal{H}(\alpha)$ fulfills RRR guarantee, if and only if

$$\forall r > 1 \exists \alpha : RRR(\mathcal{H}(\alpha)) \leq r \tag{6.12}$$

- **goal**: devise approach fulfilling RRR guarantee

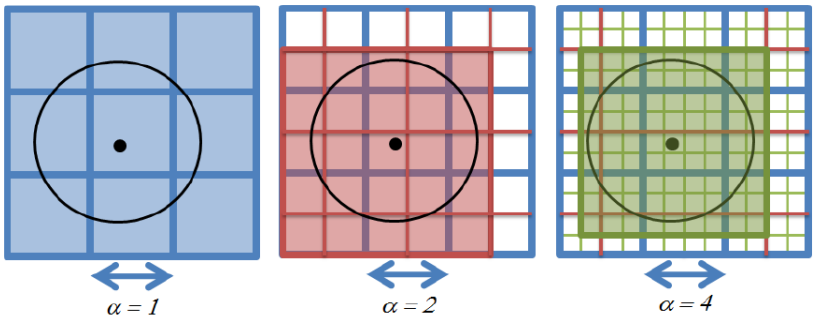
Hyppo (Distance Based RRR Approach

- main idea:
  - restriction $\delta(s,t) \leq \theta$ describes **hypersphere** (dimension n) around s
  - approximate hypersphere using a **hypercube**!
  - even better: use multiple hypercubes!
- width of single hypercube to $\triangle = \frac{\delta}{\alpha}$
  - where $\alpha \in \mathbb{N}$ is a granularity parameters defining how fine grained the space tiling will be
- **space tiling**:
  - tile $\Omega = S \cup T$ into adjacent hypercubes $C = (c_1, \ldots, c_n)$
    - contains points $\omega \in \Omega : \forall 1 \leq i \leq n : c_i \cdot \triangle \leq \omega_i \leq (c_i + 1) \cdot \triangle$



- approximating hypersphere of point $\omega \in \Omega$

- combine $(2\alpha + 1)^n$ hypercubes around ω



- **restrictions**:
  - approach works because for special spaces (Ω, δ) (orthogonal dimensions) common metrics can be decomposed into functions working on only one dimensions
  - example: **Minkowski distance** $(p \geq 2)$

$$\delta_p(s,t) = \sqrt{\sum_{i=1}^{n} |s_i - t_i|^p} \tag{6.13}$$

- **problem**:
  - no guarantees wrt. RRR in Hyppo

HR^3 (Distance Based RRR Approach)

- extends HYPPO (same restrictions and base idea)
- but additionally assign each hypercube an **index** (numerical number)

$$index(C,\omega) = \begin{cases} 0 & \text{if } \exists 1 \leq i \leq n : |c_i - c(\omega)_i| \leq 1 \\ \sum_{i=1}^{n} (|c_i - c(\omega)_i| - 1)^p & \text{else} \end{cases} \tag{6.14}$$

  where C is a hypercube and ω is a reference point
- for reference point ω discard cube C iff

$$index(C,\omega) > \alpha^p \tag{6.15}$$

- only compare ω with points from undiscarded cubes
- important **properties**
  - no loss of recall
  - $\lim_{a \to \infty} RRR(\mathcal{HR}^3(\alpha)) = 1$

6.2.4 RADON

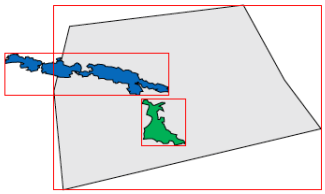
Motivation

- geo-spatial datasets
- link discovery on such datasets requires computation of topological relations
- standard approaches infeasible due to polygon structures with many coordinates
- **Dimensionally Extended nine-Intersection model** (DE-9IM) used to encode relations between regions
  - 3×3 matrix filled maximum number of dimensions in which two regions intersect wrt. different intersection definitions
  - one matrix corresponds to a special relation, e.g. all 0: disjoint

  $\Rightarrow$ if we can compute really fast if two regions share at least point, we can speed up the whole computation (as we do not have to compute the whole matrix for the two)

Basic Idea

- improved indexing approach based on **minimum bounding boxes**



- space tiling

Basic Approach

- given S and T and a relation
- three main steps

  1) swapping strategy for index size minimization
     - if S consists of a set of large regions covering many hypercubes, we need a large index to represent them for the computation
     - BUT: we can swap T and S if we compute the **inverse relation**!
     - estimate "complexity" of S and T by computing an **estimated total hypervolume** of the set of structures
  2) space tiling for indexing
     - insert all geometries $s \in S$ into $I(S)$
       - compute MBBs for all s and map each s to its MBB
     - insert only those $t \in T$ into the same index, which are potentially in hypercubes covered by $I(S)$

3) filtering of candidates
    - only compute DE-9IM for pairs where the MBBs intersect

Properties

- Radon is complete and correct

6.2.5 Gnome

Motivation

- approach assuming data does NOT fit into memory
- basic idea: most approaches rely in divide-and-merge paradigm

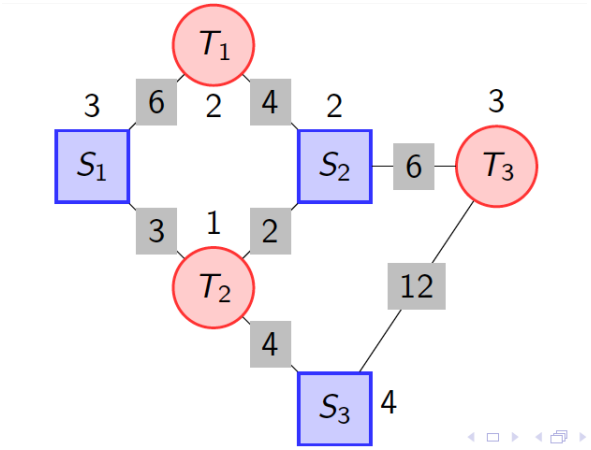
$\Rightarrow$ use this to turn them into distributed approaches

Formal Model

1) break up S and T into overlapping subsets
    - $\mathcal{S} = \{S_1, \ldots, S_n\}$ with $S_i \subseteq S \wedge \bigcup_{i=1}^{n} S_i = S$
    - $\mathcal{T} = \{T_1, \ldots, T_m\}$ with $S_i \subseteq T \wedge \bigcup_{i=1}^{m} T_i = T$
2) find mapping function $\mu : \mathcal{S} \rightarrow 2^{\mathcal{T}}$ s.t.
    - elements of S_i are only compare to elements of $\mu(S_i)$
    - union over all results of $S_i \in \mathcal{S}$ is exactly M'

Task Graph

- define **task graph** $G = (V, E, w_v, w_e)$ where
    - $V = \mathcal{S} \cup \mathcal{T}$
    - E: edge $e_{i,j}$ is present if we compare S_i with $T_j \in \mu(S_i)$
        - note: an edge represents an according **task** of comparing the two subsets
    - node weight function $w_v(v) = |v|$ (i.e. number of elements in associated subset)
    - edge weight function $w_e(e_{i,j}) = |S_i| \cdot |T_j|$ (i.e. required number of comparisons of task)



Approach Based on Task Graph

two steps

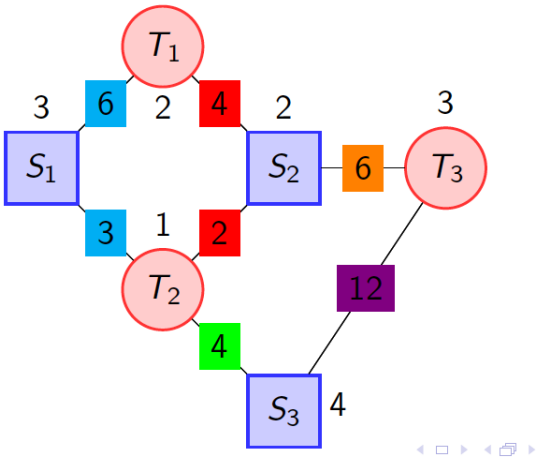
1) **clustering**: find groups of nodes which fit into main memory together

2) **scheduling**: compute sequence of groups minimizing hard drive access

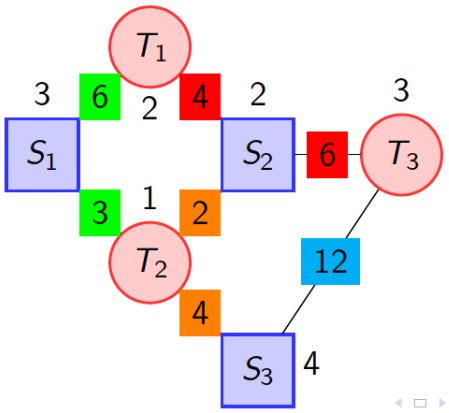
Clustering

- two approaches

  1) **naive**: cluster tasks by S_i involved



  2) **greedy**: start by largest task and add **connected largest tasks** until memory is full



- results
  - naive is more (time-)efficient
  - greedy gives better results

Scheduling

- output of clustering: sequence of clusters
- **intuition**: consecutive clusters should share data
- **goal**: maximize data overlap of generated sequence
  - **overlap** between two tasks: number of elements of S_i and T_i used in both tasks
  - **overlap** of whole sequence: sum of two-task-overlaps of consecutive pairs
- two approaches
  1) **best-effort**: select random pair of clusters - if switching them improves overlap, do so
  2) **greedy**: start with random cluster and choose next cluster with maximum overlap
- results
  - best-effort is more (time-)efficient
  - greedy gives better results

6.3 Accuracy Related Solutions

6.3.1 Link Discovery as Classification Problem

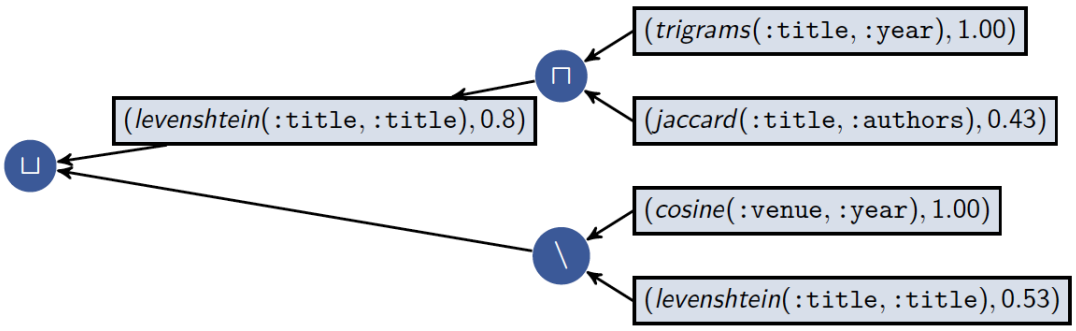
- given: sets of resources S and T and a relation $\mathcal{R}$
- find
$$M' = \{(s,t) \in S \times T | \mathcal{C}(s,t) = 1\} \tag{6.16}$$
using a classifier $\mathcal{C} : S \times T \rightarrow \{0,1\}$
- goal: $\mathcal{C}(s,t) = 1 \Leftrightarrow \sigma(s,t) \geq \theta$
- standard ML algorithms perform well, but dedicated solutions are better
- AND: need for unsupervised / active learning in most cases

6.3.2 Link Specifications

- declarative link discovery frameworks learn **link specifications**

Syntax

- two types of link specifications
  1) **atomic link specification**: $f = (m, \theta)$ where (e.g. (edit, 0.8))
     - $m : R \times R \to r$ is a **measure** (R:the set of resources)
     - $\theta \in [0, 1]$ is a threshold
  2) **complex link specifications**: $L = (m, \theta, \varphi(L_1, L_2))$ where
     - m is a measure
     - θ is a threshold
     - L_1, L_2: link specifications
     - φ: **binary link specification operator** from $\{\sqcup, \sqcap, \setminus\}$ (note: binary is enough as it allows us to represent everything we could represent with non-binary ones)
  3) if $\theta = 0$, we may write $L = \varphi(L_1, L_2)$
- measure can be **atomic** and **complex**
  1) **atomic measure** $m : R \times R \to [0, 1]$
  2) **complex measure**: **aggregations** over multiple measures (e.g. $m = \max(m_1, m_2, m_3)$)



Semantics

- $[[L]]_M$: mapping computed by link specification L from M
- we are usually interested in $[[L]]_{S \times T}$
- atomic link specification $L = (m, \theta)$

$$[[L]]_M = \{(s, t) \in M | m(s, t) \geq \theta\} \tag{6.17}$$

- **complex link specification**

$$[[(m, \theta, \varphi(L_1, L_2))]] = [[m, \theta]]_{[[\varphi(L_1, L_2)]]} \tag{6.18}$$

  i.e.: we apply first $\varphi(L_1, L_2)$ and then our atomic link specification (m, θ) on the result

- **semantics of LS operators**
  - $[[\sqcap(L_1, L_2)]] = \cap([[L_1]], [[L_2]])$
  - $[[\sqcup(L_1, L_2)]] = \cup([[L_1]], [[L_2]])$
  - $[[\backslash(L_1, L_2)]] = \backslash([[L_1]], [[L_2]])$

6.3.3 Challenges

- creation of labeled data is extremely expensive
  - solution: active learning
- need automatic means for automatic class and property matching
  - solution: statistics + hospital/resident algorithm
- need for efficient executions of link specifications
  - solution: previous section + planning
- devise dedicated machine learning approaches
  - solution: topic of this section

6.3.4 Learning Components

learning a classifier requires learning

1) two sets of restrictions defining sets S and T (we do not know what to compare if we only have two databases)
2) the components $\sigma_1, \ldots, \sigma_n$ indirectly defining a complex similarity measure σ
3) a set of thresholds $\phi_1, \ldots, \phi_n$ for the components
4) link specification operators combining the different (atomic) link specifications
   - note that we can view (σ_1, ϕ_1) as an LS

6.3.5 Raven

Assumptions

- restrictions defining sets S and T are **class restrictions**
  - i.e. we try to find similar resources between two classes from different KBs)
- classifier shape is given ⇒ no need to learn LS operators, as we only use $\sqcap$ (limitation!)

Learning Class Restrictions

- **goal**: find best class matching across knowledge bases
  1) define class similarity functions
     - e.g. string similarity, property overlap, etc.
  2) solve **hospital-resident problem** with preference ordering based on similarity function
- hospital-resident problem
  - extension of **stable-marriage-problem**
    - n males and n females each with a preference ordering on the other sex
    - goal: find one-to-one mapping which ensures no cheating, i.e. a **stable mapping**
    - **stable mapping**: there exists no pair (m, w) such that there exists a pair (m', w') with
      1) m prefers w' to w and w' prefers m to m'
      2) and vice versa
    - algorithm: while we have unmatched men, find the "best" woman the men did not try out yet and marry them if possible (unmarry woman if she prefers him to current partner)

```
STABLEMARRIAGE(Men m ∈ M and women w ∈ W)
 1  for x ∈ M ∪ W
 2  do Set x to free
 3  while ∃m : (isFree(m) ∧ ∃ w canPropose(m, w))
 4  do w = first woman on m's list he has not yet proposed to
 5     if w is free
 6        then engaged(m, w)
 7        else m' = fiance(w)
 8             if w prefers m to m'
 9                then Set m' to free
10                     engaged(m, w)
```

  - here: each hospital can accept several residents (i.e. one-to-n mapping)

Learning Similarity Measures

- learn property matchings for classes S and T similarly to class matching
- directly leads to similarity functions (i.e. compare matched properties [e.g. one similarity function per property])

Learning Thresholds

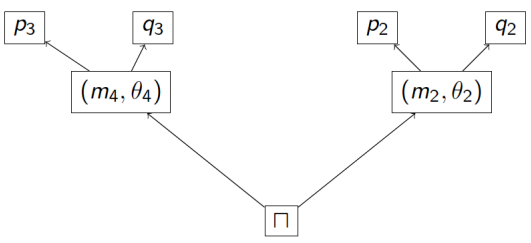
- active **perceptron learning**
- begin with guessing thresholds θ_i
- update thresholds based on **most informative examples** (MIEs)
  - MIEs: e.g. closest examples to decision boundary which are unclassified
- algorithm:
  1. Initialize $L = \sqcap(\sigma_i, \theta_i)$, e.g., $\theta_i = .9$
  2. Set $L' = \sqcap(\sigma_i, \theta_i - \Delta)$, e.g., $\Delta = .1$ relaxation
  3. Sort pairs $(s,t) \in [[L']]_{S \times T}$ by $\sum_i (\sigma_i(s,t) - \theta_i)$ in descending order why descending?
  4. MIPE = top-k pairs $\in [[L]]_{S \times T}$ (most informative positive examples)
  5. MINE = top-k pairs $\in [[L']]_{S \times T} \setminus [[L]]_{S \times T}$ (most informative negative examples)
  6. Ask oracle for correct classification of MIPE and MINE
  7. Update θ_i

$$\theta_i = \theta_i + \eta \frac{\sum_{(s,t) \in MIPE} \sigma_i(s,t) - \sum_{(s,t) \in MINE} \sigma_i(s,t)}{|MINE| + |MIPE|} \text{ (perceptron)} \quad (1)$$

  8. Iterate 2 – 7 until termination criterion is reached

6.3.6 EAGLE

- assume we have:
  - two sets of restrictions defining S and T (assume we have these)
  - property mappings (p_i, q_i) where p_i is a property of S and q_i is a property of T
  - set of labeled examples $(s,t) \in S \times T$
- main difference to RAVEN: learns **generic** classifier types (i.e. can combine LS operators)
- idea: specifications are trees ⇒ start with random tree and use a **genetic algorithm** to evolve the tree



Approach

1) generate initial population
   - assume small initial set of annotated training data
   - start with random tree (random LS operators, random matchings between measures and property pairs, random thresholds)
2) until termination criterion
   a) evolve population X times
      i. compute fitness of each individual (i.e. F-measure over training data)
      ii. **selection**: pairwise tournaments wrt. fitness decide who is allowed to recombine
      iii. **recombination**: apply **crossover** between pairs of individuals: create offspring by swapping two random subtrees
      iv. **mutation**: apply mutation to an individual with mutation probability (i.e. randomly change LS operator, measure, thresholds, etc.)
   b) compute k most informative (unlabeled) links
      - most informative unlabeled links: maximum disagreement between individuals of population, i.e. half says 0 and other half says 1
   c) ask user to label these examples and add them to training data
3) return fittest LS

Idea behind Active Learning Part

- we have many possible solutions which might disagree a lot ⇒ use more data to find out who is right

Remarks

- larger population leads to better results, but longer runtimes
- population size of 100 usually enough
- very time efficient approach

6.3.7 COALA

- rethinks idea of active learning in LS learning
- idea: use more than just **informativeness** of an example as an indicator

⇒ use similarity between link candidates!

Similarity between link candidates

- link candidate $x = (s, t)$ can be seen as a vector

$$(\sigma_1(x), \ldots, \sigma_n(x)) \in [0, 1]^n \tag{6.19}$$

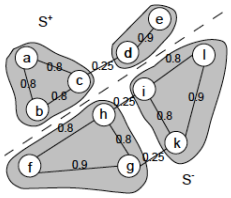
- **similarity** between link candidates x, y

$$sim(x, y) = \frac{1}{1 + \sqrt{\sum_{i=1}^{n} (\sigma_i(x) - \sigma_i(y))^2}} \tag{6.20}$$

- allows exploiting both **inter-class** and **intra-class** similarity
  - **inter-class similarity**: similarities between links of the same class (i.e. type)
  - **intra-class similarity**: similarities between links of different classes

Graph Clustering (Inter-Class Similarity Based)

1) cluster elements of classes + and − independently (i.e. class pairs in + and cluster pairs in −
2) choose one element per cluster as **representative**
3) ask oracle for correct classification of representatives



Activation Spread (Inter- & Intra-Class Similarity Based)

1) compute similarity matrix M_0 of elements both from + and −
2) initialize so called **activation vector** A (vector having one entry for each element in $+ \cup i$ containing an **informativeness** value)
3) until values in M reach a very small value
   - normalize A
   - compute $A_{t+1} = A_t + M \cdot A_t$
     - idea: propagate activation (informativeness) to neighbors (weighted by similarity)
     - result: higher activation for elements with both high inter-class and intra-class similarity
   - apply weight decay to M
4) return k elements with highest activation to oracle

6.3.8 Wombat

Idea

- mostly positive link examples on the web (negative usually not possible due to open world assumption)
- idea: two part process
  1) learn atomic LS
  2) combine atomic LS to complex ones

Learning Atomic LS

- assume we have a training dataset of annotated links and we are given S and T
- compute subset of properties of S and T with coverage above a certain threshold (i.e. a property mapping)
- create atomic measures for these property pairs (i.e. measure compares s and t by comparing the properties)
- return those atomic measures with maximal F-measure

Deriving Complex LS

- use LS operators to combine atomic LS
- compute complex LS by using an approach based on **refinement operators**
  - simple one: does not allow arbitrarily nested LS
  - complex one: allows arbitrarily nested LS (upward refinement operator)

    - Given partially ordered space $(S, \sqsubseteq)$
    - An upward refinement operator is a function ρ with
      - $\rho : S \to 2^S$
      - $\forall x' \in \rho(x), x \sqsubseteq x'$
    - A downward refinement operator is a function ρ with
      - $\rho : S \to 2^S$
      - $\forall x' \in \rho(x), x' \sqsubseteq x$
    - Characteristics of refinement operators
      - Finite if $\rho(x)$ is finite for any x
      - Redundant if there exists distinct refinement chains between x, $x' \in S$
      - Proper if $\forall x \in S\ x' \in \rho(x) \to x \neq x'$
      - Complete if $\forall y \subseteq x$ we can reach $z \equiv y$ from x
      - Ideal if ρ is finite, complete and proper
    - For specifications $L \sqsubseteq L'$ iff $[[L]] \subseteq [[L']]$

- perform an iterative search through a solution space based on a fitness function (e.g. F-measure)

6.4 Execution Optimization

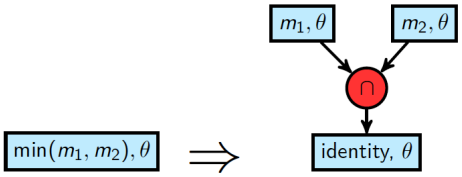
- link executions should be executable fast!

6.4.1 HELIOS

1) rewrite specifications to small equivalent specifications
2) optimize execution by finding time-efficient plans

Rewriting

1) **leaf generation**
   - use equivalence between operators to generate atomic leaves (we have efficient algorithms for atomic LS)



2) **dependency detection and propagation**
   - idea: **dependency**: LS L_1 depends on L_2 ($L_1 \rightarrow L_2$) if for all S, T: $result(L_1) \subseteq result(L_2)$
   - dependencies can be **propagated** towards the root using **set theory**
     - e.g. if $depends(L_4, L_2) \wedge depends(L_3, L_2) \Rightarrow depends((m_1, \theta_1, \cap(L_3, L_4)), L_2)$
3) reduction
   - remove unnecessary inner LS based on dependencies

Planning

- evaluation function: **runtime**
  - measure runtime on samples
  - fit a linear regression ML model to these data points
  - use predictions combined with size of S and T as runtime estimates
- evaluation function: **selectivity** (how many results will the plan generate?)
  - similar as runtime
- optimization idea: given LS L
-
- if L is atomic, return L
- for $L = (m, \theta, \varphi(L_1, L_2))$

1) compute plans for L_1 and L_2

2) combine these plans to plan which minimizes runtime

CHAPTER 7

Link Prediction

7.1 RDF as Multi-Graph

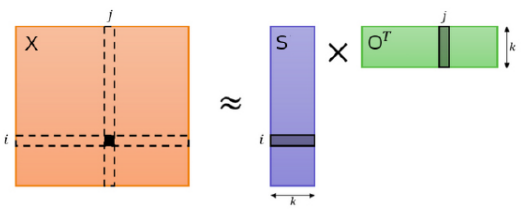
- data in RDF is **labeled directed multigraph** $G = (V, E, L)$
  - V: set of nodes (RDF: entities)
  - $E \subseteq V \times V \times L$: set of ordered triples (RDF: RDF triples)
  - L: set of edge labels (RDF: relation types)

7.2 Matrix Representation

- normal graphs can be represented by **adjacency matrix**
- more complicated for multi-graphs
- graphs can be analyzed based on the adjacency matrix!

7.2.1 Matrix Factorization

- idea: represent matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ by the product of two matrices $\mathbf{S} \in \mathbb{R}^{n \times k}$ and $\mathbf{O} \in \mathbb{R}^{k \times m}$



- **rank of factorization**: k
- rows $s_i \in \mathbb{R}^k$ and columns $o_j \in \mathbb{R}^k$ of factor matrices can be seen as **latent-variable** representations explaining the observed variables $x_{i,j} = s_i \cdot o_j^T$

Computation

- $\mathbf{X}$ is approximated by $\mathbf{S}$ and $\mathbf{O}$ s.t. the **mean squared error** is minimized

$$||\mathbf{X} - \mathbf{S} \cdot \mathbf{O}^T||^2 \tag{7.1}$$

- optimization: e.g. using gradient descent

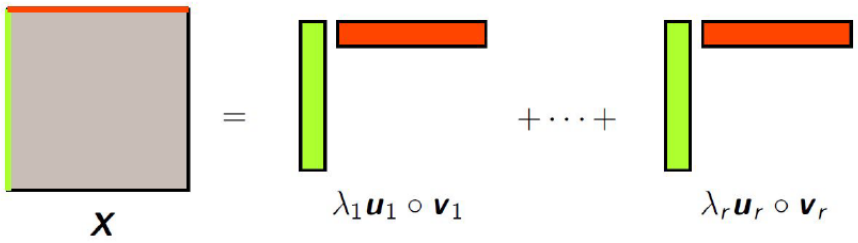
- problem: overfitting $\Rightarrow$ use **regularization**!

$$||\mathbf{X} - \mathbf{S} \cdot \mathbf{O}^T||^2 + \lambda \left(||\mathbf{S}||^2 + ||\mathbf{O}||^2 \right) \tag{7.2}$$

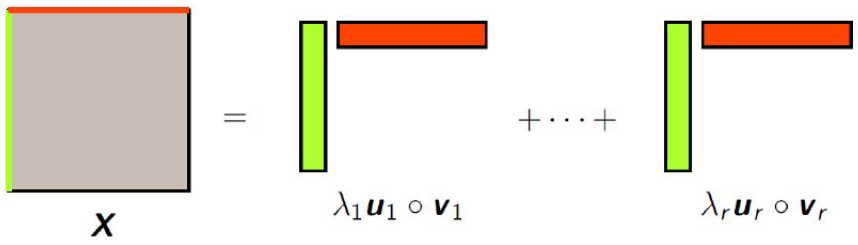
- right rank k and regularization weight λ have to be selected to avoid under- or overfitting

7.2.2 Singular Value Decomposition (SVD)

- Any matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ has a unique decomposition into a weighted sum of the outer products of pairwise orthonormal vectors, i.e. $\mathbf{X} = \sum_r \lambda_r u_r \circ v_r$ such that



- λ_i is the i-th singular value of X
- u_i is the i-th left singular vector of X
- v_i is the i-th right singular vector of X
- u_i, v_i are pairwise orthonormal, e.g., $u_i^T u_i = 1, u_i^T u_{j \neq i} = 0$
- Rank of X = minimal number r of outer products $\lambda_r u_r \circ v_r$ that generate X in their sum



- X can be written as $X = U\Sigma V^T$ with
  - $U \in \mathbb{R}^{n \times k}$
  - $\Sigma \in \mathbb{R}^{k \times k}$ (diagonal matrix)
  - $V \in \mathbb{R}^{m \times k}$ and
- Hence $x_{ij} = \sum_{\alpha=1}^{k} u_{i\alpha}\sigma_\alpha v_{j\alpha}$
- Each row $u_i(v_i)$ is the latent representation of a subject (object)
- $u_{i\alpha}(v_{i\alpha})$ is the α-th latent representation/factor
- λ_α is the overall weight (magnitude of effect) of the α-th factor
- Lower rank approximation = Only taking the first r' terms What could this be good for?

- lower rank approximation: useful for **dimensionality reduction**

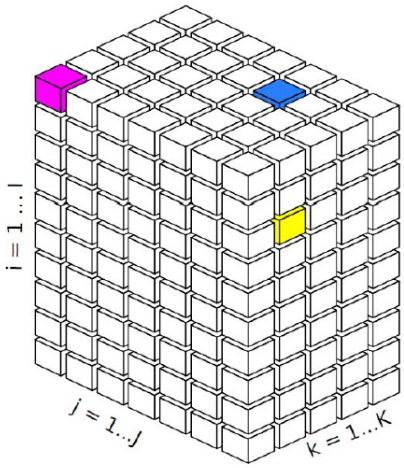
7.3 Multi-Graphs as Tensors

- multi-graphs cannot be represented by a single-matrix (in general) without information loss

$\Rightarrow$ use **tensors**: n-modal (dimensions) generalization of matrices

Definition: Tensor

- formal:
  - n-th order tensor $\mathcal{T}$: element of the tensor product of n vector spaces: $\mathcal{T} \in V_1 \oplus \ldots \oplus V_n$
- informal: tensors are multi-dimensional arrays (with more than 2 indices), e.g. $\tau_{a,b,c,d,e}$
- **order of tensor**: number of **modes** (i.e. number of indices required to identify one element)



$\mathcal{T} \in \mathbb{R}^{10\times7\times5}$

$= \mathcal{T}_{1,1,1}$

$= \mathcal{T}_{3,7,2}$

$= \mathcal{T}_{1,5,4}$

Mode-1 has dimension $I = 10$
Mode-2 has dimension $J = 7$
Mode-3 has dimension $K = 5$

7.3.1 RDF as Tensor

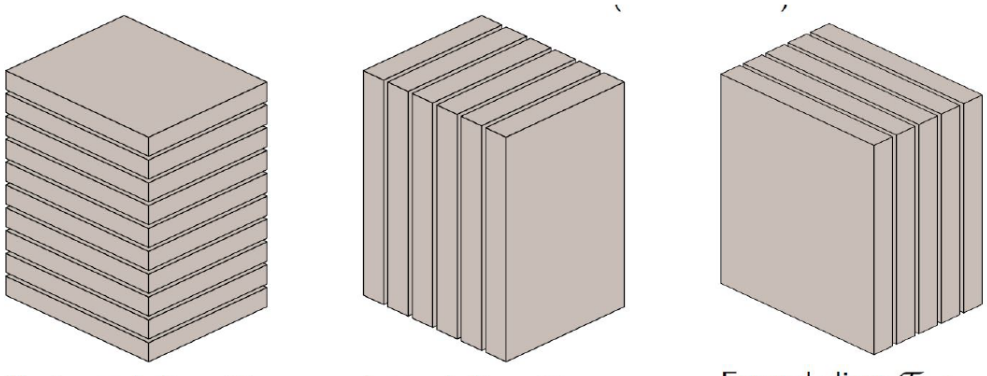
- for RDF we only need **third-order** ($n = 3$) tensors
- two modes refer to the entities, one mode to the relation type
  $|mode_1| = |mode_2| = |V| \wedge |mode_3| = |L|$
- ordering of entities in mode 1 and 2 must be identical [**ToDo**: *why is this the case?!?*]
- **adjacency tensor** $\mathcal{T}$ defined as

$$\tau_{i,j,k} = \begin{cases} 1 & \text{if triple (i-th entity, k-th predicate, j-th entity) exists} \\ 0 & \text{else} \end{cases} \tag{7.3}$$

  - contains a cell for every possible triple!

7.3.2 Slices and Fibers

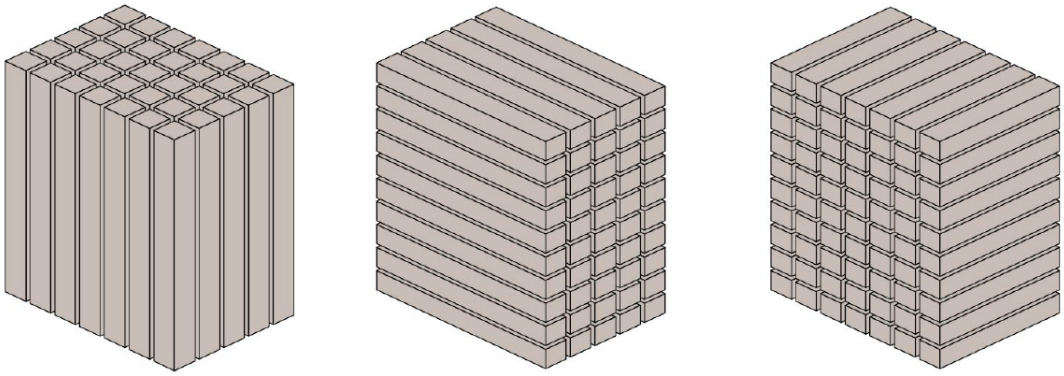
- Slices are two-dimensional sections of a tensor (i.e. matrices).



Horizontal slices $\mathcal{T}_{i,:,:}$ Lateral slices $\mathcal{T}_{:,j,:}$ Frontal slices $\mathcal{T}_{:,:,k}$

= stacked adjacency matrices in adjacency tensors

- Fibers are higher-order analogues of rows and columns in matrices.



Mode-1 fibers $\mathcal{T}_{:,j,k}$ | Mode-2 fibers $\mathcal{T}_{i,:,k}$ | Mode-3 fibers $\mathcal{T}_{i,j,:}$

7.4 Link Prediction for RDF on Tensors

- predict links by learning **adjacency tensors** of multigraphs based on **tensor factorizations**

7.4.1 Rank-1 n-Order Tensors

- rank-1 n-order tensors: result of outer product of n vectors
  - for $n = 3$: we can represent as follows $\tau_{i,j,k} = a_i \cdot b_j \cdot c_k$ and write $\mathcal{T} = a \bigcirc b \bigcirc c$
- outer product ($\bigcirc$) of two vectors $u \in \mathbb{R}^2$ and $v \in \mathbb{R}^3$

$$u \bigcirc v = u \cdot v^T = \begin{bmatrix} u_1 \cdot v_1 & u_1 \cdot v_2 & u_1 \cdot v_3 \\ u_2 \cdot v_1 & u_2 \cdot v_2 & u_2 \cdot v_3 \end{bmatrix} \tag{7.4}$$

7.4.2 Candecomp/Parafac (CP) Decomposition

- main idea: represent tensor as **weighted linear combination of rank-1 tensors**

$$\mathcal{T} \approx \sum_{q=1}^{r} \lambda_q \cdot (a_q \bigcirc b_q \bigcirc c_q) \tag{7.5}$$

- **tensor rank** $rank(\mathcal{T})$: smallest number r required to represent $\mathcal{T}$ exactly
- **rank of CP decomposition**: number r of rank-1 tensors used to approximate $\mathcal{T}$
- computation: optimize some loss, e.g. root mean square error

$$||\mathcal{T} - \sum_{q=1}^{r} \lambda_q \cdot (a_q \bigcirc b_q \bigcirc c_q)||^2 \tag{7.6}$$

7.4.3 Collective Learning

- definition: ability of a learning algorithm to include **information of related entities** (e.g., classes, relations or attributes) in the learning and prediction task for a particular entity
- hence: we want a collective learning approach for link prediction!
- **problem about CP**:
  - actually models a bipartite graph as it assumes that entries of first and second mode are not identical
  - i.e. it does NOT assume identical sets of entities for subjects and predicates

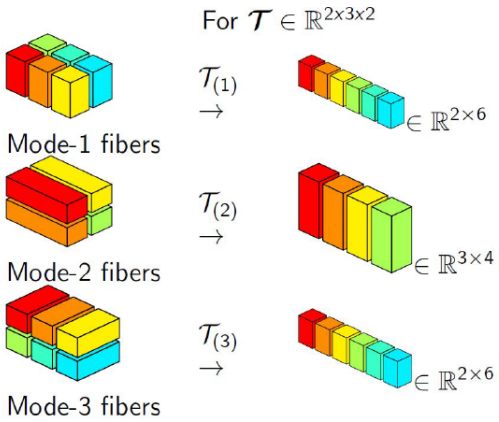
  $\Rightarrow$ we have different **latent variable representations** a_q and b_q in $\sum_{q=1}^{r} \lambda_q \cdot (a_q \circ b_q \circ c_q)$

  $\Rightarrow$ we ignore identities between subjects and objects
- **solution?**
  - model decomposition as $\sum_{q=1}^{r} \lambda_q \cdot (a_q \circ a_q \circ c_q)$
  - problem: we loose information about directions
- **solution**: **Tucker Decompositions**

7.4.4 Tucker Decompositions

Definition (Mode-n unfolding)

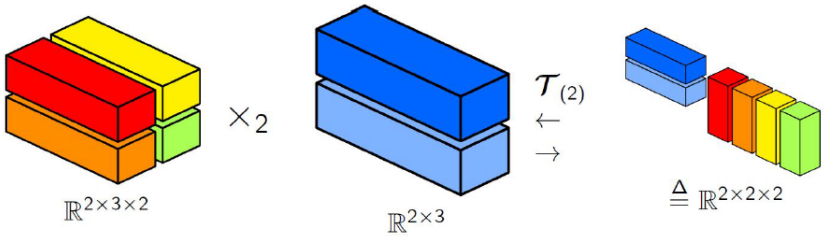
The mode-n unfolding of a tensor $\mathcal{T}$, denoted by $\mathcal{T}_{(n)}$, reorders the elements of $\mathcal{T}$ into a new matrix $\mathbf{M}$, by using the mode-n fibers as columns of $\mathbf{M}$



Definition (Mode-n product)

The mode-n product, denoted by $\times_n$, is the multiplication of a tensor by a matrix in mode n.

- Let $\mathcal{T} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ and $\mathbf{M} \epsilon \mathbb{R}^{Q \times I_n}$
- $\mathbf{Y} = \mathcal{T} \times_n \mathbf{M} \in \mathbb{R}^{I_1 \ldots I_{n-1} \times Q \times I_{n+1} \ldots I_N}$, $\mathbf{Y}_{(n)} = \mathbf{M}\mathbf{T}_{(n)}$



Actual Decomposition

- Tucker decomposes a tensor into a core tensor and separate factor matrices for each mode
- For a third-order tensor $\mathcal{T} \epsilon \mathbb{R}^{n_1 \times n_2 \times n_3}$ we get
  - A core tensor $\mathbf{R} \in \mathbb{R}^{r_1 \times r_2 \times r_3}$, and
  - Matrices $\mathbf{A} \in \mathbb{R}^{n_1 \times r_1}$, $\mathbf{B} \in \mathbb{R}^{n_2 \times r_2}$, $\mathbf{C} \in \mathbb{R}^{n_3 \times r_3}$

$$\tau \approx \mathbf{R} \times_1 A \times_2 B \times_3 C$$

- Goal: Find best approximation under some loss, e.g. $||\mathcal{T} - \mathbf{R} \times_1 A \times_2 B \times_3 C||^2$
- Two Tucker families
  - Tucker1: **B** and **C** are identity matrices, i.e. $\approx \mathbf{R} \times_1 A$
  - Tucker2: **C** is the identity matrix, e.g. $\mathcal{T} \approx \mathbf{R} \times_1 A \times_2 B$

7.5 Factorizing RDF: RESCAL

RESCAL

- Factorization of multigraphs into unique entity and predicate representations
- Corresponds to Tucker2 decomposition with the constraint that two factor matrices have to be identical

$$B = A \text{ ergo } \mathcal{T} \approx R \times_1 A \times_2 A \tag{4}$$

$$\mathcal{T}_{:,:,k} \approx A R_{:,:,k} A^T \text{ hence } \mathcal{T}_{ijk} \approx \sum_{q,r} a_{iq} R_{qrk} a_{jr} \tag{5}$$

- $A \in \mathbb{R}^{|V| \times r}$ represents the entity-latent-component space
- $R_{:,:,k} \in \mathbb{R}^{r \times r}$ is an asymmetric matrix that specifies the interaction of the latent components for the k-th predicate
- r is the number of latent components of the factorization

7.5.1 Link Prediction

1) compute **rank reduced factorization** (latent variable representations)
2) compute
$$\tau_{i,j,k} = a_i \cdot R_k \cdot a_j^T \tag{7.7}$$
3) $\tau_{i,j,k}$ is **confidence value** that the corresponding triple exists

7.5.2 Entity Resolution

- RESCALE can also be used to find duplicate instances of the true underlying entity (e.g. "Merkel" and "A. Merkel")
- A can be seen as a way of embedding entities into a simple vector space $\Rightarrow$ apply any ML technique to A to find duplicate entities

CHAPTER 8

Summary

1. Representation (semantic networks, property graphs, RDF)
2. Query languages (GraphQL, Gremlin, SPARQL)
3. Knowledge Extraction (NER, NEL, RE)
4. Link Discovery (runtime, accuracy)
5. Link Prediction (tensors)
6. Ontologies (OWL)