GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Ontology-based transformation of natural language queries into SPARQL queries by evolutionary algorithms

Dissertation

zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades

"Doctor rerum naturalium"

der Georg-August-Universität Göttingen

im Promotionsprogramm PCS

der Georg-August University School of Science (GAUSS) vorgelegt von

Sebastian Schrage

aus Hannover

Göttingen, 2021

Betreuungsausschuss

Prof. Dr. Wolfgang May, DBIS, Institut für Informatik

Prof. Dr. Florentin Wörgötter, III. Physikalisches Institut, Biophysik

Dr. Minija Tamosiunaite, III. Physikalisches Institut, Biophysik

Mitglieder der Prüfungskommission

Referent/in: Prof. Dr. Wolfgang May, DBIS, Institut für Informatik

Korreferent/in: Prof. Dr. Florentin Wörgötter, III. Physikalisches Institut, Biophysik

Weitere Mitglieder der Prüfungskommission:

Prof. Dr. Kerstin Strecker, Didaktik der Informatik, Institut für Informatik

Prof. Dr. Carsten Damm, AG Theoretische Informatik und Algorithmische Methoden, Institut für Informatik

Prof. Dr. Winfried Kurth, Abteilung Ökoinformatik, Biometrie und Waldwachstum, Büsgeninstitut

Prof. Dr. Florin Manea, Arbeitsgruppe Grundlagen der Informatik, Institut für Informatik

Tag der mündlichen Prüfung: 10.01.2022

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
www.informatik.uni-goettingen.de

First Supervisor: Prof. Dr. Wolfgang May
Second Supervisor: Prof. Dr. Florentin Wörgötter
Third Supervisor: Dr. Minija Tamosiunaite

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen,

Abstract

In this thesis an ontology-driven evolutionary learning system for natural language querying of RDF graphs is presented. The learning system itself does not answer the query, but generates a SPARQL query against the database.

For this purpose, the Evolutionary Dataflow Agents framework, a general learning framework is introduced that, based on evolutionary algorithms, creates agents that learn to solve a problem. The main idea of the framework is to support problems that combine a medium-sized search space (use case: analysis of natural language queries) of strictly, formally structured solutions (use case: synthesis of database queries), with rather local classical structural and algorithmic aspects. For this, the agents combine local algorithmic functionality of nodes with a flexible dataflow between the nodes to a global problem solving process. Roughly, there are nodes that generate informational fragments by combining input data and/or earlier fragments, often using heuristics-based guessing. Other nodes combine, collect, and reduce such fragments towards possible solutions, and narrowing these towards the unique final solution. For this, informational items are floating through the agents. The configuration of these agents, what nodes they combine, and where exactly the data items are flowing, is subject to learning. The training starts with simple agents, which –as usual in learning frameworks– solve a set of tasks, and are evaluated for it. Since the produced answers usually have complex structures answers, the framework employs a novel fine-grained energy-based evaluation and selection step. The selected agents then are the basis for the population of the next round. Evolution is provided as usual by mutations and agent fusion.

As a use case, EvolNLQ has been implemented, a system for answering natural language queries against RDF databases. For this, the underlying ontology medatata is (externally) algorithmically preprocessed. For the agents, appropriate data item types and node types are defined that break down the processes of language analysis and query synthesis into more or less elementary operations. The "size" of operations is determined by the border between computations, i.e., purely algorithmic steps (implemented in individual powerful nodes) and simple heuristic steps (also realized by simple nodes), and free dataflow allowing for arbitrary chaining and branching configurations of the agents. EvolNLQ is compared with some other approaches, showing competitive results.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 4

1.3 Structure of the Thesis 8

2 Foundational Concepts for the Approach **11**

2.1 Ingredients: Data Model, Metadata, and Natural Language Processing 11

2.2 The Big Picture: Employing an Evolutionary Algorithm 23

2.3 Miscellaneous 40

3 The Evolutionary Dataflow Agents Framework **45**

3.1 Motivation 45

3.2 Products: the Data that Flows 47

3.3 Evolutionary Dataflow Agents' Anatomy and Genetics - Nodes, Conduits and Connections 50

3.4 Evolutionary Dataflow Agents at Work 55

3.5 Evaluating the Work of Evolutionary Dataflow Agents 57

3.6 Environments 60

3.7 Simulations 69

4 Applying Evolutionary Dataflow Agents to NLQ **75**

4.1 Product Classes 82

4.2 Nodes 110
4.3 Agents and Evolution 174
4.4 Learning Data 178

5 Implementation and Usage **183**
5.1 Query Interface 183
5.2 Agent creator panel 184
5.3 Training Environment 190
5.4 Implementation 192

6 Evaluation **197**
6.1 Settings 197
6.2 Question Test Sets and Benchmarking 208

7 Conclusion **239**

Bibliography **257**

A Agents Species and Test Sets **259**
A.1 Agent Species Ellyn25545 259
A.2 Geobase250 Query Set 268
A.3 Mondial Query Set 277

Chapter 1

Introduction

1.1 Motivation

When we need information, humans learn from a very young age that they can ask others for it. Starting with crying, screaming, facial expressions and gestures up to the use of natural language. The exchange via natural language is normal for us, even if it had to be learned over years and also changes in details in the course of time. Moreover, it is inaccurate or unnecessarily complicated in several places, has irregularities and is too rich in variations. Even learning another language takes a lot of time and practice until it can be used for unrestricted information exchange. That is why it is much easier for machines to use a subset of a language and put it into a fixed, well-defined scheme. Therefore, other means of communication than natural language are used to communicate with computers. Starting with punch cards, then assembler code over C and Java code to specialized query languages like SQL, XQuery, or SPARQL the languages developed to be closer to natural language but ultimately they stayed very formal and restricted and still both the machine and the user are expected to adapt to the other's language to some degree. For the machine, this means using an interpreter or compiler; for the human, it means learning a programming or query language.
Here lies one of the biggest obstacles for database users, because often the person with the question is not the same person who writes the query, but prefabricated queries or forms are used because the person who needs information does not know the query language. This significantly limits the person asking the question and they still have to take a step towards the machine by adapting the actual thought to a pre-made query or filling a form to provide the desired information, or information from which the actual one can be derived. This is why there has long been the idea of asking queries entirely in natural language to a database and having the machine not only search for the information, but also convert the query into a logically correct and machine-understandable form. As mentioned at the beginning, however, natural language is difficult to understand and so there have been attempts to create good natural language interfaces for databases (NLIDB) since

1972 [1].

Depending on the available data structures, theories, and computational capabilities, various approaches based on graph theory [2,3], pattern matching [4], learning algorithms, state transition [5], and ontologies-based [6] have emerged over the years. While all of these approaches have had varying degrees of success, it is still not the case that one can communicate with a database as one would do with a human expert.

In many benchmarks such as the annual QALD contest [7–10], depending on the current data set, not even half of all questions (for qald9 had the best approach gAnswer [11] a precision of 0.293) are answered correctly by the winner. Another example are the voice assistants in cell phones or operating systems. In the end, one realizes very quickly that you have to ask questions in a certain way and that there are many limitations. In case of Windows voice assistant *Cortana*, which was introduced in Windows 10 and will be removed with Windows 11 again, these limitations were severe enough such that *Cortana* had no efficient practical use and nearly no one used it frequently.

However in cases with a well-defined and smaller data set, better results can be achieved by approaches, such as ATHENA [6].

In this thesis another approach on how to tackle this problem is presented. Based on the observation that with the components of a sentence of a single question and with good knowledge of the database and query language, answering the query is usually feasible for a human. One can quickly derive rules, such that the nouns play an important role, and think of operations, such as replacing synonyms with terms known to the database, that lead to an answer to the query. The problem is rather that these rules often do not have generality, may only be applied in certain cases or to certain groups, or must be executed in a certain order. Given the variety of ways to formulate a single query, one quickly realizes that simply creating small operations and rules is not enough, but that the interaction of these is crucial.

So the combination and organisation of those methods and of the input information can be seen as an optimization problem. A common approach is to solve theses problems by learning algorithms in the absence of an effective and deterministic algorithm. In particular, deep learning approaches like mentioned in [12] have been widely used in this context, but they have the problem that the training sets for database queries are laborious to create and not available in larger quantity with decent quality. Thus, they usually have to make do either with very little data or with automatically generated data which lacks the variations of normal queries and are prone to linguistic incorrectness. In both cases the quality of the learning suffers especially in the context of domain independence.

In comparison with Deep Learning, *Evolutionary algorithm*s cope better with smaller training sets, as can be concluded from the recommended sizes of training sets from [13] (for *evolutionary algorithm*s) and [14] for deep learning. Further the knowledge of the developer can much more easily be integrated in form of functions of the *evolutionary algorithm*. On the other hand, *evolutionary*

algorithm are in some sense weaker than deep learning since they *require* such knowledge of the developer to be integrated.

So, it's a matter of problem analysis, what approach is best suited for a problem, in this case, the translation of natural language questions into database queries.

1.1.1 The Evolutionary dataflow agents Framework

With the above problem of *translation natural language questions* into database *queries* (and answering them, which is then trivial as it is just sending the created query to the database, which evaluates it and returns the answer) in mind, a framework based on *evolutionary algorithm*s has been developed.

Starting Point: Natural Language Understanding An important aspect was the analysis how humans analyse written natural language sentences, especially (i) children when learning a reading in their native language, and (ii) also, adults dealing with a foreign language where they maybe do not know the vocabulary completely and are not extremely fluent with the grammar: words, there are many possible connections between words, *locality* issues in the text, (basically pointwise) *relationships* between words (verbs, subject, object, etc.), intervals of words (for descriptions or negation), and at the end, more or less subconscious, the sentence is understood.

So one driving force for the design was that there are *local* aspects, collecting *data items*, and that interferences between those induce further information, sometimes at first glance guessed, where humans rather immediately detect garden paths.

That's the point to change to a computer's view: it has less sentiment for garden paths. So, instead of sentiments, appropriate algorithms using the –atomic or collected– information have to be identified and implemented as small possible units of a global process.

Thus, the power is in (i) the data flow, (ii) the diverseness of smallest steps that allow to infer –potential– information fragments, (iii) the local power of nodes, and (iv) *any* way to combine all this. Issue (iv) is solved by a learning algorithm that learns how to combine data flow and nodes.

That is, what evolutionary dataflow agents intends to realize in a general way.

Relationship of the Goal to practical Informatics Another aspect is that the goal of the problem, and also part of the input is "hard" practical informatics: the learning process and the usage have to access a formal ontology and a database, and to create a *query* in a formal database language. So, hardwired code fragments (database access) must be included, and classical programming of the string-based mapping to the query language (i) have to be, and (ii) can be (as the designer knows them well) included. There is no need and no fun for such a learning system to learn how to write syntactically correct SPARQL queries. So, portions where the algorithms are known and can be programmed can be integrated within powerful nodes.

Internally, the nodes are programmed in Java, thus they allow practically any functionality, as some of the graph-based nodes described concretely in Section 4.2.7 show. So, any classical algorithms *can* be encapsulated into the nodes, and the contribution of the evolutionary dataflow agents framework is to allow to solve problems for which no classical algorithm can be given.

The approach is general enough to expect to be applied to other problems of the above kind, not only to the extensive use case as described in this thesis. For concrete applications, the designer has to analyze the informational fragments of the input, intermediate informational/data structures, and the final result, how they are distilled from the input data, and all possible ways how they can be combined, generating lots of potential paths towards the solution, how these alternatives can be cleaned towards the final solution(s).

1.1.2 EvolNLQ: Application of Evolutionary Dataflow Agents to the Problem

An application of evolutionary dataflow agents for the NL-to-SPARQL translation problem described above has been developed in this thesis, called EvolNLQ.

The design of EvolNLQ consists of (i) identifying the types of informational items, from NLQ parsing to the underlying (basically algebraic) structure of SPARQL queries (ii) designing intermediate knowledge structures that are needed on the way, and (iii) identifying units of work that map between them. Actually, the design of (iii) identifies further items for (ii) and so on, intermittent with an (iv)th task, namely to run and analyse test cases, until the process is considered to be sufficient.

By providing the *local functionalities* which the learning algorithm can use to create agents, it can be ensured that EvolNLQ only uses methods that are *generally* valid for their respective language (English), and that do not depend on the application-specific terms on which they operate.

The latter are provided by the ontology that is appropriately preprocessed. Here, another aspect comes into play when comparing the coverage with other approaches: the task is *not* general understanding of natural language, but *mapping* questions to a known database. The metadata of the database, i.e., the *ontology* is also used during the process to constrain the search space, and to evaluate intermediate data items wrt. their reasonability.

Furthermore, the ontological side that is used with EvolNLQ has before been developed for mapping between RDF data (SPARQL) and relational data (SQL) [15], so it is –on a generic level of data modeling– more sophisticated than usual, and it has been equipped also with specific modeling issues, e.g., reified properties that have properties themselves.

1.2 Related Work

Here is a brief overview of other state of the art approaches in this area, primarily based on [16].

In the current development in NLP systems, there are four major categories that are used either in isolation or in combination: Keyword-based, Pattern-based, Parsing-based and Grammar-based. Further there is the rather separated field of machine learning approaches.

1.2.1 Keyword-Based Approaches

The simplest approaches are keyword-based. For such approaches, an inverted index is created from data and metadata, and the incoming question is examined for known terms using this index, and certain actions are taken according to the keyword. Simplicity and customizability are the big advantages here, a disadvantage of this approach is that the questions usually need to be heavily customized.

For example, SODA [17] does not really use natural language, but expects input reduced to the keywords like "Name largest city Europe" and instead of "What is the name of the largest city in Europe?".

Then the question or rather keyword string is analyzed in 5 steps:

- the keywords are analyzed. To find out these keywords, SODA uses both the domain ontology and the synonyms that can be queried through DBPedia,
- keywords found are heuristically assigned a score,
- the tables involved are determined,
- filters are created from the keywords,
- keywords, tables, and filters are compiled into a correct question.

A more complex approach was chosen for SINA [18], where the keyword approach is extended using Hidden Markov Models, which try to determine the resources of the underlying database that match the query. In this approach, the first step is also to isolate the keywords. Then the keywords are segmented based on the available resources. The third step is then to compile the required resources by string matching. The resulting segments are then evaluated and a SPARQL query is composed from the best ones.

1.2.2 Pattern-Based Approaches

The pattern-based approach is an extension of the keyword-based approach with natural language patterns.

For example, QuestIO [19] uses this approach. First, the usual methods for isolating keywords are used. The keywords are then categorized and then examined for certain patterns that are related to a categorized word. Finally, these relations and patterns are reformatted into a query.

1.2.3 Parsing-Based Approaches

These systems parse the input query, usually using general Natural Language parsers. These then provide a parsetree with a lot of information, describing both individual words and their grammatical relationships. From these relations, corresponding rules for database queries must then be created.

For this approach, Athena [6] is a very successful representative. Athena is an ontology-based natural language interface for relational databases. Athena requires a mapping between the ontology and the SQL database to be queried in order to work. For a given ontology, Athena also requires synonyms for all terms in the ontology. The following four steps are then performed:

First of all, the sentences are annotated. Then keyword matching is performed with different methods:

- Direct matching via terms of the ontology or their synonyms.
- The search index is extended by abbreviations of names of persons or companies, such as initials.
- Furthermore, temporal and numerical expressions are searched for.

Subsequently, the determined grammatical relations are used to establish relationships between the found keywords. The tokens determined from this are combined to form so-called interpretations. From these interpretations, an interpretation tree is created which has to fulfill the following conditions:

- **Evidence**: All tokens that have been discovered must also be in the tree.
- **Weak connectedness**: The tokens must be indirectly connected to each other, the shorter the better.
- **Inheritance constraint**: tokens may only use properties of their superclass, not vice versa.
- **Relationship constraint**: the relationships found in the text must all be considered.

These trees are evaluated for their plausibility and the most promising one is translated into an interlanguage and then adapted to the SQL database and translated into a valid query. Other approaches such as NaLix and NaLir use similar methods, but rely exclusively on the parse trees and do not use an ontology.

1.2.4 Grammar-Based Approaches

While the previous approaches build on each other to some extent and can be mixed, the grammar-based approaches are distinctly different, especially how they interact with the user. These systems have a fixed set of rules (grammar) that define what questions can be asked against a database. Through these rules, the system is then able to assist and guide the users in creating their questions

and ultimately, as they write the query. The advantage of this is that it may allow the user to discover more from the database than was expected and the resulting queries may actually be accurate. While the question "what happens if one wants to ask a query which is not covered by the rules" springs to mind immediately, this is not a problem specific for this kind of approach since it is true for all other approaches as well, what they can't do, they can't do. It is more important that the concrete instance of those systems pushes this boundary as far as possible and the real problem of those approaches is above all that these rules are extremely domain-specific and have to be created by hand for each individual case. Therefore the performance depends on the rule set for a concrete domain as well as on the system which works with it.

An example for this approaches is TR Discover [20]. During question creation, the system uses a First Order Logic representation as an intermediate language. The system uses this to make two types of suggestions; autocompletion and prediction. The suggestions are generated based on the relationships and entities of the database. By the left-corner principle, during input all rules are considered whose left part of the right side of the rule is still applicable to the input and the remaining right side is then suggested. This is repeated after each completed rule, thus a parsetree is created according to defined rules, which can restrict later questions also further than only with the input. The parsetree is then traversed and translated either to SQL or SPARQL.

1.2.5 Machine Learning Approaches

Machine learning approaches are a clearly different approach. These use only very rudimentary schema, ontology-based or linguistic-based methods and focus primarily on a general architecture and good and powerful training examples.

The most commonly used approach is sequence-to-sequence modeling based on recurrent neural networks with an input encoder and an output decoder. However, to do this, the queries must first be converted into (high-dimensional) vectors that the networks can understand [21].

An example of these approaches is the work of [22], this approach uses an input encoder and an output decoder with a global attention model. Unlike pure machine learning approaches, this work also incorporates the schema to generate their training data. In a first step, training data is created by manually created templates for the natural language and SQL query pairs. Then, a synonym corpus is used to replace parts of these queries to create a larger linguistic variance. The network is then trained on this mapping.

Approaches which use deep learning have been presented with Seq2SQL [23] and sqlnet [24]. To train Seq2SQL, the new dataset WikiSQL has been created using Wikipedia. It consists of a total of 24,241 tables and 80,654 handwritten NL-SQL pairs. Ultimately, however, the approach was only demonstrated for simple, one-table queries. More successful was sqlnet, trained with the same set, but both approaches were still surprisingly inaccurate in comparison to the relative simplicity of the WikiSQL data set as Yavuz et al [25] stated in their paper.

1.3 Structure of the Thesis

This thesis is divided into seven chapters, which are briefly presented below along.

Chapter 2 first gives an overview of the foundational concepts used in this thesis. The general setting of the system to be developed is described, introducing aspects of databases and ontologies, and the basics for understanding text annotation and explains relevant terms of linguistics for these annotations. Then, the field of learning algorithms is opened, especially the general concept of an *agent*, which is the foundation of all *evolutionary algorithm*s and its functions is introduced. Among others, the Artificial Life Simulation "Sticky Feet Creatures", which has been inspiring for this approach is discussed.

Chapter 3 explains the Evolutionary Dataflow Agents framework developed in the course of this dissertation.

Chapter 4 applies the Evolutionary Dataflow Agents framework to the Natural Language Questions use case.

Chapter 5 gives insights into the use of the programs created for this thesis and how they can be extended.

Chapter 6 gives an evaluation of this work, examining the learning behavior of the Evolutionary Dataflow Agents framework and compares EvolNLQ in relation to other approaches. For this purpose, the trained program is applied to three different data sets.

Finally, Chapter 7 concludes this thesis.

Chapter 2

Foundational Concepts for the Approach

2.1 Ingredients: Data Model, Metadata, and Natural Language Processing

Different information sources are combined to achieve a translation from natural language to a query language. In this section, the information sources are described, first in general followed by the the used approaches, algorithms, and standards.

To accomplish its task of constructing a correct database query to a given natural language question, EvolNLQ ultimately has three different information sources at its disposal. These are the dataset, the natural language question itself, and domain independent knowledge in form of a thesaurus.

The data set consists of the data stored in the dataset and the metadata of this set.

The notion of *data* hereby means all information which describes entities in the application domain. Such data can be used to find entities mentioned in the query and to use them as a starting point for further conclusions. Ultimately the answer to the query is a subset of that data.

The metadata describes the structure of the data set, including a classification structure of the entities within the data set, which of those entities are in which forms of relationships to each other and restrict the set of possible classes and relationships to a finite set. With the metadata it can be checked, which conclusions from the other sources are at all possible and if necessary also conclusions can be drawn about information that is only implicitly available.

And the last component, of course, is the input question itself. In addition to the original wording, the question provides a lot of other explicit and implicit information.

2.1.1 Accessing Data: Databases, Relational Algebra and Query Languages

For efficient data access, larger amounts of data are usually stored in databases or more precisely within a *Database Management System (DBMS)*[1]. While a database could be nearly everything which vaguely contains data, beginning from a shopping list over a folder of Word and Excel documents to things like Wikipedia; a DBMS is none of that, but a computer program specialized in storing, retrieving and manipulating large amounts of data in an efficient manner. When the term database is used in the following, it always refers to a DBMS.

In order to be able to use such large data sets meaningfully, it is necessary to filter, link, aggregate or otherwise modify the data set to retrieve the desired information of it. Since the development of relational databases and *SQL* [26], most databases are based on an *algebra*, which is a set of operations for handling data. However, it is rather uncommon to state queries directly in this algebra; instead query languages are defined such as SQL [26], XQuery [27], the *SPARQL Protocol And RDF Query Language (SPARQL)* [28] [29], or Datalog [30], which in turn are translated into an algebra. The resulting operator tree is then transformed using algebraic laws to allow the queries to be evaluated as efficiently as possible. Then the algebra tree is executed on the specific structure of the database. In case of SPARQL, the query language used in this thesis, the underlying data structure is a graph, in case of the most common database *SQL* the data structure is based on relations (which is the mathematical model of tables).

These query languages must be unambiguous and capable of formulating (nearly) arbitrary queries so that they can represent the full functionality of the underlying algebra. This always happens at the expense of the accessibility of the language. The specific query languages must be learned and practised by the user. So while languages like SQL or SPARQL are relatively close to a common English sentence and easy to understand when read, it is still not like one could just write away without knowing how the languages need to be structured. Other query languages such as XPath or Datalog are even less intuitive, but shorter to write.

While the DBMS handles the transfer from *query language* to the underlying *algebra*, the users must bring their idea or question, presumably formed in natural language in their mind, into the query language. To translate a natural language question into the query language, such that the users do not have to do this step themselves, there are *natural language database interfaces*. Those, however, are not able to handle queries in the same complexity as a query language, but are very flexible about the way a question was asked.

2.1.2 Domain Ontology and Data

A data set consists of data and metadata. While data contains all the facts about instances that are stored in the database, metadata describes in an abstract way how these data are related to each

[1] Or at least should instead of being stored in a bunch of Excel tables, as is traditionally done in many administrations. Data stored in this way can then only be found again by the administrative staff who created it, and not at all on Fridays.

other. For example, "Germany has a population of eighty-two million" would be data, also, that Germany is a country is data; on the other hand, the knowledge that countries have a population and population is stated as a number is metadata. Depending on the DBMS, metadata is stored in different ways and different degrees of expressiveness. While SQL metadata still requires some expert knowledge since it only describes information such as, the table named Country has a column of type Numeric named Population so that the actual meaning still needs to be reasoned by the user. The *Resource Description Framework (RDF)* [31] stores this information as queryable information and is machine-understandable. This is important since it enables the drawing of further conclusions and the inference of additional information.

2.1.2.1 Ontologies

The term "ontology" is used in many disciplines, in this work it always means an ontology in the computer science sense. It describes a formal, (semi)-structured, machine-readable representation of a set of concepts and the relationships between them. This representation thus provides the basis for recognizing terms within the natural language and can then contour a meaningful query from the relationships formalized in it. Particularly important here are the concepts of class, domain, and range. Most ontologies are written in RDF, therefore EvolNLQ is expecting its ontologies in RDF as well. [32]

2.1.2.2 RDF - The Resource Description Framework

To ensure that the different ontologies can communicate with each other, the W3C introduced *Resource Description Framework (RDF)* [31] as a standardized data model and language, and in order to increase the expressiveness and flexibility of RDF data, *Resource Description Framework Schema (RDFS)* [33] and *Web Ontology Language (OWL)* [34] have been defined. To query those data sets, SPARQL is proposed as the standard query language by the W3C.

An RDF ontology is a collection of statements, each of which consists of three elements, therefore they are also called a triple. Similar to English sentences a triple consists of a subject, a predicate, and an object. Each triple describes a relationship between two nodes. A node can be either a resource (representing a real or an abstract individual) or a literal (representing a value, e.g., a string or a number etc.). *International Resource Identifier (IRI)*s [35] should be unique across whole world wide web, therefore often an internet domain of which the creator holds the domain is used as a prefix to ensure that this name is not already used by someone else. Other than in SQL the metadata is also stored in that way, so it is query-able data. This is crucial for machine reasoning over the data set, such that not only data contained in the set is queryable, but also logically inferred facts.

RDF can be stored in several different formats, the most important being *RDF/XML* and the *turtle format* [36] format which is an extension of the *Notation 3 (n3)* [37] . Both formats have their own advantages, which is why both are used regularly.

The *RDF/XML* format is, as the name suggests, a valid well-defined *XML* [38] [39] document. This also means that it can be queried, edited or displayed using *XML* tools. This format is mainly used for communication between different SPARQL endpoints and display in websites since most browsers have some way to represent an *XML* document. The biggest disadvantage of this format is that the *XML* tags make it much longer and they hinder the reading flow for humans.

The *N3* format maps the triples structure more clearly and is thus shorter and easier for humans to read, but then also requires separate tools that understand this format. *N3* is usually easier to work with, since there is no need to additionally take care of the compliance with *XML* syntax conventions and the associated tree structure, but only to work on the triple level.

Example 1 *Example excerpt of RDF in the turtle format format from the geographic database Mondial, for the country Germany.*

```
<countries/D/>  rdf:type :Country ;
     :name "Germany"  ;
     :localname "Bundesrepublik Deutschland (die)" ;
     :carCode 'D' ;
     :area 356910 ;
     :capital <countries/D/provinces/Berlin/cities/Berlin/> ;
     :population 82521653 ;
     :hadPopulation
        [ a :PopulationCount; :year "1950"^^xsd:gYear; :value 68230796] ,
        [ a :PopulationCount; :year "1997"^^xsd:gYear; :value 82501000] ,
        [ a :PopulationCount; :year "2007"^^xsd:gYear; :value 82217837] ,
        [ a :PopulationCount; :year "2011"^^xsd:gYear; :value 80219695] ,
        [ a :PopulationCount; :year "2016"^^xsd:gYear; :value 82521653] ;
     :populationGrowth -0.18 ;
     :infantMortality 3.46 ;
     :gdpTotal 3593000 ;
     :gdpInd 30.1 ;
     :gdpServ 69 ;
     :gdpAgri 0.8 ;
     :inflation 1.6 ;
     :unemployment 5.3 ;
     :government "federal republic" ;
     :independenceDate '1871-01-18'^^xsd:date  .
```

2.1.2.3 Classes

Similar to a phylogenetic tree in biology or classes in programming languages, classes in ontologies represent groups of individuals that have certain properties in common. Classes can in turn have superclasses and subclasses. A superclass contains all the individuals of its subclasses, and a

subclass has all the properties characteristic of its respective superclasses, in addition to possibly other properties. A class can have any number of superclasses and subclasses and can also occur as one arbitrary often. Further superclass and subclass relations are transitive. Therefore the relationships between classes then results in a tree, which is called a class hierarchy.

An individual of an ontology can have any number of (unrelated) classes, unless the ontology explicitly excludes certain class combinations. In that case, we speak of disjoint classes. The classes themselves can be defined very differently. They can be extenstionally defined (i.e., by explicitly assigning instances to them), ir intensionally, i.e., in terms of possessing certain properties.

2.1.2.4 Resource Description Framework Schema (RDFS)

RDF does not support a class hierarchy by itself, but since classification is an important task, RDFS was developed to extend RDF to that effect. RDFS is commonly used in RDF data sets.

2.1.2.5 The Web Ontology Language (OWL)

The scope of RDF is further extended by the use of OWL. [2] The Web Ontology Language OWL was created to describe terms of a domain and their relationships formally in a way that software can also process the meaning. Among other things, OWL allows more complex class and property descriptions using set operators, such as intersection, union, and complement, allowing significantly more inferences to be made by reasoning.

2.1.2.6 Classes in RDF/OWL

In an RDF/OWL ontology, a distinction is made between defined classes and blank nodes. On the one hand, defined classes are formally introduced outside of their usage and have a referenceable name. They are intended to be used more frequently, or to appear directly as part of a query. They are used for representing extensional knowledge about individuals. Blank nodes, on the other hand, are intended to be used once, and, since they do not have a name themselves, are not directly queryable. For example, in Example 1, the combination of population number and the year is stored as a blank node (square brackets are the syntax of a blank node in the N3 format). In this case, since this data construct does not have an intuitive name, and originates from the formalization in RDF, it lends itself to be translated to a blank node. Furthermore, there is still the possibility to name a class, but not to define it beforehand. For Natural Language Processing especially the defined classes are important because they provide a finite set of things that can be meant. In addition blank node constructs are usually used because they cannot be described well and can only be referenced indirectly and therefore do not usually occur in natural language queries.

[2]The acronym was chosen (original Mail can be found under [40]) beside many other reasons to make pronunciation clear. Further there is a children story of an owl which could not say owl but said wol, since they really wanted an owl as their icon since those are traditionally connected with wisdom they decide to use this story as a justification for this "wrong" acronym.

2.1.2.7 Domain and Range

As mentioned before, each statement consists of subject, predicate and object. While the subject describe individuals of the ontology, object can either be another individual or a literal value and the predicate is a relationship between them, called a property. Properties are directed relationships between two individuals. Similar to classes, properties also have superproperties and subproperties, which also have transitive inheritance. In addition, properties have a domain and a range. The domain indicates from which class the relationship of the property can start and the range to which class the relationship can go. Also symmetrical relations, i.e. relations from which subject and object can be switched, without causing the statement to became false. for example State borders State, the properties have nevertheless a range and a domain, which map however to the same class. Non literal-valued properties can also have an inverse property defined. This represents the reverse direction of the relationship, in which case domain and range are obviously swapped. An example of this would be *a State has a City,* and the inverse of it *a City is in a State*. So the property "has (city)" is inverse to the property "is in". A property itself cannot be inverse, but only in relation to another property, so there is no "right" direction in that sense, but not every property has an inverse property defined.

2.1.2.8 Reasoner

While the class hierarchy mentioned in Section 2.1.2.4 is given by the transitivity of the superclass and subclass definition, it is not present in data sets like an RDF document. Usually, while not necessarily, only the direct relations are listed there in each case. A simple query to such a document would also not return more than what is directly in the document. In order to draw the conclusion about transitive class memberships, a so-called reasoner is needed.

A reasoner is a program that draws logical conclusions from a given ontology. Depending on the reasoner or setup, these can be of different depths. For example, interheritance reasoners can only create a class hierarchy and make it queryable, they track only the transitive links. More complex reasoners, such as Pellet [41] or HermiT [42] can draw the full spectrum of logically provable inferences, such as inferring classes, using inverse properties, proving unregistered but deducible individuals, and applying rules.

The exact operating principle of a more complex reasoner is, as the name suggests, highly complex, especially in a reasonable amount of time, but it is based on *Tableau Calculus* [43] in which logic is used to try to disprove an assertion and thereby confirming provable facts (for more information, [41, 43] are recommended).

However, this process is computationally expensive, so with an ontology that does not change in the metadata, one can reuse old conclusions and only have to repeat the reasoning when adding new facts. Therefore, it makes sense to store and recall these conclusions, especially for the training phase.

Class	Property	Range	Generated Inverse
⋮	⋮	⋮	⋮
River	flowsThrough	State	false
State	border	State	false
State	hasCapital	City	false
Mountain	elevation	number	false
State	locatedIn_Inv	GeologicalThing	true
City	capitalOf	State	false
State	name	string	false
⋮	⋮	⋮	⋮

Table 2.1: Excerpt of the Mapping Dictionary of the Geo Base Ontology

2.1.2.9 RDF2SQL

RDF2SQL [15] [15] is a framework to translate an RDF ontology into an SQL database and then while still being able to use SPARQL to query the SQL database. *RDF2SQL* does not only translate the query into valid SQL but still can make use of a lot of reasoning feature and provides extensive metadata knowledge called a *Semantic Data Dictionary (SDD)* [15] in quickly accessible tables.

The *SDD* contains both the reasoning data and mapping information about how the data was stored in the SQL database. The latter is irrelevant for this work, but the pre-computed metadata is essential. Here again not all information is used, but only the *Mapping Dictionary* and the class hierarchy. The Mapping Dictionary contains one entry for each combination of class, property and range valid for the ontology. Additionally the *SDD* is extended with the *identifier table*.

This information is used to infer and exclude specific statements in the query generation (Example content is shown in Table 2.1 and usage is shown in Example 2).

Further, it also can create this Mapping Dictionary from a relational database, so this approach gets a certain flexibility in DBMS and does not necessarily need an RDF ontology, but an ontology derived from only a relational database is usually of poorer quality than one created by hand and accordingly degrades the performance of EvolNLQ

Furthermore *RDF2SQL* can be used the other way round and translate the result of EvolNLQ into SQL or XML queries.

Example 2 *As an example for the usage of the SDD, assume the question "What is the capital of Maine?" is being examined. For this purpose, the mapping dictionary and the identifier mapping are used as shown in Figure 2.1.*

1.) Check if there is an instance in the ontology that has a literal recognized as an identifier with the value "Maine".

2.) Check if there is a class or property in the mapping dictionary (and therefore also in the ontology) with

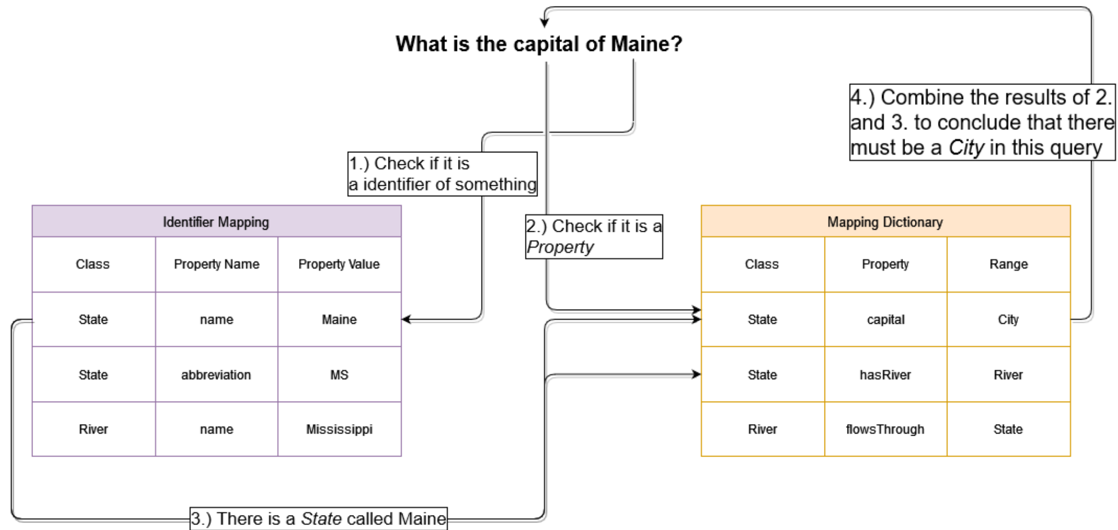


Figure 2.1: Usage of the tables of the *SDD* to infer implicit information about the individuals in the query

the name "capital".

3.) If there is something with the value "Maine" in the PropertyValue column, then it can be checked which class it is. The result is "State".

4.) The mapping dictionary can then be searched for an entry with "State" as the class as found in (3) and the property "capital" found in (2).

From this it follows that there must be an object City as an answer for the question, since this is the only range for a subject of the class State and predicate of the property capital. The value can then be found by the corresponding query against the database.

This example showed a simple case – EvolNLQ will be able to handle more complex queries, using parsing, heuristics and the help of the *SDD* data.

2.1.2.10 Importance of the different aspects of an ontology for EvolNLQ

While formally an ontology can use any names and structure, for EvolNLQ's performance it is important that the ontology is as well constructed and as meaningfully named as possible. Exactly what is the best naming or structuring for an ontology cannot be said in a universally valid way, but in general there are easily understood criteria that are important for EvolNLQ:

- **Naming:** All resources should be named by their name, or as one would name them in natural language. For example, the class for countries could be called Country or State, which does not matter much, but if this class would simply be called C1, many possibilities

to deal with this class are excluded from the beginning.

- **Domain and Range:** The domain and ranges should be restricted as much as possible. A domain of owl:Thing is nearly worthless for EvolNLQ while a restriction to the union set of its actual domains already excludes many wrong results.

Although also important for a good ontology design, the following factors have only little influence on the performance of EvolNLQ, or only in special cases:

- **Disjoint classes:** Classes that are mutually exclusive are not overly important for EvolNLQ, since the answers are sets for which every variable binding is isolated, such that different bindings for one variable can be disjoint classes. For example, it can be assumed that if there is a class *mountain* and a class *lake*, an individual cannot be both, but given a query for all geographic objects in an area, the answer set should contain both mountains and lakes.
- **Class hierarchy** A deep class hierarchy can be useful to answer queries where classes are over- or under-specialized, such as using the word *water* when the class *river* is meant or the word *road* instead of *route*. If the class hierarchy, however, represents commonly known concepts, it is often possible to infer the classes meant in the ontology from the word itself by a hypernym or hyponym (more on this in Section 2.1.3.3).

2.1.3 Understanding the Question: Natural Language Processing

Information for the construction of a query can be obtained on the syntactic and on the semantic level of a sentence. The syntactic level refers to the structure and relative relationships of individual components of the sentence, while the semantic level in turn deals with the meaning of the sentence.

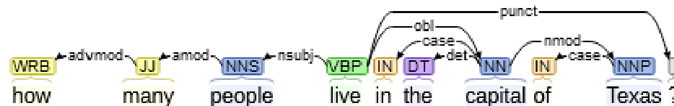
The easiest accessible syntactic information beside the words themselves a sentence can provide is the so-called *pseudocount* position of a word in the sentence. This can be derived by simply counting the words (in case of *CoreNLP* [44] starting with 0) from left to right. This information is easy to get and useful for many simple cases, such as finding compound words. Nonetheless with more complex structures in the sentence this does not reflect the actual logical structure of the sentence and it does not allow for a differentiated distinction between different parts of the sentence.

Provided the language is known, a sentence has much more meta information. Based on the specific rules of the language, this information can be determined and assigned by so-called annotators. This information includes the *Part Of Speech tags* and *Grammatical Relationship*s between the words. In addition, things like sentiment or certain entities can also be found by annotators, but neither used nor computed in this case; since annotating a sentence is actually the computationally most expensive part, the annotator should be used as sparingly as possible to avoid affecting performance too much.

Example 3 *Consider Query 16 from the Geobase query set.*

The sentence "How many people live in the capital of Texas?" is annotated by the annotator CoreNLP [44] (more details about CoreNLP is provided in Section 2.3.1.1).

The sentence itself is shown in the lower part, above are the Part Of Speech Tags of each word in colored boxes (the color is defined by the general type of the tag, for example nouns are dark blue regardless of the exact tag (NN, NNS, or NNP) to which they belong) and the grammatical relationships are displayed as labeled arrows from the governing tag to the dependent tag.



2.1.3.1 Part Of Speech Tag

The *Part Of Speech Tags* determine what role a word has in a sentence. Widely known roles include nouns, verbs, adjectives, or adverbs, but in addition to these, there are over 50 other such tags in the English language. Which tag a word belongs to is determined by the word itself, but also by its relationships to other words and its position in the sentence.

2.1.3.2 Grammatical Relationships

Grammatical Relationships are constructs derived from dependency grammar theory. This theory assumes that every sentence has a finite verb that has the central role in the sentence. In Figure 3, this would be *live*, the *VB Part of Speech*. From this finite verb, relations are formed with other words that are considered dependent of this word. From these words, further dependencies to other words can be formed. The dependent word is called *dependent* in the grammatical relation and the word it depends on is the *governor* in this relation. Thus, many words are both *dependents* and *governors*, depending on which relation is considered. The finite verb is the only non-dependent word. This creates a tree that contains all words, and has no cycles. So no word can be indirectly both *governor* and *dependent* of another word and further there is always only a single Grammatical Relationship between two words [45].

Each Grammatical Relationship has a type, which describes what kind of relationship the two words and have with each other. Some Grammatical Relationship also have a subtype, which further refines the description of the relationship. Every instance of a Grammatical Relationship is represented graphically as an arrow labeled with the type t from the governing word w_2 to the dependent word w_1, or as a slotted frame $[type \rightarrow t, gov \rightarrow w_1, dep \rightarrow w_2]$.

This is of course a severe simplification of this topic. However, no more than this is used to answer the queries. So, both the existence or non-existence of relations are considered, as well as the types and subtypes, which are used to make decisions, set priorities, and filter results.

2.1.3.3 Synonyms, Hyponyms, and Hypernyms

As mentioned in Section 1.1, natural language is often very rich in variations. In communication with people, they can be used to express moods, facts, or relationships, or it can simply arise from regional differences in the language and many more reasons. The database on the other hand, has a very limited and specific vocabulary and most of the subtext of a query simply does not matter for the query.

Thus, the rich natural vocabulary must be mapped to the restricted vocabulary of the database. In linguistics, these mappings are called *synonyms*, *hypernyms*, and *hyponyms* and are extensively studied since the 17th century, first primary in french but nowadays for most common languages.

Synonymity is the relationship between two words that have an equivalent meaning. Thus, provided that the terms of the database are characterized by meaningful and differentiated terms, the vocabulary can be expanded so that all synonyms are also mapped to their respective part in the database.

Hyponymity and hypernymity describe the relationship between a generic term (hypernym) and a more specific term (hyponym). This corresponds to the idea of a class hierarchy in Section 2.1.2.4 and can therefore be used to map expressions that are formulated too precisely for the more general concepts of the database.

Linguistics has several other concepts besides these that describe relationships between words, and hyponyms or hypernyms could be pursued arbitrarily far to identify larger relationships. However, they must all be taken with a grain of salt, as they can often be off-target quite easily, especially when the database describes a rather limited but detailed area where the precision of the terms makes serious differences.

An even greater problem is posed by incorrect synonyms, as they are very often used in the spoken language. In the following example a query is asked, with the intent to receive the population of New York.

Example 4 *Variations of the same translated query, with incorrect synonyms.*

- ***A.*** *What is the population of New York?*
- ***B.*** *How many people live in New York?*
- ***C.*** *How many citizens live in New York?*

If the three queries in Example 4 are considered, it is clear to a human reader that they actually mean the same thing, but variation C, for example, would exclude all people without registered residence. Therefore citizen is strictly not a synonym of population, also considering that the usage of population in other contexts like the population of a biological biome. Also neither citizen nor population is a hypernym of the other and even if is possible to find a common hypernym for both, that would include many other wrong words as well.

This linguistic imprecision is a severe problem for most NALIBs and often determines the performance of an approach severely. This problem can be approached with vector space mapping of words like [46] or with a specially crafted word mapping like done in [4]. In EvolNLQ, neither is used since they did not produce the desired results and those information can be better deducted from the metadata in cases of smaller datasets.

2.2 The Big Picture: Employing an Evolutionary Algorithm

*Evolutionary algorithm*s are based on the idea to use the evolutionary selection mechanism described by Darwin in the context of optimization problems in computer simulations. Different approaches towards similar concepts were developed independently from each other in the early 1960s. Those were classified as greedy optimization algorithms. Later, the category of *evolutionary algorithm* was created for them, classified as a supervised learning algorithm. It aims to simulate a selection process like it happens in nature to converge towards an optimal solution for a specific problem.

For this purpose, subprograms called agents (see Section 2.2.1) are created, which can be changed to a certain degree.

The agents themselves and also the changes come in many different designs and are used for many different application areas. An agent can be an array of bits or a program, a tree or a series of sequences. An agent can be trained for some kind of task, or just be created to find the optimal solution for a specific task. It can also be the solution itself; for example, a sequence of numbers that specifies the order of visited cities of the *Traveling Salesman Problem*.

The state of those changes is called a *configuration*. Depending on the implementation, these changes can be of almost any nature. The important thing is that they lead to different results, such that changing the configuration may lead to a different behavior of the agent.

A training session, called a *simulation*, starts normally with agents with random configurations, but there are also approaches which have other strategies for the initialization. The agents receive an input and a task which is executed according to their configuration. After that, the results of the agents are evaluated. The evaluation is done by a so-called *fitness function*, which determines the degree of success of an agent, the *fitness score*. The fitness function acts as the inverse supervising error function in this situation.

According to the chosen method (e.g. the best rated) agents get the opportunity to reproduce. Depending on the approach the reproduction is sexual or asexual, meaning one or multiple agents contribute to the creation of an offspring. These offsprings are copies of existing agents, which have a low probability of a change in their configuration, i.e. they can be *mutated*. In some approaches the configuration of agents can also be mixed, a so called *crossover* which takes some value from one agent and some from another.

The exact copies preserve successful species, while the *mutations* or *crossovers* may create better versions that drive the optimization process.

After the creation of a new *generation*, a single iteration of the algorithm, also called *run*, is completed. This is followed by the next run, however, with the new created agent of the new generation, but with the same task. This is repeated until the termination criterion of the specific implementation is fulfilled. Since these algorithms cannot tell whether the optimal solution is

actually reached, they normally terminate after a certain number of time or runs, or when a specific result or score is achieved.

2.2.1 Agents

Agents are subprograms of an *evolutionary algorithm*. Depending on the concrete implementation and the type of the *evolutionary algorithm*, agents are able to change their behavior to a certain degree. This can mean that an agent is a array of values [47], a program [48], a function which parameters are changed [49], a sequence of instructions that reorders [50], or a tree of operations [51] – which is the case in this work.

Regardless of the exact implementation, the state of an agent is called a *configuration*. All agents with the same *configuration* are called a *species*. Members of a *species* behave exactly the same. An agent has only a certain number of possible changes, which is described as its set of operations. How exactly these are executed and in which order depends on the type of algorithm. Some types perform only one possibly very complex operation, others very simple ones, but any number of them.

An *evolutionary algorithm* starts with a set of *n* either randomly generated agents or agents with a specific start *configuration*. This set and each set of agents created by an iteration of the algorithm is called a *generation*. *Evolutionary algorithm*s evolve slightly changed new generations based on the previous one to optimize the agents toward an optimal solution. In every *generation*, every created agent is evaluated by a metric described by a fitness function calculating the degree of success of an agent. During *reproduction*, the *configuration* of the new agent has a probability to change (either through recombination of already existing features or generation from random ones) in a more or less slightly manner.

The set of all newly generated agents forms the next *generation* of agents. Those new agents execute the task again and they are again evaluated by the fitness function, repeating the cycle. While the unchanged agents should get the same score assigned as their parents, the *mutations* might have a another score due to the changes in their *configuration*.

It is likely that random changes to the agent might not be an improvement or even render the agent dysfunctional, especially in later stages of the algorithm improvements are becoming increasingly rare. Therefore it is necessary not to mutate every offspring, so the next generation might contain the successful agent configuration of the previous generation. Less frequent, but all the more important, are the mutations that change the behavior of an agent in such a way that it generates a better solution with respect to the *task*s given to it. This should result in a higher score of the fitness function, provided that the function can detect an improvement of this type and magnitude. Depending on the selection function, this score does not necessarily lead to the survival of this agent (most functions, however, guarantee the survival of the best agent), but at least its chances to survive should be better. Since only a subset of the *generation* is allowed to reproduce, a higher

score increases the probability for a single agent to be part of this set. The subset of agents which are allowed to reproduce is determined by the selection strategy of the approach. Those selection strategies always refer in some way to the fitness score either by ranking agents directly or increasing the likelihood for an agent to be chosen. With most strategies the set of reproducing agents should be smaller than the whole *generation* and therefore some agents, which were either unlucky or are ranked too low, do not reproduce and drop out of the simulation. Traditionally every generation consists of fixed number n of agents. Since not every agent reproduces, some agents have to produce multiple offspring and after a few *generations* the *species* of a successful agent might provide all agents and extinct lower scored *species*, advancing the evolutionary process.

In this case the other species will no longer be part of this simulation and the new dominant species might become extinct as well[3], as soon as a new more successful *species* is created by a mutation.

2.2.1.1 Fitness Function

The fitness function calculates the degree of success of an agent with respect to its task. The function can be arbitrarily complex, but should not be too computationally intensive, since it must be executed for each new *species* of agents. The more accurately a function can assess an agent's progress, the faster convergence to the optimal solution can be expected. Originally it was defined that the fitness function can only be related to a single goal, but as [52] has shown, this can be extended to any number of goals.

2.2.2 Types of Evolutionary Algorithms

In the following the four main ideas are introduced: *evolutionary strategies*, *evolutionary programming*, *genetic algorithm*s, and *genetic programming*. All of these approaches are quite similar and a transition of one to the other should usually be easy to do, nor are their ideas mutually exclusive, but can be blended together. Therefore for many practical implementations it is not possible to clearly delineate which approach is followed. EvolNLQ, for example, primarily uses *evolutionary programming* but also approaches from *genetic programming* and *genetic algorithm*s.

2.2.2.1 Evolutionary Strategies

Evolutionary Strategies, were developed as part of Ingo Rechenberg's doctoral [49] dissertation in 1964. The work dealt with problems in hydrodynamics for which there were no classical mathematical solutions and was original called $(1+1)$-ES. This approach defines a vector consisting of the input parameters for a function $\overline{x}^0$. Another vector $N(0,\sigma)$ of independent Gaussian numbers with median zero and standard deviation σ is added to $\overline{x}^0$. With those the new Generation $\overline{x}^{t+1}$, with t referring to the current generation, is produced with following equation:

$$\overline{x}^{t+1} = \overline{x}^t + N(0,\sigma)$$

[3] Similar to the corona virus and its diverse variants

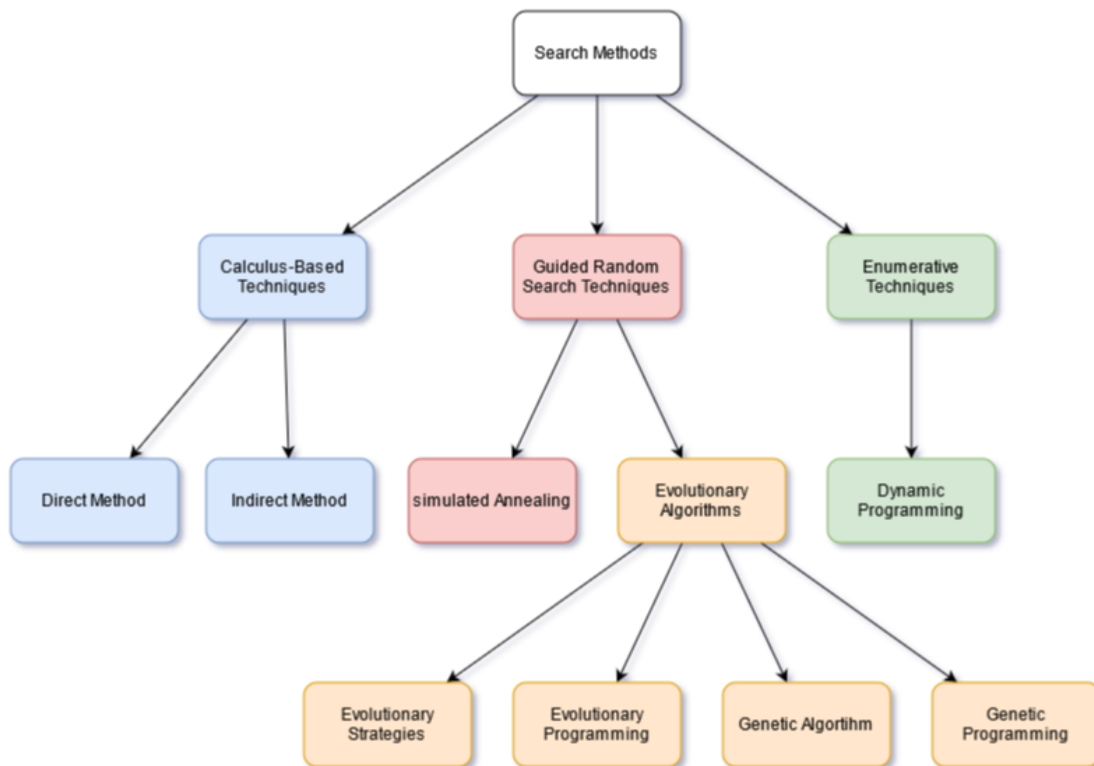


Figure 2.2: Contextualisation of *evolutionary algorithm* with other search methods based on [50]

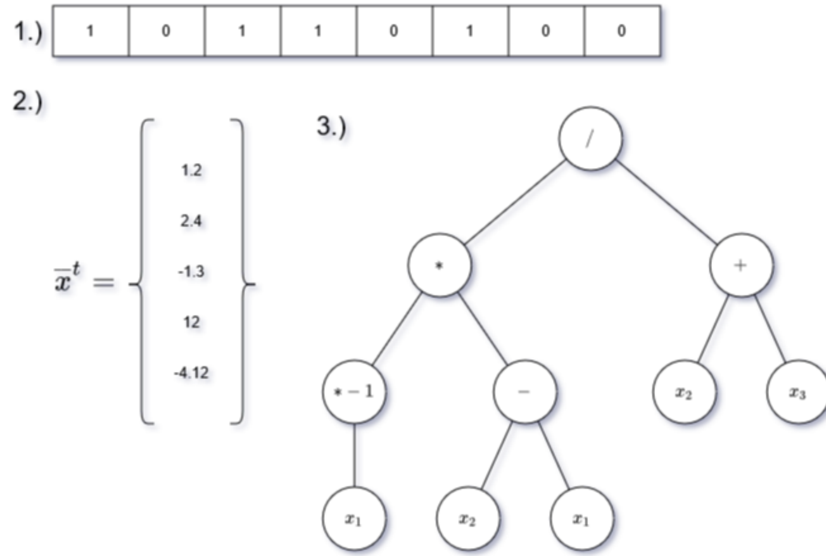


Figure 2.3: Schematic representation of the different agents of different types of *evolutionary algorithm*.

1. Is a typical agent of a *genetic algorithm*s
2. Is an *evolutionary algorithm* and
3. Could the result of either an *evolutionary strategies* or *genetic programming*

Then those parameters are applied to the problem and the better result of both is used for the next run. Starting from this base, some changes like the number of agents involved and the change of not only the parameter vector but also of the settings, like the standard deviation, were added [53].

To define σ, Rechenberg later developed the "1/5 success rule" [49]. In this rule, the standard deviation is adjusted every n mutations according to the size and success, with n being the number of parameters, so that $\frac{1}{5}$ of the mutations result in an improvement. The rule consists of the following:

$$\sigma(t) = \begin{cases} \sigma(t-n)/c & \text{if } p_s > \frac{1}{5} \\ \sigma(t-n) * c & \text{if } p_s < \frac{1}{5} \\ \sigma(t-n) & \text{if } p_s = \frac{1}{5} \end{cases}$$

Where p_s is the relative success rate and c is a constant value. The value of c was theoretically derived by Schwefel with $c = 0.817$ [54].

By dynamically adjusting σ, a faster approximation can be achieved by large deviations at the beginning and later fine tuning can be done for finding the exact solution with lower standard deviation.

[53] [54] list as examples in which Evolutionary Strategies are used

- Routing and networking
- Biochemistry
- Optics
- Engineering design
- Magnetism

2.2.2.2 Genetic Algorithms

*Genetic algorithm*s were created by John H. Holland in the early 1960s [47], at this time called *genetic reproductive plans*. Similar to natural selection, this method aims to achieve progress primarily through the recombination of proven traits and unlike *evolutionary strategies*, relies only secondarily on mutation.

Similar to sexual reproducing creatures, in which most developments result from the two sets of chromosomes of the parents being brought together in different ways, this method also relies primarily on so-called *crossover*, which corresponds to this process at the data level.

For this purpose, first of all an encoding of the behavior is designed and each agent gets its own sequence of encoded behavior. Traditionally this sequence is a bit array of fixed length and only binary encoding is allowed, but as shown later any fixed number of symbols can be used. The whole sequence is called the *genome*, while the position in this sequence is called a *chromosome* and the actual value of this position is called a *gen*. The process of encoding behavior into the *genome* is very similar to the parameters of *evolutionary strategies*, and must also be determined individually for each problem.

During reproduction, each reproducing agent is assigned a mate and the *genome* of the offspring is a combination the *genomes* of the parents.

There are different approaches for determining which agents are allowed to reproduce and who with whom. They are categorized by Coello Coello [53] as followed:

- **Proportionate Reproduction:** These methods are based on the a probabilistic selection that is proportional to the agent's performance. It is often also extended with additional functions or a part of one of the other methods is used for a subset of a *generation*. The exact method can vary; often *Monte Carlo* or *roulette wheel* selection [55], *stochastic remainder selection* [56], or *stochastic universal selection* [57] are used.
- **Ranking Selection:** This selection was proposed by Baker [58]; the population is sorted from best to worst, and each individual is copied as many times as it can, according to a non-increasing assignment function, and then *proportionate selection* is performed according to that assignment.

- **Tournament Selection:** The population is shuffled and then is divided into pairs, from each pairs the better agent is selected for reproduction. This can be repeated to refine the selection further. Therefore the resulting selection size m after k repetitions can be calculated with the formula:

$$m = \frac{\textit{population Size}}{k}$$

  This method has the interesting property that in any case the best agent is included, but also other not so successful agents. Further the worst agent is excluded for sure. The selection generated this way tends to consist of the better agents and some of the upper midfield. By allowing less successful agents, the diversity of the population increases and more potential recombination possibilities arise.

- **Steady State Selection:** This method is primarily based on maintaining the successes already achieved. For this purpose, only a few of the worst agents are replaced by the recombination of the other agents. For the determination of the pairs, one of the *proportionate reproduction* methods is used. In this way the middle field is preserved, but has a subordinate role in the reproduction. This approach is mainly used for incremental learning.

Each reproduction produces two offspring; for each section of the genome, one offspring inherited from one parent, the other from the other parent. In the end of this process no genome section is used twice or not at all. This procedure is called *crossover*.

The three major *crossover* strategies are listed in the following and are shown in Figure 2.4, the parents are noted as p_1 and p_2 and the offsprings as o_1 and o_2:

- **Single-point crossover:** The strategy chooses a single point c within the genetic code of the parents and then assigns the genetic code from p_1 up to the c-th sign to o_1 and after the c-th sign the rest of the code is taken from p_2. This is repeated with o_2 but p_1 and p_2 swap places.

- **Two-point crossover:** The two point strategy defines two random points c_1, c_2 within the length of the code and then copies the genome of p_1 to o_1 but replaces the code between c_1 and c_2 with the corresponding part of p_2. Again the same process happens, but with swapped places for o_2.

- **Uniform crossover:** This strategy determines the code of o_1 for each position based on a probability pr . If the genome of p_1 or p_2 is chosen for this position, the leftover sign is then assigned to o_2. Based on pr the severity of the changes can be influenced.

The resulting mixture of genomes is then the genome of the offspring. Subsequently, random changes can be made to the genome of the new agent, i.e. mutations can be carried out.

Some representative applications of*genetic algorithm*s are the following [48] [53]:

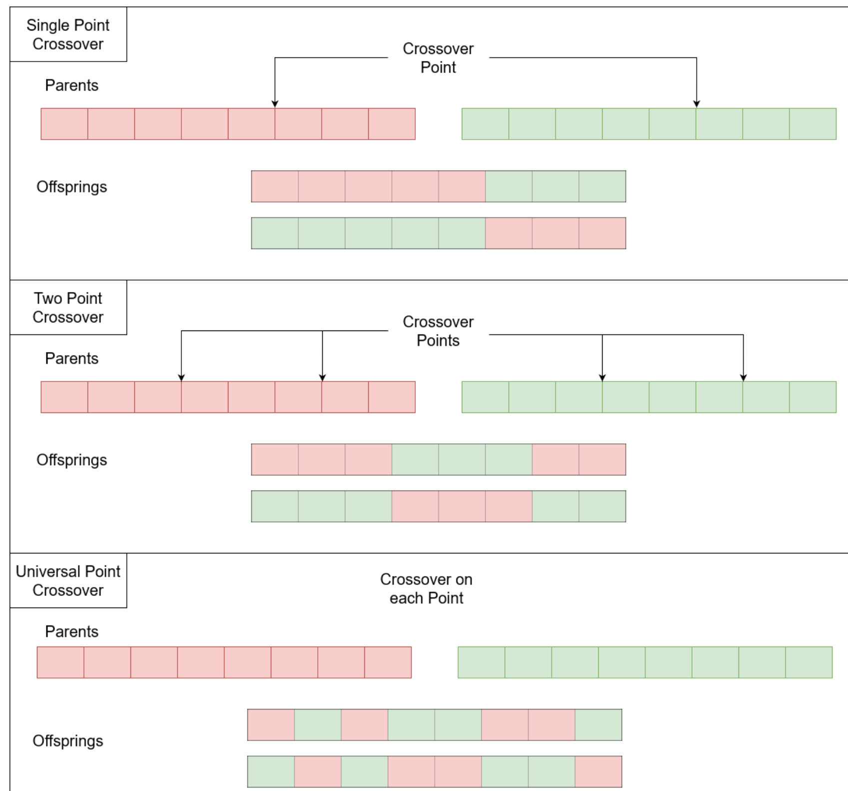


Figure 2.4: The three most important crossover strategies.
The genome is represented by a line of eight boxes, chromosomes of o_1 are red and chromosomes of o_2 are green. Note that this does not necessary mean that they are different (but might be), the color only determines the origin of the chromosomes.

- Optimization (numerical, combinatorial, etc.)
- Machine learning
- Databases (optimization of queries, etc.)
- Pattern recognition
- Grammar generation
- Robot motion planning
- Forecasting

2.2.2.3 Evolutionary Programming

evolutionary programming was developed by Forgel in the 1960th. "*Evolutionary programming* emphasizes the behavioral links between parents and offspring, instead of trying to emulate some

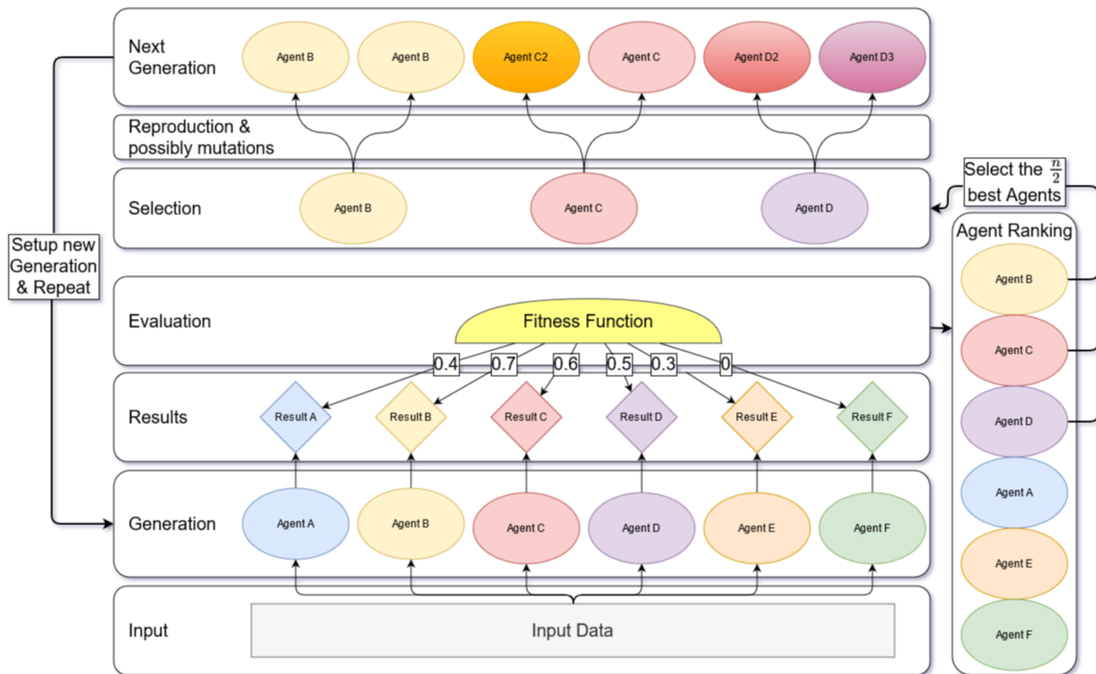


Figure 2.5: Example representation of a single run of *evolutionary programming*. The agent population size is $n = 6$ and as a selection strategy the better half of the agents is allowed to produce two offspring.

specific genetic operators (as in the case of the *genetic algorithm*s [48])." [53]

It has many similarities to *evolutionary strategies* in that it also does not rely on recombination but uses only mutations to evolve a group of individuals. Thus, reproduction is asexual and each parent produces not only one offspring. The selection of the offspring is probabilistic, which is the biggest difference to evolutionary strategies. Furthermore, no encoding of the problem is done, but a set of domain-specific methods is provided, which produces a solution.

Using these operations, agents are then mutated, evaluated on a fitness function, and mutated again. The process is illustrated again in Figure 2.5.

Some representative applications of *evolutionary programming* are the following [59] [53]:

- Forecasting
- Generalization
- Automatic control
- Traveling salesperson problem
- Route planning
- Pattern recognition
- Neural networks training

2.2.2.4 Genetic Programming

Genetic progamming was created with the ambitious idea of writing a program that writes arbitrary other programs. Since the search space for such a task was simply too large, this approach was so unsuccessful, that it was heavily criticized by other AI researchers [59]. Only much later a simplified search space was introduced by Koza [51], in which instructions are strung together tree-based, which were defined before. The depth of the tree is also restricted to prevent infinite growth. Despite existing problems, his approach was successfully used in some fields [13].

This method, as well as *genetic algorithm*s, is based primarily on the use of crossover operations and only secondarily on mutation. The difference, however, is that there is no genome. Thus, the crossover points are no longer in a genome, but at the nodes of the tree structure and subtrees are exchanged accordingly.

Unlike a fixed-length genome, it is no longer possible to say exactly which node corresponds to which node in a tree. Therefore, arbitrary subtrees are exchanged at arbitrary positions, which leads to trees of extreme size, which can block the entire learning process. Therefore, a maximum depth is usually specified and the nodes must have a fixed number of children.

Koza introduced the following operations [53] [51]:

1. Arithmetic operations $(+, -, /, *)$
2. Mathematical functions (e.g. sine, cosine, logarithms, etc.)
3. Boolean Operations (e.g. AND, OR, NOT)
4. Conditionals (IF-THEN-ELSE)
5. Loops (DO-UNTIL)
6. Recursive Functions
7. Any other domain-specific function

Those restrictions are rather recommendations, especially through the last point, which basically could mean anything.

2.2.2.5 Shared Algorithmic Properties

With infinite time and an unrestricted number of possible mutations, this algorithm finds the optimal solution within the search space restricted by its operation and according to the fitness function. However, the algorithm cannot tell by itself whether the current best solution is actually the optimal solution.

Therefore a time restriction or a score threshold is needed as long as the optimal fitness function score to terminate the algorithm is unknown. Moreover, *evolutionary algorithm* are greedy algorithms. So they are an efficient solution for many complex problems, but they also bring the usual problems of greedy algorithms (for more details see [60]).

In particular, they are prone to being trapped in a local maximum [13]. If the agents have evolved in a direction that does not lead to the actual optimal solution, but is in a local maximum, it would have to make such a drastic mutation that it not only leaves the maximum, but also finds a configuration that achieves a score that is higher than the one of their predecessor on the first try. With infinite time and arbitrary extensive mutations this is possible, but in practice this can be a big obstacle which might not be overcome in a reasonable time span. The issue is addressed with *memetic algorithms* which use so called *memes* to combine other learning techniques with *genetic algorithm*s. The *memes* can be any general learning technique, often *hill climbing* or *hyper heuristics* are used. The *memes* are used on individuals during reproduction to guide their development. This was done With a certain degree of success by [61] but nevertheless when implementing those kind of algorithm one should keep that weakness in mind. In the end, running the algorithm multiple times, with randomized or altered starting conditions and (hopefully) not falling into one of the local maxima at least once stays kind of a trick of the trade either way [13].

2.2.3 Multi-objective Optimization

In the traditional *evolutionary algorithm*s, the fitness of each agent is computed on a single objective and it is not always possible to determine whether the found solution is the best possible. This works fine for problems like the *knapsack problem* or the *traveling salesman problem* when the fitness function is focused on a single objective, like raising the value of the knapsack content or decreasing the length of the distance to travel, even without knowing when the optimal solution is found. However in case of NLQ processing and many other problems, a single objective is not enough. In so called *Multi-Objective-Optimisation (MOO)* [62,63] every part of the query must be considered an objective on its own, since it is actually not meaningfully possible to evaluate which parts of the query contribute how much to the optimization.

There are basically two different approaches how the calculation of the *fitness score* in MOO can be done, but in the end it comes to having only one single score, which can then be used to evaluate and rank the agents accordingly.

Linear Function The simplest approach is to use the sum of the fitness functions of the single objectives $f_1, f_2, \ldots, f_x$. These can then still be optionally weighted, resulting in a linear combination of the form

$$f = \sum c_i * f_i$$

where the parameters $c_1, c_2, \ldots, c_x$ are suitable constants for each objective. This method, in addition to its simplicity, is especially usable for any number of targets without any problems [62,63].

Multiplicative Function Another possibility first presented by [64] for two objective problems is to form and use the square of the error of both used functions, so as to restrict one-sided evolution. This approach can be extended arbitrarily in the following form:

$$f = \prod f_i$$

However, the problem here is that each objective must be of extreme importance. A single unmet objective will cause the complete score to be zero. This is certainly not a problem for a function with only two goals, but in the case of EvolNLQ, many goals are initially unsatisfiable for new agents and the number of goals is always much greater then two. Thus, no development would occur with this approach, since no agent would ever meet all objectives immediately ($f_i > 0$) and therefore never achieve a score above zero.

Objective Hierarchy For this approach a modification of the linear function was chosen. Goals can get subgoals, which are refined again and again. Thus, each goal is also a linear MOO function in itself, in which they use the weighted averaged sum of their subgoals and possibly also an own

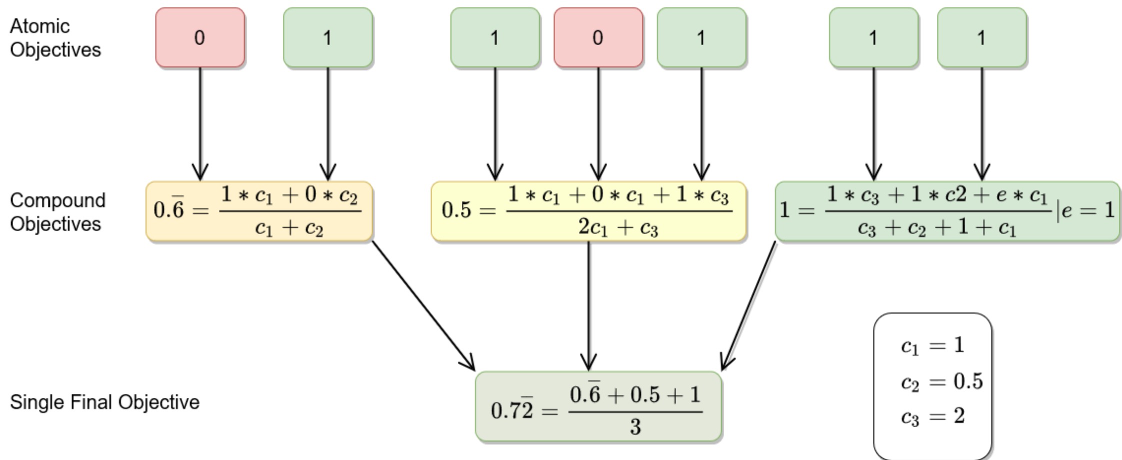


Figure 2.6: Example for a MOO composed of 7 atomic objectives and one compound objective with an own objective value e.

goal value to determine their own score. Thus, in the end, there is only one goal, which has a tree of subgoals, each composed of weighted sums of its subgoal components, up to atomic objectives, which cannot be further subdivided, but are also in themselves a single goal, which then gets a binary score.

2.2.4 Diverse Challenges and Sticky Feet

Although all types of *evolutionary algorithm* try to mimic natural evolution in one way or another, the aspect of species diversity and specialisation remains quite absent. Even in strategies that allow for a greater variance of types of agents, success is still tied to the overall success of the agent, and a highly specialized agent has virtually no chance of long-term success. Meanwhile, in nature, highly specialized species are commonly found, especially in more extreme habitats.

Considering the application area of query translation, there are always recurring types of queries with certain requirements to be answered correctly, but also rarer occurring queries, which have other requirements, have to be handled.

Now, one could create a separate query set for each of these types and separately train agents on them, but deciding which types of queries exclude which other types and for which there is a possible agent that can answer both ultimately already requires knowledge of such an agent and makes the whole process unnecessary.

To stay with the biological example, there needs to be a mechanism that enables agents that do not compete for the same food sources to also not compete for reproduction possibilities. So there would have to be a way for specialists to also be successful agents in terms of getting reproductions granted and evolve undisturbed of the agents which deal with other problems, but at the same time,

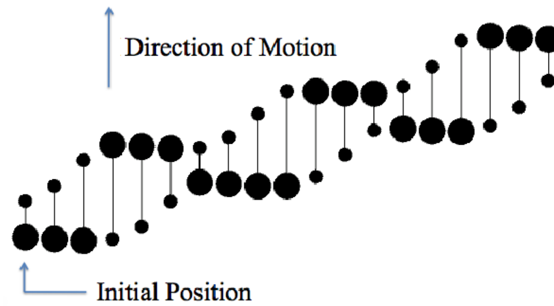


Figure 2.7: "A simple one-segment creature that moves upwards. The creatures initial state is shown at the left, and subsequent positions in time are shown to the right of this. Points with high friction have a large radius, and the smaller points have lower friction" [67]

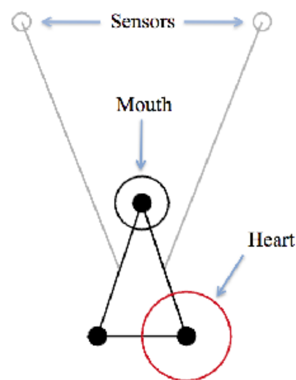


Figure 2.8: "A simple triangular creature with three pointmasses and three segments. Two of the segments have sensors attached to them that extend in front of the creature. The mouth of this creature faces in its direction of motion, and its heart is in a trailing position." [67]

it is not possible to just keep any agents which might be completely useless, without clogging the whole computational process. Further the problem is a bit more complicated, since most queries can combine several typical elements in different ways, but also parts of the queries actually appear in every query. Therefore, even specialists will have to master the basic parts and the sets of queries will not be completely disjoint in terms of typical constructs.

This problem with *evolutionary algorithm* has been studied and solutions have been found in the field of Artificial Life Simulations. Even though this area has nothing to do with database queries, there are some ideas on how to implement this specialization in the context of queries as well, in particular Turk's *Sticky Feet Creatures* [65], and the extension *Energy as a driver of diversity in open-ended evolution* [66].

2.2.4.1 Sticky Feet Creatures

Sticky Feet Creatures is an Artificial Life simulation by Turk [65] in which creatures evolve from initially very simple forms into more effective and complex creatures. An *evolutionary algorithm* is used to simulate this evolution. Each of these creatures consists of a set of nodes, edges and sensors as shown in Figure 2.8. In addition, each creature has special nodes, a heart and at least one mouth. Whenever a creature has moved its mouth on top of the heart of another creature, the creature is considered "eaten" and is removed from the simulation and the eating creature produces an offspring that may mutate. By mutations components can be added, removed or changed. Each component has a parameter that can also be changed by mutations. The edges have a changeable parameter that determines how fast they oscillate, the nodes have a parameter that indicates at which intervals they increase or decrease their mass to change how movable they are. Combined, those properties enable the creatures to move as shown in Figure 2.7. The *heart* leaves a trail on the environment and sensors that come into contact with such a track are activated, thus influencing the behavior of nodes by changing their mass or the behavior of edges by changing the oscillation speed. Correctly done, this can lead to a tracking behavior, which increases the chances of finding the heart of another creature through random movement.

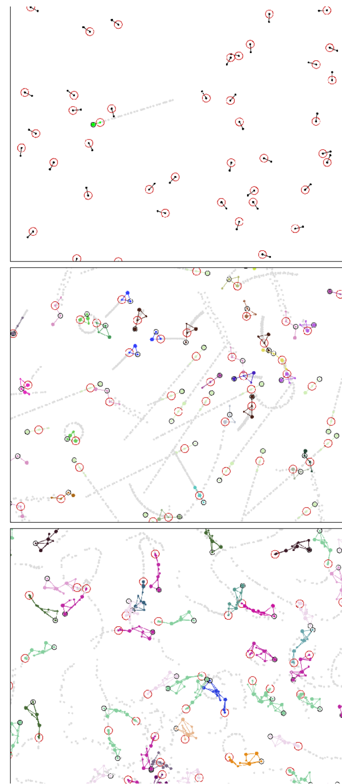


Figure 2.9: "The initial state of the simulator, with a single green moving creature (top), and later snapshots of such a simulation run (middle, bottom)." [67]

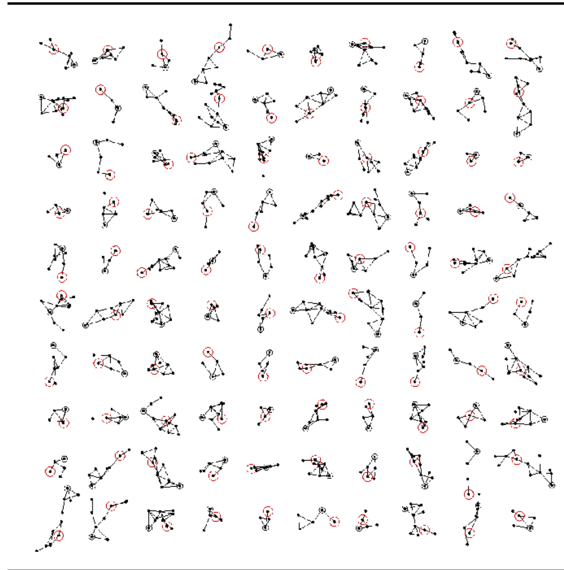


Figure 2.10: The most successful creatures from 100 different lone ancestor simulations. Each represents the most numerous type of creature at time step 2,000,000 for a particular simulation. [67]

Given enough time for each run, an effective creature will emerge and continue to optimize itself. The creatures of different runs are relatively different, even though they are ultimately optimized to move their mouths onto the hearts of other creatures. Within a run usually only one or very few species prevail.

2.2.4.2 Diversity

The lack of diversity can be a problem if the goals are more diverse as well. Assigning a single total score to evaluate an agent and to base reproduction only on that one score then leads to the problem that special cases are ignored as long as they are contradictory to more common cases. As an illustration of this problem one might assume an agent which is able to answer a very specific and possibly very difficult task. Now the fitness function should credit it with an appropriate score, but the agent is only able to do this because it cannot find the correct solution for relatively simple, perhaps frequently occurring tasks, for which the agent then receives no points or negative score points. Therefore the specialised agent will always be in the score below a generalist agent who can answer most of the "low hanging fruits". This leads to the fact that only agent configurations that can fulfil many and frequently occurring tasks will prevail. On the other hand, tasks that require a configuration that is unsuitable for most other tasks can only be solved by agents that are not among the top scored agents, and thus are unlikely to remain in the simulation.

Similar concerns are tackled by Hoverd and Stepney in their paper *Energy as a driver of diversity in open-ended evolution* [66]. They state that the diversity in Turk's simulation [65] was usually

extremely low and in the long run, a single species survived because it was superior to all others.

Sometimes a scissors, stone, paper principle may arise in which one variation was superior to the other, wiped it out, a third variation caused the second to strive out, and then a variation evolved that was again similar to the first and very well suited to hunt the third. After that, the cycle began again. While this problem arose from the fact that the agents got their legitimisation to propagate from eating another agent and thus had to respond to the dynamic evolution of their environment and a race condition arises. Hoverd [66] took a look at the inspirational natural biomes and argued that there is not only one type of very flexible omnivore, but also many specialists that use only certain, less competitive food sources. In biology one would speak of a niche, but neither the classical *evolutionary algorithm* approaches nor Turk's simulation allow such a behavior.

2.2.4.3 Energy-Based Reproduction

Motivated by the above, Hoverd and Stepney introduced two major changes to the simulation;

1. They introduce a new variable in the simulation: energy.
2. They removed reproduction through replacement, such that agents do not automatically reproduce as soon as they eat another one, but they can reproduce if they have a sufficient energy level.

In their approach, agents have a constant energy consumption, which is determined by the number and type of components they consist of as well as by their activity. Thus, the more complex or active an agent is, the more energy it consumes. Also, each part that makes up the agent has a certain energy cost. When an agent reaches a certain multiple of its component cost as its energy stock, it reproduces and divides the remaining energy between itself and its offspring. Energy can be obtained in two ways, either eating another agent, getting its supplies and part of its component cost, or collecting the energy from the *environment*. For this purpose, each environment is assigned a passive energy value, which is supposed to work about the same way as sunlight in nature. Agents can collect this energy with certain body parts, according to the area they cover. This provides less energy than eating an agent, but can be reliably collected without moving. The idea is that now "plants", " herbivores" and " carnivores" can evolve and all forms balance each other to form a stable and diverse population. Thus, in addition to the main goal of simply surviving and maintaining their species, these agents have set themselves different goals in which they evolve to fill a particular role as efficiently as possible. Applied to other problems, this is a way to get specialists for certain subtasks, defined by the algorithm itself. Provided that one can now determine the quality of a result, one can decide which agent has delivered a satisfactory result for which task and then use its behavior to generate a solution for this task.

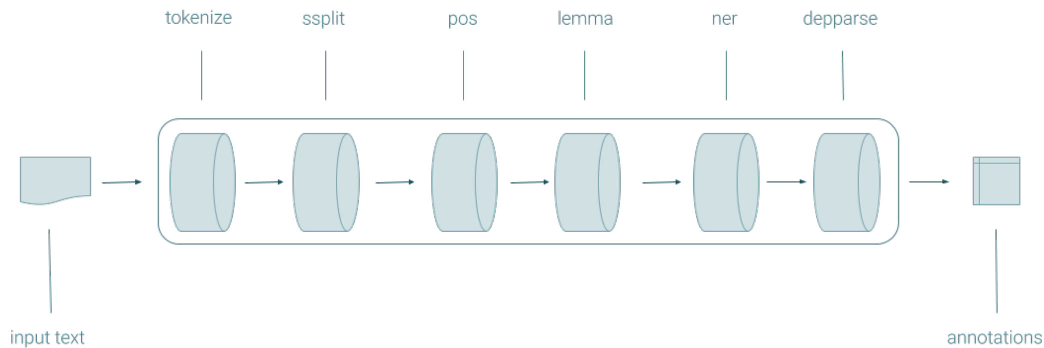


Figure 2.11: Visualisation of the Pipeline infrastructure of CoreNLP

2.3 Miscellaneous

In this section information necessary and worth to be mentioned is given which did not fit elsewhere.

2.3.1 Used third party Programs

Beside *RDF2SQL* the other most important third party program used in EvolNLQ is CoreNLP.

2.3.1.1 CoreNLP

CoreNLP [44] by Manning et al. is a natural language processing tool written in Java. According to [68], CoreNLP enables users to derive linguistic annotations for text, including token and sentence boundaries, parts of speech, named entities, numeric and time values, dependency and constituency parses, coreference, sentiment, quote attributions, and relations. CoreNLP currently supports 6 languages: Arabic, Chinese, English, French, German, and Spanish.

CoreNLP uses a pipeline architecture, thus only the intended annotators have to be used to reduce computation time. The pipeline takes raw text and then incrementally generates the desired result. For this thesis only the output of the part of speech tagger [69], the dependency parser [70] and the named entity recognition [71] are used, but since the dependency parser is last in the pipeline, shown in Figure 2.11, all components were needed.

The result is presented in form of *CoreDocument* instances, data objects that contain all of the annotation information, which then can be interacted with among others directly in Java.

For all visualization of the results of CoreNLP the online tool [4] from the official website was used, because it presents the results in an appealing and clean way.

2.3.1.2 WordNet

Additionally *WordNet* [72] play an important role for EvolNLQ. *WordNet* is a lexical-semantic network of the English language. Is has been started 1985 at the Cognitive Science Laboratory of Princeton University and is still developed and carefully curated. The corpus provides a wider range of information than simply storing synonyms, but providing a network of relation between words it is more than just a thesaurus. *WordNet* comes with an API to utilize those information appropriately. It is used in EvolNLQ to access the data about synonyms, hyponyms and hypernyms of words. This API is available in many different programming languages, among others in Java.

Further, *Apache Jena* [73] is used for SPARQL query construction and evaluation, *JDOM* [74] is used for reading and writing XML documents and *JFreeChart* [75] is used to visualize the learning progress during the training phase.

2.3.2 Pathfinding

Pathfinding algorithms are widely used and are a very thoroughly researched field. Before computing power was actually available to fully utilize them, mathematicians already developed efficient and correct algorithms like Kruskal [76] and Dijkstra [60]. These algorithms find, in a graph with weighted edges, the path between two nodes with the lowest sum of the weights of the edges used. The most obvious use for path-finding algorithms is to find paths by either interpreting intersections as nodes and the connecting roads as edges, such as for navigation systems, or by dividing the space itself into regular discrete positions that act as nodes and, depending on the nature of the space at the location, creating connections between adjacent positions that act as edges, such as is used for movement on a board in computer games.

Path finding problems are a special type of optimization problems with algorithms which can efficiently solve them. For example, in the case of EvolNLQ, an *ontology* is translated into a graph where classes act as nodes and properties act as directed edges from their domain to their range. Resources in a query can then be appropriately integrated into the graph, changing the weights for known elements such that the most favorable path between two resources inserts as few additional elements as possible. This way, intermediate resources not explicitly mentioned in the query can be found.

2.3.2.1 Yen's Algorithm

The algorithm of Kruskal [76] finds only one of the shortest paths. However, under certain circumstances further paths could be interesting, in particular paths of the same distance or such

[4] http://corenlp.run/

(possibly longer) paths that have certain properties. Therefore it can be useful for the application to find not only the best, but the k best paths.

For this purpose the algorithm of Yen [77] can be used. [5] It uses other path finding algorithms to determine one of the shortest paths, then removes one edge (or more) from this path and determines the next best path without the edge(s). This is done for all edges of the current best path. Then the shortest path, which is not already in the result set, is taken from the set of recomputed paths and added to the result set. The resulting ordered set of paths can then be considered for further evaluations.

[5]For this approach, a modified version of Brandon Smock's implementation is used [78]

Chapter 3

The Evolutionary Dataflow Agents Framework

3.1 Motivation

The agents of EvolNLQs are based on the idea of the *dataflow computer* architecture principle [79], extending it with *evolutionary algorithm*s in a way that has many similarities with *genetic programming*. Therefore they are called evolutionary dataflow agent. However, for the sake of readability and compactness, from now onwards unless stated otherwise, the term *agents* only refers to evolutionary dataflow agents.

Each agent first consumes a given input, concretely, the result of CoreNLP, and processes it in different steps, via more complex and valuable information in appropriate data structures towards the final result – a SPARQL query.

The inner structure of an evolutionary dataflow agent consists of application-specific *node*s, which implement the basic operations. There are different node *types*, and from each type there may be multiple instances. Initially, it is not known, which of them are actually necessary, in which order and in which cases they must be executed and with which settings, to reach the objectives. Thus, the *configuration* of the agents is subject to evolution. In this aspect, the approach has strong similarities with *genetic programming*.

The execution of this set of basic operations and the flow of data are executed in parallel similar to the *dataflow architecture* model.

The nodes are connected for the information flow inside the agent. Dependent on the datatype of the exchanged information, these connections are typed – mirroring the concept of input/output signatures of operations in general. Every node might produce several output results, and every output *connection*s of a node can be connected to multiple other nodes . In contrast to *genetic*

algorithm, communication is done by copies of the information (corresponding to call-by-value, not call-by-reference). As another difference to *genetic programming*, the resulting structure is not a tree, but a directed graph, potentially even containing (feedback) cycles. By this, e.g., diamond structures can be generated, i.e. structures in which a node of the graph branches and these branches reunite in a later node. Furthermore, depending on the setting, the connections may be allowed to form feedback loops (feedback loops in this context are investigated in Section 6.1.6).

An agent is a network of such nodes (for an illustration see Figure 3.2).

Structure of this Chapter The following sections will describe the structure of EvolNLQ in detail in a bottom-up way.

Starting with the encapsulation of information into products in Section 3.2.

Section 3.3 describes how the operations are embedded in nodes and how they form an agent.

Section 3.4 describes the workflow of such agents and Section 3.5 deals with the fitness function and evaluation of agents.

Section 3.6 describes how a group of agents exists together in an environment, which provides the infrastructure for presenting tasks and how agents are selected for reproduction. The process of generating the next generation of agents is then described in Section 3.6.4

Section 3.7 describes how environments are managed by the simulation. This is the encapsulation of the entire learning process and manages multiple environments, which in turn have multiple agents, each consisting of multiple nodes that produce a set of products, as shown in Figure 3.1.

Phylogenetic Relationships of the Evolutionary Dataflow Agents Framework with existing approaches

While the conventional representation of agents can be very different depending on the type of evolutionary algorithm, they are all ultimately interconvertible. As shown in Figure 2.3, the agents of *genetic algorithm*s are a sequence of finitely distinct symbols into which the behavior or result of the agent is encoded. Meanwhile agents in *evolutionary strategies* are vectors that passes parameters to a function, and *genetic programming* generates agents in a tree structure, which can ultimately also be seen as a sequence of predefined functions, similar to the agents of *evolutionary programming*s.

More crucial than the representational form of agents is, of course, the context in which they are used. Especially agents of *genetic algorithm*s depend on how they are interpreted, since without this interpretation of the *genome* the agent is just a sequence of symbols unrelated to the task it should solve. Also agents generated by *evolutionary strategies* depend on which parameter is coded at which position of the vector and its meaning for the function and most important, what this function actually does.

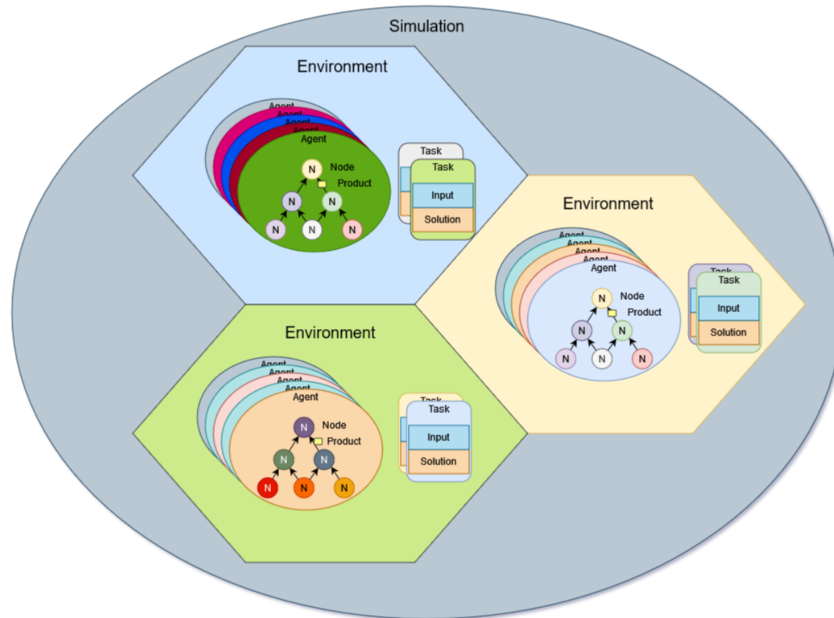


Figure 3.1: Overview of all components of the Framework.

More focused on the solution and closer in their representation to evolutionary dataflow agents are *genetic programming* and *evolutionary programming* which both represent a sequence of commands. This structure is also used in evolutionary dataflow agents. Unlike *genetic programming*, however, the information flow within the agent is not homogeneously numbers (as in Figure 2.3, for example) but can contain arbitrary, predefined data structures, which is why operations like *crossover* cannot be used effectively. Swapping one arbitrary subtree with another would too often result in incompatible data structure types.

Therefore, evolutionary dataflow agents fall into the category of *evolutionary programming*s. However, the whole approach cannot be categorized so cleanly, since it uses selection procedures different from those defined for *evolutionary programming*s, the number of active agents is not fixed, and the number of produced offspring can be more than one per parent.

3.2 Products: the Data that Flows

All information exchanged between nodes is encapsulated in so-called products, to standardize them and make them usable for as many nodes as possible independently from their creator.

The set of products is organized by a hierarchy $\mathcal{PROD}$ of *product classes*, dependent on their data (and thus their semantics). As usual in a class hierarchy, each product has all the properties of its superclasses, meaning it can be interpreted as a higher class as well for all node operations. As

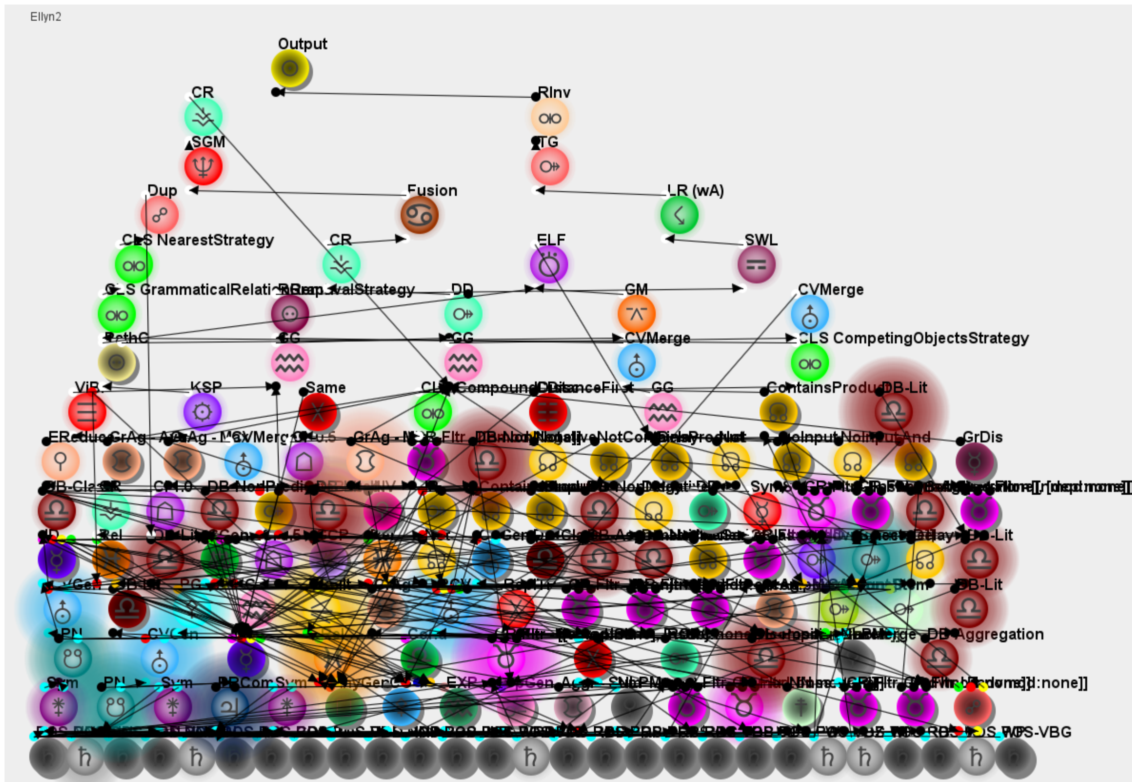


Figure 3.2: Graphical representation of a evolutionary dataflow agent of EvolNLQ in a very highly developed state

Note that input is on the bottom, and the result-producing node is depicted on the top. The black arrows are connections and bigger colored dots represent nodes, while the smaller dots represent *conduit*s (input below the node, output above the node). The color and symbol combination is unique for each class. The brighter nodes illuminate their surrounding the more active it was, respectively if the node casts a shadow it was inactive in the last query.

(The symbols were chosen without paying attention to their original meaning, but only to make it easier to find certain node types. Most of the symbols have their origin in alchemy or the zodiac. As a metaphor, the symbol of the Part Of Speech Nodes (input) is the alchemical one for lead and the Output Node has the symbol for gold (output).)

said, the nodes input and output are typed in terms of the product classes.

Furthermore, products can be compound themselves: there are *atomic product*s, which do not contain other products and are interpreted as entities, and compound products, which contain further products and are interpreted as expressing relationships between those.

While most *product classes* are specific for the use case and are detailed in Section 4.1, there are some high level product classes that are universally usable; these are listed in Table 3.1.

The set P_C of *product classes* is fixed and does not alter during run time, but can be extended

manually if it seems necessary. Instances of such a newly created product class can then be used during the training process from some agents which might be more successful with the additional product class. If this is not the case, the extension was likely neither necessary nor beneficial in the context of the current tasks and the set of other operations.

There are product classes that represent the input data (annotated questions from CoreNLP), ones that are used to represent the created output structures (SPARQL queries), and others that represent intermediate information.

Every product instance p consists of a tuple $(p_c, p_d, p.conf)$ with p_c being the class of the *product*, p_d the data contained by p, and p_r being the *confidence value* of the product.

Product Class	**Description**
Product	The least specific product class, which is the superclass of all other $p_c \in P_c$
Auxiliary product	Those *products* only contain information for other nodes and are ignored by *result nodes*
Atomic product	Contains only data values.
Compound product	Containing other products.
Input Product	Unmodified raw input data
Looking-for-Replacement Products	Those are used to mark another product as an uncertain solution, which can then be confirmed or replaced by another node
Modifier Product	Contains a product and additional information how this product is changed. Whenever such products are in the same query graph product they replace all unmodified instances of the contained product

Table 3.1: Application-independent, abstract product classes

The data p_d of a product is a set of named slots $[name_1 = v_1, \ldots, name_n = value_n]$ of data, where the slot names are defined by the specific product classes. Syntactically, for a product p, the slots can be addressed by $p.name_1$ etc.; analogously, also $p.class$ and $p.conf$ are addressed like a slot. Those can be simple data values or lists of data or even another product or a set of products.

The confidence value p_r of products is described below.

Confidence Value and Reliability

The methods for creating, processing and extending products are not all equally reliable. Therefore, each product has a confidence value, which indicates how certain the procedure in which the product was generated is evaluated by the agent. For atomic products the confidence value is set directly, for compound products it is computed using those of the products involved wrt. the semantics of the compound relationship itself similar to calculating the *fitness score*.

The confidence value $p.conf$ of a product is set when the product is created by a node or altered

by other nodes during processing. *p.conf* is used for decision making or filtering and to judge the overall confidence value in the final result. For a solution P, the overall confidence –which is the confidence of the root product of the solution– is denoted as *P.conf*.

E.g., it is used by some nodes to have another decision criterion in cases where multiple candidates are considered. It is to be noted that a low confidence value is not to be regarded automatically as wrong, but simply as "not being very sure about it". In some cases uncertain measures have to be taken to be able to solve a given task at all. What is considered unsafe or not is decided by mutations of the agent – so, the approach learns to interpret the reliability values.

For the final (qualitative) evaluation of an agent during the training phase, it is also crucial that rewards and penalties correlate with the confidence value. Therefore, the total value of the reward is multiplied with the confidence value.

By this, an agent that receives a task it is not specialized in and clearly misinterprets it, will not be penalized for this, as long as it can make it clear via the confidence value that it is not suited for this task. On the other hand, if an agent answers the query correctly, but has a very low confidence value, the reward will also be very low. Conversely, an agent that has solved a query correctly and is very confident in the answer should also receive a correspondingly high share of the reward. However, if the answer is wrong, it is also important to punish the agent accordingly high, i.e., in case of a "certain" answer that is wrong, it does not only get no reward for this answer, but also gets penalized for being too uncritical with its answer. Thus, the "ability" of the agents to estimate the reliability of its results is also a criterion, and neither permanently high nor low confidence values are advantageous independently of the result. By this, the evaluation refined the original energy reward technique from [66]; see Section 3.6.2.1 for details.

Finally, agents should use this value to make clear whether they are suitable for a query or not. After the training phase, the whole group of agents can be asked and the result with the highest confidence value is presented to the user.

3.3 Evolutionary Dataflow Agents' Anatomy and Genetics - Nodes, Conduits and Connections

3.3.1 Nodes

Nodes represent the basic items of functionality that the agents can perform. Every node class (like implementational classes in any programming language) represents a specific function. Thus, it has a specific input and output signature, specified in terms of the products. Nodes can have an internal storage to keep hold information over some time.

The complexity of the functions implemented by the node classes can vary greatly, from simple filters to extensive graph operations. The respective operations are programmed manually. Using

the node classes, evolutionary dataflow agents implements a *pluggable framework*: From each node class, arbitrarily many instances can be created. The learning process combines them into agents.

3.3.2 Connections and Conduits

The creation of agents is done by the learning process. In principle, nodes can be chained together arbitrarily. Considering the existence of *product classes* and the signatures specified by the node classes, the generation and mutation has to respect certain integrity constraints. Therefore, the idea of connections is refined: each node class n specifies a set of *conduits* n_C. Conduits are the anchor points for the exchange of information between nodes .

The set of *conduits* n_C of a node class n divides into $n_{C_{in}}$ being the input conduits and $n_{C_{out}}$ being the output conduits: $n_C = n_{C_{in}} \cup n_{C_{out}}$ and $n_{C_{in}} \cap n_{C_{out}} = \emptyset$. The notation is generalized in that for a node n, n_C etc. denotes the conduits of its node class.

Node classes with $n_{C_{in}} = \emptyset$ are considered *input node* classes, they access the input data (in case of EvolNLQ the data from CoreNLP). Node classes with $n_{C_{out}} = \emptyset$ is a *result node* classes. There is only a single node class of this configuration, which simply delivers the final results. All other node classes have both input conduits and output conduits.

Each conduit c has a set of accepted classes $class(c) \subseteq \mathcal{PROD}$ representing the set of product classes which can be received or sent respectively. To deliver or receive any products, an output conduit of a node must be connected to one or more input conduit(s) of one (or more) other nodes . For every such connections between an output conduit c_n and an input conduit c_m, $n \neq m$ and $c_n \in n_{C_{out}}$ and $c_m \in m_{C_{in}}$ and $class(c_n) \subseteq class(c_m)$ must hold.
The set $n_{C_{out}} = \{n_{Cout1}, \ldots, n_{Coutj_n}\}$ of output conduits is an ordered set and according to this order, the products are distributed. Each n_{Couti} handles all products p generated by n for which $class(p) \subseteq class(n_{Couti})$ holds and that have not been accepted by a previously enumerated node $n_{Coutj} \mid j < i$. (i.e., it works like a coin sorter: the coin falls (only) into the first hole where it fits.). Additionally, a node can allocate a product directly to some conduit for output.

There should be no products left at the end, otherwise the node is incorrectly designed and will produce products that it cannot distribute and should be reviewed. When an output conduit is connected to more than one input conduit, every such input conduit gets an individual copy of each of the products sent (here, the coin sorter metaphor is not met: the coins would be duplicated).

3.3.3 General Node Types

While the concrete operation and set of conduits depends on the application-specific node type, there are additional general, structural node classes that are handled differently within the agent. These structural classes are not mutually exclusive and a single node type can have multiple of them. Nodes belonging to such a class are activated in certain situations.

3.3.3.1 Input Nodes

These nodes have the property to be activated once at the beginning of the computation of the agent and to read data from one or more external sources (in the EvolNLQ case this is CoreNLP). An agent needs to have at least one input node because otherwise it would not have any reaction to a request. The only input node class in EvolNLQ is the *Part Of Speech (POS) node* class which receives the annotations from CoreNLP (more details see Section 4.2.1.1).

3.3.3.2 Collector Nodes

An issue for any dataflow-based process arises when it comes to check if something does not exist or whether further (e.g., more confident) items might come in later and need to be waited for or not. The nodes cannot tell that something does not exist or might be generated in the future. Therefore, collector nodes serve for synchronisation. Those nodes collect all received products until they are activated (which will be discussed in Section 3.4.2), then empty their product storage and send the products respectively to their conduits to other nodes .

The activation takes place in a learned order if there is no further activity in the agent. For this purpose every such node gets an ordering value assigned which is just a random number, but the ordering value can be changed through mutations and therefore adjust.

When no activity is recognised in the agent, but it has a *collector* node instance, the agent then checks all *collector* nodes starting with the one with the lowest number in ascending order whether it is has any products in its storage to forward. If any stored products are found, the computation continuous normally until there is no more activity again. Then the process is repeated, starting by the node with the lowest ordering number until there are no further stored products in the agent or the allowed time for computation is exceeded.

3.3.3.3 Output Nodes

The output node is a special form of a collector node, it is always the last in the order of the collector nodes and with its activation, the calculation of the agent is finished. Instead of passing on its products, all products stored in it are interpreted as the final result.

3.3.3.4 Parameterized Nodes

Many nodes have one, or more parameters. Those parameter are set to a random value at the creation of a node or changed through mutations. If a parameterized is randomly chosen to mutate, it not only has the possibility to change it connections to other nodes but additionally the chance to change its parameter. Parameters are predefined in the context of a specific domain, such that they can be meaningfully changed by a mutation. Parameters are either number-based, key-value-based, or mode-based .

Numeric parameters

Nodes with numeric parameters use a specific value for their operation as a parameter, for example a threshold limit or the exact confidence value assigned to a product can be such parameters. During the creation of the node, a random value is assigned to the parameter within its defined interval. By mutation this value can be changed, these changes are significantly smaller relative to their interval, so that an optimal value can be approached.

Key-value parameters

These nodes use one or multiple values of a list of possible key values to decide whether or not to execute a response. The values are strongly relate to the specific domain and they process only inputs with corresponding values.

In the case of EvolNLQ for example the *Part Of Speech tag* nodes are specific for one type of tag, chosen from a list of all possible tags. Although these nodes have access to all CoreNLP data, they only process data with fitting values.

Mode-based parameters

Mode-based parameters have a list of predefined modes.. The mode of a node can change its function and the performed operation entirely and the node is then more a logical set of functions than the implementation of a single operation. A typical example could be the logical node, which has many different modes like "*and*", "*or*", "*not*", "*contains*" and so on (see Table 4.5 for more details), while all theses modes are logical operations, they of course have vastly different results.

3.3.4 Mutations

The source for *evolution* to form new agents is mutation: With a small probability, which is either set in the global settings or is random for each agent. Mutations of the anatomy of an agent can occur during reproduction. Only agents resulting from mutation that fulfill the following criteria are viable:

1. An agent needs always one output node, otherwise it can not generate any results and cannot earn any score points.
2. At least one input node is necessary since otherwise the agent would not be able to start any computation, thus not produce any results.
3. The agent must be a single connected graph since otherwise it would mean, that certain parts of the agent are not able to deliver any meaningful contribution to the result.
4. Only connections between conduits with matching signatures as specified in Section 3.3.2 are allowed.

Those criteria are either met by design or are checked after the mutation.

Although there are different forms of mutation, they all have in common that they change the configuration of an agent. Mutations can either be a single operation like removing a node, or be a sequence of other mutations like "inserting a node before another one", which consist actually of "add a node", "remove connections" and "add connections" mutations executed on specific nodes and connections. The possible mutation options listed below can accrue, each with an own likelihood. Those are based on the standard mutation used in *evolutionary algorithm*s described by [13].

Node Addition Adds a single node to the configuration and connects at least one input (as far as one exists) and one output with another node.

Node removal Removes a node and its associated incoming and outgoing connections (via *connection removal*) and checks for disconnected sub-graphs and removes those (via *node removal*).

Connection addition Adds a valid connection between two conduits of different nodes

Connection removal Removes a connection and checks for disconnected sub-graphs and removes those.

Parameter Change Changes a parameter of a parameter node to another value.

Insert node insert a node n_x and appropriate conduits satisfying the signature conditions stated in Section 3.3.2 between two nodes n_i and n_j which have a direct connection, which is then removed in course of the mutation.

Generate production line Starting with a new input node n_0, this mutation generates a new nodes chain $n_1, n_2, \ldots$ with appropriate conduits. For a given $p < 1$, with probability $p * i$, node n_i is not a newly generated node but an already existing node of the agent.

Increase/Decrease Mutation Likelihoods The likelihood for a mutation at all and for each individual mutation type can also mutate.

Fusion Adds the configuration, i.e. the nodes and connections, of another agent to an agent. It merges input nodes that are of the same type and have the same parameters. These nodes are then classified as "equal" and as a pair. Then, all nodes that are connected to nodes that are classified as "equal" are checked to see if there is also such a node with the same type and the same parameters at the other agent at the partner node that is classified as "equal". If it is connected to nodes that are partners of the connected nodes of the first agent in the input and vice versa, it is added to the

list of "same". This is repeated until the list has not expanded in one iteration. All non-equal nodes are then added to the first agent and connected to the last "equal" partner nodes .

Swarm Fusion To answer a query, the set of agents is queried as single individuals and the result with the highest confidence value is used as the final response (see Section 3.6). This simple filtering can also be done within a single agent. This gives the possibility to merge all agents into a single one by copying the whole content of each agent into a single agent and, instead of all result nodes, adding a single node for all agents with filtering by confidence value which then continues into the new result node. The swarm thus merged into a single agent can now evolve further or can be used as the final result.

3.4 Evolutionary Dataflow Agents at Work

Every evolutionary dataflow agent is a network of nodes where data flows and which can be seen as a distributed process. Thus, similar to the *dataflow computer architecture*, synchronisation matters. The dataflow contains the control flow, and is organized in *ticks*.

Basically, an agent (or, in general, all agents of an active population) are started at the same time, once reading (with their input nodes) the available input to be processed.

From then on, with every tick, each node is allowed to perform its operations once in parallel. Between the ticks, the dataflow via the conduits takes place.

3.4.1 Single-Tick Node Behavior

Behavior of Common Nodes

The common nodes – i.e., not input nodes and collector nodes – perform the following steps with each tick (depicted in Figure 3.3).

1. Reading the received products of each input conduit.
2. Emptying the input conduits.
3. Saving the products, if the functionality of the node requires it.
4. Process the received input, including stored products (the operation is executed for each new input with all previous received inputs of other conduits as well as inputs received in same tick of other conduits).
5. Distribute the generated products to the output conduits according to their type.

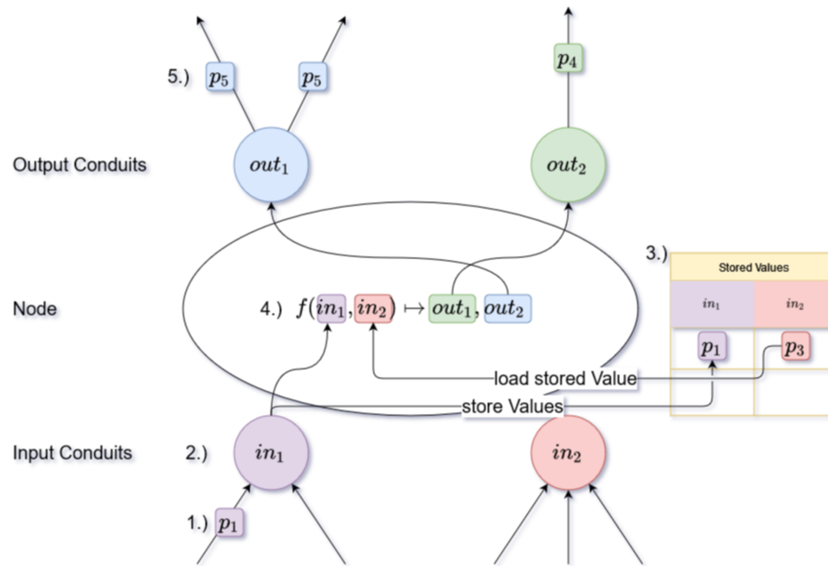


Figure 3.3: Workflow of a node. (Note again, that input is from the bottom, output towards the top.)

Behavior of Input Nodes

The behavior of input nodes differs only slightly from that of common nodes: instead of steps (1) and (2), with the first tick, they read the input of the given task. With subsequent ticks, they do nothing.

Behavior of Collector Nodes

The behavior of collector nodes *in every tick* consists only of steps (1) – (3). So far, they just collect their inputs and serve for operational stratification; see below.

Behavior of Output Nodes

Output nodes are collector nodes – they also just collect their inputs, but never distribute them to other nodes, but their stored products are considered the solution to the given task after the agent terminates.

3.4.2 Agent Infrastructure: Controlling the Ticks and Dataflow

When all nodes are terminated, the tick –performed by the individual nodes– ends. Then –performed by the agent infrastructure– all products in the output conduits are (duplicated, if needed) and forwarded to the input conduits according to the connections. Afterwards the output conduits are emptied.

If in the current tick, at least one of the nodes has generated a product, the next tick starts.

Stratification: Activation of Collector Nodes

When in the last tick, no node has generated a product, i.e., a kind of a fixpoint in the computation is reached the collector nodes come into play: they still contain stored products that can now be processed. By this, collector nodes act e.g. like the operational *stratification* in *Datalog with negation* [80] which is not only needed for negation ("wait until/whether it is clear that nothing happened"), but also for aggregation operations (count, sum, average).

In each such situation, *one* of the collector nodes is fired. Intuitively –according to the idea of stratification–, the "first" collector node in the graph should be activated first. In evolutionary dataflow agents, the order of activation is subject to learning, it develops and can be changed by mutations. So, according to the learned order, the collector nodes are checked whether they (still/again) have stored products, and the first one of those is activated. With this, the next tick starts. Obviously, in this new first tick, only that node does something, producing results, feeding its output conduit(s), and the agent returns to its standard tick-based behavior, until the next "fixpoint" is reached. If no more collector nodes contain unprocessed products, the agent's computation terminates successfully. In this state the output node contains the final answer.

An invalid termination is forced from the environment, if an agent's computation exceeds the maximal allowed computation time. During the learning phase these agents also get an enormous penalty on their score to avoid the waste of computational time as suggested by [13].

3.5 Evaluating the Work of Evolutionary Dataflow Agents

The purpose of evolutionary dataflow agents during the training phase is to solve some specific type of tasks. Each agent is afterwards evaluated how well it performed with this.

A task consists of an input I and a goal S to be achieved. In the general case, I and S can contain any kind of information. In EvolNLQ, I is a natural language question (NLQ), and S is a SPARQL query (for more information see Section 4.1).

Determining whether a task is solved or not, and if not, to what extent it has been answered correctly is not trivial, especially in the case of database queries. The number of possible correct translations into a query language for a query is infinite, just with the possibility to name variables arbitrarily, or to insert conditions that do not change the result. But even relationships relevant to the content can often be expressed in different ways. For most benchmarks, therefore, not the actual query is used, but the resulting result set.

This makes sense, if the condition for the correctness of a query is the correct answer. For most cases, this is also a sufficient condition. But for some cases, however, even an incorrect query can lead to the correct result. For example, for queries that do not filter their result set strictly enough

to really match the query, but for which there is no case in the data set for which the missing restriction would make an impact.

Nevertheless for the most benchmarks this condition is still sufficient and the only practical solution such that the expected result set, or more often the expected result set size, is compared with the generated output, or its size and a binary, correct or incorrect evaluation can be assigned.

But as soon as the solution set contains elements that should not be in it, or elements that should be in it are not present, it can no longer be said how close this solution is to the actual one.

Example 5 *For the EvolNLQ case, there are queries with vastly different result sets evaluations but nearly the same error in their SPARQL query result candidates:*

A.) *Give me the population of all cities except Berlin.*

```
SELECT ?population
WHERE {
    ?City a :City.
    ?City :name ?name.
    ?City population ?population.
    # MISSING:
    # FILTER( ?name != "Berlin")
}
```

B.) *Gives me the population of the city Berlin.*

```
SELECT ?population
WHERE {
    ?City a :City.
    ?City :name ?name.
    ?City population ?population.
    # MISSING:
    #FILTER( ?name == "Berlin")
}
```

Considering the queries from Example 5, the SPARQL structures of these two queries are almost identical, only to be distinguished by the equal sign and inequality sign in the filters. Assuming the question is correct, except that this filter was not created and there are 10,000 cities. A) would then have 9,999 correct results and one wrong one, so 99.99% of the correct set while B) would have 9,999 wrong results and one correct one. In both cases the answers were partly correct but there is also a crucial part missing. From the similarity of the solution sets, however, A) would be almost correct, and B) contains so many wrong results that a random set might be better.

For a benchmark this is not yet a problem, since it only cares for correct or wrong answers, but for *evolutionary algorithm*s it is fundamental to be able to give an as exact as possible indication of how close the solution is to the actual one, in order to be able to reward possible improvements accordingly. Therefore, this approach is not suitable for task evaluations.

Because of this, the evaluation of a generated query is based on its components, and it is done even before the conversion to a SPARQL query string. The goal S is defined on the internal product level, containing the expected products for the optimal solution. Consider e.g. that a "minus" in the relational algebra can be translated into a NOT EXISTS subquery in SQL; or in a MINUS. So, the evaluation before translating into the target language is more appropriate. This has further

the advantage that each product can already define for itself how it determines its similarity to another product and the average of these similarities can serve as an evaluation standard. The disadvantage of this method is that it cannot cover the set of all correct solutions, since there are infinitely many correct solutions.

From the practical point of view, annotating the questions is a relatively laborious and tedious process. Considering the size of the query sets, only one reference solution per query was used at a time. Usually, wrt. some kind of normal form, the definition of a reference solution on this level provides a good coverage. To compensate at least partially for this disadvantage, products are usually defined to include variations in their similarity assessment. The reference solutions are provided in form of an XML serialization of the products that can be read by EvolNLQ to recreate the reference instances.

Each product class c comes with a function $fcomp_c$ how it can compare to instances of its class, and different product classes are always evaluated as completely different.

For every task t given to the agents, the solution $P_{a,t}$ generated by an agent a, the actually generated products in $P_{a,t}$ and the products in the reference solution S_t are compared according to $fcomp_c$, filling a *similarity matrix* $SM := M^{(P_{a,t},S_t)}$.

For computing the similarity values between $P_{a,t}$ and S_t, every product p_a from $P_{a,t}$ must be assigned to *only one* p_S in S_t (mirroring compositeness etc.). While it looks like a knapsack problem at first glance, a simple greedy algorithm can be used here, as described below. In the end, it is only crucial that exact matchings are assigned to each other. In case that non-optimal distributions arise with partially incorrect solutions, no real damage is done, since they are still regarded as incorrect and the numeric value of incorrectness is arbitrary.

For this, the contents of S_t and $P_{a,t}$ are greedily paired as long as their similarity values are not below a specified threshold. For the highest value in SM, it is kept (breaking ties arbitrarily), and all other values in the respective row and column of SM is set to 0 to prevent any further matching of already matched product. This process is then continued until all rows are handled, obtaining SM'. Then, each row and each column contains at most one non-zero value (none, if the product was not matched at all).

With this, the evaluation for $P_{a,t}$ wrt. each product $s_t \in S_t$ can then be calculated by checking for every product in s_t how near the "best" product in a's solution came to it:

$$eval(s_t, P_{a,t}, S_t) := max(s_t, SM')$$

with $max(s, SM')$ defined as returning the highest similarity value from SM' for the column associated with s.

For evaluating the performance of an agent wrt. the whole solution $P_{a,t}$, the difficulty of learning

the usage of compound products compared with simply learning about atomic products must be considered. For this, every product s_t of the reference solution S_t is assigned a corresponding constant value v_{s_t}. In this approach, compound products simply get 10 times the weight of atomic products: Let at be the number of atomic products in S_t and co the number of compound products in it. Then, the share of every s_t is computed as

$$v_{s_t} := \begin{cases} 10 \cdot 1/(10 \cdot co + at) & \text{if } s_t \text{ is a compound product} \\ 1/(10 \cdot co + at) & \text{if } s_t \text{ is a atomic product} \end{cases}$$

obviously with $\sum_{s_t \in S_t} v_{s_t} = 1$. Note that the evaluation for compound products does not yet sum up the evaluations for their compounds, so the evaluation for the whole $P_{a,t}$ wrt. S_t is simply the weighted sum, normalized to [0,1]:

$$eval(P_{a,t}, S_t) := \left(\sum_{s_t \in S_t} eval(s_t, P_{a,t}, S_t) \cdot v_{s_t} \right) \tag{3.1}$$

resulting in a normalized value between 0 and 1.

These evaluations will be used next for computing the *energy-based reward* which in turn will be used together with *penalties* for defining the fitness function.

Apart from the rewards, excess or too dissimilar (i.e., with a maximum similarity below a fixed threshold) products of $P_{a,t}$ are not included in the calculation of the rewards, but are counted (named as $err_{a,t}$, which equals the number of rows of the *similarity matrix* that contain only 0s), and stored with the corresponding agent to be included as part of the penalty calculation for the *fitness function*.

3.6 Environments

The first abstraction step to have (i) a *population* of agents working together, and (ii) to have subsequent generations are the so-called environments, following the terminology from [65] and [66]. While in an Artificial Life Simulation, these environments have clear similarities with the habitat of natural creatures, the environment in this case is a bit more abstract, but nevertheless it can still be well imagined as the habitat of agents, since it defines their conditions in the same way.

The learning process advances by the fact that the subsequent generations of agents contain other, new agents. The composition of a generation is decided by the respective selection criterion, which means for the species of an agent that its continued existence, or even the further development of its line, depends not only on its own results, but also on that of the other agents of the species. For example, together with a population of agents that each gets 10% of the maximal score, an agent that achieves 15% is very dominant, and will certainly be able to evolve and its offspring will be

part of the next generation. In a population of agents that get 50%, its extinction is almost certain. What may seem a bit tragic for the single agent is of course indispensable for an algorithm that converges towards an optimal solution, after all it is based on the fact that between initialization and the end of the learning process, there are several hundreds or thousands of intermediate generations.

An environment e describes the sequence of generations that evolves around a fixed set T of tasks, which is a subset of set of all tasks which is the set of *all* training tasks and S_T the corresponding set of solutions for T. The evolution is controlled by an application-specific fitness function f for evaluating the agents and a *selection method* s that specifies the agents that will reproduce. Examples of selection methods are given in Section 2.2.2.2 on page 28 and one developed in course of this approach is introduced later in Section 3.6.2. Thus, the environment is basically formalized as

$$e := \{T, S_T, f, s, ((A_i))\}$$

where $((A_i))$ is a sequence of sets of agents (i.e., the generations A_i). Actually, it is $(\{T_e, S_{Te}, f, s, ((A_i))_e\}$ when the top abstraction of a *simulation* that contains several environments that all share the same global fitness function and selection method. In the EvolNLQ case, T is a set of natural language queries which should be converted into the SPARQL queries with reference solutions S_t for each $t \in T$.

3.6.1 Training Run

The training takes place in runs, each run concerning one generation. Each run follows the same pattern and restores the starting conditions at the end. At the beginning of each run i, there is a set of agents A_i, the current generation, and the set of tasks T, the training set. Every agent in A_i is presented with all the tasks of T as input to be solved in a certain amount of computation time. Then the results are compared with the solutions in S_T and an evaluation is made by the f of the environment. After that, the selection method s of the environment is used to select a subset of A_i based on their results (depending on the s, luck might also be involved). In this approach, the selection is based on *energy* that is rewarded for solving tasks. Those agents then reproduce, using their energy, and their offspring form the next generation A_{i+1} for the following run $i + 1$. In the reproduction process, mutations can happen as described in Section 3.3.4. This is then repeated with the same training set, but with A_{i+1} which is likely slightly altered to A_i. A graphical representation of the process can be found in Figure 3.4.

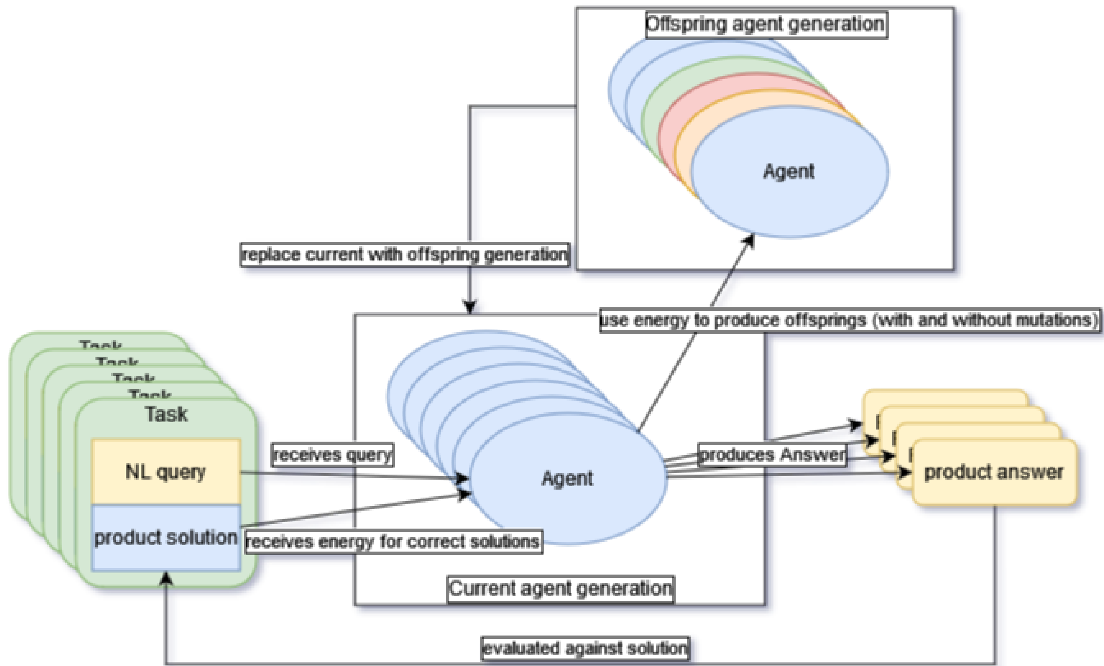


Figure 3.4: Cycle of a single training run of an environment.

1. Natural Language query from each task is given to each agent in parallel.
2. Each agent generates answer products.
3. When all agents are done or produced a timeout, the answers are evaluated against the reference solution of the corresponding task.
4. Depending on the degree of success for each task, each agent is rewarded with a certain amount of energy.
5. Agents that received sufficient energy use their energy to produce offspring which might be mutated.
6. The generation of offspring replaces the original agents generation.

3.6.2 Energy-Based Rewarding

While a single agent is static and produces a deterministic result, the actual progress in solving given tasks is made by selecting suitable offspring. The selection itself can be handled in different ways as described in Section 2.2.2.2 and [13]. Another method, *energy-based selection* has been developed in this context of this work and is described below and compared with traditional methods in Section 6.1.2.

While the selection is the main driver of the evolution of the agents, other factors besides selection play a role as well for the *convergence* of the process, as described in [13]. These are besides randomness, the initialization, but especially the choice of the fitness function and the provided set of products and nodes.

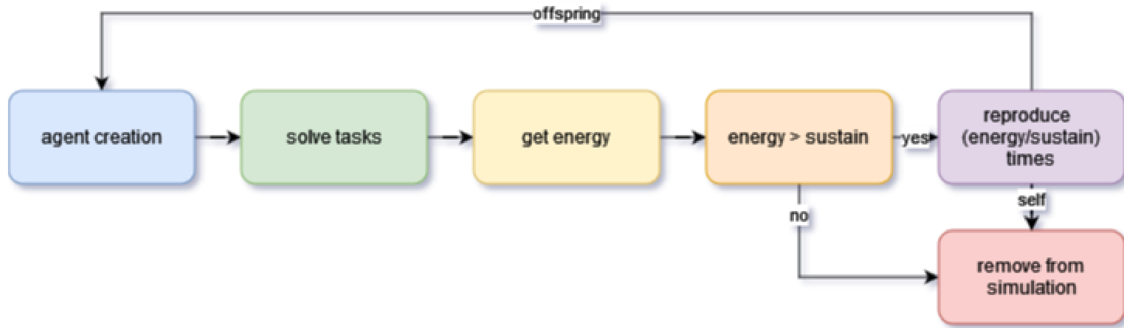


Figure 3.5: Life Cycle from the agents point of view

3.6.2.1 Energy-based Selection

The original idea of Hoverd [66] is to give an environment a passive energy level that agents can use. In turn each agent needs an agent-specific amount of energy to survive, the *sustain energy*.

Transferring these two major changes from Hoverd into this approach results in assigning an *energy value* ev_t to each task that can be earned by an agent for solving the task (formally, the $((ev_{t_i}))$ is a sequence of values that also change for every run, depending on the worth/difficulty to solve the task), and replacing the limit of the number of agents with the agent-specific *sustain energy* value. While the sustain energy value can be used as developed by Hoverd, the notion of passive energy provided by the environment must be transformed to this approach. This is due to the goal here being not survival or the consumption of other agents, but solving a task. Therefore instead of a finite number of eatable agents, each task must provide a limited amount of energy that the agents need for survival (actually, for reaching the reproduction phase at the end of the run, see Figure 3.5).

Sustain Energy To compute sustain energy, two approaches have been developed. The difference is only in the treatment of input nodes and their connections:

- "non-zero-cost adjustable input" scheme: the *sustain energy* value of a agent a is calculated by summing up the cost for all nodes and all connections:

$$sustainEnergy(a) = |a.nodes| * nc + |a.connections| * cc \tag{3.2}$$

  with nc being the cost for a node and cc the cost for a connection.

- "zero-cost fixed-all input" scheme: the input nodes and the connections from the input nodes to the other nodes are not taken into account. With this scheme, the initial agents already have every possible combination of input node and parameter, so they already get the maximum amount of input information. Also, input nodes will not be removed anymore, instead of being deleted by mutation, all connections coming from them will be removed. On the other

hand, each input node type and parameter combination can exist only once and no more input nodes can be added to prevent *bloating* [13].

$$\begin{aligned} \textit{sustainEnergy}(a) = {} & (|a.\textit{nodes}| - |\{i \in a.\textit{nodes} : i \text{ is an InputNode}\}|) * nc + \\ & (|a.\textit{connections}| - |\{c \in a.\textit{connections} : c.from \text{ is an InputNode}\}|) * cc \end{aligned}$$

analogously for a species sp of agents, $\textit{sustainEnergy}(sp)$ is defined. (3.3)

So the idea here is to always just keep all input information ready for all nodes and not punish an agent via the *sustain energy* costs for checking all available information.

Compared to the rewards for correct answers, the values of nc and cc must be relatively low, (experiments have shown that it is not so important how big the difference is after a certain point. This value is approximately 1/1000 for nc and 1/10,000 for cc), otherwise a very high success barrier is created for new agents. Since these usually have unnecessary nodes and connections and first need a certain basic infrastructure to produce anything evaluable. However, the value cannot be omitted completely, otherwise agents will suffer under so called bloating. Which means that the structure of the agent grows without contributing anything to the result, neither useful nor wrong. Thus, without this cost, the agent would not be considered worse than a agent with less components but the same result. But the bloating agent would require additional computation time. Even if this application is not designed for speed, the growth of a bloating agent would happen so fast and excessive that the learning process would practically come to a standstill or the program would crash from a heap space overflow.

This low cost also has the effect that agents who find additional solutions are scored significantly better, even though they have probably created many unnecessary structures through their mutation to achieve this goal. Otherwise if the costs are high, a solution which needs a complex structure to be achieved might just not provide enough reward to be worth it.

During phases in which no new successes are achieved, the only slightly lower chance of being included in the next run nevertheless leads to more effective agents, i.e. with fewer unnecessary structures, surviving in the long run and replacing the wasteful ones.

Thus, the costs here have an effect on the bloating behavior of agents that is actually desirable, at least to a small extent, since a more complex structure may be needed for a better solution that requires more than one mutation step.

Energy Provided for Solving Tasks In every run, every task t is assigned an amount of *energy* $en(t)$ that can be distributed amongst agents that solve it more or less perfectly (as it is dependent on the run i, a more global notation would be $en(t, i)$, but for here, $en(t)$ suffices).

Roughly, $en(t)$ is available for distribution amongst the agents that solved that task, considering (i)

how good they solved it, and (ii) a considering requirements that are important for survival of the population:

If $en(t)$ would be equally distributed over all agents that solved t, sudden extinction of a species sp_a may occur once it reaches a number n of agents such that the received energy for each agent of sp_a is lower than the energy necessary to produce a single offspring:

$$\mathit{feed}(a) := \frac{en(t)}{n} < \mathit{sustainEnergy}(a) \ .$$

Which means, that not a single exemplar of sp_a would be in the next generation and sp_a would be extinct.

Thus, there is a fixed value $ep \in \mathbb{N}$ that limits the number *energy portions* that are distributed for a task to the agents that found the ep best solutions.

(Details for tie breaking: Inside agents of the same species, there is a fixed ranking, such that the same individuals of a species always receive energy first. If there are more than ep candidates from several species, the species that is more successful overall, is preferred.)

Thus, enough agents a that solved the task *perfectly* are guaranteed to get $feed(a) := en(t)/ep$ energy.[1]

As described above, the the *evaluation* function given in Equation 3.1 determines how well each agent solved each task, resulting in a value in the interval [0,1] with 1 if the solution of the agent exactly matches the expected solution and 0 for nothing or a completely wrong result. For every task, a list of agents is created using this score. Agents with the same score (which is mostly the case for agents of the same species, or when agents from different species solved one or more tasks perfectly) get a random position according to their score. Now for each task, the top ep agents can claim energy for their solutions, but their claims will be adjusted by confidence of the agent wrt. the solution, and may be reduced later due to penalties. All other agents get nothing:

$$\mathit{claim}(a, P_{a,t}, t) = \begin{cases} \text{if } eval(P_{a,t}, S_t) \in \text{Top}_{ep}(\{eval(P_{a,t}, S_t) | a \in A_i\}) \text{ then } eval(P_{a,t}, S_t) \cdot en(t)/ep \\ \text{0 otherwise} \end{cases} \tag{3.4}$$

By this, the claimable energy (and further, the actually rewarded energy) is computed on a very detailed granularity to support already minimal learning steps. For agent learning, *micromanagement* proves to be useful. Also, punishment is useful, as described next, while *motivation* does not work with these agents as the analysis of the behavior of the framework will show in Section 6.1.7.

[1]This corresponds to nature: if there is not enough food available for a pack of hyenas, the higher-ranked eat, and expel the lower ranked ones completely. Same for two packs: the overall stronger wins. Or "he that has plenty of goods shall have more" –german– "wer hat dem wird gegeben".

Penalties So far, for a solution of a task an agent can either get energy if it is correct or at least partially correct, or it gets no energy if there is no solution, or it is not good enough (or the agent has no luck in the ep-based preselection). Furthermore, there is a way even to "punish" an agent by withdrawing energy if the solution is wrong.

Such punishments, however, can become a problem relatively quickly if such penalties are too high. This can lead to the situation that an actually not bad agent that finds many correct things, is punished so heavily that it does not receive enough energy to survive and the actually good approach is lost. Too low penalties, however, lead to the situation that, for example, a required more complex structure for filtering wrong results would generate higher energy costs than the penalty for not filtering. Especially for many tasks, individual errors are then hardly relevant and the evolutionary pressure is correspondingly low. Therefore, to be able to weigh errors it was decided to include the penalties into the fitness function (defined later) not additive but as a multiplicative factor.

For a task t, consider an agent a returned a solution $P_{a,t}$ with confidence $P_{a,t}.conf$, where the number of errors as $err_{t,a}$ was computed from the *similarity matrix* above. Additionally, there is a global value $pe \in [0, 1]$ for weighting penalties (for example 0.98 or 0.95, the exact value it set by the simulation settings; the lower pe the worse is the punishment). Then,

$$penalty(a, P_{a,t}, t) := pe^{(err_{a,t} * P_{a,t}.conf)} \tag{3.5}$$

To prevent too harsh punishment for solutions with a low confidence value, the value of e is multiplied with the confidence value to reduce the punishment accordingly.

3.6.3 Fitness Function

Using the previously defined functions of Equations 3.4 and 3.5, the total score for an agent a that computed solutions $P_{a,t}$ (with a confidence $P_{a,t}.conf$) for the tasks $t \in T$ that have reference solutions S_t, the fitness function, denoted by $energy(a)$ is defined as follows:

$$energy(a) := \left(\sum_{t \in T} claim(a, P_{a,t}, S_t) * P_{a,t}.conf \right) * \left(\prod_{t \in T} penalty(a, P_{a,t}, t) \right) \tag{3.6}$$

Thus, with each error, the total score is further reduced, but the amount of reduction per error decreases, so that especially errors which are repeated over and over again are less important. The limit of this function due to errors is still 0, which is fine, if an agent actually produces arbitrary many errors.

Agents that receive less energy than they need as *sustain energy*, do not survive. In this approach, this is simply encoded into the generation of the next generation of agents, where only agents

Variable	Meaning
sim_o	maximal amount of offspring an agent can have
sim_{mp}	probability that a mutation will occur during reproduction
sim_{msv}	variance of the mutation severity
sim_{msm}	Mean of the mutation severity
sim_{pe}	severity of penalties
sim_{tv}	energy value of a single task

Table 3.2: Global Settings of the simulation

reproduce that received enough energy.

3.6.4 Generation of the Next Generation

As usual in *evolutionary algorithm*, agents themselves are not part of the next generation (like in nature), but the whole next generation is created by reproduction, i.e., identical or mutated offspring. The decision which agents will reproduce is determined by the selection method s. Instead of selecting the agents with the highest energy for reproduction, as is usual in a classical fitness score selection procedure, in evolutionary dataflow agents, each agent a such that

$$energy(a) \geq \textit{sustainEnergy}(a)$$

produces offspring for the next generation for each multiple that its collected energy exceeds its *sustain energy* value.

For this purpose, it is determined for each individual agent whether and how many offspring it will produce. There ist a globally defined value sim_o that constrains the maximal number of offspring an agent can have, independent from how much energy it has. This prevents a single species from becoming too numerous and taking up too much computing time since it can only reach a size of $ep * sim_o$ (all other agents of this species would not get any energy due to ep) . For each new agent to be created as offspring of some agent a of generation A_i, it is determined with a probability of sim_{mp} set by the simulation, whether it will mutate. If the offspring is not mutated, a new individual of the species of a is added to the generation A_{i+1}. If the offspring agent is mutated, the severity of the mutation is first determined. The severity of a mutation is a stochastic variable m of how many mutations are performed on the agent. m is Gaussian distributed with a preset variance $\sigma = sim_{msv}$ and a mean value $\mu = sim_{msm}$ resulting in the distribution:

$$m \sim |\mathcal{N}(sim_{msm},\ sim_{msv}^2)|$$

Then, a sequence of m mutations according to the Section 3.3.4 is then executed on a new individual of a's species one after the other, and the resulting agent is added to generation A_{i+1}. The agent then has a different configuration than its ancestors and thus defines a new species (w.l.o.g., no

such mutations results in a species that already existed).

Lifespan of an Agent in the Implementation Quite similar to Darwin's theory, the individual agent has completed its purpose during the training phase by producing one or more offspring for the next generation during its lifetime, which is in its case a single run. For the next run, the agent is no longer part of the simulation, since it is itself static and cannot contribute new results and therefore, does not make any progress in the search for an optimal solution.

However, to save computation time, if the agent has at least one non-mutated offspring, all its results of the tasks are stored. The agents produce deterministic results and therefore the results do not have to be recalculated in each generation for each agent or even for each agent species, but only once, when a new species is created by a mutation.

In case a whole species does not produce any offspring (which happens especially frequently for new mutations) the data is not preserved, since the likelihood to have the exact same configuration again is extremely low and the effort to check whether a mutation results in a species that already existed in a previous run is far greater than just to recalculate the result.

3.6.5 How to Motivate Agents to learn

The energy $en(t)$ can be used to motivate the agents to learn to solve up to now unsolved tasks, or not to focus on already learned tasks. For this, $en(t)$ can be modified with every run.

At the initialization, every $t \in T$ has the same energy value, let's shortly call it e_0. In case there are more than eu agents that solve a given task t perfectly, then

$$en(t) \leftarrow en(t) * (1 - \epsilon) \text{ for some } \epsilon < 1 \tag{3.7}$$

i.e, the energy assigned to this task in the next run is reduced up to a minimum value of $e_0/4$ since this can be considered a rather easier task and increases the pressure to develop into another direction.

In case that for some task t there is no agent which solves t, the value of the task increases to

$$en(t) \leftarrow en(t) * (1 + \epsilon) \text{ for some } \epsilon < 1 \tag{3.8}$$

up to a maximum of $e_0 * 4$.

Over time, the energy offered by a task increases as long as the agents were not able to solve the task, and it decreases once an agent is able to do so.

The idea behind this is that an agent found a solution for an unsolved task, the respective agent gets a huge energy boost to replicate more and to spread its species this way. As soon as the species

has stabilized, the energy provided by this task decreases and reduces the species to a reasonable number. These energy boosts are important to reduce the likelihood that a single agent has a configuration that can solve a previously unsolved problem, but then only gets offspring that are mutated in a detrimental way, and thus that configuration is immediately lost again. The effect of this is investigated later in Section 6.1.3).

3.7 Simulations

The simulation is the highest level in the structure of the framework and contains (indirectly) all other elements. Similar to how the environment controls and synchronizes the processes of the agents, the environments are in turn coordinated by the simulation.

The simulation always (except in a very late phase) runs several environments in parallel. A commonly used number is to have 5 – 10 initial environments. For all environments usually the set of tasks they actually provide to their agents is only a subset of T_{full}, which is much smaller in size, such that agents are trained on rather few random tasks.

The idea behind this is that highly specialized agents are created first, such that for each task of the training set at least one agent configuration that has good abilities to solve it is found. Eventually, however during a second learning phase, the tasks sets should become bigger and the set of agents should become more diverse since a configuration that covers a large number of problems, thus is more generalized, is more likely to be able to solve a previously unknown problem, especially if the problem is only a combination or variation of known problems. In a third learning phase, the number of agent species should reduce to develop agents that can solve several tasks (recall from Section 3.3.4 that two agents can combine into one by fusion mutation, and a set of agents can mutate as a swarm fusion). Therefore, the different $e \in E$ are not hermetically separated from each other, but there are operations that can operate on them.

A simulation *sim* consists of a set of environments E, the complete set T_{full} of learning tasks, the *global relocation likelihood* p_{reloc}, and the *global fusion likelihood* p_{fusion}:

$$sim := \{E, T_{full}, p_{reloc}, p_{fusion}\}$$

Initially, the set T_{full} of learning tasks is randomly partitioned between the initial environments. Then, all currently active environments in E start running, controlled and synchronized by the *simulation*, i.e., they all complete their first run, stop, complete their second run and so on. After each run, the simulation *can* interfere with a certain, low probability, namely given by p_{reloc} and p_{fusion}. Common values are p_{reloc}=1/1000 and p_{fusion}=1/10000 to occur for an environment, i.e., when there are 10 environments, such events are expected about all 100 and 1000 runs, respectively. Such an interference means to execute

- with probability p_{reloc}, an *Agent Relocation* occurs, i.e., a single non-mutated offspring switches to another environment, or
- with probability p_{fusion}, an *Environment Fusion* occurs, i.e., two environments are fused, doubling in size of their tasks and their agents, or
- with probability $p_{fusion}/2$, an *Environment Split* occurs, i.e., an environment splits in two, each with the half of the tasks, but with the full set of agents (copies).

Since environment fusions occur with double probability than environment splits, the number of parallel environments shrinks over time. These operations are described in detail below. After such an operation, the new set of environments (in general, there is one new environment, and the others are unchanged), the process continues as before. The official termination criterion is that there is only one environment running, and all tasks are solved by the current population. The other way is to stop the process after some time, grabs one agent of each species, fuses them manually and keeps the resulting agent.

In Section 6.1.4, the effect of this behavior is examined in more detail and compared with static environments.

3.7.1 Agent Relocation

Agent relocation means that a non-mutated offspring of an agent is not created in its own environment, but into another environment. Since this should only be done with previously successful agents, this can only happen if the species is already present in the next generation in its "home" environment (a schematic depiction is given in Figure 3.6). In the early stages of the simulation, such an agent will not be able to survive in an environment for which it is not adapted. But later with a sufficiently generalized configuration the agent should be able to solve at least some tasks in the other environment. However, it is unlikely that the new agent performs better there than the agents that are already present there.

In rare, but very valuable cases, the relocated agent does not die out, but can solve one or more tasks that were previously unsolved in its new environment, but that its species learnt in its earlier environment. In such cases, the relocated agent, and then its species, turns out to be very successful at the beginning, claiming a lot of energy from tasks that were unsolved before. Then, the species produces more offspring, often mutating. By this, that offspring evolve apart again, and often unlearn abilities to solve tasks from the original environment where the species came from, and which are not represented in the tasks of their new environment. It further turned out that the relocated agents do indeed make a difference, however, their performance when applied to the previous tasks is, as expected, worse than it were if all happened in a single environment - they lose abilities where there is no more demand for, (e.g., mutations drop nodes and connections that provided abilities that are not needed any more, and by this reduce the required sustain energy).

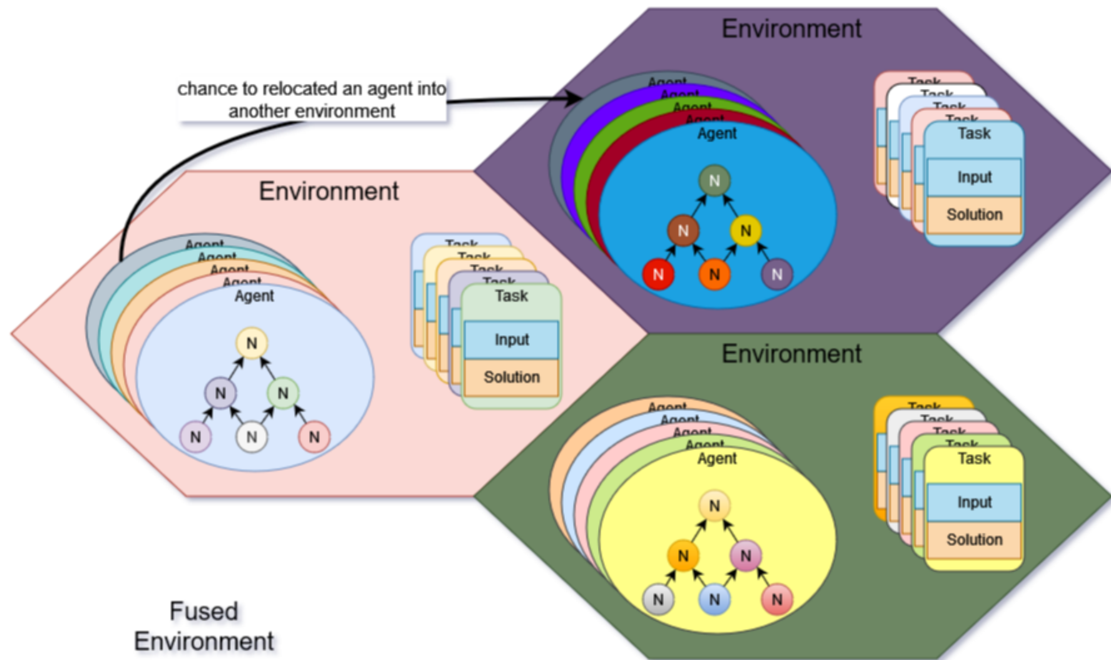


Figure 3.6: Schematic Overview of the learning simulation.
Each simulation contains multiple environments (hexagons), each of which contains a set of agents (ellipses) and a random selection of *task*s (rounded rectangles). Agents contain nodes (circles) and connections (black arrows), *task*s consist of an input and the corresponding solution. There is a low probability that agents invade another environment, therefore agents of the same species can occur in different environments

This observation lead to the idea to allow for *fusing* environments.

3.7.2 Fusion of Environments

An environment can fuse with another environment: The two original environments are removed and new environment is created with the combined agents and tasks of both environments. Depending on the difference of the tasks, the successful species of both former populations coexist or compete for common solutions. Instead of the effect of losing abilities as described before for relocated agents, these abilities still pay after the fusion. Even more, *agent* fusion mutations (see Section 3.3.4) may afterwards also result in hybrids of species from the two original populations that are better suited than each of the original species. A schematic overview is given in Figure 3.7.

Over a longer period of time, this leads to the fact that at some point all active environments merge, containing (but not necessarily solving) the entire set T_{full}. At this point the set of agents gets the opportunity to practice with T_{full} for final evolution.

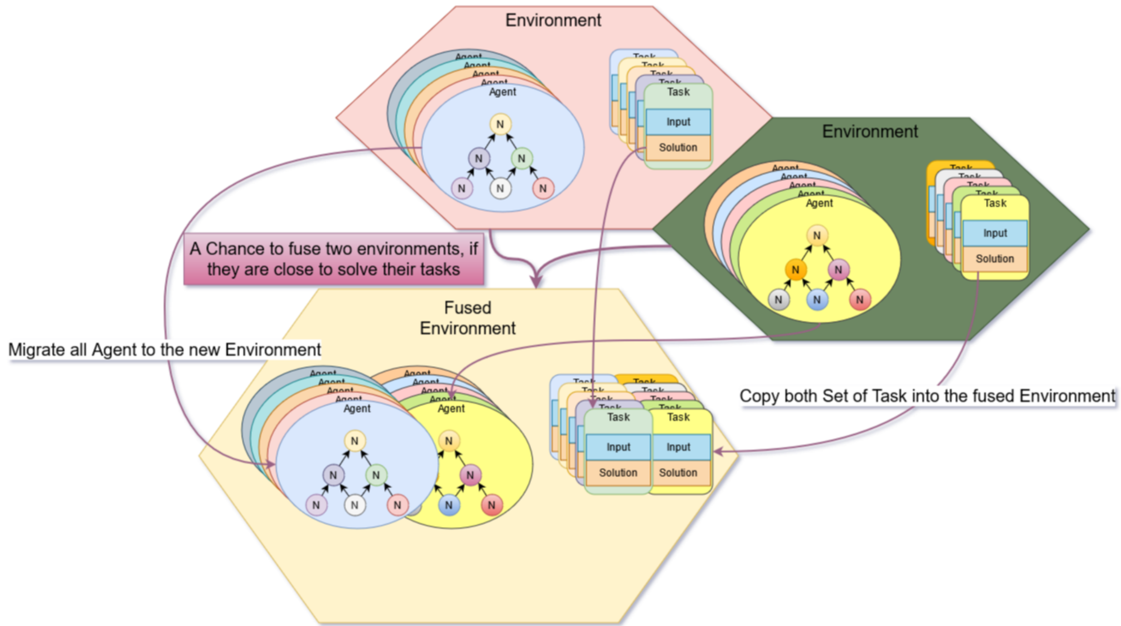


Figure 3.7: Schematic representation of a Environment fusion.

3.7.3 Environment Split

A split of an environment can also happen with a lower probability than a fusion, namely $p_{fusion}/2$), where an environment splits, splitting the task set, but both environments receive a copy of the entire next generation, so that for each task the best-suited agents are always present. The reduction of the available energy by the halving of the number of tasks, leads nevertheless to a massive extinction. Which also means that the computation expenditure remains constant after the first run when the population size over the whole simulation normalizes.

Chapter 4

Applying Evolutionary Dataflow Agents to NLQ

In EvolNLQ, the concept of evolutionary dataflow agents is employed for the concrete task of translating natural language questions into SPARQL queries.

The following example illustrates first what can be expected, and also introduces the different graphics that are used in the frequent examples.

Example 6 *Consider the question GeoBase 191 "which states border Kentucky?". Figure 4.1 shows on top the question text, annotated with Part Of Speech tag and grammatical relations, that is the input to the system generated by CoreNLP. In the middle, the query graph product, an internal representation that is generated by the agent, is shown. It shows the "question graph", consisting of nodes and edges that -roughly-have the following meaning:*

The main part of the graph analyzes the text and maps it to the notions of the application domain, similar to what the entity relationship notation [81] does:

- *rectangles (consider the entity types in ER notation) represent object-valued variables, i.e., ones that range over classes (here e.g. "state")*
- *ovals (consider the attributes in the ER notation) represent literal values of properties (e.g., "name").*
- *lines represent relationships and properties.*
- *diamonds (other than in the ER notation!) denote constants that are used in a query (e.g., "Kentucky").*
- *Every such element carries a name which is usually derived from the word in the question text and its* pseudocount *position in this text.*

For the above ones, green color means that the agent has a good confidence in what it found out, the more towards red, the weaker the confidence.

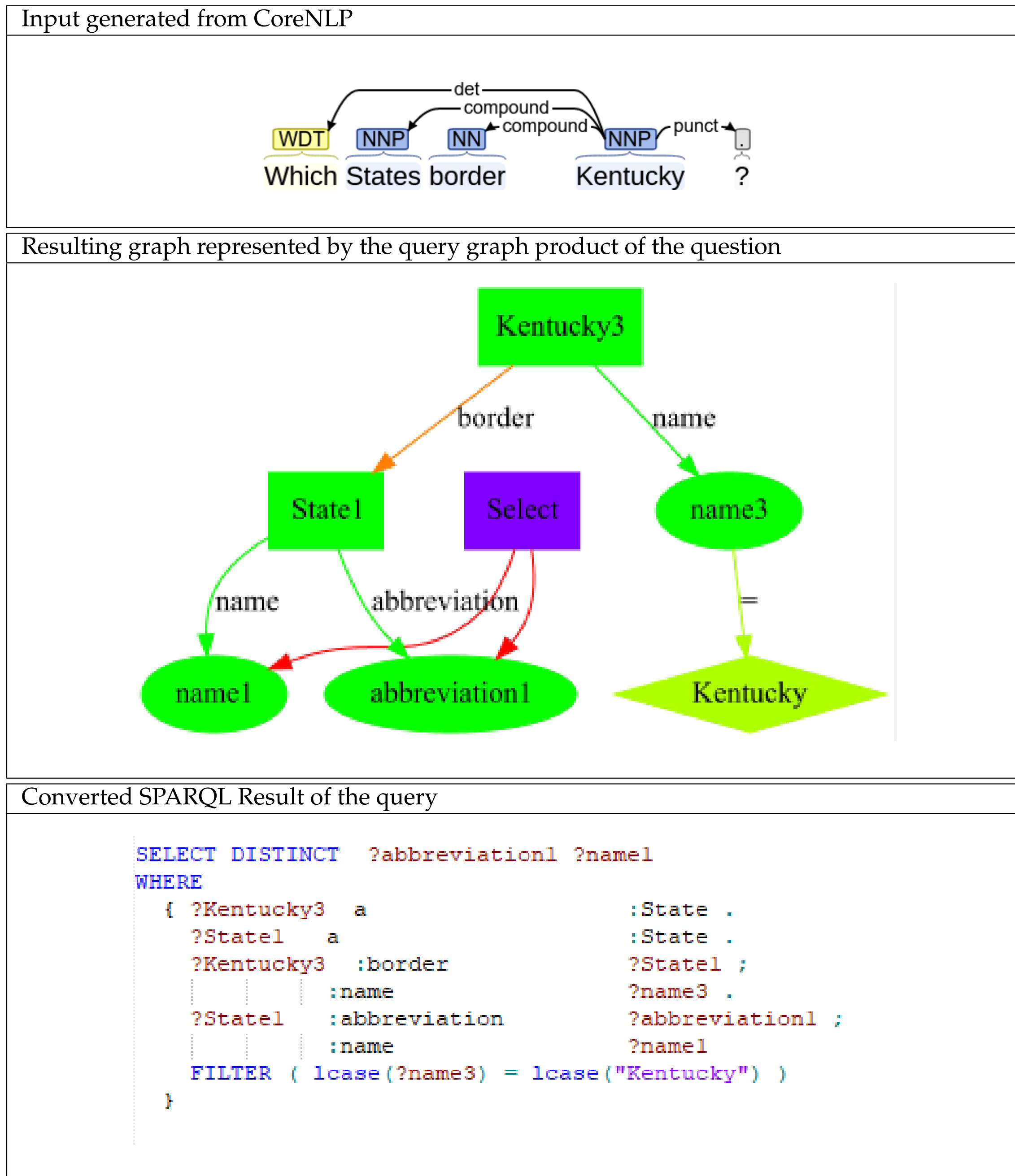


Figure 4.1: Question Geobase191 with CoreNLP anntotations, query graph, and resulting SPARQL query

Furthermore, this graph contains additional elements that represent the structure and properties of the question itself:

- *The "Select" node points to the items that contain the values (names etc.) that should be given as answer.*
- *Further such items will be introduced later, e.g., for negation, for aggregations (counting, maximum, etc.)*
- *Other graphics will later be used especially to give more insights into the representation of such* structural *issues of the representation of the question.*

In the following, sometimes, forward references will be made to get an intuition of some aspects of later examples without yet having discussed all details. In a clickable PDF, the readers can follow the references directly, for others, sometimes the page numbers are added explicitly.

General Considerations when Designing an Evolutionary Dataflow Agents Application

Independent of the *evolutionary algorithm* used, the selection of the operations (in the evolutionary dataflow agent case: the node classes) is crucial for success. When designing operations, the first decision is how specialized these operations are created. Very specific and powerful operations have the advantage of a faster development, since in the long run fewer steps are needed. In addition certain –unwanted– possibilities can be excluded, by simply not providing operations that would support them. In this approach for example, no operations are used, which access domain-dependent information or can generate arbitrary access to the database and therefore the agents cannot construct a combination of operations to access the domain-specific data (so they have to "learn" to handle generic questions). Finally, in such a way also generally valid rules and existing expert knowledge can be used by providing powerful high level operations and not rely on the algorithm to develop similar operation combinations. It is, however, a disadvantage that the solution possibilities are strongly limited. The resulting program might be very close to the original idea of the creator and in extreme cases might better be written directly without the *evolutionary algorithm*. Which means that unforeseen situations are more difficult to cope with, or possibly not at all.

On the other hand, too general methods are also not suitable either. While they offer more possibilities to find a solution, the search space quickly becomes too large.

In particular, the accuracy of the fitness function determines how general and fine-grained the operations can be. If an operation makes so little progress, such that the fitness function is not able to measure it, the progress will not improve the rating of an agent, but increases its costs due to the additional components. Therefore this agent would have a disadvantage compared to its ancestors, even if it is theoretical closer to the optimal solution. This means for the agent, to

actually increase its score, that it had to generate not just a single operation or a short sequence of operation but a longer sequence of correct operations to generate any measurable success in just one run. Depending on the complexity, this can quickly become practically impossible. Further for such applications, other learning techniques like *neural networks*, are much better suited. Therefore, a case-dependent assessment must be made here as to which methods can be used in order to find solutions that are as flexible as possible, but also terminable in a finite amount of time.

Considerations on the Use case

Basically, there node must implement operations for *analysis* of the input question. This can use the individual words, their positions, and the grammatical annotations. From these, small possible fragments of relationships, up to larger contexts must be synthesized. Often, parsing natural language is ambiguous, so always several possibilities have to be considered. So, having an agent architecture that is tailored to massive dataflow, it is possible to instantiate lots of possible, and at least plausible relationships, and to wait and see whether and how they fit. All conclusions are assigned with a confidence value to distinguish between most probable ones and –in case that these turn out to be garden paths– fallback ones. The analysis of the explicit question is only one part towards the query that can be stated against the database which implements a given ontology. This is where the step from a question to a more formal query takes place, marking also the transition from *analysis of the question* towards *synthesis of the query*. Nevertheless, the synthesis steps using he more formal ontology more intensively also help as constraints in the analysis. Often, in a question some entities or relationships, or both are implicit. In the query, these steps must be made explicit. In this case, the ontology can be asked what *paths* are there that can be used to make the query connected by using items that seem to be underspecified or have kind of "free" properties. On the other hand, there are properties that are not yet connected to a specific subject or object, instead having a placeholder. Again, there are often many possibilities, so guessing and heuristics are again to be applied, this time, not on the level of small relationships or contexts, but on a larger graph level. Graph algorithms are not subject to learning, they are well-known. So they are implemented in powerful nodes. This leads again to larger amounts of larger data (that then already represents possible linkages of the whole query) that must be narrowed down, exploiting constraints (coming from the ontology) and eliminating alternatives that have been produced by excessive guessing and have low confidence. Then, a smaller set of possible, per se consistent solution "proposals" is found, and that with the highest confidence is used as an answer. The later the process, the more happens by operating on the structural level of a formal query. Note that this is not the algebraic structure of an SQL or SPARQL algebra query, but a logical structure containing notions like "triples forming the query graph" (so the basis is already SPARQL), "negate a context", "variable x has to be aggregated", "group by that".

Finally mapping this into a valid SPARQL query is again not subject to learning, but is done by the programmer and implemented in Java. This is not implemented in an own node type, but

provided by the respective node classes.

Consequently, the evaluation of the work of the agents is also not based on the (syntactic) SPARQL query, but on the "final" synthesis step before: the structural level of the returned query. The reference solutions are also given on this level, using an XML serialization.

As the agent framework allows for absolute modularity. So, at first, the infrastructure for conjunctive queries was established, then followed by add-ons for more advanced features. This modularity also makes it

Structure of this Chapter

This chapter describes the individual building blocks of EvolNLQ. First, the external and high-level dataflow of EvolNLQ as a whole is described. Then, the individual product classes are introduced, their specific methods are formalized, and application examples are given. Afterwards, the different types of nodes are introduced, as well as which operations they execute, which input and output products are used and how the mapping from the internal products structure to a SPARQL query is executed.

In addition, various examples are used to explain the idea behind the respective nodes and where they are used. It should be kept in mind that none of the nodes or procedures contained can be claimed to be absolutely correct or proven to deliver always correct results. Here lies the actual strength of the learning approach: the usefulness of the nodes is validated by the learning process and nodes whose behavior does not lead to desired results are not used in successful agents and are not used in the long term. Not all developed nodes classes are presented. If it turned out that some kinds of nodes are simply not useful in agents, they are not included in this listing.

For the formalization of operations, sets or functions are occasionally necessary that are not contained in the internal information flow of the agents, e.g. the output of CoreNLP, or information from the Mapping Dictionary. These are introduced in Table 4.1.

Because the product structure is in most cases very close to the structure of SPARQL (especially, the underlying SPARQL algebra), this mapping can be done very intuitively in almost all cases. Nevertheless, the product structure is less expressive and subject to certain limitations discussed in this chapter.

External and High-level Dataflow of EvolNLQ

In the *high-level* dataflow (i.e. the dataflow between the core EvolNLQ evolutionary dataflow agents and the surrounding tools and resources), a distinction must be made between initializing the system, training of the system, and using the system, i.e., querying. A visual overview is given in Figure 4.2:

<table>
<tr><th>Symbol</th><th>Description</th></tr>
<tr><td>$object\ name.slot\ name$</td><td>As usual, the dot notation is used to access the data slots of objects. So the $object\ name$ is the name of an object followed by a dot and then $slot\ name$ the name of the slot of that object means that this value is used or set in the formalization.
In the case of products, it is used to obtain the respective values of the product.
In the case of nodes, this notation is used to access the parameters.
It is also used for MD entries, referring to a specific cell of the table.</td></tr>
<tr><td>P_{Tag}</td><td>Set of Part Of Speech tags defined by CoreNLP</td></tr>
<tr><td>GR_{Tag}</td><td>Set of Grammatical Relationships defined by CoreNLP</td></tr>
<tr><td>T</td><td>Ordered set of words from the raw input text. The ordering is the same as in the sentence itself.</td></tr>
<tr><td>NC</td><td>Set of output data from CoreNLP with T as input</td></tr>
<tr><td>$NC[i]$</td><td>the word at position i (starting with 0) in the input</td></tr>
<tr><td>NC_{Gr}</td><td>Set of all Grammatical Relationships in NC (wrt. the current T; subset of NC)</td></tr>
<tr><td>$\mathcal{CLS}$</td><td>Set of all classes defined in the ontology</td></tr>
<tr><td>$\mathcal{PROP}$</td><td>Set of all properties defined in the ontology</td></tr>
<tr><td>$\mathcal{LIT}$</td><td>Set of all XML Schema literal datatypes [82] (includes string, several numeric datatypes, date, time, etc.)</td></tr>
<tr><td>MD</td><td>Set of entries of the Mapping Dictionary with the configuration $md(class, property, range, inverse)$ with the following functions:
<table>
<tr><th>Name</th><th>Signature</th><th>Mapping</th></tr>
<tr><td>class</td><td>$MD \rightarrow \mathcal{CLS}$</td><td>$in \mapsto \pi[class](in)$</td></tr>
<tr><td>property</td><td>$MD \rightarrow \mathcal{PROP}$</td><td>$in \mapsto \pi[property](in)$</td></tr>
<tr><td>range</td><td>$MD \rightarrow \mathcal{CLS}$</td><td>$in \mapsto \pi[range](in)$</td></tr>
<tr><td>inverse</td><td>$MD \rightarrow \{true, false\}$</td><td>$in \mapsto \pi[inverse](in)$</td></tr>
<tr><td>MD.getConcreteSubcls</td><td>$\mathcal{CLS} \rightarrow \mathcal{P}(\mathcal{CLS})$</td><td>returns all concrete subclasses (not abstract) of a given class</td></tr>
<tr><td>getInverse</td><td>$PROP \rightarrow PROP$</td><td>returns the inverse of a property</td></tr>
</table>
e.g. from Mondial: (country, hasCapital, city, isCapitalOf)</td></tr>
<tr><td>ID</td><td>Set of selected identifier entries, which consist of tuples $id(ClassName, PropertyName, PropertyValue)$ with the following functions:
<table>
<tr><th>Name</th><th>Signature</th><th>Mapping</th></tr>
<tr><td>class</td><td>$ID \rightarrow \mathcal{CLS}$</td><td>$in \mapsto \pi[ClassName](in)$</td></tr>
<tr><td>property</td><td>$ID \rightarrow \mathcal{PROP}$</td><td>$in \mapsto \pi[PropertyName](in)$</td></tr>
<tr><td>value</td><td>$ID \rightarrow WORD$</td><td>$in \mapsto \pi[PropertyValue](in)$</td></tr>
</table>
e.g. from Mondial: (city, name, "New York"); see Figure 2.1 for an example</td></tr>
</table>

Table 4.1: Definitions of sets and functions related to the CoreNLP output and Mapping Dictionary information that are used on multiple occasions

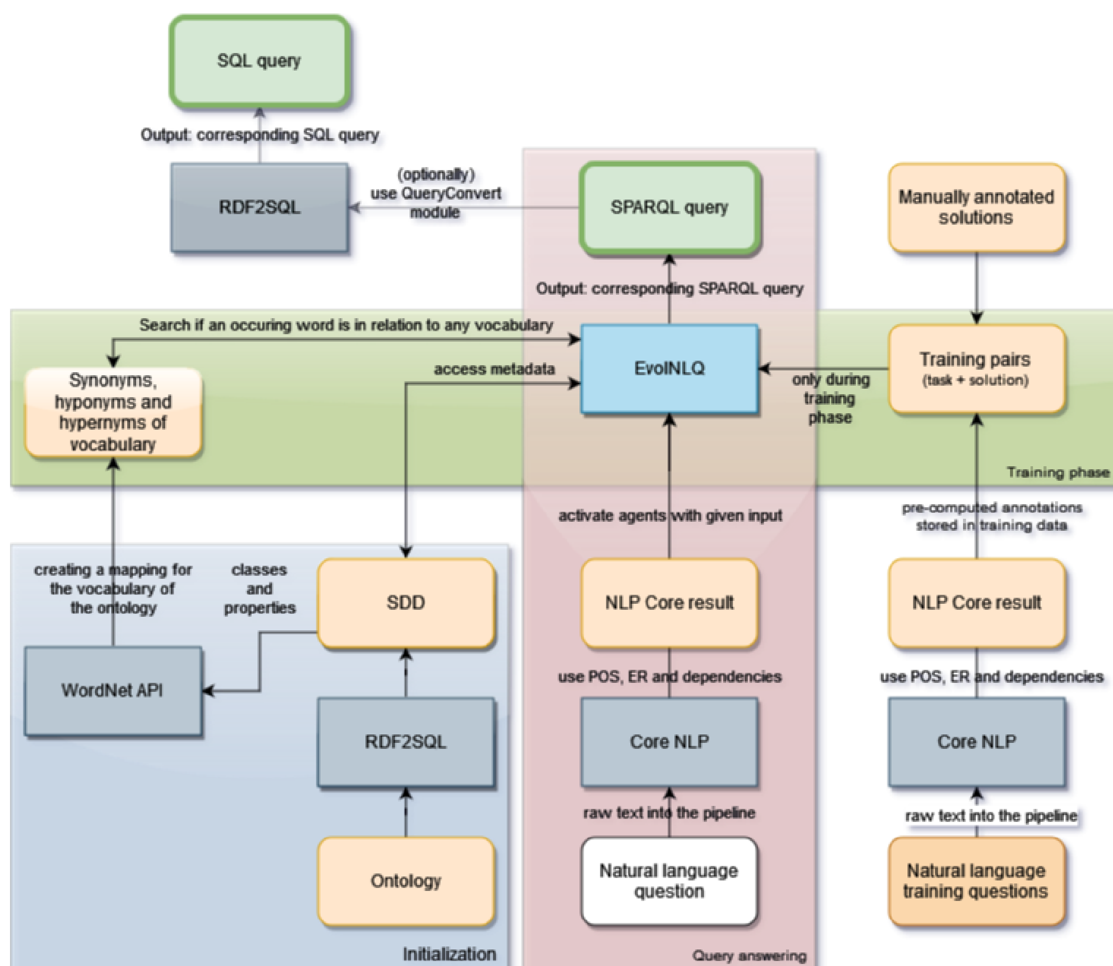


Figure 4.2: External Dataflow of EvolNLQ. Rounded orange boxes are data, programs are in squared boxes. *WordNet*, *RDF2SQL*, and CoreNLP are described in Section 2.3.1.

As depicted in the box on the lower left, before the system can be used, it must be initialized once. For this, *RDF2SQL* is used to create the Mapping Dictionary. The vocabulary of the Mapping Dictionary is then passed to *WordNet*, and EvolNLQ creates a mapping from the results of *WordNet* of all synonyms, hyponyms and hypernyms of the vocabulary of the ontology to the actual terms.

For the training phase (described in detail in Section 3.6.1), the training set is precomputed: the training questions are once ran through CoreNLP and the obtained annotations are stored, while reference results (which are not the query results, but the SPARQL query equivalents to the training questions) are manually created (an example can be found in Section 4.4.1). Thus, during the actual training, CoreNLP is not accessed. The data of the Mapping Dictionary and *WordNet* (see Section 2.1.2.9) is looked up regularly during the training phase.

During the regular querying, CoreNLP is needed for each user's question to annotate the question, since these questions are not known ahead of time. Therefore it is always used once before EvolNLQ starts its own computations. The output data from CoreNLP serves as input to EvolNLQ, which –via the learned agents– transforms it into a SPARQL query. During runtime, the *WordNet* API and the Mapping Dictionary are used from specific nodes in the agents, which might access them several times during a single query translation.

In case that SPARQL is not the target language *RDF2SQL*, (or any other query language translation program, but *RDF2SQL* metadata is already created anyway) can be used to transform the query into SQL.

4.1 Product Classes

The concept of products introduced in Section 3.2 is complemented in this section by the concrete product classes of EvolNLQ.

The different kinds of products used in EvolNLQ are organized in a class hierarchy as shown in Figure 4.3. In addition to the (non-disjoint) generic product classes like CompoundProduct, and Auxiliary already mentioned in Table 3.1 (page 49), there are two EvolNLQ-specific top-level product classes:

- Positionable product are product classes that refer to a word or a range of words in the input question;
- SPARQLing product are product classes that represent parts of the resulting SPARQL query.

In general, every product class has its own properties visualized in the following format:

- **Product class name**: name of the product class.
- **Symbol**: a symbol denoting the set of all instances of this class.

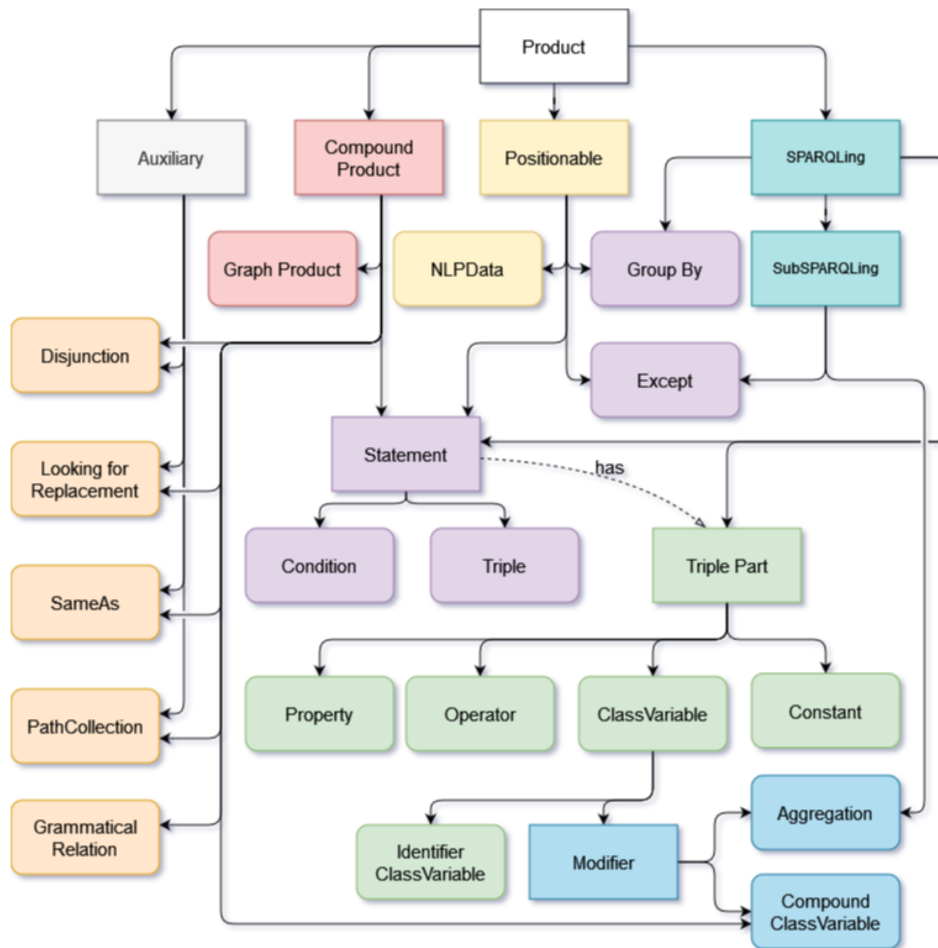


Figure 4.3: Class hierarchy of products. Rectangles represent abstract classes and rounded rectangles represent concrete classes

- **Hierarchy position**: the product classes this class is derived from. Sometimes also non-direct superclasses are listed in parentheses[1] from better understandability.
- **Slots**: property names and their ranges. It is also noted whether the properties refer to *components* of the product, and in which order of those (or unordered), or to simple slots (usually literal-valued or references).
  Possible ranges are:
  - *words* of the input sentence,
  - *numbers* (positions in the sentence),
  - *identifiers of products* (specific strings),

[1]just for the record: compound, positional

  - the class $NODE$ of the nodes of the agent,
  - the class $PROD$ that contains all products,
  - sets of one of the above; formally the domain is then the powerset, denoted by $\mathcal{P}(_)$ of one of the above domains.

- **Functions**: Formal introduction of additional functions that the superclasses do not have (a product class can inherit from multiple direct superclasses) with their arguments (the class itself always listed as first argument) and ranges.
- **SPARQL:** Any product class that is a subclass of the SPARQLing product class supports its translation into SPARQL. This consists of
  - a method *getSPARQLName()*, which is usually defined if it is an atomic expression like a variable;
  - a method *toSPARQL()* that describes how it is mapped into a SPARQL query. If not indicated otherwise, this is just a call of *getSPARQLName()*.

  The exact translation mechanism is explained for each product when it is introduced, if it is not inherited from its SPARQLing superclass (the SPARQL product classes have only one direct SPARQLing superclass).

  The whole SPARQL generation is a very straightforward process in almost all cases, due to the fact that SPARQL is an algebraic, i.e. term-structured, language and its atomic terms are triples (subject, predicate, object). Therefore, the products resulting from certain parts of the sentence (resp. its CoreNLP annotation) can very often directly form substructures of the SPARQL query. The *toSPARQL()* method does not only recursively form a SPARQL query string, but for the correct conversion to SPARQL, Jena's query builder [83] is used. New variables and statements can be added, which are then converted into syntactically correct queries; especially triple patterns, conditions, or `VALUES` clauses can be added to the innermost surrounding query. Furthermore, each query builder can be given another query builder as a subquery. The description describes such operations in an intuitive way; for details it is recommended to have a look at the original program code.

  EvolNLQ is, however, more limited than SPARQL in its expressiveness, so some shortcuts can be taken here that simplify more complex structural constructs significantly. In particular, subqueries are usually only marked up by a single product and then have to be generated during translation. For example, for an except product (e.g. in "all cities that are *not in a certain relationship*"), the except product product is simply an interval from a position up to the end of the sentence which indicates everything which is located within the interval is part of the "not exists", while its translation is a subquery that subtracts its result set from the main query.

This simplification can be used as it is very complicated to describe situations in which those would lead to conflicts in natural language and thus would become too complex for EvolNLQ anyway.

4.1.1 Product

Space: $PROD$

Addition(s) to slots:

Name	Range	Comp. Pos.
$origin$	$NODE$	-
$name$	string?	-
confidence	number	-

Each product has at least the slots ($origin$, $name$, *confidence*):

- *origin* is the node it was created by,
- *name* is the name of the node instance. It is optional, because it is not used for all node types.
- *confidence* is a value between 0 and 1 assigned by the origin node.

4.1.2 Compound product

Space: *CPROD*

Subclass Of: product

Compound products are products that contain a set of *components*, i.e. a set of other products (all other products are of the class atomic product which does not add any generic slots or functionality). An important usage of them is to synchronize the transport within the information flow, and to operations that are applied to the set. A generic subclass are graph products (see below).

The compound product class also serves for the case that the component set is unary, containing a single product of some class, and adding information and/or functionality to it, e.g. the projection product operation.

For some subclasses of compound product, the set of components is ordered and of fixed size: the size is 3 for product classes that represent relations of (atomic) products, e.g. the triple products and comparison products. These ordered compound products have a special meaning, for example subject, predicate, and object for the components of a triple product. An example that uses several compound product classes is given later in Figure 4.4.

Additional or overridden Function(s):

Name	Signature	Mapping
$getComponents$	$CPROD \rightarrow \mathcal{P}(PROD)$	$cp \mapsto \mathcal{P}(p) : \mathcal{P}(p)$ is the content of all slots with a "comp. Pos" in noted order.

4.1.3 Query graph product

Space: QGP

Subclass Of: compound product

Addition(s) to slots:

Name	Range	Comp. Pos.
$components$	$\mathcal{P}(PROD)$	unordered

The query graph product type is not a generic product type, but is EvolNLQ-specific, intended to handle SPARQL query pattern graphs. Thus, query graph products are compound products with unordered components. They interpret statement products as edges and atomic products of the Variable and Constant classes as nodes and provide operations on that graph. Such atomic products having the same *name* are considered the same node in the graph. In case that products have the same name, but differ in some property values, the more restrictive one is preferred (this case occurs only if one product carries more specific information than the other).

The components of a query graph product usually include also one or more projection products, and several auxiliary products that belong to the query.

An example of such a graph product (and sample compound products) is shown in Figure 4.4.

A statement product, with subject and object as nodes and the property forms a named directed edge between the two nodes. Sequences of such statements as directed edges are then interpreted as a paths in the graph.

Additional or overridden Function(s):

Name	Signature	Mapping
findPath	$QGP \times TP \times TP \rightarrow \mathcal{P}(STMT)$	shortest connection between two TriplePart products (interpreted as nodes) via statement products using Kruskals pathfinding algorithm [76]
connected	$QGP \rightarrow \{true, false\}$	$g \mapsto$ if $\exists\ tr_a, tr_b \in components(g)$, $a = subject(tr_a)$, $b = subject(tr_b)$ and $FindPath(a, b) = \emptyset$ then *false* else *true*
replace	$\{PROD\} \times \{PROD\}$	$toReplace \times replacement \mapsto QGP([components = (self.component \cup replacement)/toReplace])$

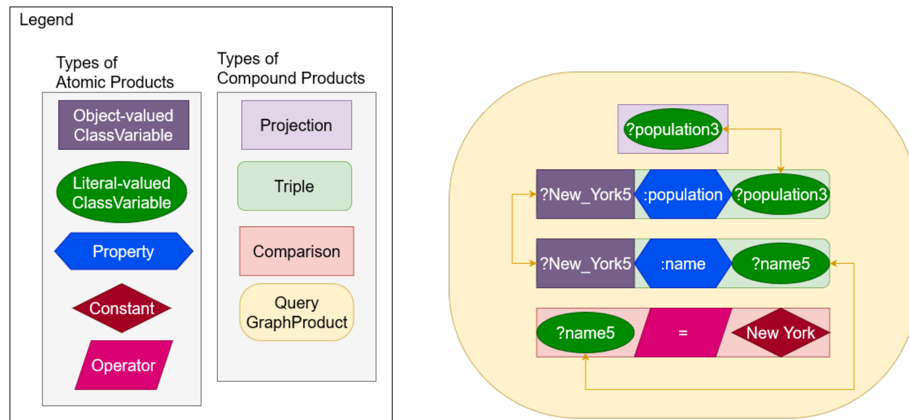


Figure 4.4: Diagram of the encapsulation of compound and atomic products for the question "What is the population of New York City" (Geobase Query 68); the graph itself is also shown in Figure 4.13 (page 102).

4.1.4 Positionable product

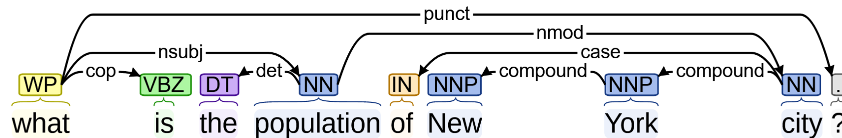
Space: *POS*

Addition(s) to slots:

Name	Range	Comp. Pos.
p_{min}	$\mathbb{N}$	-
p_{max}	$\mathbb{N}$	-

Positionable products are products that refer to a specific position, or to an interval of words. A positionable product has the signature (p_{min}, p_{max}) with p_{min} the beginning of the interval and p_{max} the end. Wrt. CoreNLP's annotations, the positionable product is considered to be in all relationships that refer to at least one position of the interval.

Example 7 *Consider the CoreNLP-annotated query Geobase68:*



During the process, "population" is derived to be an instance of class property product (which is a subclass of positionable product) at position 3 (the CoreNLP pseudocount starts with 0). "New York City" will be derived to belong together, and that it is a product of the class ClassVariable product (which is also a subclass of positionable product). It ranges over three words and therefore has a positional interval [5,7]:

Product Class	*source word*	p_{min}	p_{max}
property product	*population*	3	3
ClassVariable product	*New York City*	5	7

All grammatical references to any of these positions (e.g. that "population" has an "nmod" relationship with the POS NN tag "city" at position 7) are concluded to apply to NYC.

Additional or overridden Function(s):

Name	Signature	Mapping
$scope$	$POS \rightarrow \mathbb{N} \times \mathbb{N}$	$pos \mapsto [p_{min}, p_{max}]$
in	$POS \times POS \rightarrow \{true, false\}$	$p_1 \times p_2 \mapsto$ if the intervals $[p_1.p_{min}, p_1.p_{max}]$ and $[p_2.p_{min}, p_2.p_{max}]$ have a non-empty intersection, then return *true* else *false*

4.1.5 NLPData product

Space: *NLPD*

Subclass Of: positionable product

Addition(s) to slots:

Name	Range	Comp. Pos.
$word$	string	-
$type$	P_{Tag}	-
$subtype$	P_{Tag}	-
$entityType$	P_{Tag}	-
$normalizedNamedEntity$	P_{Tag}	-

The NLPData product class is the direct product implementation of the output of CoreNLP. Each NLPData product contains a word $t \in T$, its position in t and its Part Of Speech tag (type) pt and if it is of any more specialized subtype $ptsub$ of pt, then this subtype information as well. If the entity recognizer founds an entity the corresponding tag is stored in $entityType$ and the normalized named entity in which is the concert values is stored in $normalizedNamedEntity$.

Example 8 *Question Geobase40: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase40 are depicted in Figure 4.5. The NLPData product for the same query is shown in Table 4.2.*

4.1.6 SPARQling

Space: *SPA*

Subclass Of: product

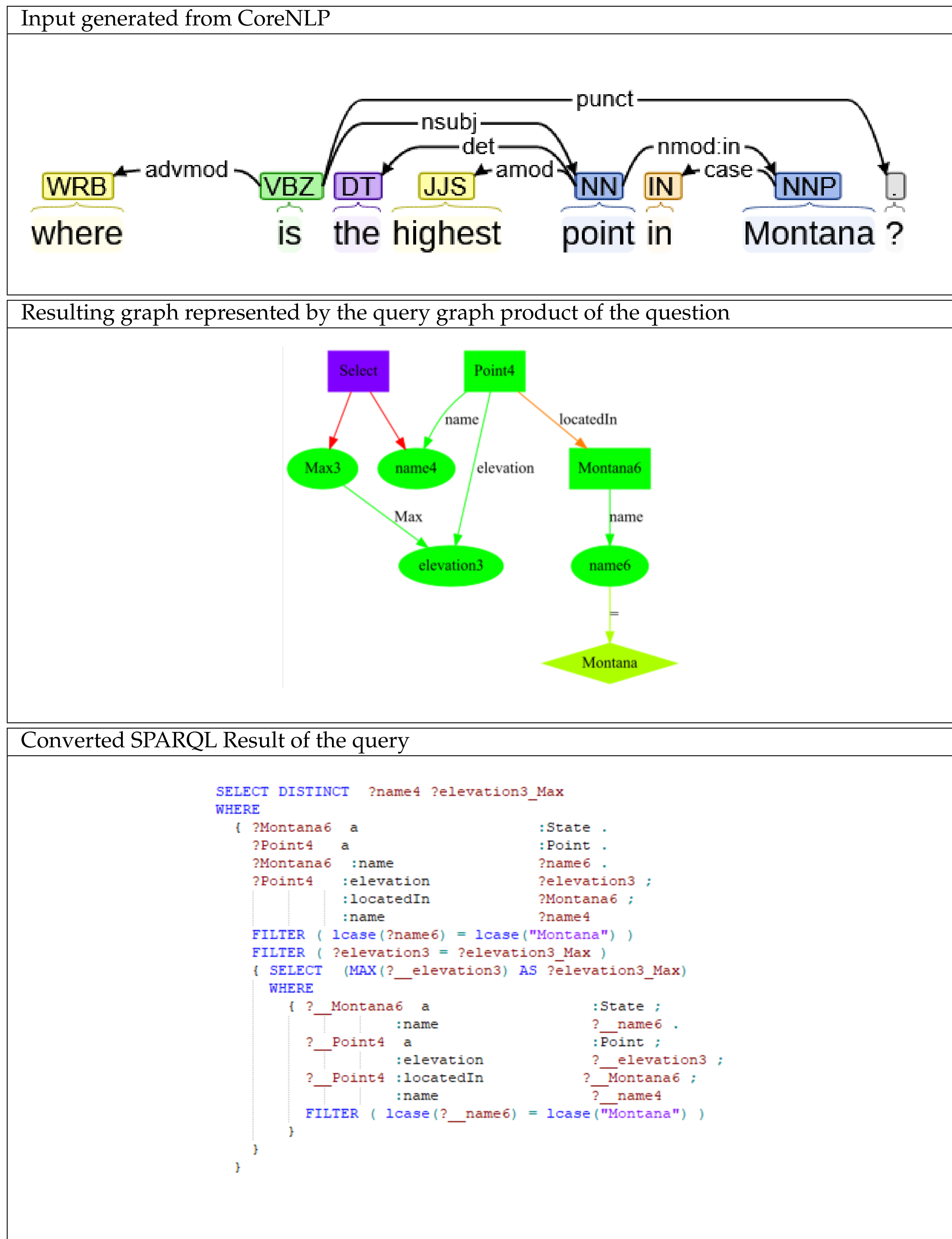


Figure 4.5: Question *Geobase40* with CoreNLP anntotations, query graph, and resulting SPARQL query

Word	(Part of Speech) type	(Part of Speech) SubType	Pseudo count
Where	WRB	-	0
be	VBZ	-	1
the	DT	-	2
highest	JJS	-	3
point	IN	-	4
in	FW	-	5
Montana	FW	in	6
?	.	-	7

Table 4.2: NLPData product for example 8

The SPARQLing class is the superclass of all products that have been designed as constituents of the final SPARQL query – into which they will be converted. It provides an abstract function to generate a SPARQL fragment.

Additional or overridden Function(s):

Name	Signature	Mapping
toSPARQL()	$SPA \rightarrow string$	abstract, implemented for each subclass individually

4.1.7 SubSPARQLing product

Space: *subSPA*

Subclass Of: compound product

A subSPARQLing product is a nested *sub*query that is in fact an application of an operation like NOT EXISTS or an aggregation to a query. It is a compound product, containing a single component (its subquery). Its *toSPARQL* method consists of writing a separate subquery, i.e., renaming the variables, and putting the operation into the SELECT clause.

Algorithm 1: Subquery generating Algorithm *createSubquery*

Result: A subquery statement for a SPARQL query for a given SPARQLing product
Input: Set of SPARQLing products S (i.e., its components)
SPARQLBuilder $builder \leftarrow new\ SPARQLBuilder$
foreach $s \in S$ **do**
  recursively rename all variables v: $v.name \leftarrow$ "__" $+ v.name$
  $builder.Add(s.toSPARQL())$
end
return $builder$

Additional or overridden Function(s):

Name	Signature	Mapping
$createSubQuery()$	$subSPA \rightarrow TP$	see Algorithm 1

4.1.8 Except product

Space: *EXC*

Subclass Of: positionable product, subSPARQLing product

The except product class is the counterpart for the SPARQL keyword EXCEPT. For this purpose, the except product defines an interval $i = [p_{min}, p_{max}]$ (referring to a part of the question) which is to be negated. Structurally, p_{min}, p_{max} are the slots that is inherits from being a positionable product.

Note that by this, an except product is *not* a compound product that would contain its own graph, but that it is just a component of the surrounding query graph product, as illustrated in Example 9.

For the translation to SPARQL, all components of the query graph products where the except product itself is a component of (i.e., the surrounding query) that have at least one component or position in the "negated" interval i are pulled into a subquery and this in turn is prefixed with an EXCEPT.

It the query graph itself is not disconnected, there is (at least) one variable that occurs both inside the subquery and outside.

Converting into SPARQL:

Conditions	SPARQL Translation
true	see Algorithm 2

Algorithm 2: Except's *toSPARQL()*

Result: Changes SPARQL Builder content fitting to the except product
Uses: the SPARQLBuilder $builder$ of the surrounding query, the Except product exc, and the Query graph product g where it is contained in (i.e., of the surrounding query)
foreach $p \in components(g)$ **do**
  if $class(p) \in$ *POS* ***and*** $p.in(exc)$ **then**
    builder.removeFromWhere(p)
    builder.addtoMinus(p)
  end
end

Example 9 *Question Mondial2: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Mondial2 are depicted in Figure 4.6. In this query is shown how the except product is built. It specifies the interval on which the negation keyword applies, and then all statements that contain positionable products that are within that interval are moved to the minus part of the query. It should be noted that for the triple product (Country6 :encompasses Africa12) its positionable products interval is*

defined from the component with the lowest position up to the product with the highest position and the triple product is located at any point in that interval, therefore it is in the except product but Country6 itself is not. The ClassVariable product "Country6" (denoting that it comes from the word "country" at the pseudocount position 6) is the variable that connects the outer query and the negated subquery.

Example 10 *Question Geobase181: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase181 are depicted in Figure 4.8. A schematic representation of the product structure is shown in Figure 4.9. Note that the aggregation product is just one of the components of the query graph product of the whole query. It does not itself contain a (sub)graph, but consists simply of naming the aggregation operator and a pointer to the ClassVariable product on which it is applied.*
The resulting SPARQL query illustrates that the subgraph (query pattern for geographical points/mountains) is created, the SELECT clause with the MAX(...) is added to it, and the result becomes a subquery of the main query, where the FILTER is added.
Furthermore, it illustrates that it is necessary to pull a copy of the outer query inside (although, it's not the perfect example from the point of view of the application semantics): assume that a mountain m could be given, which is not *located in any state and has the highest elevation. If the subquery did not include the complete content, the result would be that mountain m. The outer query, however, would only bind mountains that are located in states to the variable, and the elevation would not be equal to Max_Elevation (which contains the elevation of m) and the result would ultimately be empty, although there are mountains in states and have an elevation and consequently also one or more with the maximal elevation.*

4.1.9 Statement product

Space: $STMT$

Subclass Of: compound product, positionable product, SPARQLing product

Statement is the abstract, collective term for triple products and comparison products. It is used to summarize all relation-forming products. Statement products are compound products that consist of three ordered components.

Additional or overridden Function(s):

Name	Signature	Mapping
$subject$	$STMT \to CV$	abstract
p_{min}	$STMT \to \mathbb{N}$	$stmt \mapsto min(subject.p_{min}, predicate.p_{min}, object.p_{min})$
p_{max}	$STMT \to \mathbb{N}$	$stmt \mapsto max(subject.p_{max}, predicate.p_{max}, object.p_{max})$
$scope$	$STMT \to \mathbb{N} \times \mathbb{N}$	$stmt \mapsto [p_{min}, p_{max}]$

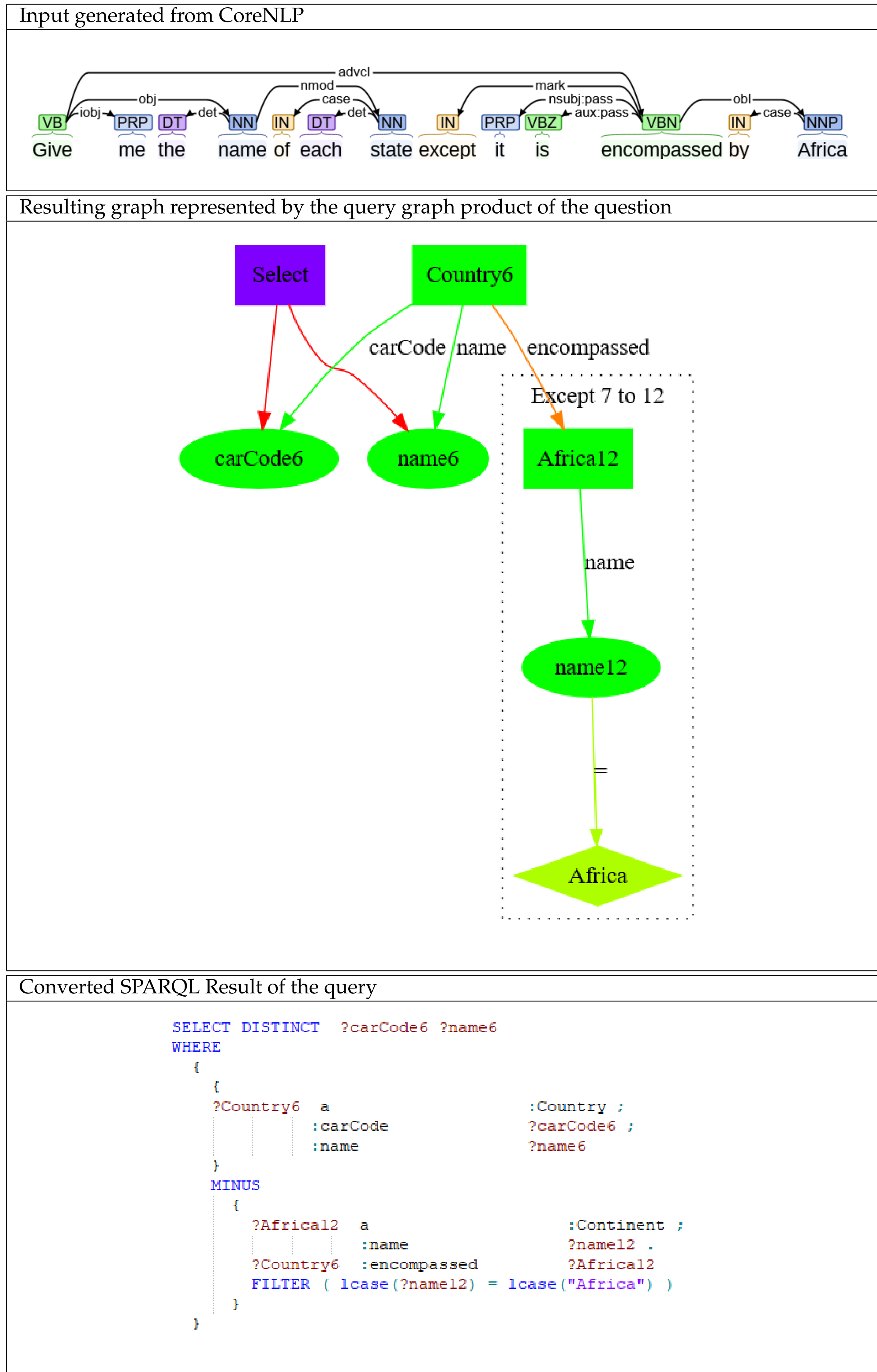


Figure 4.6: Question *Mondial2* with CoreNLP anntotations, query graph, and resulting SPARQL query

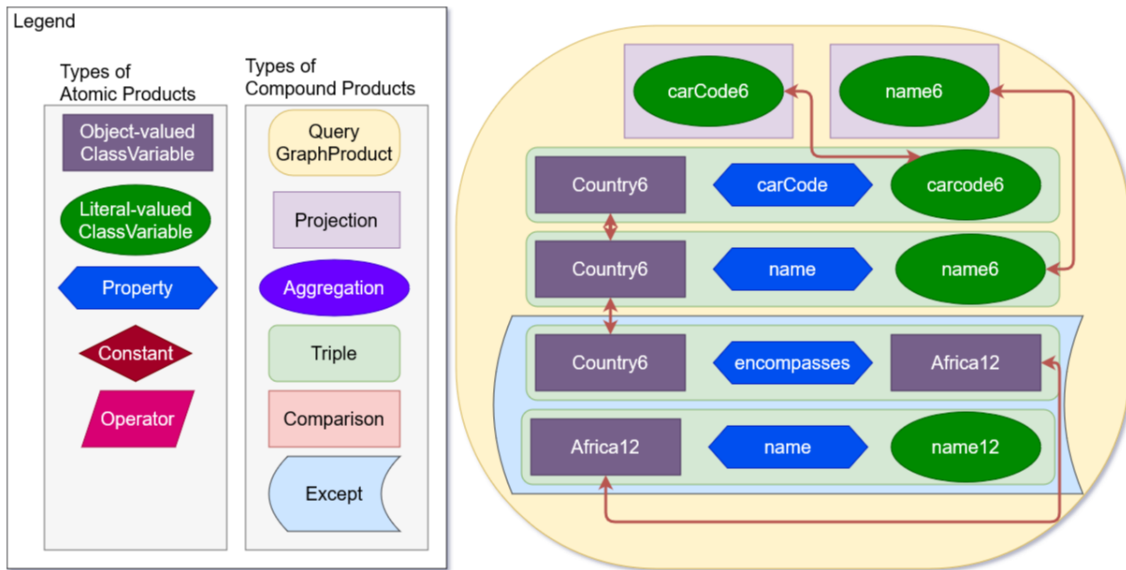


Figure 4.7: Query graph product structure of query Mondial2, which is depicted in Figure 4.6. It illustrates the use of a except product.

4.1.10 Triple product

Space: *TR*

Subclass Of: statement product, (compound product, positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
$subject$	CV	0
$predicate$	$PROP$	1
$object$	$CV \cup CONST$	2

The triple product corresponds to the SPARQL triple pattern and contains in each case an ordered set of three TriplePart products that act respectively as subject, predicate, and object and thus form the basic statement products of the query. The subject has to be an object-valued ClassVariable product, the object can be an object-valued ClassVariable product, a literal-valued ClassVariable product or a constant product, while the predicate has to be a property product (see also Figure 4.10). Each triple also contains a positional interval corresponding to the minimal and maximal pseudo counts of its components.

Additional or overridden Function(s):

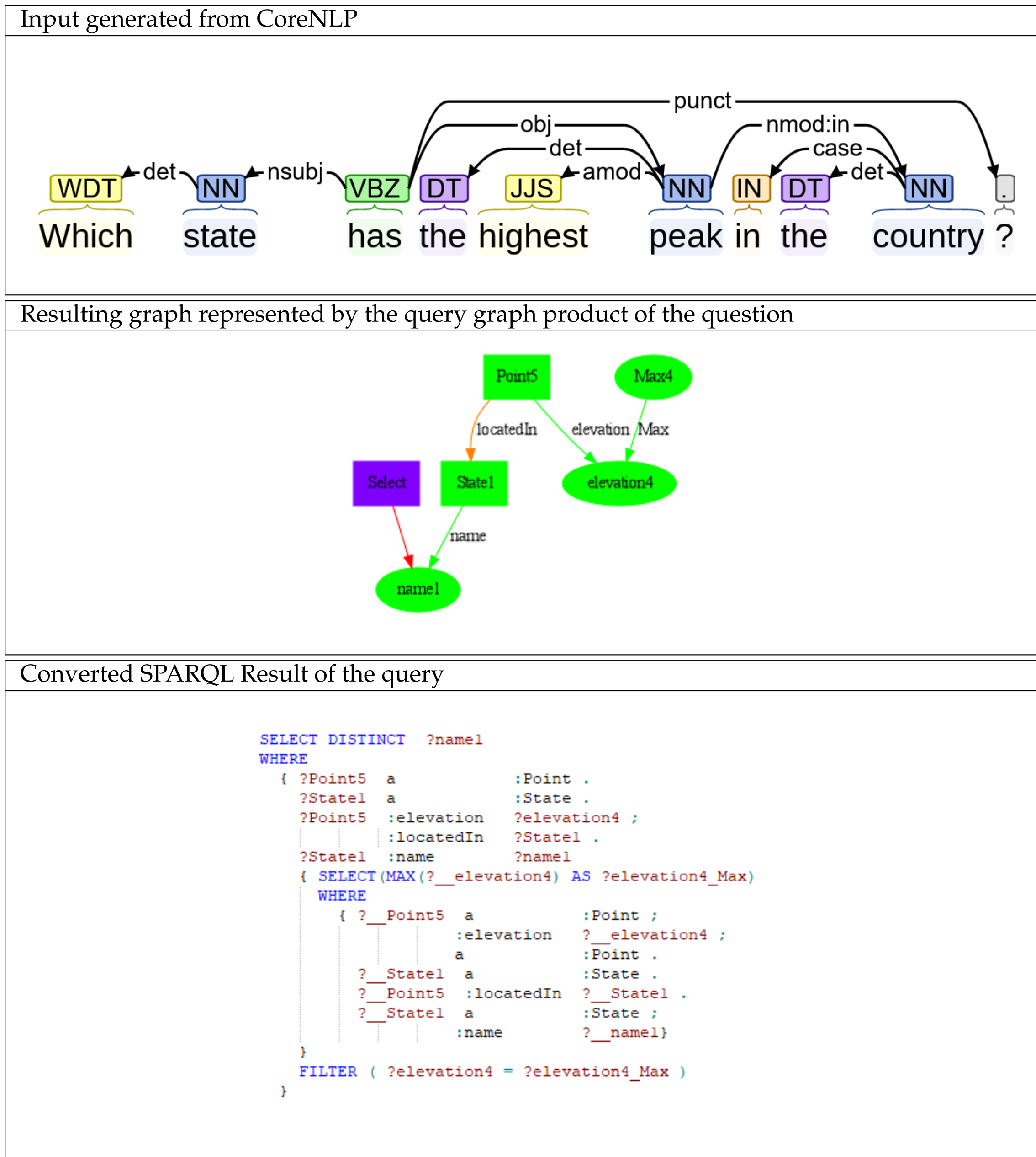


Figure 4.8: Question *Geobase181* with CoreNLP anntotations, query graph, and resulting SPARQL query

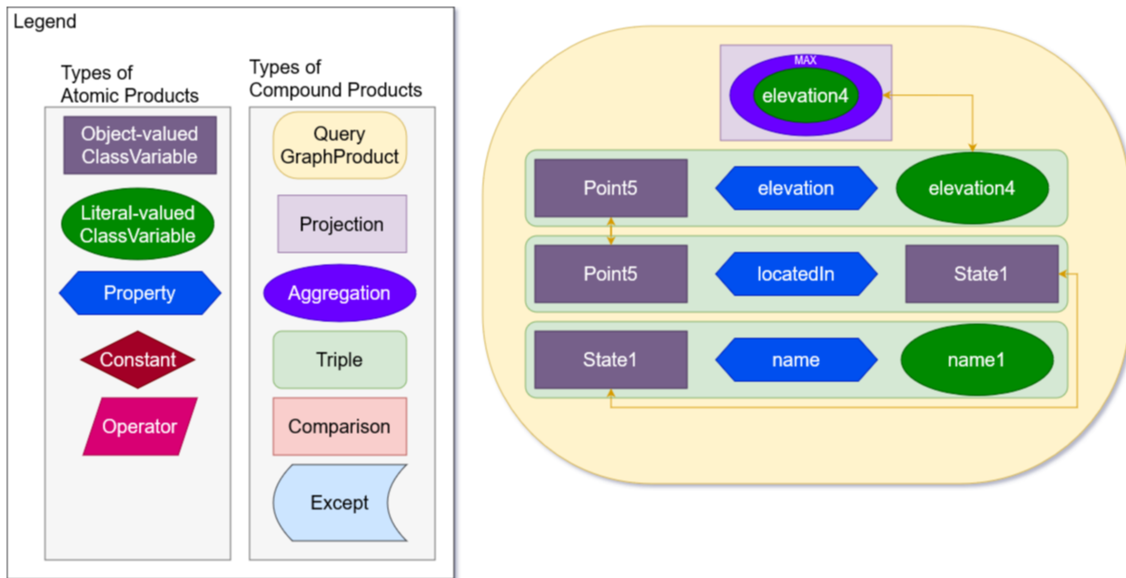


Figure 4.9: Query graph product structure of query Geobase181, which is depicted in Figure 4.8. It illustrates the use of a "max(elevation)" aggregation.

Name	Signature	Mapping
$subject$	$TR \to CV$	$tr \mapsto tr.subject$ $tr \mapsto TR([subject := tr.object,$ $predicate := tr.predicate.invert(),\ object := tr.subject])$

Converting into SPARQL:

Conditions	SPARQL Translation
true	triple $(subject.toSPARQL(), predicate.toSPARQL(), object.toSPARQL())$

4.1.11 Comparison product

Space: *COMP*

Subclass Of: statement product, (compound product, positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
$left$	$CV \cup CONST$	0
$operator$	OP	1
$right$	$CV \cup CONST$	2

Comparison products are the equivalent of the *FILTER* keyword in SPARQL. Very similar to

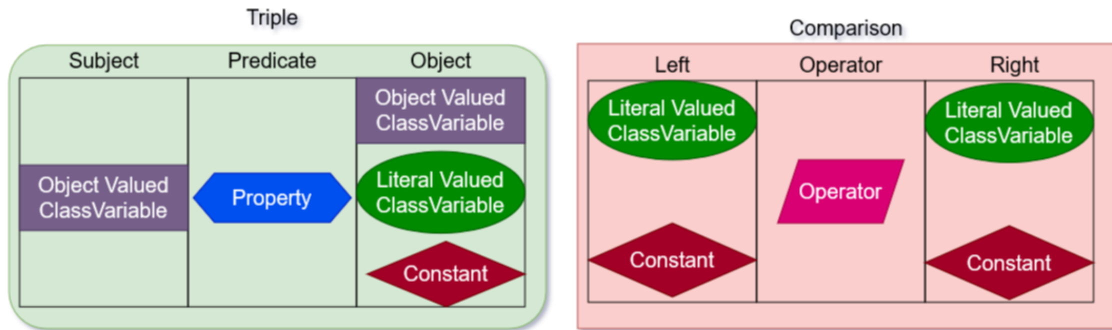


Figure 4.10: Structure of the two different types of statement products
Inner structure of triple product (left) and comparison product (right). Both statement products contain an ordered set of three TriplePart products with a special meaning and type restrictions for each position individually.

triples, conditions are also tripartite statement products. The first part and the last part must be literal-valued ClassVariable products or a constant product, but the second part for conditions must be an operator product (see also Figure 4.10). The $subject()$ function is adapted to deliver any variable of the statement if exists.

Additional or overridden Function(s):

Name	Signature	Mapping
$subject$	$STMT \rightarrow CV$	*comp* $\mapsto$ if *comp.left* is a ClassVariable product then *comp.left* else if *comp.right* is a ClassVariable product then *comp.right* else $\emptyset$

Converting into SPARQL:

Conditions	SPARQL Translation
true	condition (*left.toSPARQL()*, $operator.toSPARQL()$, $right.toSPARQL()$)

4.1.12 TriplePart product

Space: *TP*

Subclass Of: positionable product, SPARQLing product

TriplePart product is an abstract class of products that can be anything that can be part of a statement product. As such, it always has a more or less direct connection to a position within the input sentence and is therefore also a positionable product.

Additional or overridden Function(s):

Name	Signature	Mapping
getSPARQLName	*TP* $\rightarrow$ string	abstract, implementation the subclasses

4.1.13 ClassVariable product

Space: CV

Subclass Of: TriplePart product, (positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
$domains$	$\mathcal{CLS} \cup \mathcal{LIT}$	-

A ClassVariable product is the representation of a set of individuals or literals. Besides the general properties of a TriplePart product, it also has a set of classes (domains) that represent the domains of the ClassVariable product. The distinction between object-valued ClassVariable products and literal-valued ClassVariable products is decided based on whether all of the domains are in $Class(MD)$, in which case it is an object-valued ClassVariable product, otherwise, if none of them is in $Class(MD)$, it is a literal-valued ClassVariable product; if there is a mix between classes and literal datatypes, the product is considered invalid and the product is removed.

Example 11 *Question Geobase175: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase175 are depicted in Figure 4.11. The example illustrates the use of a ClassVariable product that ranges over multiple domains.*

Additional or overridden Function(s):

Name	Signature	Mapping
$isLiteral$	*CV* $\rightarrow$ {*true*,*false*}	$cv \mapsto$ if $cv.domains \subseteq \mathcal{CLS}$ then *false* else *true*
getSPARQLName	*CV* $\rightarrow$ string	$cv \mapsto$ "?" $+ domains[0] + pos_{min}$

Converting into SPARQL:

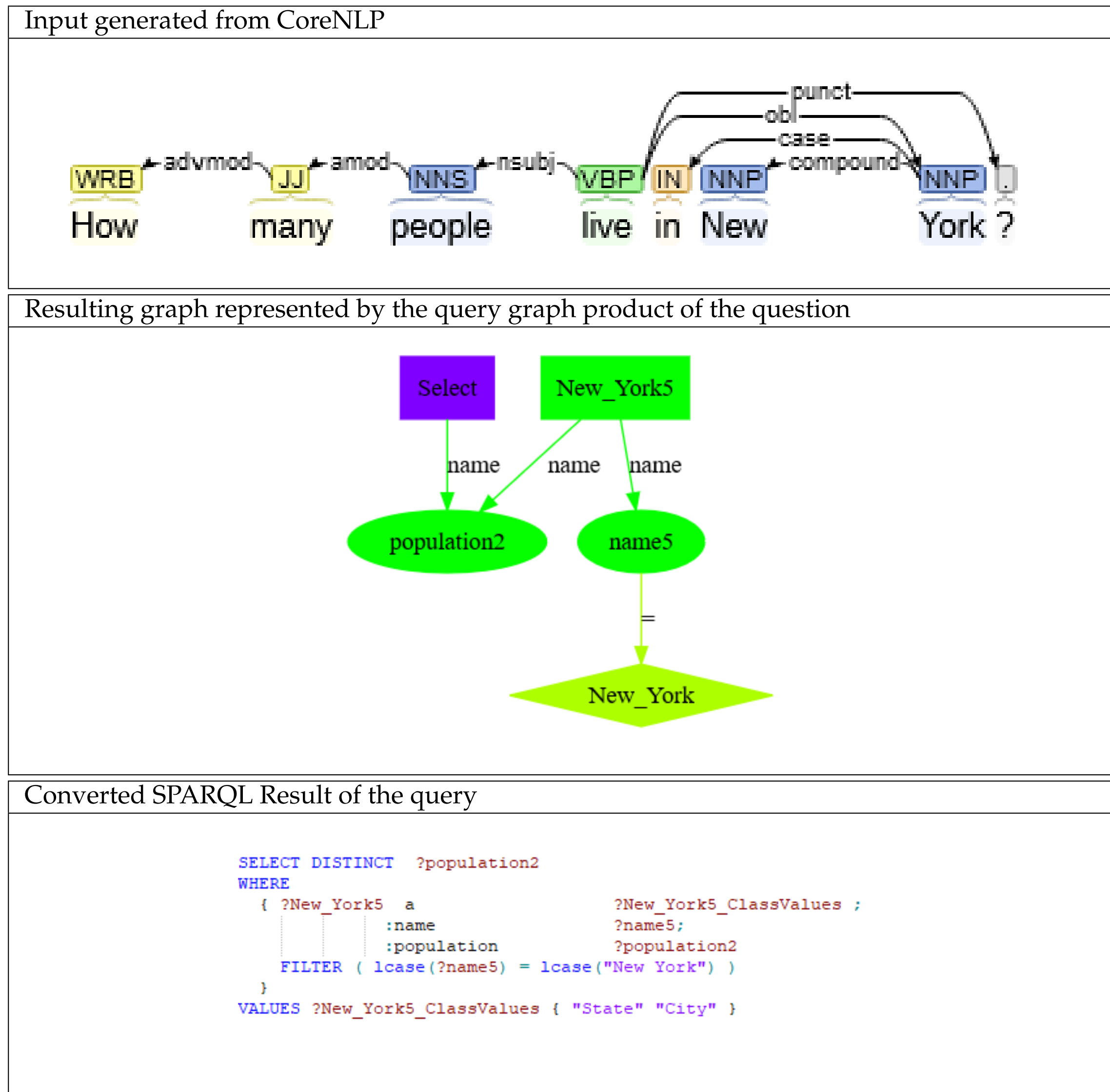


Figure 4.11: Question *Geobase175* with CoreNLP anntotations, query graph, and resulting SPARQL query

Conditions	SPARQL Translation
$isLiteral$	$getSPARQLName()$
$\neg\, isLiteral$ and $\|domains\| = 1$	$getSPARQLName()$ add to surrounding query: triple($getSPARQLName()$, rdf:type, $domains[0]$)
$\neg isLiteral$ and $\|domains\| > 1$	$getSPARQLName()$ add to surrounding query: triple($getSPARQLName()$, rdf:type, $getSPARQLName()$ + "_ClassValues") and "VALUES " + $getSPARQLName()$ + "_ClassValues" + "{" + $domains$ + "}"

4.1.14 IdentifierClassVariable product

Space: *IDCV*

Subclass Of: ClassVariable product, (positionable product)

Addition(s) to slots:

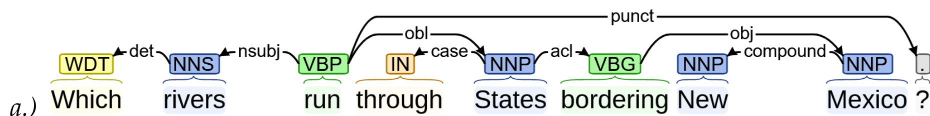
Name	Range	Comp. Pos.
tp_{id}	*TR*	-

IdentifierClassVariable products are ClassVariable products that refer to a specific individual. Note that they are *not* compound products. They also have exclusive claim to the property product that gives them their identity. The IdentifierClassVariable product extends the signature of the ClassVariable product by referring to a triple product tp_{id} illustrated in Figure 4.12. The tp_{id} that identifies it as itself cannot be used by another TriplePart product.

Additional or overridden Function(s):

Name	Signature	Mapping
getSPARQLName	*IDCV* $\rightarrow$ string	$icdv \mapsto$ "?" $+ tp_{id}.subject.name + tp_{id}.subject.pos_{min}$

Example 12 *Geobase Query 1: a.) output of CoreNLP b.) query graph product of the Query c.) Converted SPARQL Result of the Query*



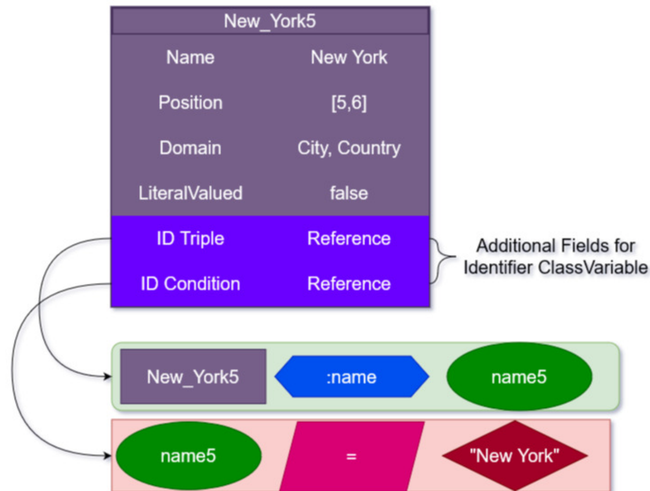
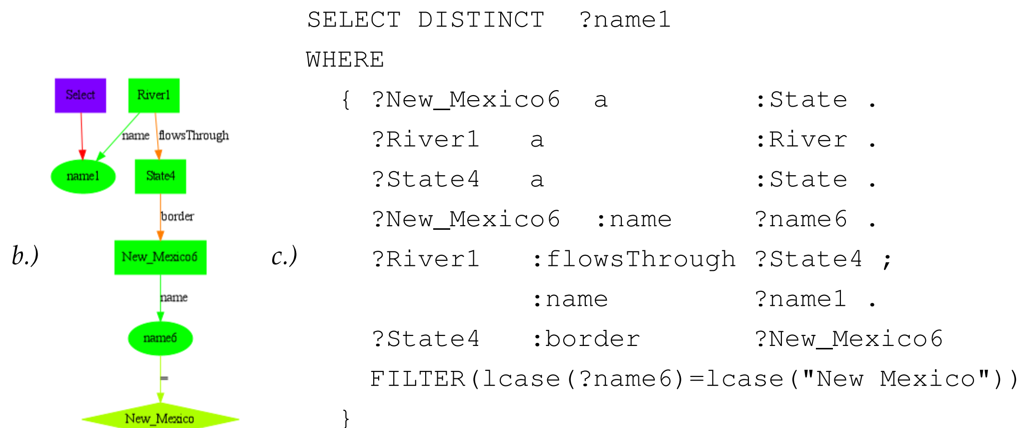


Figure 4.12: ClassVariable product for New York in Geobase Query 68 in example 13 at a point, where the domain is not finally determine between city and country. In mauve properties which every ClassVariable product has and in light purple the additional references of IdentifierClass-Variable product




```
SELECT DISTINCT  ?name1
WHERE
  { ?New_Mexico6  a           :State .
    ?River1     a             :River .
    ?State4     a             :State .
    ?New_Mexico6  :name       ?name6 .
    ?River1     :flowsThrough ?State4 ;
                :name         ?name1 .
    ?State4     :border       ?New_Mexico6
    FILTER(lcase(?name6)=lcase("New Mexico"))
  }
```

Example 13 *Question Geobase68: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase68 are depicted in Figure 4.13. This query is an example of an ambiguous identifier, in this case New York, which in the ontology is both the name of the state and the city. Here it must be recognized that there is not an additional city, but that the identifier is subsequently clarified with its class.*

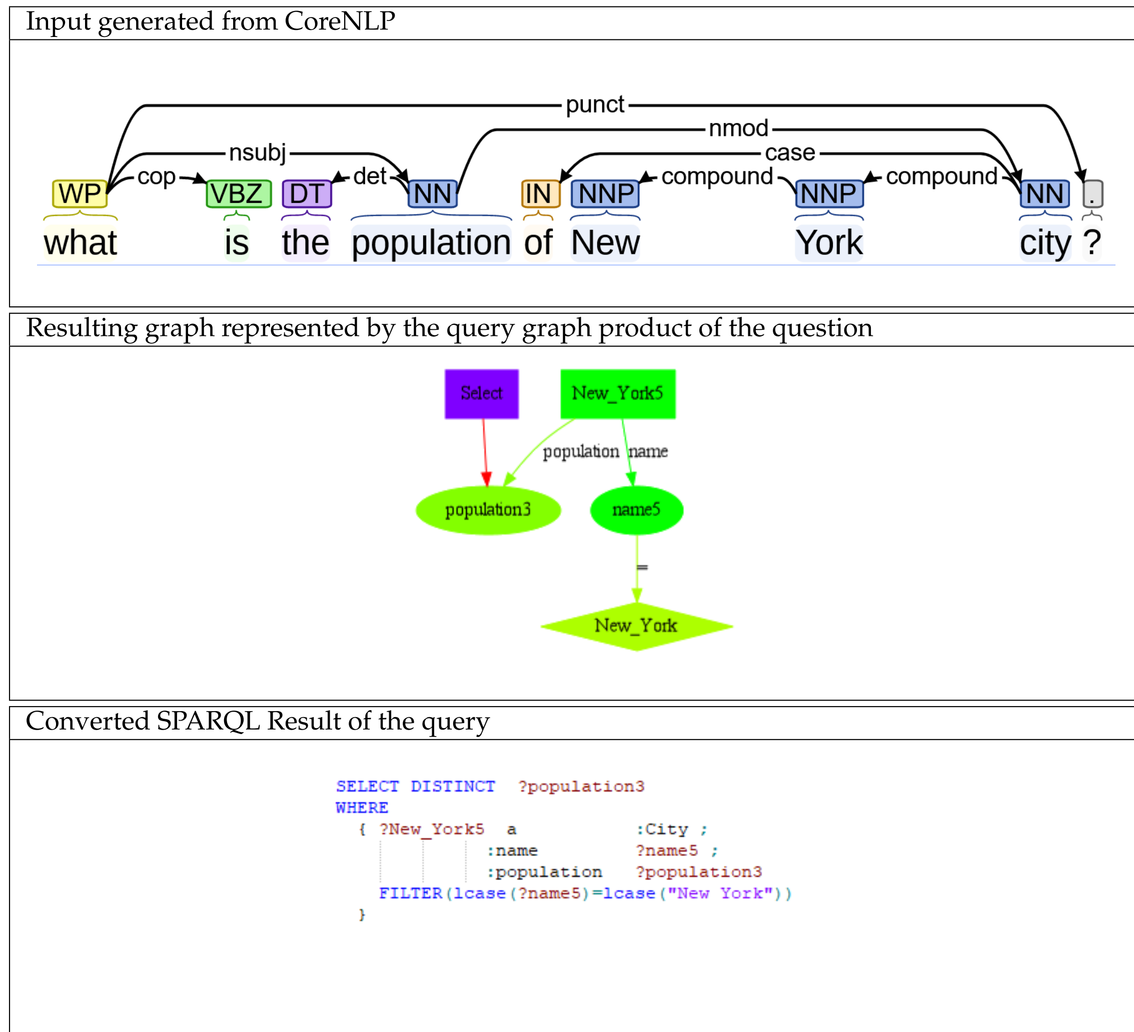


Figure 4.13: Question *Geobase68* with CoreNLP anntotations, query graph, and resulting SPARQL query

4.1.15 CompoundClassVariable product

Space: *CCV*

Subclass Of: compound product, ClassVariable product, (positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
cv_1	*CV*	0
cv_2	*CV*	1
$operator$	*OP*	2

CompoundClassVariable products are ClassVariable products used for the calculation of composite values. These values can be given either directly in the query in the form of a calculation or can be taken from previously learned concepts. An example of this would be *density* understood as a numeric value divided by the value of a (sub)property of *area*. A CompoundClassVariable product contains an ordered set O consisting of two ClassVariable products or further CompoundClass-Variable products as well as one operator product.

Converting into SPARQL:

Conditions	SPARQL Translation
true	BIND (cv_1.*toSPARQL*() + " " + *operator*.*toSPARQL*() + " " + cv_2.*toSPARQL*() + " AS " + $content.getSparqlName()$)

4.1.16 Property product

Space: *PROP*

Subclass Of: TriplePart product, (positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
$names$	$\mathcal{P}(\mathcal{PROP})$	-

A property is the representation of the corresponding predicate in a SPARQL query. However, a property product in the EvolNLQ context often represents a union of individual properties, therefore a set of properties $names \subseteq Property(MD)$ is defined instead of a single name. This is necessary since during processing it is often not clear whether such a product refers to exactly one property or to several ones. For instance, when considering properties between parents and children, *hasDaughter* or *hasSon* are possible links.

Additional or overridden Function(s):

Name	Signature	Mapping
getSPARQLName	*PROP* $\rightarrow$ string	$p \mapsto$ ":" $+ names[0] + position_{min}$

Converting into SPARQL:

Conditions	SPARQL Translation
$\lvert names\rvert = 1$	*getSPARQLName()*
$\lvert names\rvert > 1$	"?" $+ names[0] + position_{min} +$ "_PropertyValues" add to surrounding query: "VALUES ?" $+ names[0] + position_{min} +$ "_PropertyValues" $+$ "{" $+ names +$ "}"

4.1.17 Constant product

Space: *CONST*

Subclass Of: TriplePart product, (positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
$value$	$\bigcup_{DT \in \mathcal{LIT}} DT$	-

In addition to implicit variables ranging over some class of the application domain, constant values can of course also occur in a query. The RDF world allows to handle all literal datatypes that are defined in XML Schema [82], including strings, numerical values, dates etc. They are handled by constant products. A constant product defines a field $value$ in which its value is stored, usually in its XML string representation.

Additional or overridden Function(s):

Name	Signature	Mapping
getSPARQLName	*CONST* $\rightarrow$ string	$const \mapsto value$

4.1.18 Operator product

Space: *OP*

Subclass Of: TriplePart product, (positionable product)

Addition(s) to slots:

Name	Range	Comp. Pos.
$symbol$	$\{<, \leq, =, \neq, \geq, >\}$	-

Operators are the counterparts of ordinary math comparison operators. They are used exclusively in conditions. They have an additional $symbol$ field in their tuple, which determines which operator

they are representing.

Additional or overridden Function(s):

Name	Signature	Mapping
getSPARQLName	$OP \rightarrow$ string	$op \mapsto symbol$

4.1.19 Aggregation product

Space: AGG

Subclass Of: ClassVariable product, (TriplePart product), (positionable product), subSPARQLing product

Addition(s) to slots:

Name	Range	Comp. Pos.
type	{Count, Sum, Min, Max, Avg, Max Count, Min Count}	-
content	CV	0

The aggregation product has a relatively simple product structure. It is a subclass of ClassVariable product, and it acts as a modification of an existing ClassVariable product v (which is listed by the slot $content$). Indirectly it is assumed that the only relevant value for the query result of v is the now aggregated value. Note that by this, an aggregation product is *not* a compound product that would contain its own graph, but that it is just a component of the surrounding query graph product, as illustrated in Example 10.

For SPARQL or any relational-algebra-based query language, however, this viewpoint is not directly appplicable. Therefore, significantly more effort has to be put into translating this structure into a SPARQL query. An aggregation in a query algebra always requires a grouping of the other values. For example, it would not be possible to compute the sum of the population of all the states of the USA and output the name of *one* of the states, because the sum is formed from the group, but the name belongs to an individual.

In order for the aggregation function to still be used along with all other variables, it must be created in a subquery and then set equal to a variable of the outer query. However, it is generally not sufficient to drag only the literal and the related object-valued variable into this subquery, since the other statements may by all means contain further constraints on the object-valued variable.

Additional or overridden Function(s):

Name	Signature	Mapping
getSelection	$AGG \rightarrow TP$	$agg \mapsto type +$ "(" $+ content +$ ")" AS $getSparqlName()$

Converting into SPARQL:

Conditions	SPARQL Translation
true	add to surrounding query subquery: $createSubQuery().setSelection(getSelection())$ and condition($content.getSparqlName() = getSparqlName()$)

4.1.20 Group by product

Space: *GRP*

Subclass Of: SPARQLing product, positionable product

Addition(s) to slots:

Name	Range	Comp. Pos.
$aggregation$	AGG	0
$elements$	$\mathcal{P}(CV)$	1

If an aggregation also requires grouping, this information is stored in a Group By product. It stores the corresponding aggregation and all ClassVariable products that act as grouping variables.

Additional or overridden Function(s):

Name	Signature	Mapping
getSPARQLName	*GRP* → string	*op* ↦ Add to surrounding query: "GROUP BY " + comma-separated list of $elements[1..n].getSPARQLName()$

4.1.20.1 Product merging and differentiation

If several nodes, especially nodes of different types, create products (usually, ClassVariable products, that –at the end– describe the same entity, these products may be different. Therefore, a way must be found to decide how and which ones to unify, since they describe the same entity, and which ones to keep separate, since they describe different entities.

The decision whether and which products describe the same entity is essential for answering the queries. If it can be ruled out that two products describe the same entity, a distinctness product is created, which prevents further attempts to merge these products within a query graph product. On the other hand, if it is clear that two products describe the same entity, their information is intersected and turned into one product. To ensure that later products describing the same entity can be converted directly into the unified product, a sameAs product is then added to the query graph product.

Example 14 *To illustrate the use cases of sameAs product and distinctness product consider Geobase25 (see Example 17) and Geobase68 (see Example 13).*

On the one hand, in Geobase 25 the words "Colorado" and "river" appear. "Colorado" could be a state or a river in the ontology. Therefore the "river" could be a specification of "Colorado" or, which is actually the case, a separate entity. In this case a distinctness product prevents the agent from merging those two entities into one. On the other hand in Geobase 68 appear the words "New York" and "City". "New York" can be a State or city in the ontology (the city New York is stored as New York, not New York City) but this time the "City" is indeed a specification of "New York" and describes the same entity, therefore a sameAs product should be part of the result set to ensure that there are no relations between "New York" and "City" and they are treated as the same entity.

Therefore the agent has to figure out somehow, which one is actually the case. This can be done by different nodes and derived from different sources. Therefore those two products are needed to carry the information to other nodes.

4.1.21 Auxiliary product

Space: AUX

Subclass Of: product

Auxiliary products are products which transfer additional information specifically for other nodes, but not for the result of the agent. Typical examples are information like the disjunction of two products or the equality of them, but also things like possible completions of the graph or a relaxation of the merging rules for specific products. This abstract class is named only for documentation purposes. It does not actually contribute any properties or methods.

4.1.22 Projection product

Space: $PROJ$

Subclass Of: compound product, SPARQLing product

Addition(s) to slots:

Name	Range	Comp. Pos.
$content$	*TP*	0
$rename$	$string$	-

While all TriplePart products and compound products become part of the query, the response usually does not consist of all variables involved, let alone the constants. For the conversion into a SPARQL query, projection products are used to specify which variables go into the SELECT part of the SPARQL query. Each projection product contains a TriplePart product that specifies the part to be selected for the output, and, if applicable, its renaming.

Converting into SPARQL:

Conditions	SPARQL Translation
$rename \neq \emptyset$	Add to Select: $component.getSPARQLName() + "\text{ AS }" + rename$
$rename = \emptyset$	Add to Select: $component.getSPARQLName()$

4.1.22.1 Comparison and Equality of Products

Many TriplePart products can be created in different ways and to avoid duplicates they should always be merged. Testing for absolute data equality does not paint the whole picture, because on the one hand the name generation depends strongly on the origin of the product and the same products may have been named differently depending on the source. On the other hand, TriplePart products that describe the same thing can have different information. Therefore, it is not a trivial task to decide, in which cases two TriplePart products describe the same entity and in which not. Since this cannot be done in a generally valid way the agent has to learn in which cases which method is correct. Which means it cannot be done in every node but only in special nodes or with certain products to give the control to the learning algorithm. Collector nodes, however, first use a safe method to determine equality and only match ClassVariable products/property products with the same name, the same position and the same domain/names.

4.1.23 Distinctness product

Space: *DIS*

Subclass Of: auxiliary product, compound product

Distinctness products are necessary to express evidence of two products not being equal. The distinctness product itself is a compound product that contains an unordered set of products, which are assumed to be distinct, i.e., may never be equated or merged. distinctness products are kept in a query graph product, as they still need to be checked form other nodes. For an example and a comparison with it counterpart sameAs product see Example 14.

4.1.24 SameAs product

Space: *SAME*

Subclass Of: auxiliary product, compound product

SameAs products are necessary to express evidence of two TriplePart products being equal. The sameAs product itself is just an ordered set of products, which are always replaced by the first added product of the set. Basically it does not matter which of the TriplePart products replaces

all others, since they are all considered the same. In case a compound product c has two sameAs products $s_1, s_2 \in components(c)$ such that $components(s_1) \cap components(s_2) \neq \emptyset$, the components of both sameAs products are merged and one is removed from c. In all other cases sameAs products are kept in a query graph product, as they still need to be checked for new products and replacement them with their counterpart if necessary.

For an example and a comparison with its counterpart distinctness product see Figure 14.

4.1.25 Looking-for-replacement product

Space: LFR

Subclass Of: compound product, auxiliary product

Addition(s) to slots:

Name	Range	Comp. Pos.
toReplace	*TP*	0
context	*TR*	1

Often products must be inferred since they cannot be found directly in the input. In some cases, however, there are references elsewhere that may not be available to the node, or other nodes may find the same content but name it differently or have a wider or narrower range of values. To communicate this uncertainty to other nodes and prioritise the replacement of these products, there is the possibility to use a Looking-for-replacement product. Depending on the node, different strategies beyond equality can be used to replace this value with something else. A Looking-for-replacement product contains the TriplePart product which should be *replaced* and a compound product as the *context* in which the TriplePart product is considered uncertain. The context is given to help nodes in solving the merging problem since those uncertainties are usually based on derived values which are necessary for the context triple product.

E.g. in the Geobase 68 query (see Example 13), a replacement can enforce the convention, that the Variable is named *New_York5* and not *City6*.

4.1.26 Grammatical relationship products

Space: $GRAP$

Subclass Of: *CPROD*, *AUX*

Addition(s) to slots:

Name	Range	Comp. Pos.
gov (*governing*)	*POS*	0
dep (*dependent*)	*POS*	1
type	GR_{Tag}	-
subtype	GR_{Tag}	-

Auxiliary product to describe a grammatical relationship (cf. Section 2.1.3.2) between two positionable products. Similar to the NLPData product type, this product is also the direct implementation of the CoreNLP annotations into a product.

Additional or overridden Function(s):

Name	Signature	Mapping
relation	*POS* × *POS*	$p_1 \times p_2 \mapsto$ if $p_1.in(gov) \wedge p_2.in(dep)$ then *true* else *false*

4.1.27 Path network collection product

Space: *PColl*

Subclass Of: auxiliary product

Addition(s) to slots:

Name	Range	Comp. Pos.
$paths$	$\mathcal{P}(\mathcal{P}(TR))$	-

This is an example for an auxiliary, intermediate product that its tailored to a very specific problem: If a query graph product is disconnected, an attempt can be made to establish the missing links using the ontology and path algorithms. For this, path network collection products will be used.

4.2 Nodes

The agents itself consist of nodes. The nodes can be partitioned in different types of nodes depending on their input/output behavior:

- Input nodes: nodes that read the external input; in the EvolNLQ case, from CoreNLP (the MD and WordNet do not count as input to the process in that sense because they are no data flow, but access to persistent data sources).
- Generator nodes are all nodes that generate products of a different class than their input class. Unlike the input nodes, however, they do not use the CoreNLP output for this purpose.
- Processing nodes *modify* their input products and forward them.

Each node maintains a storage that assigns sets $storage(1), \ldots, storage(n)$ to each if its n input conduits. If not specified otherwise, whenever a node receives input y_i at some input conduit in_i,

it processes all possible n-combinations

$$(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, x_n) \text{ such that } \quad x_i \in storage(i)$$

(keeping the storage, and adding then new y_i to $storage(i)$) and produces output products. This incremental processing guarantees that in absence of control flow, all relevant products are produced during time.

A node does not only have functional behavior, but can create multiple outputs from an input that are then available at conduits $out_1, \ldots, out_n$ as described in Section 3.3.2.

In the following, for every node type,

- its parameter(s) (which are set when the node is instantiated),
- its input conduits (note that there can be multiple input conduits that have the same type).
- its formal operation on $(in_1, \ldots, in_n)$, generating zero to many products for each combination of products read from the input conduits. For the product constructors, the syntax $prodclass([name_1 := value_1, \ldots, name = n := value_n])$ is used; for positionable products, if $p_{min} = p_{max}$, then only p_{min} is listed, leading to e.g. *CONST*$([value := 3.1415, p_{min} = 3])$ for a constant product that represents π and occurs in pseudocount position 3 of the input question.

  The formal operation focuses on the informational aspects. Additionally, every new product is assigned with a *confidence value* that is usually created from the confidence values of the input nodes. Often it is the average confidence of the input products. Input nodes create products with confidence 1 - these are doubtless. The exact values for products that are generated by nodes that implement some *guessing* are usually between 0.3 and 0.6, the details are a matter of taste during experimenting with it.
- its output conduits, and which products are allocated at which output conduit. The mnemonic names are those that are used in the formal operation. As for regular expressions, the signature is annotated with symbols x, $x?$, x^*, x^+ are used for denoting how many products are delivered when a successful computation occurred, i.e.,
  - if the node has only one input: when processing an input product, or
  - if the node has more than one input: when successfully pairing products from these inputs (for the recombinations with earlier inputs, recall Section 3.4 on page 55).

  If a product that is listed for being provided at an output conduit is not assigned a value (in case that the formal operation contains conditional parts), there is just *nothing* provided at that output conduit.

4.2.1 Input Nodes

4.2.1.1 Part Of Speech Node

This node type is the most basic one. It is the only "pure" input node type of this application, meaning the only node type that does not need any other input to start its operation. An agent without it, therefore, cannot start any meaningful calculation.

This node activates once at the beginning of a run and searches for all words in the CoreNLP output that are annotated with a tag matching to its parameter and creates corresponding products from them. These are then forwarded to other nodes. Since it has no input conduits of its own as an input node, this node is not activated again after the initial activation.

Parameter:	a Part Of Speech tag p_{tag}
Input Conduits:	1. none
Formal Operation:	$P_{tag}^{+} \to \mathcal{P}\,(NLPD)$ $(q_1, \ldots, q_n) \mapsto out := \{nlpdata([$ $word := q_i.word,$ $type := q_i.tag,$ $p_{min} := q_i.position,$ $p.entityType := q_i.entityType,$ $p.normalizedEntityName := q_i.normalizedEntityName])$ $\mid q_i.tag = p_{tag}\}$
Output Conduits:	1. **NLPData product** - *out* - Product containing the word, its position and its POS-tag

4.2.1.2 Grammatical Filter Node

The grammatical filter node has the task of forwarding only pairs of positionable products that are in the grammatical relation that was passed to it as a parameter gt. For this, each input pair of positionable products is checked depending whether they stand in a given grammatical relation. The information about these relations is taken directly from the NLP Core annotations output (cf. Section 2.1.3.2).
If this is the case, a corresponding Grammatical Relationship is output.

Parameter:	Grammatical relationship tag gt
Input Conduits:	1. **Positionable product** - in_1 2. **Positionable product** - in_2
Formal Operation:	*POS* $\times$ *POS* $\rightarrow$ *GR* $\times$ *POS* $\times$ *POS* $(in_1, in_2) \mapsto$ if there is a $gr \in NC_{Gr}$ such that $gt.type = gr.type \wedge$ $gt.gov.in(in_1) \wedge gt.dep.in(in_2)$ then $grp := GR([type := gt, gov := in_1, dep := in_2]), gov := in_1$ and $dep := in_2$ else if there is a $gr \in NC_{Gr}$ such that $gt.type = gr.type \wedge$ $gt.gov.in(in_2) \wedge gt.dep.in(in_1)$ then $grp := GR([type := gt, gov := in_2, dep := in_1]), gov := in_2$ and $dep := in_1$ else no output is generated.
Output Conduits:	1. **Grammatical Relationship**(gt, gov, dep) - grp - (basically, the product instance that represents the $gr \in NC_{Gr}$) 2. **Positionable product** - gov - governor part 3. **Positionable product** - dep - dependent part

Analysis: such output could be used to *combine* the query incrementally from incoming relationships. Another strategy would be that some node gets candidate pairs (w_1, w_2) from previous processing, and just does a lookup for the grammatical relationships between them. This shows that multiple functionality is *offered* to the learning process that can then choose what to use.

4.2.2 Nodes that Create the Representation of Basic Sentence Structures

Many atomic parts of the resulting query (=query pattern graph) can rather directly be derived from the text of the question, e.g. a ClassVariable product ranging over countries, a "(has) capital" property, or, given a proper name like "Berlin", a ClassVariable product ranging over cities whose binding must have a property "name" with a value (represented by a constant product) representing the string literal "Berlin". They are generated by generator nodes. The choice of the provided generator nodes mirrors the human way to parse a sentence and to create a model for its meaning.

The generator nodes can be differentiated between two large groups; *schema-based generators* and the *individual-based generators*. The *schema-based generators* use primarily the Mapping Dictionary [15] in order to infer the necessary components for the answer of the question from the structure of the ontology. In particular, the class hierarchy as well as the domain and range of properties are used.

While properties are mostly found directly in the questions, classes often appear more indirectly via their properties or via their identifiers for certain individuals in the questions. The latter then have to be found and created by *individual-based generators*.

In an RDF database, each individual is assigned a unique URI, but the user will usually neither know it, nor wants to get it as an answer. Even though URIs are sometimes mnemonic, they are not as clearly understandable for humans as other properties of individuals, such as their name, abbreviation, or at least an ID number. In order for EvolNLQ to handle these names, an agent learns which properties make good identifiers. By default this list already contains "name", "ID", "identificator", "abbrev" and "abbreviation", but can be extended if needed. For these properties a separate table is created in the internal database (a sample is shown in Table 4.3) which contains the class, the name of the property and the value for each known individual. It should be noted that none of these columns can be unique, because often different individuals have the same names and the like. If there is more than one possible way to refer to an individual, this class has an entry for each property. So that, for example, the European Union could also be referred to as the EU and be understood what is meant. Conversely, when referencing an individual of a class in a result that has more than one identifier property, string-valued literals are preferred to numeric ones. If multiple string-valued identifiers exists, all of them are presented to the user, since the identification might not work with a single one in some cases, e.g. if a database has separated first and last names.

Class Name	Property Name	Property Value
State	stateNumber	1
State	number	1
⋮	⋮	⋮
City	name	Abilene
City	name	Abingdon
State	abbreviation	Ak
City	name	Akron
State	abbreviation	Al
State	name	Alabama
City	name	Alameda
State	name	Alaska
City	name	Albany
City	name	Albuquerque
City	name	Alexandria
City	name	Alhambra
River	name	Allegheny
City	name	Allentown
City	name	Altoona
Mountain	name	Alverstone
⋮	⋮	⋮

Table 4.3: Excerpt from the identifier table ID for the GeoBase Test Set, sorted by *Property Value*.

However, for most questions, both types of generators are necessary, since most questions are structured in a way that they either want to know something about classes in general and start

from an individual (e.g. Example 12 asks for connections of the individual named "New Mexico" to instances of the class *state* via the property *bordering*, and for connections of these instances of *state* to the instances of the class *river* via the property *run through* (which is not known in the ontology)), or ask for individuals of a certain class that have certain properties or characteristics (e.g. Example 16 asks which individual of the class *point* is located *in* something that is identified by "USA" and has the highest elevation amongst those – note that the Geobase ontology covers only the US, meaning everything in the ontology is automatically "in the USA" but there is no individual named *USA*).

4.2.2.1 Class Variable Generator Node

The *ClassVariable Generator Node* is the main source for ClassVariable products. The intention is that its inputs should be nouns. It compares the word as well as the lemma of the input NLPData product with all known classes in $\mathcal{CLS}$. For this purpose, the Mapping Dictionary is queried and then either in case of a match, the corresponding ClassVariable product is generated and sent through the first output, otherwise, the non-matched NLPData product is just forwarded through the second output (because then it could be a proper name).

Parameter:	none
Input Conduits:	1. **NLPData product** - in - a word to be checked whether it is the name of a class (in $\mathcal{CLS}$)
Formal Operation:	*NLPD* $\rightarrow$ *CV* $\cup$ *NLPD* $in \mapsto$ if there is an $md \in$ *MD* such that $in.word = md.class$ then $cv :=$ *CV*$([name := in.word,\ domains := \{in.word\},$ $p_{min} := in.p_{min}])$ else $nonm := in$
Output Conduits:	1. **ClassVariable product**? - cv - ClassVariable product of the class whose name is the input word 2. **NLPData product**? - $nonm$ - forwarded non-matched input data – note: either cv or $nonm$ is provided

4.2.2.2 Property Generator Node

The Property Generator Node is responsible for converting words that correspond to properties of the ontology into property products. For this purpose, the Mapping Dictionary is queried whether the input word is a property, and if so, a property product is created. Otherwise, nothing is output.

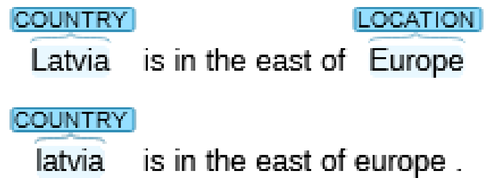


Figure 4.14: Example how the Entity Recognizer works for recognizing countries and continents, and the dependency of capitalization.

Parameter:	none
Input Conduits:	1. **NLPData product** - in - a word to be checked whether it is the name of a property (in $\mathcal{PROP}$).
Formal Operation:	$NLPD \rightarrow PROP$ $in \mapsto$ if there is an $md \in MD$ such that $in.word = md.property$ then $prop = PROP([name := md.property,$ $names := \{md.property\},\ p_{min} := in.p_{min}])$
Output Conduits:	1. **Property product**? - $prop$ property product of the property that is mentioned in the input word

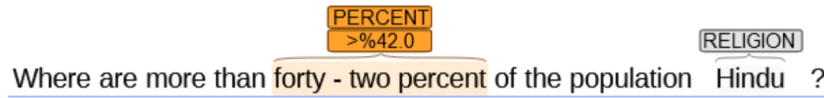
Handling Concrete Entities and Constants

Concrete entities and constant values and can also be found in questions. In general NLP, *constants* mean both *entities* such as *Germany* that are named in the text (by using their name, e.g., "Germany"), and literal-valued constants such as numbers or dates. In CoreNLP, this functionality is both included as *entity recognition*.

However, the recognition of proper *entities* is restricted to very common entity types, like countries, locations, and well-known persons etc. These are detected very reliably if names are correctly capitalizated (as shown in Figure 4.14).

The intention of EvolNLQ is that it should work for questions against arbitrary, often very specialized databases. There, this entity recognition will probably not work in general. Furthermore, the *operations* (i.e, the node functionality) of EvolNLQ are not domain-specific, i.e., they cannot (and should not) profit from knowing that something is e.g. a country. Thus, in this context, such names are usually identifying values – that are contained as values of properties in the database, that can be used to *identify* the corresponding objects.

Figure 4.15 shows that another example of entity recognition, and that is also detects percentage values.



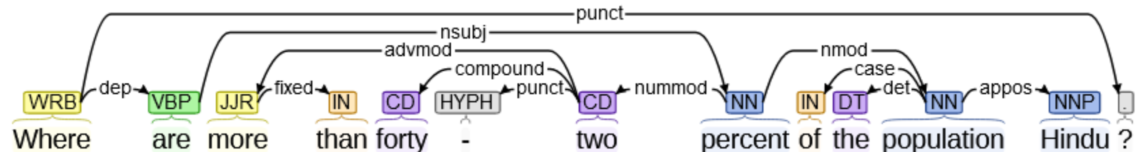


Figure 4.15: Examples for the Entity Recognition from CoreNLP for numbers and dates.

With numbers and dates it is already a little bit more unreliable, these are usually recognized, but for example everything that is a four-digit number in the one-thousands or two-thousands interpreted as a year number.

As shown in Figure 4.16 on the left, for the question whether there are more than 200 dog species, it is still recognized correctly that it is about a >200, while the same question turns into a date when used with the number 2000. Therefore it was decided for EvolNLQ to make such four-digit "data" entities also normal numbers. However, the recognition of dates that are specified more precisely is still very helpful; as also shown in Figure 4.16 on the right, the dates are even already converted into the format that is required by XML Schema [82].

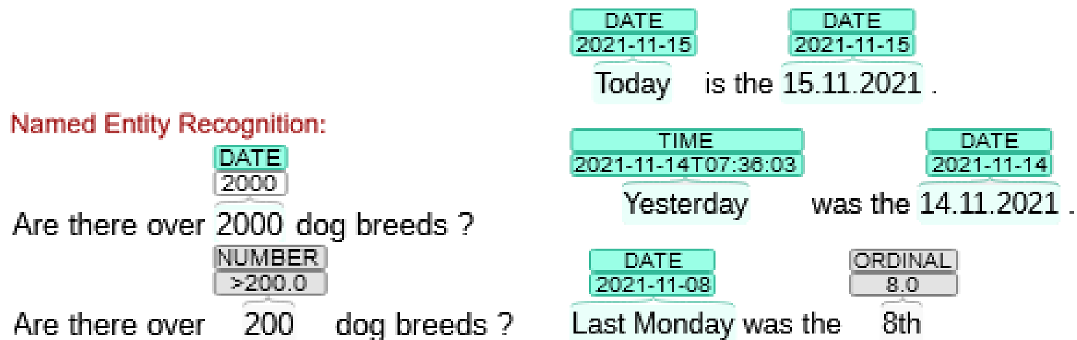


Figure 4.16: Examples for the Entity Recognition from CoreNLP for numbers and dates.

4.2.2.3 Constant Generator Node

The constant generator node uses this named entity output of CoreNLP to check whether an incoming NLPData product can be a numeric constant and creates a corresponding constant product.

(Note: CoreNLP annotates words as "City", "Country", "Date", "Time", etc. where considering the database community's terminology, only City and Country are considered to be *entity types* while Date and Time are called *literal data types* — the latter are actually relevant here:

Parameter:	ET - Accepted entity tags from CoreNLP, to avoid domain dependency it is restricted to *number*, *date* and *percent*, but could be extended
Input Conduits:	1. **NLPData product** - in - word which might be a number
Formal Operation:	*NLPD* $\rightarrow$ *CONST* $nd \mapsto$ if $in.entityType = ET$ then lit := *CONST*([*value* := *in.normalizedNamedEntity*, $p_{min} := in.p_{min}$]) where *normalizedNamedEntity* is the string representation of the value according to XML Schema [82] (which is delivered by CoreNLP)
Output Conduits:	1. **Constant product**?- lit - constant with the value of the input

4.2.2.4 Identifier Generator Node

The identifier node has the task of finding individuals named in the query by their name or, more precisely, named by a literal value of one of their properties. The identifiers if objects known to the system are stored in the Identifier Table ID (cf. Table 4.3). Usually this property is the name or an abbreviation. Which properties are concretely used is learnt from the training data.
The node as two different strategies to identify: perfect and partial matching. In the perfect matching mode, the input word must be exactly the same as the identifier stored in the ID relation of the database. In the partial matching, the input only has to be a word (subsequence) of such an identifier. To avoid too many false results for very short words, the node only considers whole words. If "both" is used, the partial strategy is only executed if the perfect matching does not have any results.

Then, the necessary products are created such that the individual can be connected to its identifying property, and provides them at the output conduits. This is specified below in the formal operation, and illustrated by the following example:

Example 15 *Consider that the word of the input NLPData product is "New York" (ProperName nodes as described later can generate such multi-word NLPData product products). Then, it uses the ID entry (City,name,"New York") and creates the following:*

- *a ClassVariable ranging over the classes of the found individual, here* $cv := CV([name = \text{"New York"}, domains := \{City\}])$,
- *a property product* $prop$ *for* $id.property$, *i.e., for "name",*

- *another IdentifierClassVariable product (with literal-valued domain)* $cv_{lit} := CV([name = "name", domains := \{string\}])$
- *and a triple product* $triple := TR(cv, prop, cv_{lit})$ *(that, as a compound product,* contains *these products).*
- *a constant product* ny *that represents the string "New York",*
- *a comparison product* $comp := COMP(cv_{lit}, =, ny)$ *(that contains* cv_{lit} *and* ny *– recall that always copies are provided at and sent via the output conduits)*

All these together just represent the SPARQL query pattern (cv name cvlit, cvlit = "New York").

Parameter:	Matching type: Perfect | Partial | Both
Input Conduits:	1. **NLPData product** - in – a word which might be the name of an individual
Formal Operation:	$in \rightarrow COMP \times CV \times PROP \times TR$ $NLPD \mapsto$ if there is an $id \in ID$ such that $in.word = id.value$ then $pos := [p_{min} := in.p_{min}, p_{max} := in.p_{max}]$, $cv := CV([name := in.word, domains := \{id.class\}, pos])$, $prop := PROP([name := id.property, names := \{id.property\}, pos])$, $cv_{lit} := IDCV([name := id.property, domains := \{range\}, pos])$, where $range$ is the range of $id.property$ when applied to $id.class$ which is looked up in the Mapping Dictionary, $triple := TR([subject := cv, predicate := prop, object := cv_{lit}])$, $comp := COMP([left := cv_{lit}, operator := "=",$ $right := CONST([value := in.value, pos]), pos])$
Output Conduits:	1. **Comparison product**? - $comp$ - comparison product which connects the constant product with the identifying literal 2. **ClassVariable product**? - cv - ClassVariable product of the individual 3. **Property product**? - $prop$ -property product literal property which identifies the individual 4. **Triple product**? - $triple$ - contains the individual as subject, the identifying property product as predicate and the literal as object – note: either none or exactly one output for each of them is provided.

4.2.2.5 Proper Name/Property Node

Since generator nodes analyze only a single product, but names or properties can consist of several words, or even their individual components have a different meaning than their combination (e.g. Salt Lake City, describes a specific city and not lakes and cities in general) it is necessary that multi-word NLPData products can be generated. Proper Name nodes start with a given word, check if this word is part of any known identifier, then extend the word with the word before respectively the word after the input (accessing the CoreNLP data). This is done until neither the addition of any previous words nor following words is contained in any identifier.
Note that A perfect match, meaning an identifier is exactly equal to the combination of words considered at that time would not be a suffcient termination criterion, considering the possibility that one identifier might be part of a larger one. E.g. "Sea of Japan" contains Japan, but does not mean the country or seas in general. On the other hand, if the algorithm terminates and the result does not contain a perfect matching, that might still result in a correct identification therefore it is considered a termination criterion. E.g. "Pacific" would be not perfectly matched with "Pacific Ocean" but has to be considered sufficient, as it is usually called so.

Parameter:	$PN \leftarrow \mathcal{P}\,(property(ID))$ if searching for proper names of entities $PN \leftarrow \mathcal{PROP}$ if searching for a property name
Input Conduits:	1. **NLPData product** - in - word which might be part of a proper name
Formal Operation:	*NLPD* $\rightarrow$ *NLPD* $in \mapsto$ $(ext, min, max) :=$ ProperName$(NC, in.p_{min}, in.p_{max})$, if $ext \neq$ "" then $out :=$ *NLPD*$([word := ext,\ p_{min} := min,\ p_{max} := max])$, For the ProperName algorithm (that uses PN) see Algorithm 3.
Output Conduits:	1. **NLPData product**? - out - whole proper name

4.2.2.6 Get Identifier Node

When the question asks for a specific individual that meets certain criteria, like the biggest city for example, an identifying property value (e.g., its name) is expected and not a URI. This is the responsibility of the Get Identifier Nodes. They use the Identifier Table to find the names of the identifying properties of a class (e.g., name, abbreviation, ID), and then do a lookup in the MD for the ranges of these properties (each range must be a literal datatype). An example for this can be found in Example 12, the node is responsible for creating the property *name* and the literal *?name1* for *?State1*, such that this can be selected by the projection product.

Algorithm 3: Proper Name Algorithm *ProperName*

Result: Longest possible matching word sequence
Input: text $T = (t_1, \ldots, t_n)$, list PN of proper names, $0 \leq n_1 \leq n_2 \leq n \in \mathbb{N}$
$w \leftarrow t_{n_1}, \ldots, t_{n_2}$
if $\exists id \in PN$ *such that* w *is a substring of* id **then**
    $(p, m_1, m_2) \leftarrow ProperName(T, n_1 - 1, n_2)$
    if $p =$ "" **then**
        $(p, m_1, m_2) \leftarrow ProperName(T, n_1, n_2 + 1)$
    end
    if $p =$ "" **then**
        return (w, n_1, n_2) // not extensible
    else
        return (p, m_1, m_2) // a subcall returned a longer one
    end
end
return ("",-1,-1)

Parameter:	All identifiers or Average longest identifier
Input Conduits:	1. **ClassVariable product** - in - which needs an identifer
Formal Operation:	$CV \to \mathcal{P}(TR)$ $in \mapsto$ $out := \{TR([subject := in, predicate := id.property,$ $object := CV([name := id.property,$ $domains := \{md.range \mid \exists md \in MD : md.class \in in.domains \wedge$ $md.property = id.property\},\ p_{min} = in.p_{min},\ p_{max} := in.p_{max}])])\}$ (note: $md.range$ is a single literal datatype)
Output Conduits:	1. **Triple product***- out - setting the input in relations with its possible identifying literals

4.2.2.7 Any Generator

An Any Generator is a node that has a set of keywords as parameters. If one of these keywords is in its input, the Any Generator creates a ClassVariable product which has as domain all classes from the Mapping Dictionary. The set K of keywords typically consists of the namesake "any", "everything" or "all". Mutations can add random words that do not occur in the Mapping Dictionary to the set of keywords K or remove them from K. This node does not make any further checks whether the bespoken term could really be of any class, but relies on the fact that via confidence, filter and further processing the wrong any-class is filtered out or further limited by restricting its domain.

Parameter:	Node-specific set K of keywords
Input Conduits:	1. **NLPData product** - in - which word might be in K
Formal Operation:	*NLPD* $\rightarrow$ *CV* $in \mapsto$ if $in.word \in K$ then $cv := CV([name :=$ "ANY", $domains := \mathcal{CLS},\ p_{min} := in.p_{min},$ $p_{max}; = in.p_{max}])$
Output Conduits:	1. **ClassVariable product**? - cv - which ranges over all known classes

4.2.3 Nodes that Introduce Structural Components of the Query by Keywords

There is a number of nodes that uses obvious keyword in the question to create structural components of the query such as comparison operators, negation, aggregations, and the final projections for the output.

4.2.3.1 Operator Generator

Operators $=, \neq, <, \leq, >$ and $\geq$ can usually be derived directly from the question very easily and domain-independently. Therefore, the operator generator uses a lists of keywords for each of the these operators. The lists are initially already filled with common keywords, since they depend on the language and not on the application domain, but can still be individually extended by each node with additional learned words (by parameter mutation).

Parameter:	K - set of pairs from words to operators $k := \{word, symbol\}$
Input Conduits:	1. **NLPData product** - in
Formal Operation:	*NLPD* $\rightarrow$ *OP* $in \mapsto$ if there is a $k \in K : k.word = in.word$ then $op := OP(symbol := k.symbol, p_{min} := in.p_min, p_{max} := p_{max})$
Output Conduits:	1. **Operator product**? - op

4.2.3.2 Except Node

The Except Node is a keyword node and accordingly has a list of words. If the input product is sich a word, the the node creates an except product with an interval from the p_{min} of the input product up to the end of the sentence.

Typical keywords for this node are "not", "without" and of course "except".

Parameter:	Node-specific set K of keywords
Input Conduits:	1. **NLPData product** - in - which word might be in K
Formal Operation:	*NLPD* $\rightarrow$ *EXC* $in \mapsto$ if $in.word \in K$ then $exc :=$ *EXC*$([p_{min} := in.p_{min}])$
Output Conduits:	1. **Except product**? - exc - from the position of the input to the end of the sentence

An example that uses negation was already given in Example 9, depicted in Figure 4.6 (page 93).

4.2.3.3 Aggregation Node

The aggregation node is a keyword node that creates aggregation functions, i.e., *count*, *sum*, *minimum*, *maximum* and *average*. Each aggregation node is only responsible for a specific type of aggregation, where is holds a list of keywords that are specific for its aggregation function type ty. The node is based on two input products: an NLPData product which acts as the keyword, and a ClassVariable product to apply the function to. This node relies on before filtering the provided input nodes, or on afterwards filtering by other nodes to determine plausibility, since it does no checks itself.

In addition to the above-mentioned aggregation functions, compound aggregations *maximum count* and *minimum count* are implemented. From the SPARQL point of view, a *maximum count* would be first a *count* function over a variable and then the *maximum* function that selects the tuple with the highest value, but in natural language those are normally expressed in a single word as a combination, therefore this nodes also is able to handled them as combined.

Depending on the ClassVariable product and the ty, it proceeds differently to create the exact aggregation product.

If the ClassVariable product ranges over a literal class, it is checked whether this is a numeric datatype; if not, then nothing further is done. Applying an aggregation function other than *count* to strings is usually not required in NLQs (in a program, something like $min(name)$ makes sense). If it is a numeric datatype, the aggregation product is created with the position of the NLPData product and contains the ClassVariable product as a modified TriplePart product. A typical use case is shown in Example 16.

If the ClassVariable product is not a literal, then a distinction must be made depending on whether the aggregation type of the node is *count* or not. In case of a *count*, the aggregation can be done directly via the ClassVariable product, similar to the literal variant. As illustrated in Example 17 the count is just applied to the ClassVariable product and the translation to SPARQL must handle the rest. Note that EvolNLQ always generates a subquery in those cases with all statements of the

main query. Even if not necessary in this example, in some cases it important to validate that all conditions of the main query are also fulfilled in the subquery (for more details see Section 4.1).

It becomes more difficult when a non-counting aggregation is executed on an object-valued ClassVariable product. This step needs knowledge about the instance as such to be answered correctly and is not even for humans always easy or unambiguous. For example, "largest state/city in the US" could mean asking for the state/city with the largest area, or the largest population. While in the case of the state, it is likely that the area is meant, in the case of the city, it is certainly the population that is meant. The decision, which of the numeric attributes is actually meant, is an issue of its own and would go beyond the scope of this work (e.g. discussed in [84]). Instead a property product is created whose names is the set of all numeric properties of ClassVariable product.

The property also gets a renaming to the word that created it, such as the union over "Population" and "Area" is then called "biggest". This then names the table column in the result so that the user can see according to which criterion the result was created.

Parameter:	Node-specific list K of keywords, associated with the node's aggregation type $ty \in \{Count, Sum, Min, Max, Avg, MaxCount, MinCount\}$
Input Conduits:	1. **NLPData product** - in_1 - which might be equal to k 2. **ClassVariable product** - in_2 which might be directly the target for an aggregation of type ty or can have property which can be target of ty
Formal Operation:	$NLPD \times CV \rightarrow AGG \times PROP \times CV \times LFR$ $(in_1 in_2) \mapsto$ if $in_1.word \in K$ then $pos := (p_{min} = min(n.p_{min}, cv.p_{min}), p_{max} = max(n.p_{max}, cv.p_{max}))$, if $cv.isLiteral()$ or $ty =$ "count" then $agg := AGG([belongsTo := cv,\ type := ty,\ pos])$ else $prop := PROP([name :=$ (take-first-of-names), $names := \{md.property \vert$ $md \in MD \wedge md.class \in cv.domains \wedge md.range =$ "number"$\}, pos])$, $var := CV([name := prop.name,\ domains := \{"number"\},\ pos])$, $tr := TR([subject := cv,\ property := prop,\ object := var])$, $agg := AGG([typ := ty,\ content := object,\ pos])$, $lfr := (LFR(tr, prop), LFR(tr, var))$ (note that this means that two LFR are provided at that output conduit)

<table>
<tr><td>Output Conduits:</td><td>
1. Aggregation product? - agg - of the type ty over cv
2. Property product? - $prop$ -Union of properties which could be target of ty (only if cv is object valued)
3. ClassVariable product? - var - which is subject of the ty operation (only if cv is object valued)
4. Triple product? - tr - relation between cv via $prop$ to var (only if cv is object valued)
5. Looking-for-replacement product? - lfr - information for other nodes that there might already be products that represent var or $property$ (only if cv is object valued)
- note that either nothing, or one aggregation product, or one output to (1)–(4) and two outputs for (5) are generated.
</td></tr>
</table>

Example 16 *Question Geobase195: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase195 are depicted in Figure 4.17. The ClassVariable product* cv *that is input to the "max" aggregation is applied is ?Point4. It is object-valued. Thus, EvolNLQ looks which numeric properties Points have, which in this case is only elevation, which is created as property* $Prop$ *and as literal-valued ClassVariable product* var $=?elevation3$ *over which then the aggregation Max3 is executed. The created triple is* $(?Point4, elevation, ?elevation3)$.

consistsconsists

4.2.3.4 Grammatical Aggregation Node

This node is a variant of the aggregation node that instead of taking a ClassVariable product and an NLPData product, takes a grammatical relationship product gr as input that expresses an aggregation. In Example 16, the relation "point" $\stackrel{\text{amod}}{\longrightarrow}$ "highest" is such an instance. The governing part is "point" (which is mapped into a ClassVariable product), the depending part is "highest".

If $gr.dep$ is an NLPData product that matches the aggregation keyword (e.g. "highest"), and the other component, i.e., $gr.gov$ is a ClassVariable product or a property product, then it creates a suitable aggregation. If the other component is a property product, then the ClassVariable product needed for the aggregation is created from the range of the property product. Starting with this, the same procedure is used as for Aggregation Node (see page 123).

Parameter:	List of grammatical relation types GR, Node-specific list K of keywords, associated with the node's aggregation type $ty \in \{Count, Sum, Min, Max, Avg, MaxCount, MinCount\}$
Input Conduits:	1. Grammatical Relationship - in Relation which might be considered an aggregation over the gov/dep

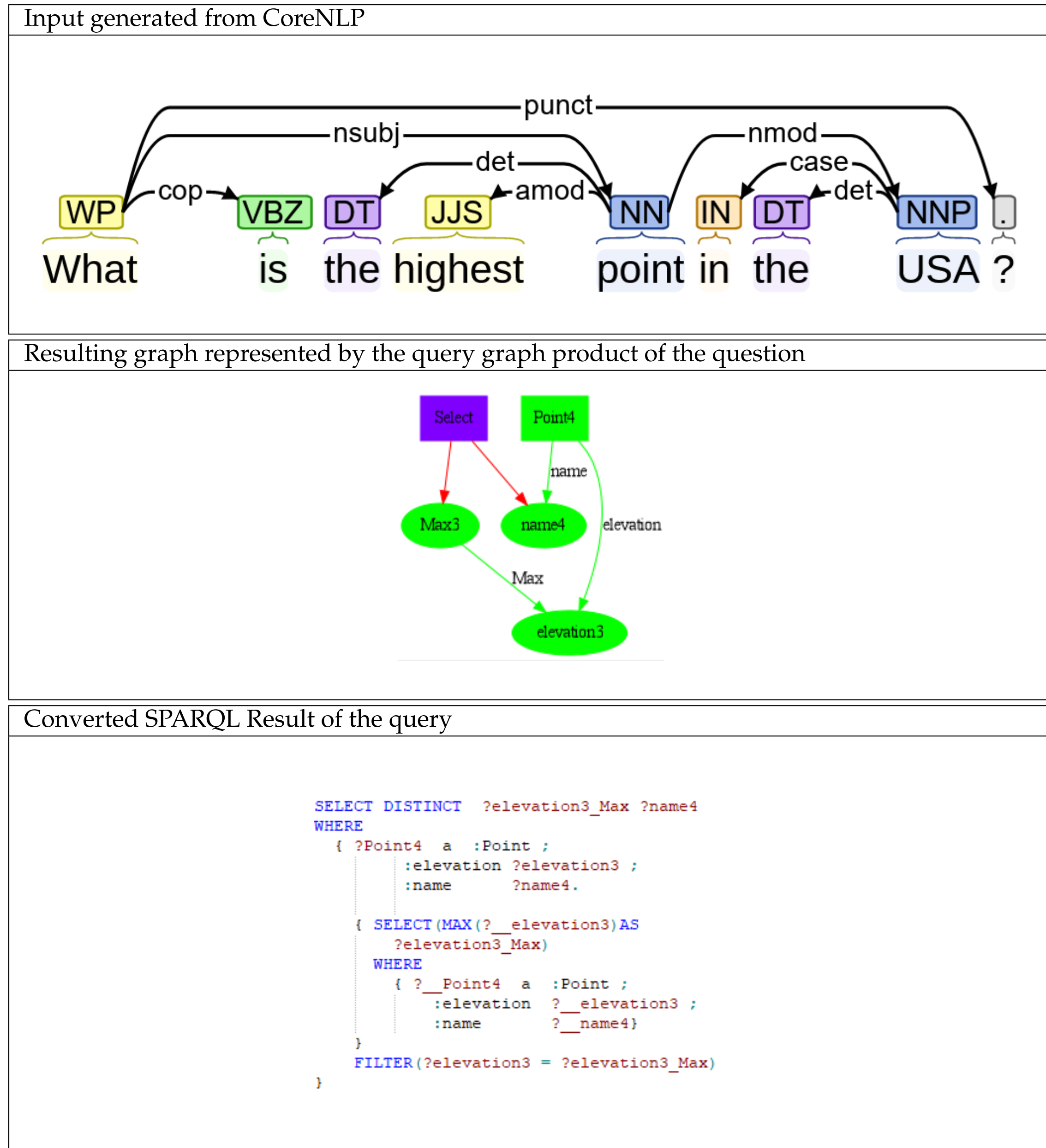
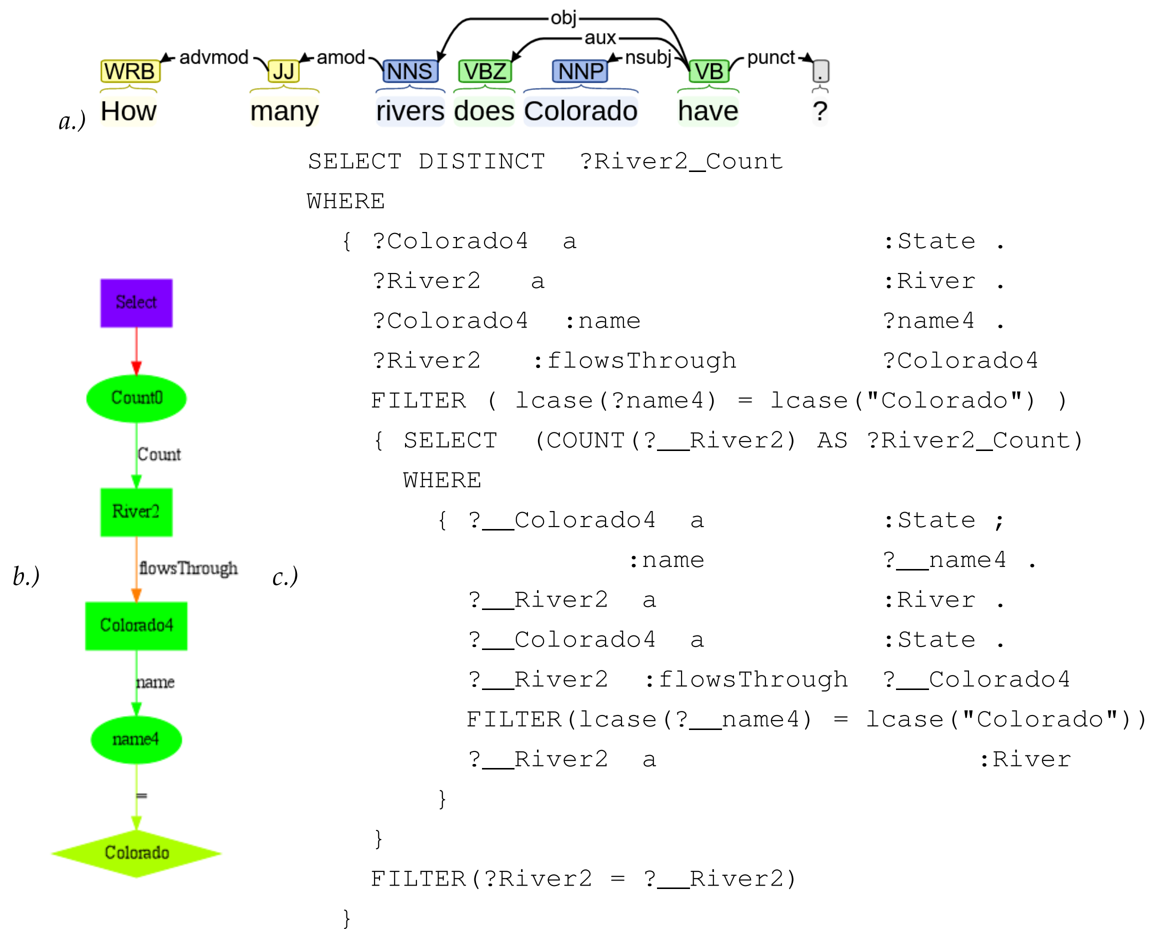


Figure 4.17: Question *Geobase195* with CoreNLP anntotations, query graph, and resulting SPARQL query

Formal Operation:	$GR \rightarrow AGG \times PROP \times CV \times TR \times LFR$ $in \mapsto$ if $in.type \in GR$ and $in.dep.word \in K$ then the same is done as in the aggregation node, with $in_1 := in.dep$ and $in_2 := in.gov$.
Output Conduits:	1. same as for the Aggregation Nodes

Example 17 *Geobase Query 25: a.) output of CoreNLP b.) query graph product of the Query c.) Converted SPARQL Result of the Query*



4.2.3.5 Projection Node

Not all variables involved in the query are ultimately destined to be part of the result set given to the user. In particular, variables that are already bound in the query do not have to be output again

with the already known value, for example in Example 17 there must be a ClassVariable product for "Colorado" but the result set should not contain it, since the questioner already has given this value in the input query and does not need it again.

For the creation, of the SPARQL query the projection products (see page 107) are used, these are created in the *Projection Node*. This node just inserts a received TriplePart product (on most cases, this is a ClassVariable product – which is also the intuitive case) into a projection product and outputs it. The filtering of which are suitable and which are not is done by the agent structure and other nodes.

There are multiple possibilities to do that – and the depending on how the learning process configures the agents:

- a-priori: some node(s) "before" learned to select which ClassVariable products should be in the answer (e.g. noun-subject), or
- a-posteriori: some node(s) "after" learned what projections to select, or how to reduce a set of proposed projections,
- or a mixture of both.

Example 18 *Consider the filter structure shown in Figure 4.18. Here an advanced agent uses various filters to limit the input for its selection nodes and priorities certain selection over other by checking for certain grammatical relations. Inputs of the selection node are filter for aggregations, literals from adjective based POS tags and identifiers generated by classes. After the selection is created, it is checked if there are any partners that have an "nsubj" relationship with the selected variable or an "adverb" modifier relationship. By filtering nodes with "Not Contain", nsubj selections are preferred, then adverb and only if neither of them can be found for a select, other selections are allowed.*

Note that in general SPARQL, a projection (output) cannot only be something that is represented here by a *ClassVariable product* (which ranges over some class), but an output variable in SPARQL can also be bound to a *property*. This is rarely needed in NLQ processing, but it must be considered here, so the input can be any TriplePart product which then also includes property products, and also constants (which are then simply output).

Parameter:	none
Input Conduits:	1. **TriplePart product** - in - which should be part of the selection of the query
Formal Operation:	$TP \rightarrow PROJ$ $in \mapsto$ if $tp \in CV \cup PROP$ then $out := PROJ([content := in])$
Output Conduits:	1. **Projection product**? - out - with in as the selected product

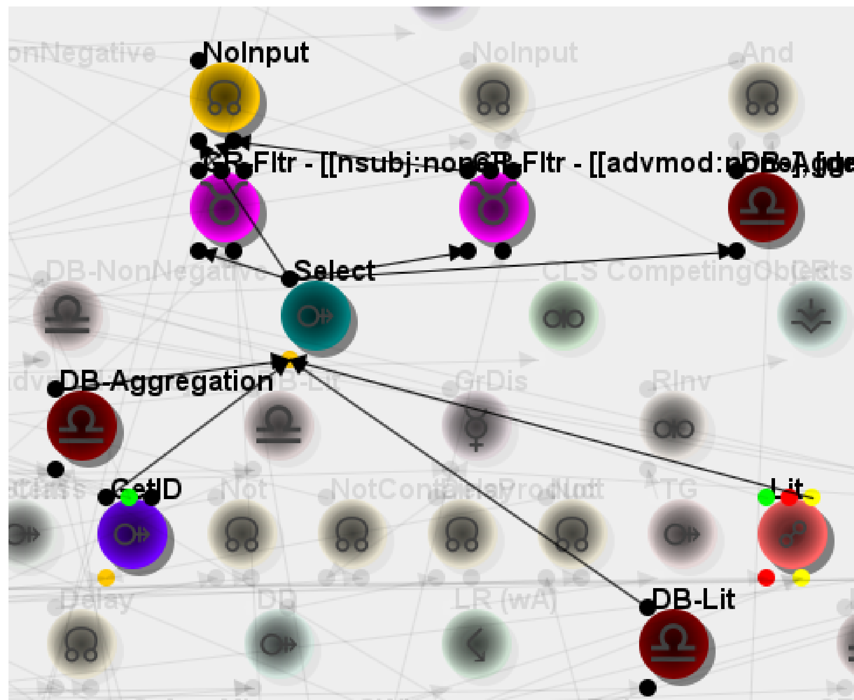


Figure 4.18: Example how a Select node is nested in filter nodes

4.2.4 Nodes that Introduce Statements or ClassVariables by Guessing

There are a number of node types that introduce rather large sets of statements by some kind of guessing. They contribute especially possibly implicit connections in the question.

4.2.4.1 Relator Node

The Relator node has two inputs and relates them in the form of a triple, if the ontology allows it. In that case, this will create the third missing part as concrete as the ontology allows. The idea behind this node is that natural language sentences often imply one of the parts that would be necessary to form a triple, because humans can infer it from the context (e.g., in "all cities located at a river and their country", the "located in" between the city and the country is missing). So this node creates every relation that is possible wrt. the ontology, regardless of the position of the words or other input-based criteria and relies on other nodes that later combine and filter the correct triples. For that, the confidence value assigned to its guesses is relatively low: the new TriplePart product gets a confidence of 0.5, and the other TriplePart products are unchanged, so the triple gets –maximally– a confidence $(0.5 + 1 + 1)/3 = 0.83$ (which is actually often reached, if the input components are "on the safe side").

Example 19 *For the geobase query 1 "Which rivers flow through states bordering New Mexico" (full question evaluation in Example 12), the relator node finds a set of possible connections and several new*

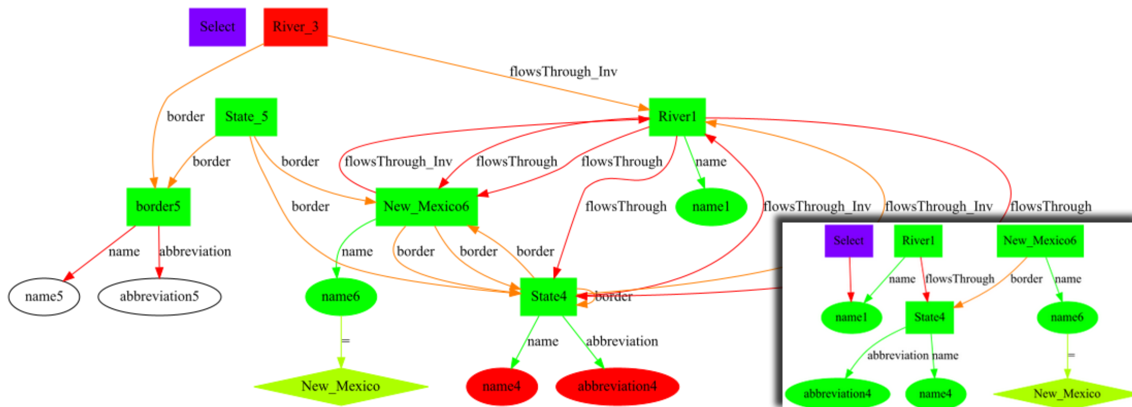


Figure 4.19: The intermediate result of an relator node (central) and the final query graph (bottom right) in comparison for 12

ClassVariable products shown in Figure 4.19. Of these, most are incorrect, but the node opens up possibilities for other nodes here to find the correct connections by exclusion.

Parameter:	"object": completes Subject-Predicate Relations, "predicate": Subject-Object Relations, both
Input Conduits:	1. **ClassVariable product** - in_1 - Subject 2. **TriplePart product** - in_2 -Predicate or Object
Formal Operation:	$CV \times TP \rightarrow TR$ $(in_1, in_2) \mapsto$ $pos := [p_{min} := in.p_{min}, p_{max} := in.p_{max}]$, if $in_2 \in PROP$ and there are one or more entries $md_1, \ldots, md_n \in MD$ such that $md_i.class \in in_1.domains \wedge md_i.property \in in_2.names$ then $TR([subject := in_1, predicate := in_2,$ $object := CV([name := md_1.range,$ $domains := \bigcup_{i=1..n}(MD.getConcreteSubcls(md_1.range)),\ pos])])$ else if $in_2 \in CV$ and there are one or more entries $md_1, \ldots, md_n \in MD$ such that $md_i.class \in in_1.domains \wedge MD.getConcreteSubcls(md_i.range) \cap in_2.domains \neq \emptyset$ then $TR([subject := in_1, predicate :=$ $PROP([name{:=}md_1.property, names{:=}\bigcup_{i=1..n}\{md_i.property\}, pos]),$ $object := in_2])$
Output Conduits:	1. **Triple product**? - *out* - relationships between inputs

4.2.4.2 Property-Based ClassVariable Generator

The idea behind this node is that often from parsing, it is not clear at the beginning what items are connected by a property, especially if one of their connecting subject/objects is only implicit in the sentence. In the graph of a query a property represents an edge which must of course connect two nodes.

The node takes an NLPData product as input, and depending on the mode the node checks for exact matches only, or if nothing was found for partly matches of the NLPData product with the properties from the Mapping Dictionary. The partly match takes into consideration that in RDF properties are often compound words like *cityIn, believedBy, methodFor* and so on. Those properties might appear as separate words in the query, but often are altered for grammatical reasons or intercepted by additional words, therefore those are not reliably found and only the core of the property name is sufficient for those partly matches, but the confidence value is reduced for those.

By this, the node finds out which words (probably) correspond to properties, and creates a most general triple for each of them: It creates a property product itself with the names it found. With the information stored in Mapping Dictionary, the triple products can be limited by the property domain and range to a certain set classes. Thus, for a property "hasCapital", e.g., a ClassVariable product ranging over the classes {"Country","Province"} can be generated for its domain, and a ClassVariable product ranging over the class "City" can be generated for its range. These products are created by the Property-Based Variable Generator.

So this node creates a triple that covers every possible usage of the property wrt. the ontology, regardless of the position of the words (the other words are yet still unknown) and relies on other nodes that later combine and filter the correct triples. For that, the confidence value assigned to its guesses is relatively low: the guessed subject and object new TriplePart products each get a confidence of about 0.25, and the property itself a relaive ly high value as motivated above, so the triple gets –maximally– a confidence $(0.25 + 0.25 + 1) = 0.5$. As said for the relator node, the process relies on other nodes that later combine and filter these products.

Since the variables for subject and object can often be found in other ways, and can be specified based on the query, these created products are additional accompanied by an appropriate looking-for-replacement products so that they can be merged with other ClassVariable products describing the same entity. (Note that e.g. p_{min} and p_{max} of the new ClassVariable product are those of the property's word, because it is not known better – thus, if e.g. due to grammatical relationships there is more knowledge about the domain and the position of the ClassVariable product that ranges oder domain/range, this can be combined).

Parameter:	Matching type: exact matching or containment, the parameter defines the function $match(a, b)$ accordingly
Input Conduits:	1. **NLPData product** - in - word which might be a property

Formal Operation:	$NLPD \rightarrow (TR \times LFR \times LFR)$ $in \mapsto$ if there are one or more entries $md_1, \ldots, md_n \in MD$ such that $match(md.property, in.word)$ then $pos := [p_{min} := in.p_{min}, p_{max} := in.p_{max}]$, $subj = CV([name := md_1.domain,$ $domains = \bigcup_{i=1..n}\{md_i.domain\}, pos])$, $pred = PROP([name := md_1.property,$ $names := \{md_1.property\}, pos])$, $obj = CV([name := md_1.range, domains := \bigcup_{i=1..n}(MD.getConcreteSubcls(md_i.range)), pos])$, $tr := TR(subject := subj, predicate := pred, object := obj)$, $lfrs = LFR([toReplace := s, context := tr])$, $lfro = LFR([toReplace := o, context := tr])$
Output Conduits:	1. **Statement product*** - *st* - derived triple with a subject and object satisfying the domain and range of the property 2. **Looking-for-replacement product*** - *lfrs* and *lfro* - for the subject and the object

4.2.4.3 Comparison Generator

The comparison generator is the simplest comparison generating node. Ultimately, it only needs all the components of a comparison, i.e. a ClassVariable product, an operator product and a TriplePart product and creates a suitable comparison from them. Besides checking $domain(left) \cap domain(right) \neq \emptyset$ and making sure that $<, >, \leq$ and $\geq$ are only used for numeric domains, this node is a storage node, i.e. the cartesian product of the inputs is formed. Therefore this node also needs a good pre or post filtering.

Parameter:	none
Input Conduits:	1. **ClassVariable product** - $left$ - part of the comparison 2. **Operator product** - $operator$ - middle part of the comparison 3. **TriplePart product** - $right$ - part of the comparison
Formal Operation:	$CV \times OP \times TP \rightarrow COMP$ $left, operator, right \mapsto$ if $left.domain \cap right.domain \neq \emptyset$ $\wedge (operator.symbol \in \{=, \neq\} \vee$ $left.domains \cap right.domains = \{numeric\})$ then $COMP([left := left, operator := operator, right := right])$

Output Conduits:	1. **Comparison product** - with the content: $\{left,\ operator,\ right\}$

4.2.4.4 Local Comparison Generator

This node uses an operator, grammatical relationships $gr_1 \ldots, gr_n$ and all ClassVariable products that currently exist in the agent to create comparisons. It uses the relative position and the grammatical relations of those ClassVariable products to generate new comparisons.

Operators and the right component of a comparison often have a specific grammatical relation, and the node learns which ones to use on its own, based on the test set. Comparison operators are the easiest amongst the three comparisons parts to detect from the input question by simply looking for keywords, and from typical sentence structures, the right comparator can be guessed. The node checks all at this time created ClassVariable products if there is a grammatical relationship of one of the types $gr_1 \ldots, gr_n$ with a ClassVariable product at the other position of gr.

Additionally it is common in natural language to preserve the left component, operator, right component structure of the comparison in terms of their relative position, meaning the assumption $left.p_{max} < operator.p_{min} < right.p_{min}$ is likely to hold, especially for "lower than" and "greater than" comparison since their meaning is based on this order. Therefore, in many cases, the left component can also be found to the left of the operator, i.e. with a lower position number than the operator. By the grammatical relation, the right component is already known and for comparisons, it is necessary that the intersection of the domains of the left and right component must not be empty.

Therefore, any ClassVariable product cv with $left.p_{max} < operator.p_{min} \wedge left.domains \cap right.domains \neq \emptyset$ could now be used to complete a formally correct comparison, whether it is really the intended comparison is another thing. If there are multiple possible candidates, the one with the smallest difference between $left.p_{max}$ and $operator.p_{min}$ is used, since this is also a common convention in natural language sentences.

The available set of ClassVariable products in an agent depends on the current state of the agent. So that at a very early stage of the query evaluation, it could be that the searched ClassVariable product has not yet been generated. But once found the searched ClassVariable product is not expected to be discarded again later, since it must be in some further relations outside the comparison to have a real meaning and not be an unbound variable. Therefore, these nodes can start their operations later than the other nodes without risking to lose the searched ClassVariable product, but with having a broader access to potential ClassVariable products. These nodes therefore learn to wait for a certain amount of *rounds* before they start checking their conduits, but while waiting, these nodes are considered "active" as long as they have at least one product in one of their conduits, so they are always executed before any collector nodes.

Optionally, this node can also still restrict the possible candidates for $left$ to certain Part Of Speech tags, it has been shown that these are actually nearly always *NNS* or *NN*, but this restriction applies to almost all ClassVariable product and is therefore not particularly meaningful.

Parameter:	Gr - List of relevant grammatical relationships applicable for local comparison generation
Input Conduits:	1. **Operator product** - op which triggers the generation
Formal Operation:	$OP \times GR_{Tag} \rightarrow COMP$ $op \mapsto$ if $(sign(op) \in \{=, \neq\} \vee$ $left.domain \cap right.domain = \{numeric\}) \wedge$ $\exists left, right \in CV(agent):$ $left.domain \cap right.domain \neq \emptyset$ $\exists gr \in Gr : gr(right, op) \in NC_{GR}$ then $COMP([left := left, operator := op, right := right])$
Output Conduits:	1. **Comparison product** - generated comparison

4.2.5 Modifier Nodes

The evolution of successful agents usually results in agents that first generate a large amount of products and later merge them, and then remove excess ones (that have been generated by the above guess-based nodes), and finally add missing ones. In particular, the generation of products is –intentionally– based only on the locally accessible context (i.e., the input products, and further products that are stored in a node). Thus, before the products have been merged (usually, collected in a query graph product, the generating nodes have no information about what products have been created by other nodes, and whether their products are compatible in the given context. As a consequence, nodes that *modify*, usually, refine, products play an important role. Such *Modifier nodes* are nodes that modify the products flowing though them content in one way or another. So input and output products are of the same type, but have modified values of their slots.

4.2.5.1 Synonym Node

The synonym node accesses the synonym table and tries to find a synonym that is an ontology term for the given word, if so, the word of the input (which is often not used in the ontology) is changed to a synonym term contained in the ontology and the product is output at the first output. If such a term does not exist, the second output is used to forward the original word.

Parameter:	Table of Synonyms $Syn(word, term)$
Input Conduits:	1. **NLPData product** - in - word which might have a synonym term
Formal Operation:	$NLPD \rightarrow NLPD$ $in \mapsto$ if there is a syn : $(in.word, syn) \in Syn$ then $out_1 := NLPD([word := syn,\ p_{min} := in.p_{min}])$ else $out_2 := in$
Output Conduits:	1. **NLPData product**? - out_1 - ontology term which is a synonym of the input word 2. **NLPData product**? - out_2 - input word, if no synonym is known – note: either out_1 or out_2 is provided

4.2.5.2 Confidence Changer Node

Usually, when a product is created, the node that creates it always gives it a confidence value. This is learned by the node by mutation (see Section 3.3.4). However, if a product reaches a certain point in the agent or if the value of a certain information flow is to be changed uniformly, confidence value changer nodes come into play. Their only function is to change the confidence value for all atomic products by a value specified in the parameters (see Section 3.2 for additional information on the confidence value).

Parameter:	v - Value of the change in the confidence of a product (positive increase or negative decrease)
Input Conduits:	1. **Product** - in - product to change confidence value of
Formal Operation:	$PROD \times \mathbb{R} \rightarrow PROD$ $in \mapsto out : out = in$ except all confidence values of atomic products incremented/decremented by v, limited to [0,1].
Output Conduits:	1. **Product** - out is same product as in with changed confidence

4.2.5.3 CustomAggregation Node

Some concepts are so general that a user might well expect that the interface should understand them. However, from a machine perspective, it requires additional knowledge that is not given in either the ontology or query. A typical example would be the density of something or *value a* per *value b*.

To be able to implement this, there are Custom Aggregation Nodes. They learn to convert certain keywords into a combination of literals, properties and operators. So in the case of density this would mean "*density*" is a keyword that has a relation via a Grammatical Relationship to another position. At this position has to be a numeric ClassVariable product and that must be a property of a ClassVariable product which is object-valued and has a (sub)property of area or volume. If so, that ClassVariable product is then divided by a (sub)property of area or volume of the ClassVariable product which has that property product. This is expressed by a newly generated CompoundClassVariable product.

On the one hand this might seem a bit far fetched, but the criterion is so strict, that false positives are extremely rare up to impossible. On the other hand no agent has learned the correct parameter on its own so far, but they can be added (editing the XML representation of an agent) by a curator. This is not particularly satisfying, but sufficient since there are only very few such concepts. Each of such concepts requires a single node, so one cannot expect that a unknown concept would be correctly interpreted.

Although it would of course be better if this were also done automatically, it is not surprising since in [6], for example GeobaseQuery 196 (see Examples 20 and 21) was considered unsolvable and has been removed from the test set. So these concepts are rare enough that it is quite possible to create them by hand.

Parameter:	K - Keyword List (e.g. density) D - List of possible properties whose value is used as second operand (e.g. *area*, *volume*) op - Type of the operation (e.g. /,*,+,-)
Input Conduits:	1. **NLPData product** - in_1 - the keyword, e.g. "density" 2. **Triple product**- in_2 - the triple that yields the first operand, e.g. population

Formal Operation:	*NLPD* $\times$ *TR* $\rightarrow$ *CCV* $\times$ *TR* $(in_1, in_2) \mapsto$ if there are one or more entries $md_1, \ldots, md_n \in$ *MD* such that $in_1.word \in K$, $in_2.object.domains = \{$"numeric"$\}$ $\wedge\ compound(in_1, in_2) \in NC_{Gr}$ $\wedge\ md_i.domain \cap in_2.subject.domains \neq \emptyset$ $\wedge$ *MD.getConcreteSubcls*$(md_i.range) \cap in_2.object.domains \neq \emptyset$ $\wedge\ md_i.property \in D$ then $prop$:= *PROP*$([name := md_1.property, names := \bigcup_{i=1..n}\{md_i.property\},$ $p_{min} := in_1.p_{min},\ p_{max} := in_1.p_{max}])$, $obj := CV([name := md_1.property, domains := \bigcup_{i=1..n}\{md_i.range\},$ $p_{min} := in_1.p_{min}, p_{max} := in_1.p_{max}])$, $tr :=$ *TR*$([subject := in_2.subject, predicate := prop, object := obj])$, $ccv := CCV([cv_1 := in_2.object, cv_2 := obj, operator := op])$
Output Conduits:	1. **CompoundClassVariable product**? - ccv - returns the compound value 2. **Triple product**? - tr - contains the relation of the newly introduced literal from D to its ClassVariable product. - note that either nothing, or a CompoundClassVariable product and a triple product are generated.

Example 20 *Question Geobase178: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase178 are depicted in Figure 4.20. This is a simple example of how a concept based on an arithmetic operation is used, in this case density.*

Example 21 *Question Geobase196: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase196 are depicted in Figure 4.21. This example is more complex, here also implicitly an arithmetic operation is used by specifying a concept, but also at the same time implicitly a max aggregation over the same value is required.*

4.2.5.4 Stemmer Node

Depending on the language, words can be changed by many different factors, these can be conjugations, tenses, plural, gender, and much more. For their interpretation in this process, these changes play a minor role if at all, but are problematic for the recognition of terms used in the ontology. In information retrieval, linguistic computer science or also in search engines therefore often so-called *stemming* is used. The aim is to reduce different words from the same stem to a

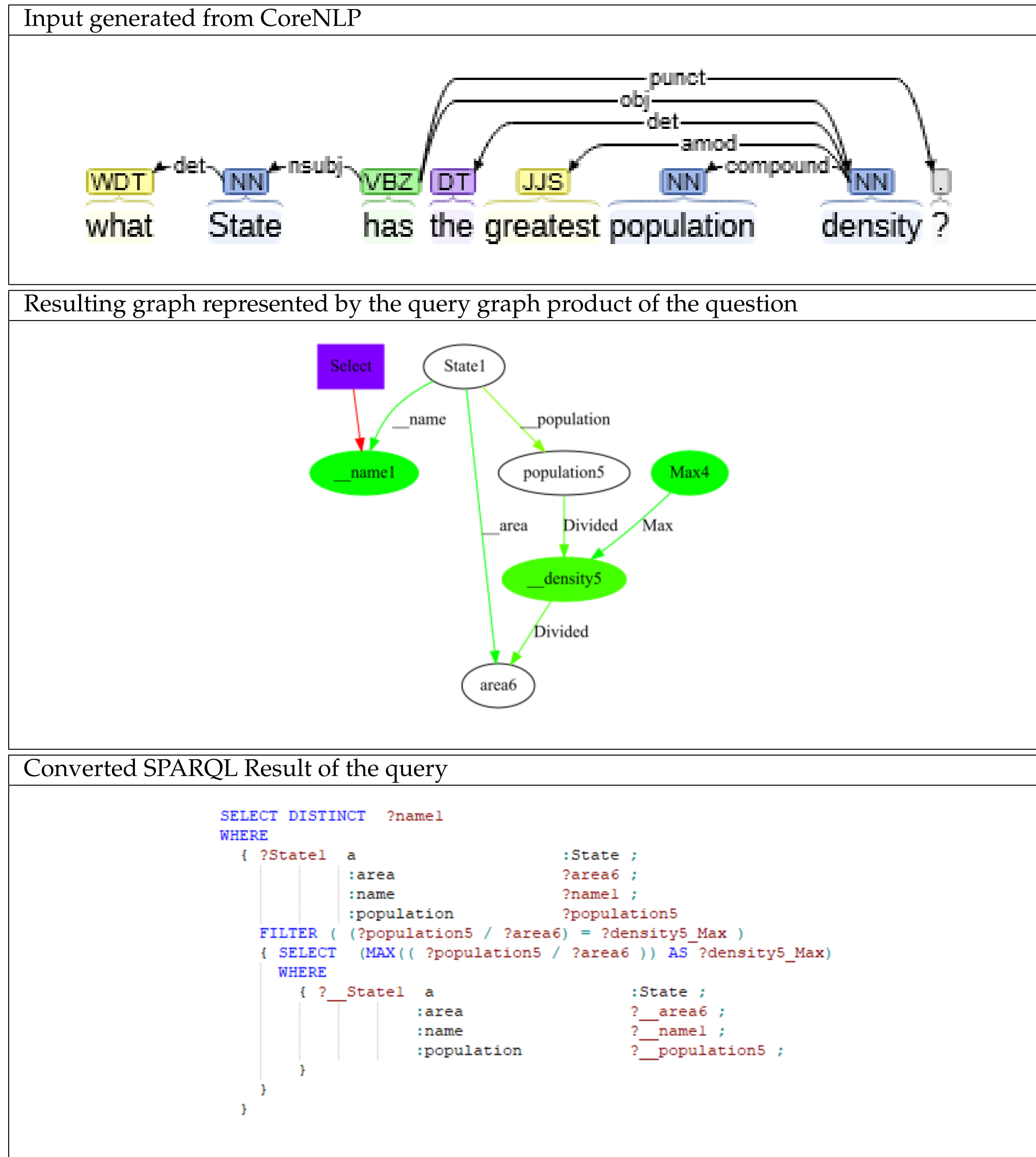


Figure 4.20: Question *Geobase178* with CoreNLP anntotations, query graph, and resulting SPARQL query

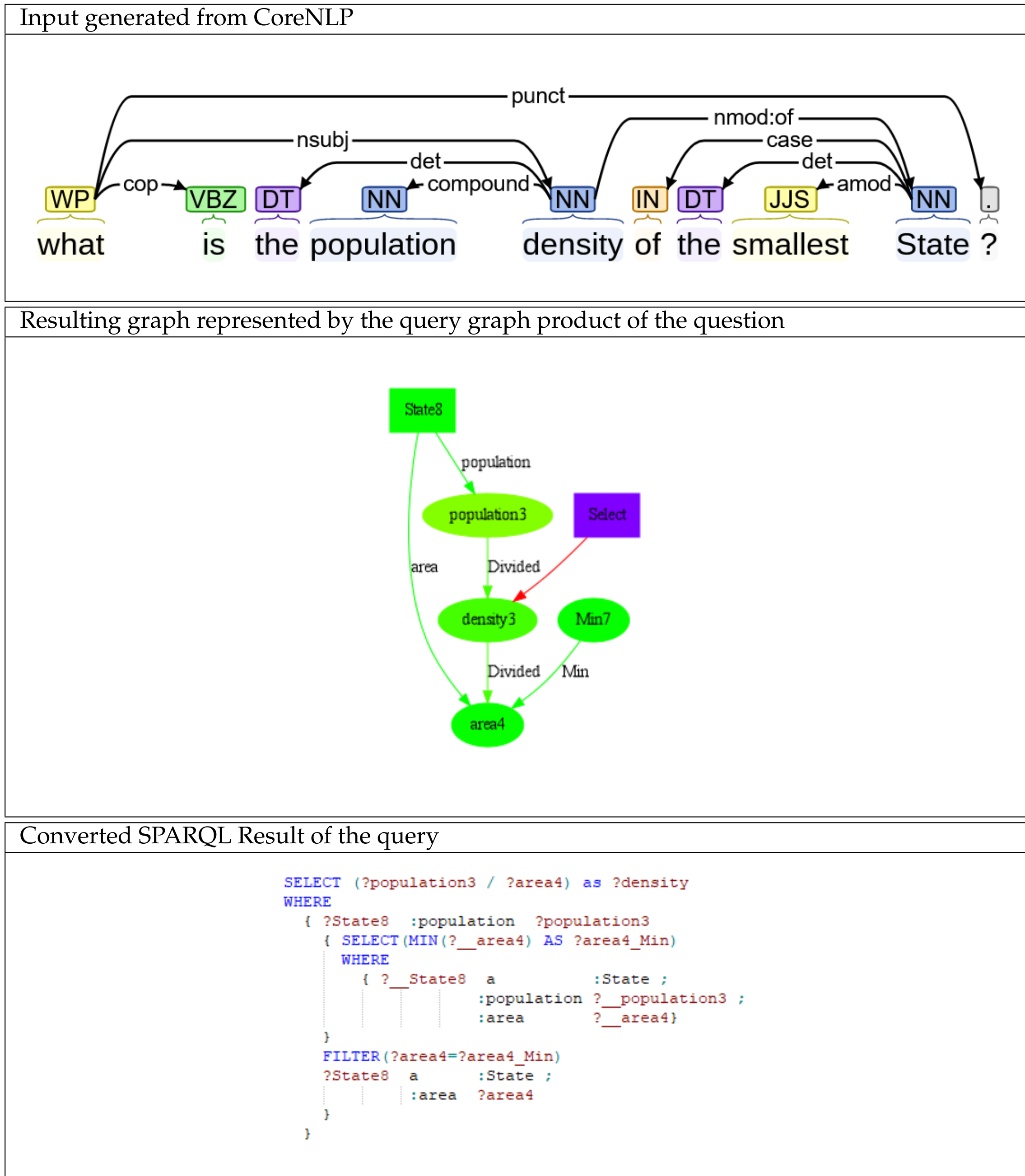


Figure 4.21: Question *Geobase196* with CoreNLP anntotations, query graph, and resulting SPARQL query

common *stem form*. It is not so important that the *stem form* really matches the base form of the word, it is much more important that words from the same stem are mapped to the same *stem form*, and only those.

For stemming itself, there is a variety of algorithms and methods. Here, a widely used algorithm by Martin Porter is used, the *Porter Stemmer* [85]. The algorithm is based on a set of shortening rules. Those rules are applied to the classes, properties, and individuals of the ontology and to the input words of the question. If there is a match between the *stem forms* of the ontology and the question, the input is changed to the matching ontology expression.

To a certain degree, CoreNLP already stems words, but the Porter algorithm is more thorough and can match a bigger variety.

Parameter:	none
Input Conduits:	1. **NLPData product** - in word from the input query
Formal Operation:	Porter Stemming Algorithm as described in detail in [85]
Output Conduits:	1. **NLPData product** - if an ontology expression has the same stem, the NLPData product value is changed to the term of the ontology otherwise the input word stays the same.

4.2.5.5 Distinctness Node

The simplest way to create distinctness products is the distinctness node. It creates a distinctness product from all incoming TriplePart products without any further checks and forwards it. Suitability must be checked by other nodes before or after. To be able to collect all TriplePart products, this node is a collector node, i.e., it serves for operational stratification and processes a *sequence* of inputs that wait for processing (see Section 3.3.3.2). A more reliable and general-purpose variant is the Distinctness Detector Node, which is described afterwards.

Parameter:	none
Input Conduits:	1. **TriplePart product*** - in - which are distinct from each other
Formal Operation:	$\mathcal{P}\,(TP) \rightarrow DIS$ $in_1 \ldots in_n \mapsto distinctness(\{in_1, \ldots, in_n\})$
Output Conduits:	1. **Distinctness product** - of all input TriplePart products

4.2.5.6 Distinctness Detector Node

For ClassVariable products directly created from words in the text, there is a high probability that they also mean different entities. Therefore, this node creates a distinctness product that declares all such ClassVariable products to describe different items.

For that, it does not only process the ClassVariable products that it just read from its input, but remembers all ever received ClassVariable products in its storage, and with each new one, created an appropriate large distinctness product.

Nevertheless, ClassVariable products positioned directly after each other can more often be a combination of individual and descriptive class like "New York City" or "the state of Texas", so they must have a learned minimum distance (≥ 0). In such cases, always one of these two ClassVariable product describes a class and the other one describes an individual.

Parameter:	$dis_{min} \in \mathbb{N}$ - minimal distance between two ClassVariable products to be considered to mean different things m - if true ignore dis_{min} whenever o_1 and o_2 are not one an individual ClassVariable product, and the other one a class ClassVariable product
Input Conduits:	1. **Product** - in - a products $\{in\}$ which might contain ClassVariable products which are disjunctive to each other and explicitly mentioned in the Part Of Speech tags
Formal Operation:	$PROD \rightarrow DIS$ $in \mapsto$ if $NC[in.p_{min}] = in.name$ (i.e., the ClassVariable product has been created from a word) and $\forall c \in storage(1) : distance(c, in) \geq dis_{min}$ then add c to $storage(1)$, $dis := DIS(storage(1))$
Output Conduits:	1. **Distinctness product**? - dis - a distinctness product

4.2.5.7 Replace POS Slots Node

Replace POS Slots Nodes mainly replace pronouns by the noun for which they are used.

This node exchanges the NLPData product in_2 by a copy of the replacing NLPData product in_1, but the position of the copy is changed to the value of in_2.

Additionally, a sameAs product is generated that can later be used in a compound product to signal that the copy and the original actually describe the same entity.

Here the time of merging is important, because an immediate replacement would not take the positioning of the copy into account and would make the whole procedure superfluous, but ultimately the copy should not be seen as independent in the final result. From there the NLPData product or the results from it and the sameAs product are passed on by different conduits, but it can be assumed that they meet again at a later time in a *collector node*.

Parameter:	none
Input Conduits:	1. **NLPData product** - in_1 - replacing Word 2. **NLPData product** - in_2 - word to be replaced
Formal Operation:	$NLPD \times NLPD \rightarrow NLPD \times SAME$ $(in_1, in_2) \mapsto$ $rep := NLPD([word := in_1.word,\ p_{min} := in_2.p_{min},\ p_{max} := in_2.p_{max},$ $type := in_1.type, subtype := in_1.subtype,$ $entityType := in_1.entityType,$ $normalizedNamedEntity := in_1.normalizedNamedEntity])$ $same := SAME([content := \{rep, in_2\}])$
Output Conduits:	1. **NLPData product** - rep - copy of in_1 with the position of in_2 2. **SameAs product** - $same$ between in_2 and rep;

4.2.5.8 Establish Replacement Node

Often looking-for-replacement products are created and elsewhere there is actually something they can be replaced with, but not always. In such cases, it may have to be accepted at some point that nothing can be found that is more suitable. For this purpose there is the *Establish Replacement Node*, which turns the contents of looking-for-replacement products that up to this node have not found a partner that could replace them into full-fledged products.

Parameter:	none
Input Conduits:	1. **Query graph product** - in - where the looking-for-replacement products did not take place
Formal Operation:	$QGP \rightarrow QGP$ $in \mapsto out := QGP([components := in.components/(in.components \cap LFR)$ $\cup \{p.components \mid p \in in.components \cap LFR\}])$
Output Conduits:	1. **Query graph product** - out - same as input, but with unpacked looking-for-replacement products

4.2.5.9 Individual Preciser Node

Due to the ambiguity of names or simply by convention, it sometimes happens that an individual is described in a query by both an identifier and its class (e.g. "the river Thames" or "New York City"). Here it is important to recognize whether those are two different entities or an individual which is described both with its name and with its class or an identifier which contains a class. The latter can be determined most easily with Algorithm 3. If this case can be excluded, it must still be decided whether one or two entities are described.

This is done by first checking that the ClassVariable products are no further away than d, with respect to their position, and that the intersection of the domains is not empty. If this is the case, the domain of the identifying class is reduced to the intersection and the other ClassVariable product is removed and a sameAs product is created for both. If this is not the case, a distinctness product is created for both and is added to the result.

Parameter:	$d \in \mathbb{N}$: maximal allowed distance between two ClassVariable products
Input Conduits:	1. **ClassVariable product** - Identifying variable in_{ID} 2. **ClassVariable product** - class description variable in_{cv}

Formal Operation:	$\times CV \times CV \rightarrow CV \times AUX$
	$(in_{ID}, in_{cv}) \mapsto$
	if $in_{cv}.domains \cap in_{ID}.domains \neq \emptyset \wedge distance(in_{cv}, in_{ID}) \leq d$ then
	$out := CV([name := in_{ID}.name,$
	$domains := (in_{cv}.domains \cap in_{ID}.domains),$
	$p_{min} := min(in_{cv}.p_{min}, in_{ID}.p_{min}),$
	$p_{max} := max(in_{cv}.p_{max}, in_{cv}.p_{max})]),$
	$aux := SAME([content := \{in_{cv}, in_{ID}, out\}])$
	else
	$aux := DIS([content := \{in_{ID}, in_{cv}\}])$

Output Conduits:	1. **ClassVariable product**? - out - combined variable 2. **Auxiliary product** - aux - both inputs set same as with the output variable or a disjunction between the inputs

4.2.5.10 CV Merger Node

It is not a trivial task to decide which ClassVariable products actually refer to the same entity and which do not. The CVMerger node uses the context, grammatical relations, and the positions of products to make merges as accurate as possible.

This nodes replaces greedily ClassVariable products which are either encapsulated in a looking-for-replacement product or have a negative position with the best matching ClassVariable product

(details see Algorithm 4) which still preserves the validity of the query graph product.

Parameter:	none
Input Conduits:	1. **Query graph product** - in - graphProduct in which ClassVariable products should be merged
Formal Operation:	$QGP \mapsto QGP$ $in \mapsto out := CvMerge(in)$ for CvMerge see algorithm 4
Output Conduits:	1. **Query graph product** - out with potentially merged ClassVariable products

4.2.6 Filter Nodes

The following nodes are used to filter products according to certain criteria.

4.2.6.1 Ontology-based Filter Node

This node is responsible for filter operations based on the ontology database. Depending on its parameters, it can filter positively or negatively on almost any property. The filter functions available are shown in Table 4.4. Apart from structural conditions, also the provenance of a product can be tested.

Parameter:	$p()$ is a filter function as listed in Table 4.4
Input Conduits:	1. **Product** - in - Any product
Formal Operation:	$PROD \rightarrow PROD$ $in \mapsto$ if $p(in) = true$ then $out := in$
Output Conduits:	1. **Product**? - out - the product, if it meets the filter criterion

4.2.6.2 Product Comparison Filter Node

These nodes realize comparisons based on products, slots of products, and membership in compound products. Each node implements an logical operator $lop(i_1, i_2)$ which is based on the parameter of the node. Each newly received product at in_i is compared with the whole set $storage(in_{3-i})$ of the other input conduit. The implementation of negations requires some kind of stratification, since it cannot be known beforehand that no further information will arrive in the future. Thus, "Not"-nodes are collector nodes, i.e., they wait until there is no more activity in the agent and only then they are activated.

Algorithm 4: CvMerge($compoundProduct$)

Result: GraphProduct with LFRs replaced
Initial: c = GraphProduct;
$bind \leftarrow All\ cv \in c.components \cap CV$
$unbind \leftarrow All\ cv \in bind : cv.p_{min} < 0 \vee \exists\, l \in c.components \cap LFR : cv = l.toReplace$
$triples \leftarrow All\ t \in c.components \cap TR$
$c.remove(unbind)$

$matchingMatrix \leftarrow M^{Bind \times Unbind}$
foreach *pair of* $b \in bind$ *and* $u \in unbind$ **do**
  if $b.domains \cap u.domains \neq \emptyset$ **then**
    $matchingMatrix^{b,u} \leftarrow \frac{|b.domains \cap u.domains|}{max(|b.domains|, |u.domains|)}$
  else
    $matchingMatrix^{b,u} \leftarrow= -1$
$binding \leftarrow (row, col)$ of the entry with the highest value in $matchingMatrix$
while $binding.row \neq -1$ **do**
  $unbindVariable = unbind.get(binding.col)$
  $bindVariable = bind.get(binding.row)$
  if $\nexists t \in triples\ :\ unbindVariable, bindVariable \in components(t)$ **then**
    $new_var \leftarrow copy(bindVariable)$
    $new_var.domains \leftarrow unbindVariable.domains \cap bindVariable.domains$
    foreach $t\ \in\ triples$ **do**
      if $unbindVariable \in t.components \vee\ bindVariable \in t.components$ **then**
        $t_{copy} \leftarrow copy(t)$
        $t_{copy}.replace(unbindVariable, new_var)$
        $t_{copy}.replace(bindVariable, new_var)$
        if *checkProduct(copyTriple) // see Algorithm 5* **then**
          $c.replace(triple, t_{copy})$
  $matchingMatrix_{(unbindVariable, bindVariable)} \leftarrow -1$
  $binding \leftarrow (row, col)$ of the entry with the highest value in $matchingMatrix$
return c

Parameter:	p() is a filter function from Table 4.5
Input Conduits:	1. **Product** - in_1 - product which should be forwarded if condition is met 2. **Product** - in_2 - product which is part of the condition, but not forwarded

Formal Operation:	$PROD \times PROD \rightarrow PROD$
	$(in_1, in_2) \mapsto$
	if in_1 was read then // recall: for each $i_2 \in storage(2)$
	  if $\exists i_2 \in storage(2) : p(in_1, in_2) = true$
	    then $out := in_1$ and in_1 is not added to $storage(1)$
	    else in_1 is added to $storage(1)$
	else // then i_2 was read
	  for each $i_1 \in storage(1)$
	    if $p(in_1, in_2) = true$
	    then $out := in_1$ and remove in_1 from $storage(1)$

Parameter	**Condition for in**
Property	in is a property product
Literal	in is a ClassVariable product and $in.domains = \emptyset$
Class	in is a ClassVariable product and $in.classes \neq \emptyset$
Aggregation	$in \in AGG$
Graph Product	$in \in QGP$
Instance	in is a ClassVariable product and $origin(in)$ is an Identifier Node
Non-Instance	in is a ClassVariable product and $origin(in)$ is not an Identifier Node
Relator-based	in is a TriplePart product and $origin(in)$ is a Relator Node
Non-Relator-based	in is a TriplePart product and $origin(in)$ is not a Relator Node
Symmetrical	$MD.getInverse(in) = in$
Non Symmetrical	$MD.getInverse(in)$ does not exist or $MD.getInverse(in) \neq in$
The following filters are used to detect junk:	
Contains Reflexive (junk)	in is a compound product and does contain a statement product where $subject = object$
Non Reflexive	in is a compound product and does not contain a statement product where $subject = object$
Negative (junk)	in is a positionable product and $Position(in) < 0$
Non Negative	in is a positionable product and $Position(in) \geq 0$

Table 4.4: Parameter of a Database Filter Node

Output Conduits:	1. **Product**? - out - product which has met the condition of the parameter p

4.2.6.3 Nearest Node

The Nearest Node searches for a given position in the sentence the closest Part Of Speech tag which matches its parameter and adds another Grammatical Relationship of type "nextTo" between the tag and the input. This can then be used by other nodes, for example to affect conflict solver metrics. The node also can be set to the modes "only forward", "only backward" and "both directions".

Parameter	Condition for $i_1 \in I_1$, wrt. $I_2 := storage(2)$
And	$i_1 \in I_2$
Or	$true$
Not	$i_1 \notin I_2$
Same Class	$i_1 \in CV \land \exists i_2 \in I_2 : i_1.domains \cap i_2.domains \neq \emptyset$
Not Same Class	$i_1 \in CV \land \forall i_2 \in I_2 : i_1.domains \cap i_2.domains = \emptyset$
Same Position	$i_1 \in POS \land \exists i_2 \in I_2 : i_1.scope \cap i_2.scope \neq \emptyset$
Not Same Position	$i_1 \in POS \land \forall i_2 \in I_2 : i_1.scope \cap i_2.scope = \emptyset$
Contains Product	$\exists i_2 \in I_2 : i_1 \in i_2.components$
Not Contains Product	$\nexists i_2 \in I_2 : i_1 \in i_2.components$
No Input	$I_2 = \emptyset$
Highest confidence value	$\nexists i_2 \in I_1 \cup I_2 : i_1 \neq i_2 \land i_2.confidence > i_1.confidence$ With this settings the node becomes also a collector node
Delay	$true$

Table 4.5: Parameter Values of Product Comparison Filter Nodes

Parameter:	tag: POS tag NLPData product to be searched for,
	$mode$: search direction{
	backward: $mode(a, b) \leftarrow a < b$
	forward: $mode(a, b) \leftarrow a > b$
	both: $mode(a, b) \leftarrow a \neq b$
	}

Input Conduits:	1. **Positionable product** - in - for which a NLPData product of the type tag should be found
Formal Operation:	$POS \rightarrow GR$ $in \mapsto$ if there is an $o \in NC_{Tag} : mode(o, in) \ \land tag = o.type \land$ $\nexists\ x \in NC_{Tag} : x.type = tag \land mode(o, in)$ $\land\ x.distance(in) < o.distance(in)$ then $gr := GR([dep := o, gov := in, type := \text{"}nextTo\text{"}])$
Output Conduits:	1. **Grammatical Relationship**? - gr - new relation of the type "nextTo" between in and a NLPData product of the type tag which is closest according to the $mode$

4.2.6.4 POS Filter

The POS Filter checks each incoming positionable product if it is on the same position as a word from the input sentence which has a POS Tag annotated from its internal list of tags. Depending

on its mode, the node has either a positive or negative list for filtering, meaning either the position of such a tag is mandatory to be forwarded by this node or everything except those positionable products are forwarded.

Parameter:	$type$: List of POS tags, $mode$: $\{positive, negative\}$ decides if $types$ is a positive or negative list
Input Conduits:	1. Compound product - in - which should be filtered for specific tags
Formal Operation:	$CPROD \times P_{Tag} \rightarrow CPROD$ $in \times type \mapsto$ if there is (not) a $s \in in.components \cup in$: $\exists c \in NC_{Tag} : c.in(s) \wedge c.type = tag$ then $out := in$
Output Conduits:	1. Compound product - out - if it meets the criteria

4.2.6.5 Extract Product Node

While the purpose of query graph products is to bundle other products and let them move together through the agent, to be able to execute context-aware actions, agents should nonetheless have the ability to select individual products or unpack a query graph product again.

This is the purpose of Extract Product Nodes. Depending on their filter set spc, this node can filter for product classes, specific Grammatical Relationships, Part Of Speech tags, position intervals or just unpack a compound product. The input is also sent on another conduit without but without the components that fulfill the filter functions criterion.

Parameter:	Depending on the mode, the filter specifies a set of products that will be unpacked: spc is a set either of: • product classes: $spc \subseteq PROD$ • Grammatical Relationships: $spc \subseteq GR$ • Part Of Speech tags: $spc \subseteq NC_{Tag}$ • positions n to m: $spc := \{i \in POS : i.p_{min} >= n \wedge i.p_{max} <= m\}$ • unpack $spec := CPROD$
Input Conduits:	1. **Compound product** - in which should be filtered

<table>
<tr><td>Formal Operation:</td><td>$CPROD \to \mathcal{P}(spc)$
$in \mapsto$
$p := \{c : c \in in.components \cap spc\}$
$in.components := in.components/(in.components \cap spc)\})$
if $in \in STMT \wedge |in.components| = 3$
  $c_{rest} := in$
else if $in \in QGP \wedge |in.components| > 0$
  $incomplete := \{c\} : c \in in.components \cap STMT \wedge |c.components| < 3$
  $c_{rest} := in.components/incomplete$
else
  $c_{rest} := \emptyset$</td></tr>
<tr><td>Output Conduits:</td><td>1. Products* - p - which fullfil the filter criteria
2. Compound product - c_{rest} - rest of the input which does not fullfill the chosen filter criterion</td></tr>
</table>

4.2.7 Graph Nodes

In many cases, questions are incomplete and require some degree of interpretation and completion to figure out what was implicitly meant. While query languages cannot handle those implications, in natural language they are quite common. The problem here is, that a natural language is not bound to an ontology and most implications are understood through background knowledge and reasoning based on this knowledge.

While reasoning to a certain degree is possible, the vast background knowledge of an human cannot be matched by EvolNLQ. Therefore the next best possibility is to derive the implicit knowledge by completing the query to a connected graph. The above-mentioned guess-based node types *Relator* an *Comparison Generator* already contribute a lot to this. Further nodes for completing the graph will be described below. To facilitate and "motivate" such implicit completions, EvolNLQ is limited to answering questions that are not based on a cartesian product (which would be all pairs of "x such that ... and y such that ..." without any join condition between them). Queries using a cartesian product without a join condition are rather uncommon, and also the largest sample of training queries, Geobase does not contain any of them.

Additionally, the graph must be valid in terms of the ontology, i.e., respecting the knowledge about domains and ranges of the properties.

4.2.7.1 Graph Collector Node

The Graph Collector Node has the primary task of bundling products and forwarding them together, so it is a direct implementation of the abstract collector node (see Section 3.3.3.2 on

page 52). Besides this task it can optionally remove duplicates, reduce the domains of class variables to the intersection of their occurrences, replace all TriplePart products that occur in sameAs products with the first TriplePart product of the sameAs product and find matchings for looking-for-replacement products.

Parameter:	The following flags can be set true, if the operations should be executed before outputting. • rdu - remove duplicates (inverse included) • rdd - reduce domains of class variables • sar - apply sameAs product replacements • *lfr* - handle looking-for-replacement products
Input Conduits:	1. **Product*** - in - set of all kinds of products that should be collected into a query graph product
Formal Operation:	$\{PROD\} \rightarrow QGP$ $\{in_1, \ldots, in_n\} \mapsto$ $g := QGP([components := \{in_1, \ldots, in_n\}$ if rdu then remove duplicate components (by deep-equality), if rdd then reduce domains of class variables (as formally defined for the Corrector Node type below (page 150)), if sar then merge overlapping sameAs products and replace ClassVariable products according to the sameAs products, if *lfr* then handle looking-for-replacement products (as formally defined for the CV Merger Node type above (page 143)) $out :=$ the result of the above
Output Conduits:	1. **Query graph product** - out - collection of all received products

4.2.7.2 Corrector Node

A common case for applying refinements to existing products is that ClassVariable products and property products have a larger domain or form a larger union at the beginning and can later be reduced using the context.

This reduction is done by corrector nodes. Such nodes check all compound products whether they are valid at all and remove invalid domains and properties. Note that *standalone* atomic products do not need to be corrected because they cannot be invalid in themselves.

The *domains* of a (standalone) ClassVariable product represent the union of these domains, and *names* of a (standalone) property product also represent the union of the such-named properties.

The formal domains and ranges of properties are known in the Mapping Dictionary.

Restrictions can come up due to the signatures of property products in triple products, and due to intersecting constraints on the same ClassVariable product or property product *between* occurrences in different triple products:

- For *triples* that *combine* such products, an obvious check can compare the domain of the named properties with the ClassVariable product that forms the subject of the triple, and the range of the properties with the object, which can be a ClassVariable product or a constant product. Additionally, information about the inverse properties is taken into account since the domain of the inverse of a property can be more restricting then its non inverse range (this is somewhat specific to the design of the MD, whose domains are based on concrete classes, while the range is always a single, maybe overestimated class). The algorithm is shown in Algorithm 5. This check can result in severely restricting the domains of the ClassVariable products, rule out some of the named properties, and even turn the triple impossible if one of these sets is cut down to the empty set. Algorithm 5 in that case removes the respective component.
- For all TriplePart products tp which occur in multiple parts (=triples and comparisons) of c, the found constraints are combined: in every occurrence of tp, its domain/names are reduced to the intersection of all found occurrences (considering subclass/superclass information from the Mapping Dictionary) as shown in the first part of Algorithm 6. Again, if a restriction results in the empty set, the components containing that tp are removed.

These two steps have to be repeated as long as a step changed something to track down the restrictions as far as possible.

In case of inconsistencies, a repairing algorithm as shown in Algorithm 6 is applied. As not much is known about the details of the products and their reasons, the strategy is to remove as few as possible components (i.e., triples or comparisons), trying to keep the maximal sum of confidences of the rest. The reduced product is then forwarded to the conduit.

Parameter:	*MD*- Mapping Dictionary
Input Conduits:	1. **Compound product** - in_1 - to validate
Formal Operation:	*CPROD* $\rightarrow$ *CPROD* $in \mapsto$ ReduceToValid(in) see Algorithms 5 and 6
Output Conduits:	1. **Compound product**? - same as input with refined constraints and without invalid products

Algorithm 5: ReduceProduct($product$,$checkInverse$)

```
Result: true, if the reduced product is Valid; changed it as a side-effect
Initial: c = compound product, MD = Mapping Dictionary, default: checkInverse = true
if c ∈ TR then
    // apply MD knowledge in forward direction s ← copy(c.subject)
    p ← copy(c.predicate)
    o ← copy(c.object)
    s.domains ← ∅; o.domains ← ∅; p.names ← ∅
    foreach md ∈ MD such that md.domain ∈ c.subject.domains ≠ ∅ ∧ md.property ∈
      c.predicate.names ∧ MD.getConcreteSubcls(md.range) ∩ c.object.domains ≠ ∅ do
        s.addDomain(md.domain)
        p.addProperty(md.property)
        o.addDomain(MD.getConcreteSubcls(md.range))
    c.subject ← s
    c.predicate ← p
    c.object ← o
    if checkInverse ∧ ¬ c.object.isLiteral() then
        // apply MD knowledge in inverse direction i ← newTR([
        subject := c.object,
        predicate := c.predicate.inverse()   // a property product whose names are the
          inverses of c.predicate.names
        object := c.subject])
        c := checkProduct(i, checkInverse = false)
    return |c.subject.domains| > 0 ∧ |c.predicate.names| > 0 ∧ |c.object.domains| > 0
else
    foreach p ∈ c.components ∩ TR do
        // if c is a graph: over all its triples:
        if ¬ checkProduct(p) then
            remove(p, c)   // else it has been reduced as a side effect
    return true
```

4.2.7.3 Remove Disconnected Node

This node removes all TriplePart products not connected to another product from a given query graph product. This can happen when other nodes have removed triple, for example, but the components are still present, or when a TriplePart product has been inserted that was simply never part of a compound product. This does not necessarily mean that the graph is connected afterwards, since each subgraph with more than one would still be there and unconnected to the other subgraphs.

Algorithm 6: ReduceToValid($compoundProduct$)

```
Result: valid Product, potentially reduced content
Initial: c = compound Product
//See Algorithm 5 for ReduceProduct
cc ← copy(c)
while the previous iteration changed something and |cc.components()| = |c.components| still holds do
    checkProduct(cc)   // changes cc as side effect
    m ← map < string, TP >
    foreach stmt ∈ cc.component ∩ (TR ∪ COMP) do
        foreach tp ∈ stmt.component ∩ (CV ∪ PROP) do
            if m.contains(tp) then
                if tp ∈ CV then
                    tp.domains ← tp.domains ∩ m.get(tp.name).domains
                else
                    tp.names ← tp.names ∩ m.get(tp.name).names
            else
                m.put(tp.name, tp)
    foreach name ∈ m.keys do
        new := m.get(name)
        foreach occurrence of tp in cc (maybe nested) s.t. tp.name = name do
            if (tp ∈ CV and new.domains ≠ ∅) or (tp ∈ PROP and new.names ≠ ∅) then
                replace this occurrence of tp by m.get(name)
            else
                remove parent (triple or comparison) of tp from cc
if |cc.components()| = |c.components| then
    return cc
else
    d ← copy(c)
    foreach x ∈ components(c) ∩ TR do
        d_x ← copy(c)   // create d_x as c without x
        d_x.remove(p_x)
        d_x ← ReduceToValid(d_x)
        conf_x := Σ_{y∈d_x.components} y.confidence   // sum(conf) of the graph without component x
    return the d_x such that conf_x = max_{x∈components(c)}(conf.x) if exists (i.e., not empty)
```

Parameter:	none
Input Conduits:	1. **Query graph product** - in - with disconnected products
Formal Operation:	$QGP \rightarrow QGP$ $in \mapsto in/d \ : d = \{ \forall p \in in.component : \nexists\, p_2 : c \in in.component \wedge p, p_2 \in components(c) \cap TP\}$

Output Conduits:	1. **Query graph product** - *out* - *in* without disconnected products

4.2.7.4 Confidence Graph

Since a, at least to some degree, answerable queries should also not leave out too much implicit information either for a machine nor a human to avoid misunderstandings, the concept of adding as few new resources as possible to a query to turn it into a connected graph is reasonable. This also happens, if an ontology does not contain all notions of a domain directly, such as e.g. for "all mountains in Europe" – *Mondial* contains only the relationship between mountain and countries, and between countries and continents; thus a path via countries must be added to the query graph.

Therefore the *confidence graph* is constructed to make a query graph connected with the least amount of derived content as possible. It is a classical *graph data structure*, external to the products world of the agents. For better and explicitly distinguishing it, for the confidence graph the terms *graph nodes* for its nodes, and for an agent the term *agent nodes* are used in this subsection to avoid confusion.

Different weights are used for the edges of the graph, depending on how far-fetched the current connection is. For example, connections between resources that are already in the query graph product are very favorable, since other agents nodes have already concluded that these must be included in the query. Similarly, edges between sub- and superclasses are light weighted edge since it is not particularly daring to refer to the properties of a superclass in a question, since the user is not necessarily aware of the classification although it is modeled in the ontology. Significantly higher costs, on the other hand, should be incurred by adding completely new entities to the question. While not absolutely infeasible, as few of them as possible should be created.

The computation of the weights is parameterized. For every type of edges, a constant, or a function f that is allowed to use the slots of the product from which it is generated can be specified (typical values are constants, or $1 -$ *confidence(n)*):

function	edge type
ov	Object valued relations in the ontology
lv	Literal valued relations in the ontology
f_{gv}	statement products, ClassVariable products, constant products of the query graph product ,
f_{iv}	interconnecting edges between ClassVariable products of the query graph product and classes in the the ontology
clv	Class hierarchy relation

The exact values for the weights of edges are determined by each agent node through mutation, however, the order for actually successful used agent nodes is always as follows (which makes the

learning somewhat obsolete):

$$f_{iv}(\textit{confidence} = 1) < f_{gv}(\textit{confidence} = 1) < clv < lv < ov$$

A confidence graph $cg := cg(g)$ to a query graph g is constructed by the following steps:

1. Create a new empty graph cg
2. Add all classes and literal datatypes of the ontology as graph nodes to cg.
3. Connect each subclass graph node with its superclass graph nodes and vice versa with an edge with weight clv.
4. For each property of the ontology, create directed edge(s) between the graph nodes representing domain and range, labeled with the property name: for every $md \in MD$, an edge labeled with $md.property$ from $md.class$ to each $getConcreteSubcls(md.range)$ is added. Parallel edges (sc, p_1, oc) and (sc, p_2, oc) are merged to $(sc, \{p_1, p_2\}, oc)$. The weight depends on the range of the property, for object-valued properties the bigger weight ov is used while literal-valued properties get the smaller weight lv.
5. Each ClassVariable product cv in g is turned into a graph node as well; the confidence of the node is set to $f_{gv}(cv)$. For object valued ClassVariable products connect the created graph nodes with an edge labeled "*instanceOf*" to all graph nodes representing the classes of its domains. Otherwise if the ClassVariable product is a literal with graph nodes representing this literal, respectively the same name. All these edges get the weight f_{iv}.
6. All constant products c are added as graph nodes as well, the confidence of the node is set to $f_{gv}(c)$.
7. For each statement product s of g, (property products and comparison products add an edge between the other parts (subject and object, or left and right) of the statement product. These edges are labeled with the property name or the operator symbol and get a weight $f_{gv}(s)$. Depending on the usage of the path, f_{gv} can e.g. be constant, proportional to the confidence of the statement, or antiproportional to it. With the former, products with a very high confidence give a high "value" and the min/max value on a path can be computed, and with the latter products with a very high confidence contribute low values to a sum, basically creating nearly free paths, while speculative products have a higher weight and will be rarely used in a optimal paths.

If the ontology itself is connected (which should in general be the case), then also the *path finding graph* is a connected graph.

4.2.7.5 Finding Possible Candidates for Missing Connecting Entities

The confidence graph is used for finding possible candidates for missing connecting entities and linking properties between them from the ontology, as illustrated in the following example:

Example 22 *Consider again the question from Figure 4.15, here with a different goal to illustrate. The question is "Where are more than forty-two percent of the population Hindu". In the question, the knowledge that "Where" refers to a country (or a province or something like that) is only implicit. Also the knowledge that "Hindu" is a religion is implicit. Moreover, the questioner knows that there is a percentage, but by far not how it is stored in the database (in this case, in an entity type that reifies the relationship "believe in"). The confidence graph provides all such connections of the ontology, and it can be used to find them correctly.*

The input query graph product g only contains the obvious products directly derived from query text: The literal ClassVariable products percent6 (number) and population9 (number) and the triple product (Hindu11 :name name11) with the ClassVariable products hindu11 (Religion) and name11 (String). Which means, that this query graph product consists of three disconnected subgraphs. Figure 4.22 shows the relevant fragment of the graph containing the Mondial ontology and the products g.

With only that few information, population could be a literal property of a country, a city or a province, while percentage could only be the literal-valued property of a reified relationship between country and either religion, continent, language, or ethnic group. Choosing the connecting paths with summing up the costs for each path (i.e., counting edges that are used in multiple paths also multiple times) does not deliver a desired result, since any of the shown reified relationships connecting population9 - Country - percentage6 could be added for the same costs. In contrast, reusing the "believed In" relationship when connecting population and percentage, which is also the shortest path from population6 to Hindu11, and from Hindu11 to percentage6 yields the cheapest solution in terms of the sum of the edges costs.

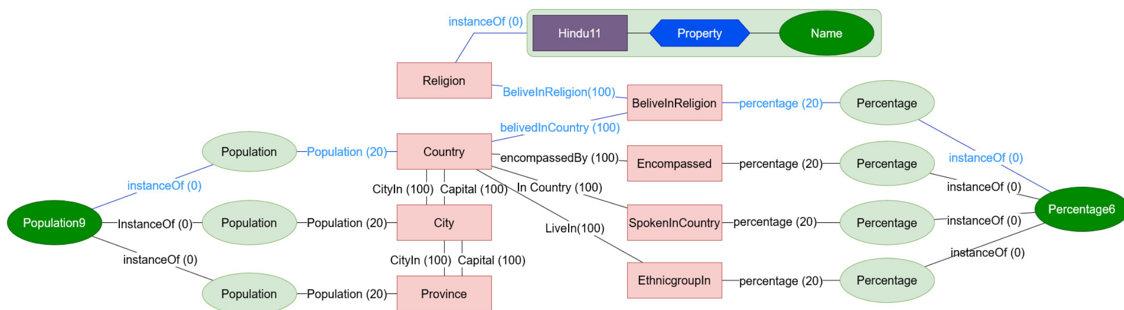


Figure 4.22: Example Graph for the question from Figure 4.15
Dark green nodes represent literal-valued ClassVariable products from the question graph g, purple ones represent object-valued ClassVariable products from g, red rectangles mark classes from the ontology, and light green ovals mark literal-valued ClassVariable products. Each line represents a property between the nodes and its respective weight is noted in parentheses behind the name. The edges forming the path with the lowest weight have blue labels.

For finding possible candidates for missing connecting entities in a query graph g, its set of maximal connected subgraphs S is computed, and also the confidence graph $cg(g)$ for a query is built. Intuitively, both together will be used to extend the query with selected paths from $cg(g)$.

For this, first, Yen's algorithm [77] (see Section 2.3.2.1 on page 41 for a brief explanation) is applied to find the k-shortest paths, between each pair of subgraphs in S, wrt. confidence graph cg. Note that if $|S| > 2$, then a path from s_x to s_y can be chosen, which contains one or more *graph nodes* of another s_z even if $x \cap y \cap z = \emptyset$ holds. Thus, for every such (unordered) pair x, y, a set $P_{x,y}$ containing k paths connecting them is computed. Altogether, $S \cdot (S-1)$ sets are built, where $P_{x,y}$ and $P_{y,x}$ contain mutually inverted paths due to symmetry.

Afterwards all combinations containing one path from each path set –modulo symmetry– i.e., for every pair (x, y), only one of $P_{x,y}$ and $P_{y,x}$ is used) are computed (note that this such a combination then is usually not a connected path, but just a set containing $S \cdot (S-1)/2$ paths that connect the subgraphs), i.e. $k^{S \cdot (S-1)/2}$ such combinations. Each combination can be seen as a graph (where maybe some of its edges are contained in multiple of the contributing paths). For each such combination, the weights its edges are summed up (i.e., the cost of edges used by multiple paths in such a combination, are not added multiple times); as illustrated in Example 23 below. This ensures that paths which add as little as possible to the existing information and use content that is already known have lower weight than paths which add a lot of new content.

The idea here is: The less content is added the less likely a wrong assumption is made and to keep a query human-understandable, it could not leave out arbitrary many entities, therefore adding as less as possible should be closer to the implications of the query.

Example 23 *To illustrate this, lets assume a query graph product g is given, which contains 3 separate maximal connected subgraphs a, b and c. Then the k shortest paths are computed to reach from each subgraph to all other subgraphs, so the path sets $P_{a,b}$, $P_{a,c}$, and $P_{b,c}$ are created. Therefore each such $P_{x,y}$ contains the k shortest paths, denoted $P_{x,y}[i]$ with $i = 1, \ldots, k$. Now assume $k = 3$, with the sample costs:*

Paths combination	*Total cost*	*cost with reused*
$P_{a,b}[1], P_{a,c}[1], P_{b,c}[1]$	15	15
$P_{a,b}[2], P_{a,c}[1], P_{b,c}[1]$	15	15
$P_{a,b}[1], P_{a,c}[2], P_{b,c}[1]$	16	16
$P_{a,b}[1], P_{a,c}[1], P_{b,c}[2]$	16	15
$P_{a,b}[2], P_{a,c}[2], P_{b,c}[1]$	18	14
$P_{a,b}[2], P_{a,c}[1], P_{b,c}[2]$	18	12
$P_{a,b}[1], P_{a,c}[2], P_{b,c}[2]$	21	20
$\vdots$	$\vdots$	$\vdots$
$P_{a,b}[3] + P_{a,c}[3] + P_{b,c}[3]$	45	41

For each path combination the total cost value is simply the summed cost of each path, while the "cost with

reused" imposes only one-time charges for edges that are used multiple times. Therefore the combination of the three shortest paths has of course the lowest sum, but since both metrics compute the same value, this path combination never reuses a path. Other then the path $P_{a,b}[2] + P_{a,c}[2] + P_{b,c}[1]$ which has actually a higher total cost but a lower cost with reuse value, so it uses some edges multiple times, which means, that overall less new content would be added to g if this path combination is realised, which make this path the most favorable in this case.

The result is a *path network collection* – i.e., a collection of *path networks*, where every individual path network is sufficient to make the graph connected (so one of them can be chosen later); so it can be called a *spanning network* for the underlying query graph g. Obviously, every path network consists of several paths. And the cost of such a path network –consider it as a train lines network– would also profit from the use of edges in multiple lines.

Basically a *path network collection* is a list of lists of lists of edges:

(1) **Collection:** An (unordered) list of networks (above: "combinations")

(2) **Network:** every network is an (unordered) list of $S \cdot (S-1)/2$ "connecting" paths $P_{x,y}[i_{x,y} \in 1..k]$,

(3) **Path:** Each $P_{x,y}[i]$ is an (ordered) list of edges.

As motivated above, level (2) (network) can be omitted, seeing every network not as $S \cdot (S-1)/2$ paths, but just as a list of edges. Denote each such *spanning network graph* by $span(i)$ with $i = 1..k^{S\cdot(S-1)/2}$.

Spanning Network Collections From a higher level aspect, every such $span(i)$ together with the query graph represents a way to connect all ClassVariable products of the query via edges from the query, or if not connecting completely, by notions of the underlying ontology. Note that by this, it is a *spanning graph* for the query using the ontology, but usually not a minimal spanning graph (components $a, b, c \in S$ can have direct connections (a, b), (a, c), and (b, c)).

So, the result is called *Spanning Network Collection* (although, it in fact contains a set of such graphs, from which one will be used for a given query). It will be used in the following for finding additional, implicit connections between notions used in the query.

The *Spanning Network Collection* is stored in a path network collection product (which thus contains graphs, but from the usage point of view represents connecting paths), which for that reason are mathematically elements of type $\mathcal{P}(\mathcal{P}(TR))$.

4.2.7.6 Spanning Networks Generator

The *Spanning Networks Generator* nodes are responsible for finding possible candidates for missing connecting entities. In a first step, it checks whether the received query graph product g is contiguous. If this is the case, it is forwarded unchanged, if not, its set of maximal connected

subgraphs S is stored and its path finding graph cg is created. Then, the above *spanning network collection* is built (i.e., the edgs are just translated into triple products):

For each path network $span_i$, for all its edges (scn, pn, ocn) (sc and oc are the subject and object class names, pn is the property name from the ontology), the according triple products

$$\begin{aligned} t := \; & TR([subject := CV([name := scn,\ domains := \{scn\},\ confidence) = 0.05]),\\ & predicate := PROP([name := pn,\ names := \{p\},\ confidence) = 0.05]),\\ & object := CV([name := ocn,\ domains := \{ocn\}, confidence) = 0.05]), \end{aligned}$$

are generated. If there is also an edge (that connects the ontology edge with a the query graph) $(cv, \text{"instanceOf"}, scn) \in span_i$ for some ClassVariable product cv, then replace the subject of t with cv, analogously for the object position. Then, all products of $span_i$ are collected in a set $paths_i \in \mathcal{P}(TR)$. Finally, the path network collection product $sncoll := \{paths_1, \ldots, paths_n\}$ (n stands for $k^{S\cdot(S-1)/2}$ from above) is created and is added to g. Later, other nodes will decide which of those possible graphs should be added to g.

Parameter:	gv - weight factor of edges from inside the query graph product (default 0) ov - weight of edges between object-valued resources (default 100) lv - weight of edges between a literal and an object-valued resource (default 20) clv - weight of edges in the class hierarchy (default 0) iv - weight of interconnecting edges between resources from g (ClassVariable products) with the spannning network (classes of the ontology) (default 0) k - Maximal number of path alternatives to be considered
Input Conduits:	1. **Query graph product** - in - possibly disconnected graph which should be turned into a contiguous one
Formal Operation:	$g \mapsto$ if g is contiguous (or contains already a Spanning Network Collection that is up to date) then $out := g$ else $sncoll :=$ the Spanning Network Collection wrt. g (and its confidence graph cg) as described above $out := QGP([in.components := g.components \cup sncoll])$
Output Conduits:	1. **Query graph product (+ Path network collection product)** - out - the input query graph product g, extended with the spanning network collection if g was not contiguous

4.2.7.7 Connections Choice Node

The Connections Choice Node type contributes to developing non-connected query graph products into connected ones. In an advanced stage of the processing, non-connected query graph products have been enriched by an above Spanning Network Collection Generator Node with a Spanning Network Collection (in form of a path network collection product) which actually contains multiple alternatives to make the query graph products connected.

The Connections Choice Node selects a spanning network from a path network collection product according to its strategy which is set as a parameter.

The node executes changes on the received query graph product according to its strategy, depending on whether the graph is afterwards contiguous or not, it is passed to the appropriate output.

Shortest Path Strategy This strategy takes the path network collection product whose summed weight is the lowest. The weights are anti proportional to their confidence. Since the path network collection product already consists of the k-shortest paths, the first path can be taken simply.

Quality Path Strategy This strategy calculates the average confidence value for each path and selects the path with the highest average confidence.

Union Strategy The approach of this strategy is to merge paths which use different edges with the same costs between the same nodes and make unions out of their values. The paths that include the most other paths are selected. Unlike the previous strategies, the Union Strategy tries to broaden the possibilities for other nodes rather than restricting them.

Parameter:	S: Strategy to decide how to pick a specific path
Input Conduits:	1. **Query graph product** - g - which is maybe disconnected and contains with a path network collection product
Formal Operation:	$g \mapsto$ $sncoll := g.components \cap PColl$ if $sncoll \neq \emptyset$ and not $g.isConnected()$ then evaluate all $sn \in sncoll.get$ according to strategy S, $snbest :=$ the best one (ties broken arbitrarily), $out := g \cup snbest$ else $out := g \cup snbest$
Output Conduits:	1. **Query graph product** - g - without the path network collection product but connected according to the strategy S

4.2.7.8 Confidence-Based Graph Reducer Node

Nodes that generate results on a very speculative basis should, at least in the long term, learn to give them a very low confidence value. Depending on whether there are better alternatives for these products or not, these must then be kept or removed for the final result.

The idea of the *confidence-based graph reducer node* is to remove all edges of a query graph product g that are not necessary to connect those parts of the graph where the agent is sure that they are part of the solution.

The *confidence-based graph reducer node* does this by creating the confidence graph cg as described above but without adding any content of the ontology. Instead, the graph edges get a non-zero weight: the edge weights function f_{gv} is anti-proportional to the confidence value of the statement product s that they represent: $f_{gv} \leftarrow (1 - s.\mathit{confidence})^w$ is used, where w is a learned factor that is supposed to regulate how much the confidence value is weighted.

Then, the cheapest paths (i.e., the most reliable ones) are calculated between all ClassVariable products that meet or exceed the confidence threshold ct. This might result in multiple paths with the same cost for some ClassVariable products, all of them are treated as part of a shortest path. All edges that appear in one of these paths are marked as necessary.

Then, all edges in cg and their associated triples in g that have a confidence value $< ct$ and are not marked as necessary are removed. So far, only edges were removed, so non-connected nodes can result from this computation. This problem is passed on to other nodes and with it the decision whether to include them again or to remove them.

Parameter:	$ct \in [0, 1]$ - confidence threshold to decides at which value a connection is considered to be sufficiently reliable. $w > 0$ - weighting of the confidence value
Input Conduits:	1. **Query graph product** - g - which should be reduced to necessary parts
Formal Operation:	$QGP \rightarrow QGP$ $in \mapsto ConfidenceRemove(in, ct)$ *ConfidenceRemove* see Algorithm 8
Output Conduits:	1. **Query graph product** - input query graph product without connections that have a confidence value below ct and are not necessary to connect parts with a confidence value $\geq ct$

Algorithm 7: isConnected(QGP)

Result: true if the whole GraphProduct is connected
Input: GraphProduct g
$firstNode \leftarrow components(g).first()$
foreach $node \in components(g)$ **do**
  $distance \leftarrow kruskal(node, firstNode)$ // [76]
  if $distance = \infty$ **then**
    return *false*
return *true*

Algorithm 8: ConfidenceRemove(QGP)

Result: Returns query graph product without unnecessary low-confidence statement product
Input: GraphProduct g**, threshold** ct **[0,1]**
▷ for isConnected see Algorithm 7
create confidence graph cg from g
foreach *maximal connected subgraph sg in cg* **do**
  $belowThresholdList \leftarrow \{c \in edges(sg) : confidenceValue(c) < ct\}$
  foreach $\{c_1 \in nodes(sg) : c_1.confidence \geq ct\}$ **do**
    foreach $\{c_2 \in nodes(sg) : c_2.confidence \geq ct \wedge c_2 \neq c_1\}$ **do**
      $conn := sg.findPath(start := c_1, end := c_2))$
      $belowThresholdList := belowThresholdList/edges(conn)$
  foreach $e \in belowThresholdList$ **do**
    remove the corresponding statement from g
return g

4.2.7.9 Subgraph Merge

The *Subgraph Merge* node tries to join disconnected graphs by finding nodes that could potentially describe the same entity.

It does this by searching for looking-for-replacement products and examining the referenced triple t_1 whether it is an object-valued relationship. If so, a search is made in every other subgraph for a triple product t_2 in which the contained property products match (i.e., have non-empty intersection of names) and the domains of subject and object of t_1 and t_2 each have a non-empty intersection.

If there are several such triples (throughout all other subgraphs), the one with the largest intersection, and then with the highest confidence, is chosen.

The thus determined triple, should there be such a triple, then replaces all TriplePart products of t_1 by the found triple, and the looking-for-replacement product that contained t_2 is removed from the query graph product. With the replacement, these two subgraphs are now connected. This is executed for all further subgraphs.

Parameter:	none
Input Conduits:	1. Query graph product - in - which might contain mergable subgraphs
Formal Operation:	$QGP \to QGP$ $g \mapsto MergeSubgraph(g)$ //see Algorithm 9
Output Conduits:	1. Query graph product - out - with possible merged subgraphs and some looking-for-replacement product removed

Algorithm 9: MergeSubgraph(QGP)

```
Result: Returns query graph product more connected by possible connections via LFR
Input: GraphProduct g, minCoverage ∈ [0, 1]
subgraphs ← all maximal connected subgraphs of g
potentialConnections ← new List
foreach g1 ∈ subgraphs do
    foreach l ∈ g1.components ∩ LFR do
        foreach g2 ∈ subgraphs : g2 ≠ g1 do
            foreach cv ∈ g2.components ∩ TR do
                domainCoverage ← |cv.subject.domains∩l.content.domain| / |cv.subject.domains∪l.content.domain|
                if domainCoverage ≥ minCoverage then
                    potentialConnections.add((domainCoverage, l, g1, cv, g2))
Sort potentialConnections descending by domainCoverage
foreach p1 ∈ potentialConnections do
    g.replace(p1[2], p1[4])
    // remove connection candidates between subgraphs that just have been connected
    foreach p2 ∈ potentialConnections : g.findPath(p2[2], p2[4]) ≠ ∅ ∨ p1[2] = p2[2] do
        potentialConnections.remove(p2)
return g
```

4.2.7.10 Conflicting Literals Solver Node

While it would be allowed in SPARQL to use the same literal-valued variable in multiple triples (with different subjects), that is not something EvolNLQ does or allows.

Example 24 *This example illustrates some of the limitations of EvolNLQ regarding the translation into SPARQL:*

For the query "Which city has the same population as Liechtenstein", the literal "population" can be used in multiple triples with different subject. While a shorter formulation is possible in SPARQL, in EvolNLQ the variables ranging over literals are always unique to an object-valued variable and constants are always used in a filter, not as part of a triple. The variations can be found in Figure 4.23.

Therefore, if more than one ClassVariable product have a relationship to a literal-valued ClassVariable product, one of the associations is actually wrong. This is often the case for triple products that have been introduced by guessing-nodes like Relator Nodes. Thus, in such situations, it must be decided which of the triples is removed.

Resolving these conflicts is not trivial, so the Conflicting Literals Solver Node has several strategies that can be executed depending on the parameter set. The idea in the design of the node is that a strategy only makes a change if it is promising or the criterion by which it is judged is applicable at all. Thus, different Conflicted Literals Solver nodes can be executed one after the other and thus a prioritization of the respective strategies can be realized.

Currently, the following strategies are implemented:

Grammatical Distance The grammatical relations are seen here as a graph, where the relation itself is an edge and the words it connects are the nodes in this graph. The distance is then calculated by determining the number of edges needed for the shortest path . The triple with the lowest distance amongst the conflicting ones is kept, the others are deleted.

This has proven to be a very good criterion as long as there are no equidistant candidates, which is very often the case. Therefore, this strategy is used by many agents before other conflict solvers.

Nearest Strategy This is the simplest stategy, but not necessarily the most reliable. If there is a conflict, the node whose position is closest to the disputed literal is taken.

Conjunction Strategy The Conjunction Strategy removes all conflicting triples that have a Grammatical Relationship of type "conjunction" with the conflicting literal. Such relations arise for example in Example 25: Literal-valued ClassVariable products exist there for name3, population5, and area7. Reasonable triples having name3 as object could have been created for capital4 or country10. The Nearest Strategy would associate the name4 with capital10, although they are only in the enumeration of properties of Country. The Grammatical Distance would assign a distance of 1 for (name4 - conj:and - capital5) and (name4 - nmod:of - country10). The conjunction strategy would remove the former and –correctly– keep the latter.

So this strategy deals with problems which occur if an enumeration of properties contains object-valued and literal-valued items which are transformed into ClassVariable products, of which both the actual modifier and a modifying property, share other modifying properties. While this strategy does only solve that rather uncommon special case, it does not disturb anything else, as far as observed, so it might be one of the first solving strategies.

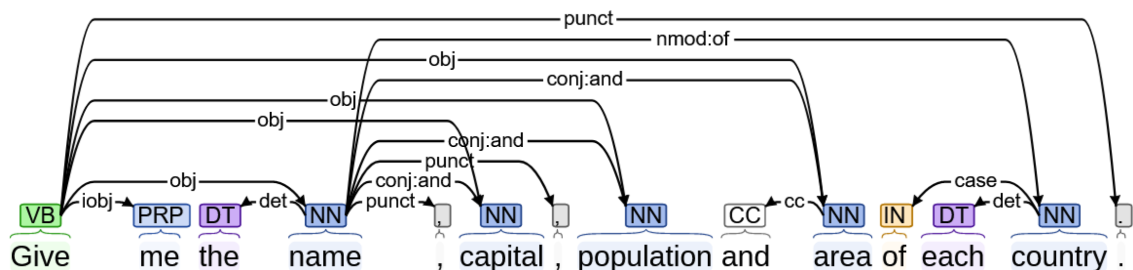
Grammatical Relation Removal Strategy Removes conflicting triples which have a grammatical relation gr between subject and object of the specified type. Note that the above ConjunctionStrategy is an instance of this where $gr = conj{:}and$.

Compound Gov First, Compound Depend First & Compound Distance First These three strategies can be applied to cases where the subjects of both conflict triples are in a Grammatical Relationship of type "compound". Preference is given to triples that have either their subject in the governor or the dependant part of the relation or which one is closer.

Competing Objects Strategy Strategies in the above style can also be applied for object-valued relationships, instead of literal-valued ones. This situation is not necessarily a conflict (i.e., such graphs would correspond to some usages of pronouns), but sometimes it can help to remove excess triples.

Parameter:	Strategy S = one Of {Grammatical Distance, Nearest Strategy, Conjunction Strategy, Grammatical Relation Removal Strategy, Compound Gov First, Compound Depend First, Compound Distance First, Competing Objects Strategy}
Input Conduits:	1. **Query graph product** - in - which might contain conflicted TriplePart product
Formal Operation:	$QGP \to QGP$ $in \mapsto$ application of one of the strategies $=:$ out
Output Conduits:	1. **Query graph product** - out - input, some triples removed according to S executed

Example 25 *CoreNLP output for: "Give me the name, capital, population and area of each country."*



4.2.7.11 Redirect Grammatical Relation Node

The Redirect Grammatical Relation redirects a grammatical relationships instance gr from one word in the query to another one.

This plays a particularly important role with pronouns. Pronouns usually have a *conj:and* relationship with their noun in CoreNLP, just like enumerations connected by *conj:and*, or sometimes they

A valid way to ask the query in SPARQL

```
SELECT ?city
WHERE{
    ?country4 :population ?p .
    ?city6 :population ?p .
    ?country4 :name "Liechtenstein"
}
```

How EvolNLQ has to construct the query

```
SELECT ?name6
WHERE{
    ?country4 :population ?population4;
            :name    ?name4.
    ?city6 :population ?population6.
            :name    ?name6
    FILTER(?population4 = ?population6)
    FILTER(?name4 = "Liechtenstein")
}
```

Figure 4.23: Short valid SPARQL query in comparison to the variant of EvolNLQ for the query "Which city has the same population as Liechtenstein"

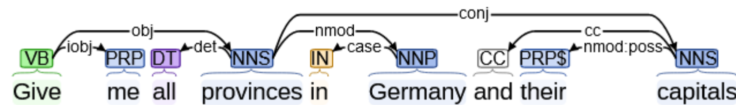


Figure 4.24: Example Query with a pronoun which relates to the more distant noun

do not have a relation at all. Note that *conj:and* does not contribute to calculate the grammatical distance, so the redirection does not change anything.

Nevertheless pronouns almost always refer to the subject of the sentence. Thus, the pairings between pronouns and nouns can be determined directly by the Part Of Speech tags. Usually, agents learn in the middle of their evolution that NNS and PRP$ are a very good combination for this substitution if $position(NNS) < position(PRP\$)$.

In most cases this plays a rather minor role, since incorrect assignments can often be avoided by validating the results with the ontology, other than in the following example:

Example 26 *Consider the question in Figure 4.24 based on the Mondial ontology: "capital(s)" either could relate to "Germany" or to "provinces".*

The Word Count distance is greater to province than to Germany and the conj relation is not reliable and therefore not used for the distance determination, which means the grammatical distance is infinite in both cases. This would make a Conflict Solver Node assign "capital" to "Germany". However, by changing the nmod:poss("capitals","their") Grammatical Relationship from the pronoun to the subject noun, this nmod:poss is rewritten to nmod:poss("capitals","provinces"), and the grammatical distance then leads to a correct assignment.

Parameter:	none
Input Conduits:	1. **Positionable product** - in_1 - which should be replaced 2. **Positionable product** - in_2 which should be replacement 3. **Grammatical Relationship** - in_3 - relation, in which the replacement should happen
Formal Operation:	$POS \times POS \times GR \rightarrow GR$ $in_1 \times in_2 \times in_3 \mapsto$ if $in_3.gov = in_1$ then $GR([gov := in_2, dep := in_3.dep, type := in_3.type])$ else in_3
Output Conduits:	1. **Grammatical Relationship** - with the replacement if applicable otherwise in_3

4.2.7.12 Group By Node

Detecting a "group by" semantics is not an easy task, but a certain constellation of grammatical relations in combination with an aggregator is a reliable indicator that a "group by" operation is necessary.

This node is a consequent implementation of the "look and recognize" principle mentioned in the introduction. Here a regularity was recognized and implemented as a node, the usability or the restrictions are thus the problem of the agent or must be learned.

The exact circumstances are described in the *Formal Operation* row of the following table.

Parameter:	Grammatical Relationship type: $verbToAgg, verbToG_1, aggToG_2$
Input Conduits:	1. **Query graph product** - in - which might need a group by product added
Formal Operation:	$QGP \rightarrow QGP$ $g \mapsto$ if $\exists\, agg, verb, g_1, g_2 \in components(g) \cap TP$: $GR(verb, agg, verbToAgg) \in NC_{Gr} \wedge$ $GR(verb, g_1, verbToG_1) \in NC_{Gr} \wedge$ $GR(agg, g_2, aggToG_2) \in NC_{Gr} \wedge$ $agg \in AGG$ then $out := QGP([components := g.components \cup$ $GRP([aggregation := agg, elements := \{g_1\}])])$
Output Conduits:	1. **Query graph product** - out – same as input, extended with resulting group by products added

4.2.7.13 Trim Graph Node

This node has two different modes to clean up a query graph product, depending on its mode parameter, it either executes one or both function.

Projection Mode:
Different nodes may find a TriplePart product suitable for selection for being output and bind it to a corresponding projection product. Due to the time-dependent structure of EvolNLQ, it is not always possible to ensure that this selection is actually valid since there might be other products created later which turn this projection product wrong. This especially occurs when ClassVariable products which are selected, are later modified by an aggregation product and then the aggregation product has to be selected instead.

Therefore, it is necessary to check in a later stage, after merging the products, whether there are projection products that reference a variable which is also part of an aggregation product which in turn is the content of another projection product and is used in another aggregation product. Then the former projection product is deleted.

Disconnected Mode:
In addition, the Trim Graph Node can examine a query graph product to see if there are any TriplePart products that are not used in any statement product of the query graph. These are removed if the parameter is set to "disconnected" or "both".

Parameter:	$mode$ - {projection / disconnected / both}
Input Conduits:	1. **Query graph product** - in - which should be trimmed
Formal Operation:	$QGP \rightarrow QGP$ $in \mapsto out :=$ $sub_1 := \emptyset,\ sub_2 := \emptyset,$ if $mode =$ "projection" or $mode =$ "both" then if $\exists pcv, pagg \in components(in) \cap PROJ,$ $cvagg \in components(in) \cap AGG :$ $pcv.components[0] \in CV \wedge cvagg = pagg.components[0] \wedge$ $cvagg.components[0] = pcv.component[0]$ then $sub_1 := \{cvagg\}$ if $mode =$ "disconnected" or $mode =$ "both" then $sub_2 := \{tp \in in.components \cap TP : \nexists s \in in.components \cap STMT :$ $tp \in s.components\}$ $out := QGP([components := in.components \setminus (sub_1 \cup sub_2)])$
Output Conduits:	1. **Query graph product** - out - trimmed version of in

4.2.7.14 Triple Fusion Node

Some questions explicitly mention all three parts that are needed to form complete triple products. This can usually not be detected in a single step using any kind of "local view" on the question – note that there is no node that directly creates a completely specified triple.

As described above, since many questions do not explicitly mention all three parts that are needed to form complete triple products, the predicate or the object must often be estimated, as done by the diverse guess-based nodes.

If actually all three parts of the triple product are present, the nodes which guess triples based on only one or two components still produce those, which are –obviously– less restrictive than they could have been when knowing everything. In particular if the predicate ranges over several classes and/or the domains of subject and object have more than one potential property to be connected with, the guessed predicate is a larger union than it should be, and the guessed domain of the object is larger than it should be.

In such cases, some three words, e.g., a (subject), b (predicate), c (object) of the question correspond to the completely specified triple. The guess-based relator nodes create all possible combinations (based on the grammatical annotations), so they will have guessed a property p and a triple $(a\ p\ c)$ and an object o and a triple $(a\ b\ o)$ (without knowing that these will fit together) In fact, the complete information is scattered over these.

The Triple Fusion Node merges such triples and restricts their TriplePart products (i.e., p and o) accordingly.

Therefore it first checks if some of the triple products was created based on subject and predicate and the other one based on subject and object, then it checks that the intersections of the properties and the domains of the objects are both not empty, but also not exactly equal (in this case simpler collector nodes would be able to merge it), and that there is not a distinctness product to prevent the merging of any of the relevant subproducts. Then, a triple product is generated from the shared subject and the predicate and object which are the intersections of the specifications.

Example 27 *Consider the query "Give me all capital cites of all provinces" and an agent in a state, where it already found ClassVariable products city4 and province7 of their respective types and "capital" was correct mapped to the property capitalOf.*

If Relator Nodes (see Section 4.2.4.1 on page 129) receives all three TriplePart products, the node produces two triple products, (city4, capitalOf3, o[domains={country,province}), and (city4, p[names={capitalOf, cityIn}], province7) each with a derived part which is not unequivocal in it.

This is were the triple fusion node comes in and merges both triples into one by combining the information that has a high confidence, and disregarding the guesses. The process is also illustrated in Figure 27.

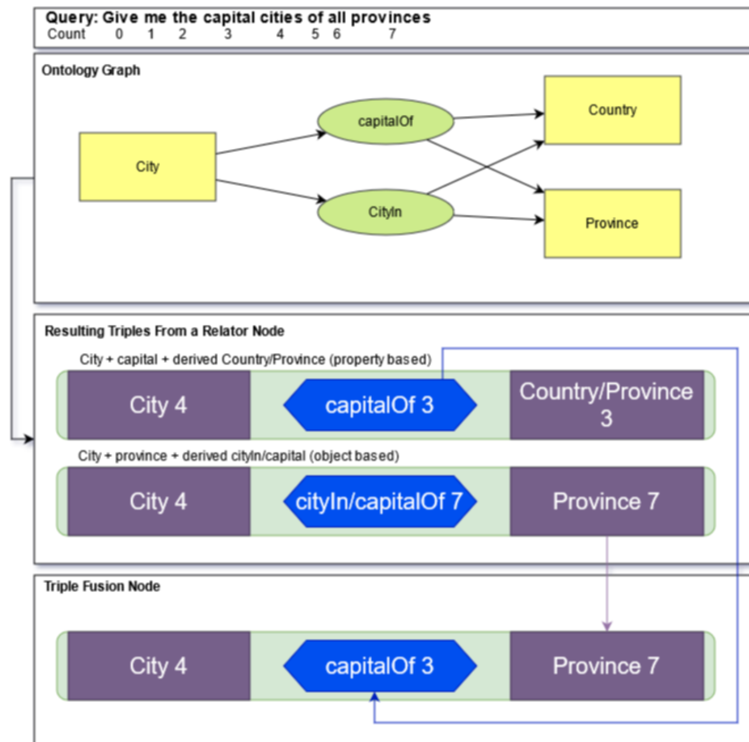


Figure 4.25: Example how a *Triple Fusion Node* (see Section 4.2.7.14) reduces the domains and properties of TriplePart products, when all parts of a triple are explicitly mentioned.

Parameter:	none
Input Conduits:	1. **Query graph products** - in- a graph potentially containing triples which might describe the same relation but the information content of the TriplePart products is different
Formal Operation:	$QGP \to QGP$ $(in \mapsto$ for each pair (t_1, t_2) such that $t_1.subject = t_2.subject \land$ $t_1.predicate.names \subsetneq t_2.predicate.names \land$ $t_2.object.domains \subsetneq t_1.object.domains :$ $out_1 := TR([t_1.subject, t_1.predicate, t_2.object])$, for each pair (t_1, t_2) such that $t_1.subject(in_1) = t_2.object \land$ $t_2.object.domains \subseteq \mathcal{CLS} \land$ $t_1.predicate.names \subsetneq$ $\{md.getInverse(name) : name \in t_2.predicate.names\} \land$ $t_2.subject.domains \subsetneq t_1.object.domains :$ $out_2 := TR([t_1.subject, t_1.predicate, t_2.subject])$, $out := QGP([components := in.components \cup out_1 \cup out_2])$
Output Conduits:	1. **Query graph products** - out - which contains a fused triple with the highest confidence

4.2.7.15 Reified View Handler Node

Attributed relationships need to be reified in RDF due to the concept of triple representation. Reification is a basic notion in modeling, for casual readers best illustrated by an example:

The information what percentage of a country is on a continent cannot be stored meaningfully for either individual, but only makes sense wrt. the relationship "encompassed by" between countries and continents. The relationship instance (:germany :encompassed :Europe) already requires a triple and no further property can be assigned to it. Note that

```
:germany :name Germany; :encompassed :europe; percent = 100.
```

would assign percent=100 to Germany, not to the relationship of being located in Europe.

So the correct modeling in RDF is

```
[ rdf:type :Encompassed;
  :encompassedArea :germany; :encompassedBy :europe; :percent 100 ]
```

where the reified class is named "Encompassed" and has properties "encompassedArea" ranging over countries, and "encompassedBy" ranging over continents.

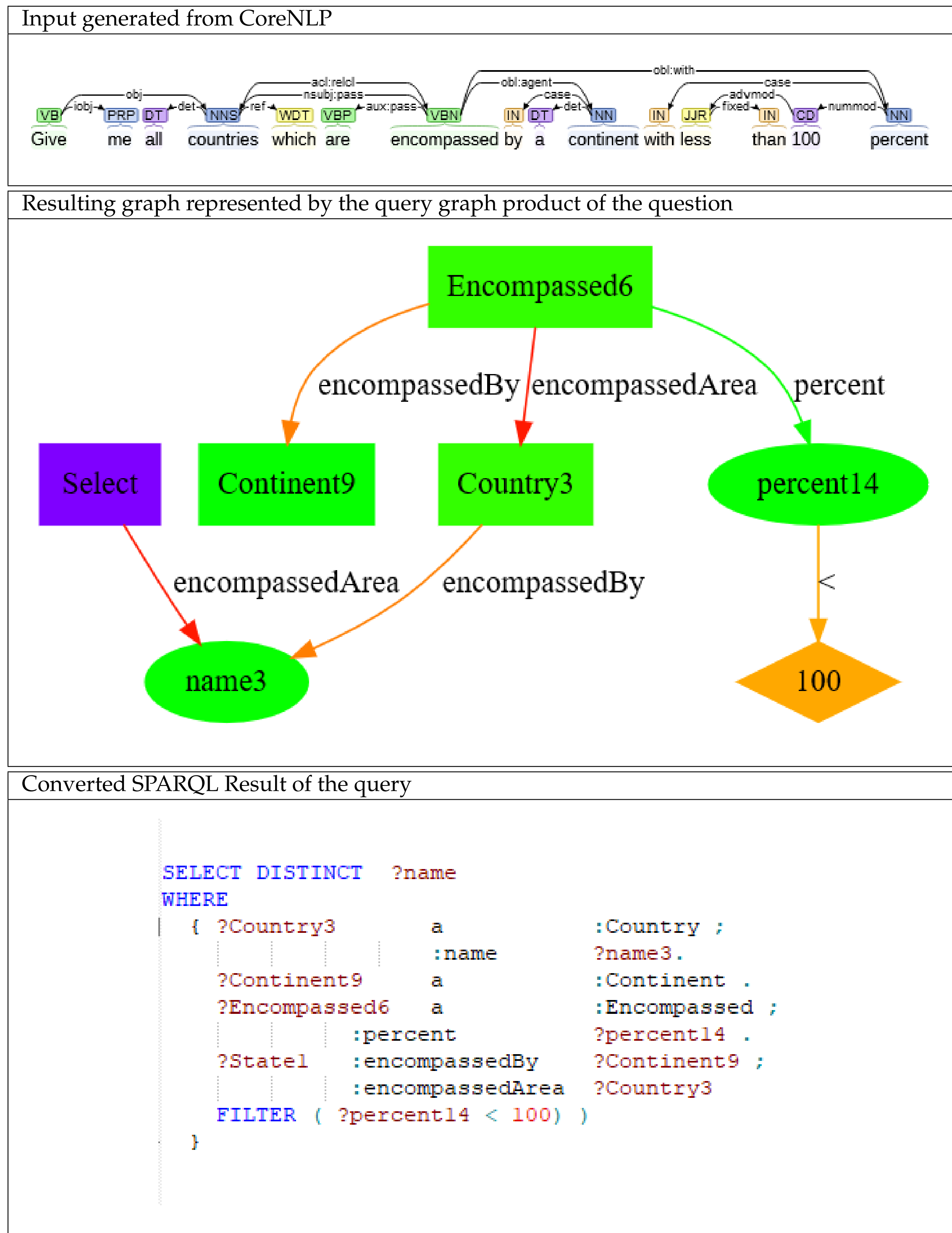


Figure 4.26: Question *MondialReified* with CoreNLP anntotations, query graph, and resulting SPARQL query

An example question with query graph product containing the generated reification triples is shown in Figure 4.26.

In the formal operation of the node, the variables match this example as follows:
The triple is ($Country encompassed $Continent) containing two ClassVariable products. The predicate's names are {encompassed} (which has the inverse "encompassed-"). Then, the Mapping Dictionary entries are searched (and found): md_1 *and* md_2 *are the Mapping Dictionary entries for the properties "encompassedArea" and "encompassedBy", defined on the class "Encompassed", ranging over "Country" which is in the domains of the subject, and "Continent" which is in the domains of the object.* $name_1$ *and* $name_2$ *are "encompassed" and "encompassed-" in any order, both stored in the table "Encompassed" (which is the class name of the reified RDF class), which is used as name for the new ClassVariable product. With this, the property products "encompassedArea" and "encompassedBy" are created, and the chain is generated by two triples ($Encompassed, "encompassedArea", $Country) and ($Encompassed, "encompassedBy", $Continent), which successfully connects $Country and $Continent.*

Therefore this relation has to be reified and another individual has to be introduced that has a relation to both the subject and object of the relation and some more literal properties that describe the relationship instances. In order to keep the original relation, the relation is defined by the Mapping Dictionary as a view.

These detours are not made by in the natural language (and also not by conceptual modeling with the Entity-Relationship Model [81]), since it is not restricted to triples. So this gap must now be overcome with special measures, recognizing in which cases a notion is a reified relation and how it can be resolved in triple form.

The view breaker node looks for triple products that have a predicate whose property in the Mapping Dictionary is marked as reified and then creates two more triple products, each connecting subject or object to the reification. Since this can only be successfully executed on relatively advanced graph products that are sufficiently completed.

Parameter:	MD : set of mapping dictionary entries
Input Conduits:	1. **Query graph product** - in - which might contain relation which can be expressed as reifications

Formal Operation:	$QGP \rightarrow QGP$ $in \mapsto$ $out := \emptyset$, for each $tr \in n.components \cap TR$: if $\exists md_1, md_2 \in MD$: $md_1.class = md_2.class$ // the reified class $\wedge\ md_1.property \neq md_2.property$ $\wedge\ MD.getConcreteSubcls(md_1.range) \cap s.domains \neq emptyset$ $\wedge\ MD.getConcreteSubcls(md_2.range) \cap o.domains \neq \emptyset$ $\wedge \exists name_1 \in tr.predicate.names$:$MD.isReificationOf(md_1.property, name_1)$ $\wedge \exists name_2 \in tr.predicate.names$:$MD.isReificationOf(md_2.property, name_2)$ $name_1, name_2$ are the reified property and its inverse then $r := CV([name := MD.getTable(md_1), domain := MD.getTable(md_1),$ $rr_1 := PROP([name := md_1.property, names := \{md_1.property\},$ $p_{min} := p.p_{min}, p_{max} := p.p_{max}])$ $rr_2 := PROP([name := md_2.property, names := \{md_2.property\},$ $p_{min} := p.p_{min}, p_{max} := p.p_{max}])$ $out := out \cup QGP([components := in.components \cup$ $\{triple([subject := r, predicate := rr_1, object := tr.subject]),$ $triple([subject := r, predicate := rr_2, object := tr.object])\}])$ else in
Output Conduits:	1. **Query graph product** - out - same as in but extended with appropriate tripels

4.3 Agents and Evolution

With the nodes and products presented in the previous sections, application-specific agents can be created. However, they still need an environment to develop and a simulation that coordinates the whole process and sets the framework parameters presented in Table 4.6. Among other things, there is set the set of environments included in the simulation and how many Questions each of these environments has.

So far, only the case of how an agent is created when it is created by another agent has been explained, but for the start of a simulation it needs an initial agent.

Option	Description
Runs	Number of runs performed until the training is finished.
Number of environments	Specifies the number of environments at the beginning.
Number of questions per environment	Specifies the number of random training questions per environment, but can be increased by environment fusion (see Section 3.7.2).
Question value	Specifies the value of $en(t)$, i.e. the maximum amount of energy per question.
Reduction for wrong answers	Specifies by which factor (< 1) the claimed energy reduced per incorrect result.
Mutation rate	Sets the probability that an offspring agent will mutate when it is created.
Mutation severity	Sets the mean value for the Gaussian-distributed number of mutations an agent will get in case it mutates.
Probability to be relocated to another environment	Sets how likely it is that an agent will be relocated from one environment to another.
Connections between nodes can only be forward	can be used to significantly reduce the computation time of agents without significant quality loss.
Input nodes are not removed	Determines whether input nodes are treated differently from other nodes, in that they cannot occur more than once, but are also not deleted, but are always present once in each agent.

Table 4.6: Description of the setting options available from the training's view

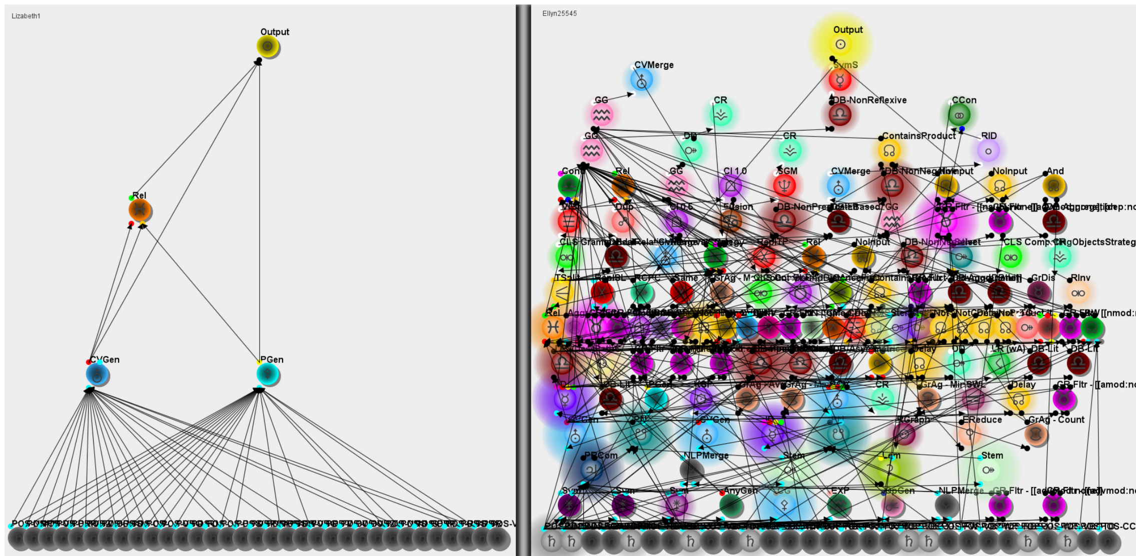


Figure 4.27: Illustration of the initial agent (left) compared to a fully developed agent (right).

4.3.1 The Initial Agent Species

The initialization of new agents, either at the start of a training run or in case that all agents become extinct, is created in *evolutionary algorithms* either with a fixed configuration or randomly. The latter makes sense if it is not clear how a viable agent is created, but that is not the case in this application. Therefore, initial agents for EvolNLQ are of a fixed configuration, but mutated with the given mutation probability before the first training run begins. An instance of the inital agent species is depicted in Figure 4.27 together with a highly developed agent to illustrate the development. The Initial Agent species' anatomy already consists of all Input Nodes, a ClassVariable Generator Node (described in Section 4.2.2.1 on page 115) and a Predicate Generator Node (described in Section 4.2.2.2 on page 115), both of which are connected to all inputs and each of which has a connection to a Relator Node (described in Section 4.2.4.1 on page 129). In addition, the ClassVariable Generator Node, Predicate Generator Node, and Relator Node have each a connection to the output node. While not for all requests, most should contain either a useful property or a class and give this configuration enough power to be able to father offspring.

In case that, for example, only individuals with synonyms of properties are connected, they would not find them, but at least over a period of time the mutations would find something to get enough energy to reproduce.

4.3.2 Final fusion

At the end of a simulation, a *swarm fusion* mutation is performed. Assume that – as usual at the regular end of a simulation– there is a single environment e that stopped (or is stopped) after some

run. Instead of starting another run, the set A of agents that would have been qualified to produce offspring are are used to build the "final" agent by *swarm fusion*. This fusion is done by creating an empty agent F and adding the nodes and connections of one agent of each species in A to it. For each input node type and parameter, one instance is kept, and all other nodes of the same type and parameter are removed and their outgoing connections instead connected to the appropriate output conduits of the kept instance. In the next step, an Ontology-based Filter Node ofn with the parameter "Graph Product" and a Product Comparison Filter Node pfn with the parameter "Highest confidence Value" are created. Then, all output nodes are removed and the links that lead to them are connected to ofn's input.

The later function of that combination is as follows: Each agent will send its results there, it filters and forwards only the graph products; usually one or two, rarely none from each agent (which are now subagent-like structures). From ofn's output, one connection is created to each of pfn's two inputs. pfn is a collector node, so its collects everything that comes in its $storage(1)$ and $storage(2)$ - both then contain *all* graph products. When there is no more activity in the whole agent (i.e. all the subagent-like structures are finished), it will be activated. Only the product(s) in its $storage(1)$ with the highest confidence pass and will be output as the result (in case that two or more results are output, it is up to the surrounding system which one is submitted to the database).

Since the agent F now has no output node, a new one is created and a connection is added from pfn's output conduit to that node. This realizes that the query graph product with the highest confidence is output.

4.3.3 Energy calculation

During the learning phase, the agents' rewards are computed directly on the products in their result set, since the reference solutions also consist of products, the comparison operation of the products can be used directly. Overall, the process is relatively simple, the only products that can bring actual rewards (but also punishments) are from the classes that are contained in the reference solution, i.e., the (compound) Statement class, with $100 *$ *similarity value* energy units, or from the (atomic) TriplePart products class (and its subclasses), each providing $10 *$ *similarity value* energy units. The similarity value is computed similarly for all products by comparing relevant slots and components and assigning a value of 1 if they are equal and a value of 0 if they are not equal. The average of these values then forms the similarity value. The concrete comparison methods are implemented in each case within the respective product class and only return a non-zero value when compared with a product of their class, since two different classes are regarded as incomparable. An exception are superclasses and subclasses here, the method is called in each case in both directions and the average is taken and a superclass makes no difference when compared with a subclass and evaluates normally, while the subclass returns a similarity of 0. Averaged, such products receive thus a similarity of at most 0.5. For two class variables this is the cardinality of the intersection of the domains divided by the cardinality of the union of the domains plus

one, if the position intervals are the same, and then divided by 2. Analogously, this is also done with properties, only with names instead of domains and for constants in this case with the value instead of domains, etc. The name is not compared here, because it is not actually important. The position is not very important either, but it is crucial if there is more than one variable of this class or property, so that it can be assigned correctly. When creating the test set, it should therefore always be considered at which position in the sentence this value can probably be inferred. However, the numerical value of the position itself is not so crucial, what is important is that there is the correct number of different instances, each with the correct relations. Therefore, up to a certain degree the positions can be remapped by the evaluation to the position in the solution, if the question already contains many correct elements. If it contains almost none or a lot of wrong elements, the mapping of the positions is hardly possible and will not be done. A triple or a comparison is evaluated by first determining the similarity values of its components, summing this up and then dividing by 3. The individual components of the statement were already worth energy units and are not counted again; the statement is much more valuable because it is usually also harder to create.

4.4 Learning Data

The general settings for a simulation have effects of different severity on its course, this will be examined in more detail in Section 6.1, but the possible settings are predefined here and listed in Table 4.6:

4.4.1 Test Sets

Equipped with nodes and products, the agents are now ready for their tasks. The tasks are stored in an XML document. Every task T and its reference solution S_T are stored together as subelements of a Query element.

For the training, test sets are necessary that contain as detailed solutions as possible, from which the fitness function can derive as accurately as possible how good the achieved results of an agent are. Furthermore, the training can also be accelerated significantly, if the task T is not just the question text and the CoreNLP annotations must be re-computed every time. Thus, the task T consists of a serialized form of the annotated question. While T can usually be easily serialized and stored, the solutions S_T are unfortunately manual work after a certain point: If one could reliably generate the solutions from T by a program, one would already have such a program that solves the task and the whole process would no longer be necessary.

The exact representation of S_T depends on the domain, but ultimately it should be a serialized form of the products that make up S_T. For evaluation of the agents, their products P_S and the products P_T as described in the serialization can then be compared by the fitness function.

The XML serialization of the training set for a query thus consists of an NL element and a SPARQL

element, that contain the serialization as described above. The NL element has as subelements (ii) for each word a POS subelement and (ii) for each grammatical relation a GR subelement, and an attribute QueryText, with the question text itself. The POS elements contain as attributes the Part Of Speech Type and optionally the Subtype, as well as the word in Lemma Form as Value and its pseudocount in the sentence as Position. The GR elements have as attributes (i) the type of relationship stored as Type, (ii) optionally a Subtype attribute and then the position of the Governor stored as From and the position of the Dependent stored as To.

The SPARQL element consists of a serialization of products. Atomic products that can be unambiguously created from a composite product do not need to be explicitly serialized. Primarily these are standalone operator products, constant products as well as property products and ClassVariable products, if they refer only to a property or a class and are not part of the selection. To name them anyway is also possible and sometimes makes the queries clearer for control. The individual elements are listed in Table 4.7.

Example 29 *In the following is shown how* T *and* S_T *are represented as an XML document for the question Geobase1 (the query is introduced in Example 12 on page 101) . Everything in the NL element is given to EvolNLQ while everything in the SPARQL Element, is used to create the solutions in* S_T*.*

```
<Query Name="GeoBaseQuery1" Type="Simple">
    <NL QueryText="which rivers run through States bordering New Mexico?">
            <POS-Component Type="WDT" Value="which" Position="0"/>
            <POS-Component Type="NNS" Value="River" Position="1"/>
            <POS-Component Type="VBP" Value="run" Position="2"/>
            <POS-Component Type="IN" Value="through" Position="3"/>
            <POS-Component Type="NNS" Value="State" Position="4"/>
            <POS-Component Type="VBG" Value="border" Position="5"/>
            <POS-Component Type="JJ" Value="New" Position="6"/>
            <POS-Component Type="NN" Value="Mexico" Position="7"/>
            <POS-Component Type="." Value="?" Position="8"/>
            <GR-Component Type="det" From="1" To="0"/>
            <GR-Component Type="nsubj" From="2" To="1"/>
            <GR-Component Type="case" From="4" To="3"/>
            <GR-Component Type="nmod" SubType="through" From="2" to="4"/>
            <GR-Component Type="acl" From="4" To="5"/>
            <GR-Component Type="amod" From="7" To="6"/>
            <GR-Component Type="dobj" From="5" To="7"/>
        </NL>
        <SPARQL>
            <Variable Name="River" Position="1" Class="River" Selection="true"/>
            <Variable Name="State" Position="4" Class="State"/>
            <Variable Name="New Mexico" Position="6" Class="State"/>
            <Constant Value="New Mexico" Position="6"/>
            <Triple>
                <Subject Value="River" Position="1"/>
```

```
            <Predicate Value="flowsThrough" Position="1"/>
            <Object Value="State" Position="4"/>
        </Triple>
        <Triple>
            <Subject Value="State" Position="4"/>
            <Predicate Value="border" Position="4"/>
            <Object Value="New Mexico" Position="6"/>
        </Triple>
        <Triple>
            <Subject Value="New Mexico" Position="6"/>
            <Predicate Value="name" Position="6"/>
            <Object Value="name" Position="6"/>
        </Triple>
        <Condition>
            <Left Value="name" Position="6"/>
            <Operator Value="=" BelongsTo="6"/>
            <Right Value="New Mexico" Position="6"/>
        </Condition>
    </SPARQL>
</Query>
```

Element	Attributes and sub elements		
Variable	Attribute	Content	
	Name	Name of the variable	
	Class	Domain(s) of the variable	
	Position	Position of the variable	
	Selection	True if variable should be in the select	
Constant	Attribute	Content	
	Value	Value of the constant	
	Position	Position of the constant	
Aggregation	Attribute	Content	
	Type	Type of the aggregation	
	BelongsTo	Name of the ClassVariable product which this aggregation is over	
	Position	position from which the aggregation can be derived	
	CVPosition	Position of the ClassVariable product of belongsTo	
Except	Attribute	Content	
	Begin	Position from which the except interval starts	
	End	Position at which the except interval ends	
Triple	Sub element	Sub element content	
	Subject	Attribute	Content
		Value	Name of the Variable
		Position	Position of the Variable
	Predicate	Attribute	Content
		Value	Name of the property
		Position	Position of the property
	Object	Attribute	Content
		Value	Name of the Variable
		Position	Position of the Variable
Condition	Sub element	Sub element content	
	Left	Attribute	Content
		Value	Name of the variable
		Position	Position of the variable
	Operator	Attribute	Content
		Value	Name of the operator
		Position	Position of the operator
	Right	Attribute	Content
		Value	Name of the variable or constant
		Position	Position of the variable or constant

Table 4.7: Elements and attributes of the sub-elements of the SPARQL-Element in a testset

Chapter 5

Implementation and Usage

The EvolNLQ implementation consists of three programs, one for each use case: The most minimal one is the query interface, which can either receive a question directly through the console or via a text field. Further it is used for initialization to create the necessary *SDD* entries for a given ontology.

Secondly, there is the *Agent Creator Panel*, where new agents can be created manually or existing agents can be examined and modified. Further, the *Agent Creator Panel* can answer all or specific questions of a test set and display the results in CSV and LaTeX for further display and analysis.

Finally, there is the training environment, where learning and simulations can be started.

5.1 Query Interface

The query panel is used to directly issue questions, i.e. for the actual use of the NLQ interfaces. Behind it, there is the best agent that has ever been created. The questions are first annotated by CoreNLP, translated by the agent, converted to SPARQL and finally evaluated against the data model. Not only the results are displayed, but also the internal product graph and the SPARQL query, in case the user wants to analyze or debug the process. As shown in Figure 5.1, the interface can be divided into three subelements.

Element A is the input field, there the question can be entered and the conversion to annotations from CoreNLP is displayed when ready. In area B the intermediary steps are displayed. The top bar shows the current activity of the program, since depending on the question it may take a moment to be analyzed and evaluated. The left part of B shows the SPARQL query when it is ready and the right element shows the graph of the products contained in the generated query. The lower right area C shows a table with the result of the query. In the upper left corner D there is still the possibility to switch to the Agent Creator Panel (described in Section 5.2) to be able to examine or change the agent used.

Figure 5.1: The Query Interface of EvolNLQ

A Input Panel
B Query Panel
C Result Panel
D Switch between Query and Agent Creator Panel

5.2 Agent creator panel

As shown in Figure 5.2, the *Agent Creator Panel* interface splits into several subinterfaces, which are briefly explained below. It performs three main tasks: First, it is intended for agent creation by the user and provides a simple clickable interface where new nodes can be added, connections between them can be drawn and parameters can be set. This is also possible via an XML serialization of the agent. Both variants have their advantages but mostly it is easier to create the agent in a graphical environment, because in the XML document it is harder to add connections using identification numbers, and the overview is missing.

More important than the creation of new agents, however, is the examination of existing ones in case that some (new) tasks are not satisfyingly solved. Since EvolNLQ does not develop its own node types but only chains them together and parameterizes existing ones, new node types must be designed to give the agents further tools to solve problems.

Identifying where the problem is located is not a trivial task and can be effectively supported by a GUI that can be used to analyze how, where and when the agent reaches a certain result or not. To simplify this, each node has a history that shows all its inputs and outputs, and if it is a graph product, also a graph representation of the results to allow for quick and easy analysis by the user.

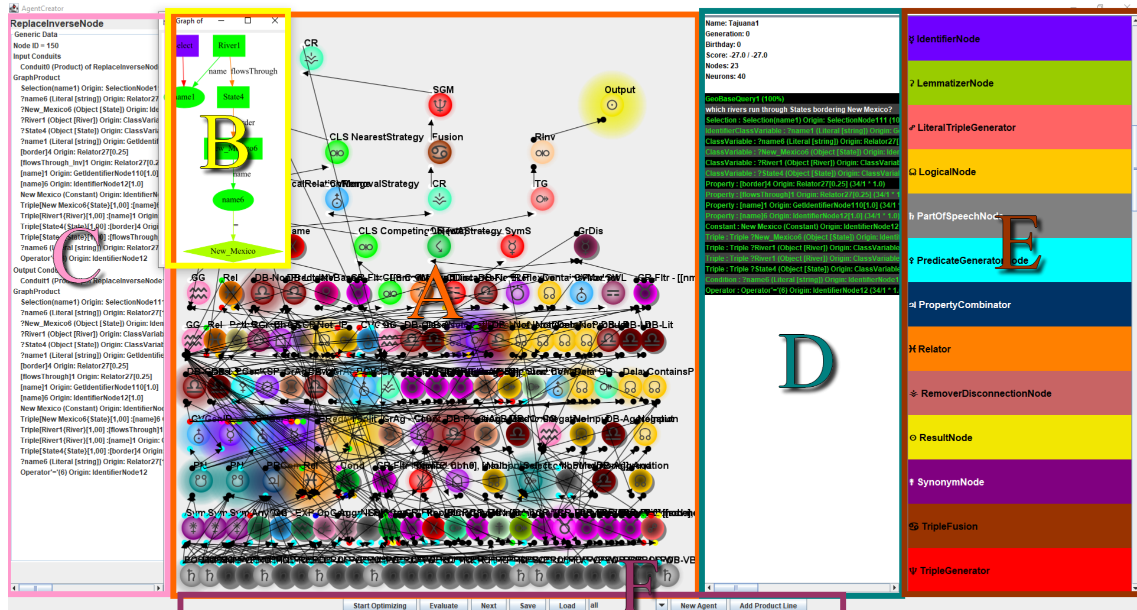


Figure 5.2: Labeled Overview of the *Agent Creator Panel*

A Interactive Agent Panel
B Graph representation of a query graph product
C Input and Output history of a node
D Agent status and evaluated result of a task
E Node creation selection
F Operation Buttons

5.2.1 Interactive Agent Panel

The Interactive Agent Panel (shown in Figure 5.2 in the orange box labeled with **A**) displays the agent as clearly as possible by trying to divide the nodes into different levels. This is done without more complex algorithms, as used for example in [86] to save display space and computational resources, so that the agents can also be displayed during training. During the training it can happen that new agents have to be displayed several times a minute and the display should not reduce the computing power significantly. For a better representation there is the possibility to translate the agent into the DOT language and to display it with *GraphViz* [86], but better still does not mean good. The hundreds of nodes and connections become enormously space consuming in this representation and are still not well understandable. In the end, graphs of this size, cannot really be displayed well for humans on a screen or especially on paper and a reduction of the complexity is necessary,if the user should be able to understand the information flow at least partially, in order to be able to develop new, so far missing methods. Therefore the interface offers the possibility to display the agent in a simplified way as shown in Figure 5.4. By hiding the other nodes and only showing the connections and directly linked nodes, it is easier to understand what exactly is happening at that node.

In the panel, connections between nodes can also be created or removed. To do this, all what is needed is to click both conduits. Since multiple connections are not possible or useful between conduits (multiple connections between two conduits would only create duplicates, which are filtered out), this can be used for creating new conduits or removing existing ones.

After the new connections have been created, the agent presentation in the panel is restructured.

The best agent that has been created up to now (and guinea pig for everything), Ellyn2554 which is in described in detail in Section 6.1.9, was trained in long runs, several times being stored, and placed again in a population later, and manually extended, and again trained in a learning population.

5.2.2 Node Detail Panel

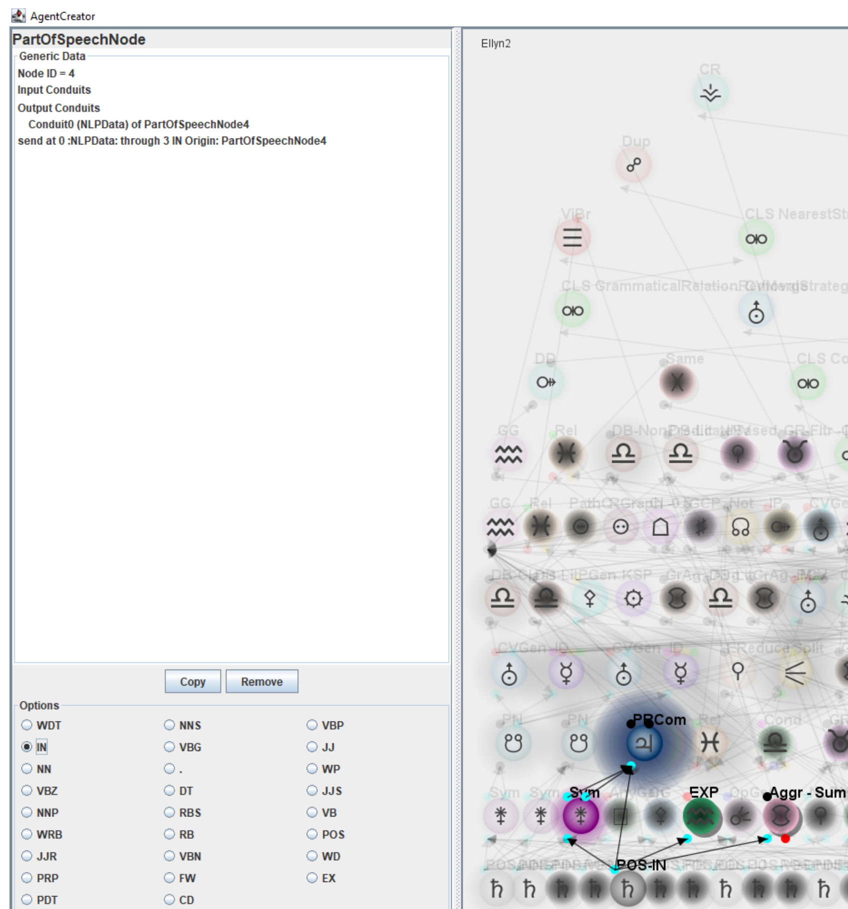


Figure 5.3: Possible Settings for a *Part of Speech Node*
(see Subsection 4.2.1.1 for details about those nodes). The possibilities are derived from the test sets.

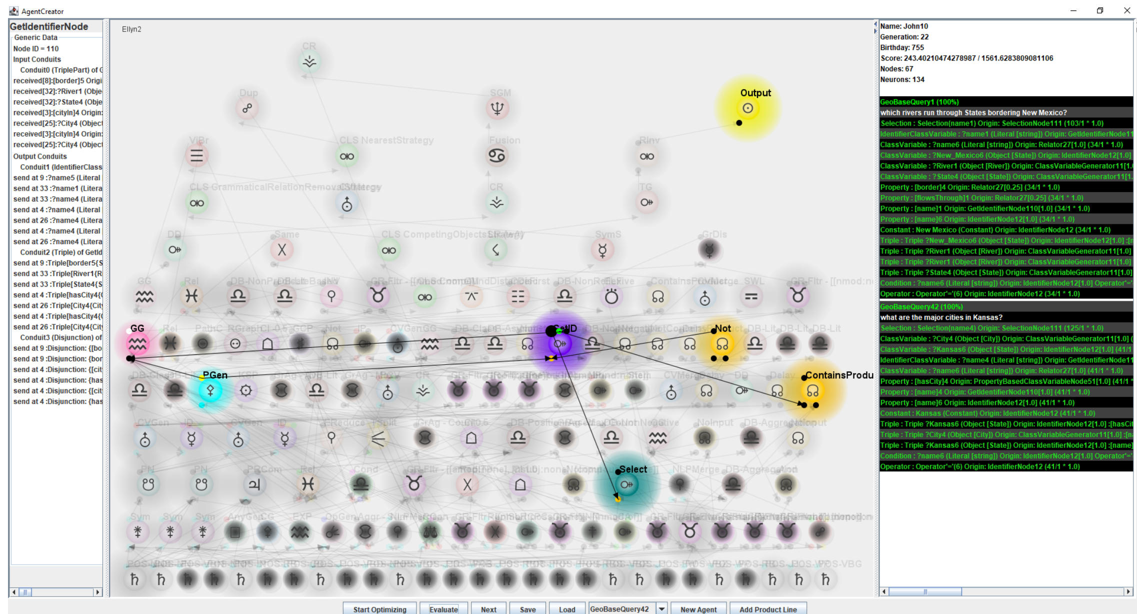


Figure 5.4: The *Agent Creator Panel*
Not directly connected nodes are faded out when selecting one node to make the overview easier with hundreds of nodes and connections.

In the node detail panel (shown in Figure 5.3) all details about the currently selected node can be viewed. For all its conduits all connected nodes are listed together with all products sent through these connections. For these products it is also listed from which products they are composed, from which node they were generated and which confidence value they have.

In addition, the parameters of the node can be changed here. The possibilities are either inherent to the class of the node or the parameters are derived from the test sets given to EvolNLQ. Depending on the kind of node it can be a single or multiple choice.

5.2.3 Graph representation of a query graph product

For the analysis of the nodes' behavior it is important to be able to compare the input and output products of one node with another. This task is much easier for humans if the query graph product is represented graphically and is not just as a textual list. So to quickly get an overview of a particular product, it can be clicked on in the node history and a plot of the query graph products is created using GraphViz. In this case, the representation of GraphViz is very suitable because a query usually does not create more than ten nodes in a graph and is also not used during training but only on user demand.

Those graphs are also used for the middle part of query examples in this thesis. The coloring indicates the respective confidence values of the products. The nodes and connections are labeled

with the name of the product that they represent. Furthermore, literal-valued ClassVariable products are represented as ellipses, object-valued ClassVariable products as rectangles, and constants as diamonds.

5.2.4 Agent Panel

Name: Sharice1
Generation: 0
Birthday: 0
Score: -27.0 / -27.0
Nodes: 23
Neurons: 40

GeoBaseQuery1 (100%)
which rivers run through States bordering New Mexico?
Selection : Selection(name1) Origin: SelectionNode111 (103/1 * 1.0)
IdentifierClassVariable : ?name1 (Literal [string]) Origin: GetIdentifierNode110[1.0] ID of: River1 (34/1 * 1.0)
ClassVariable : ?name6 (Literal [string]) Origin: Relator27[1.0] (34/1 * 1.0)
ClassVariable : ?New_Mexico6 (Object [State]) Origin: IdentifierNode12[1.0] (34/1 * 1.0)
ClassVariable : ?River1 (Object [River]) Origin: ClassVariableGenerator11[1.0] (17/2 * 1.0)
ClassVariable : ?State4 (Object [State]) Origin: ClassVariableGenerator11[1.0] (34/1 * 1.0)
Property : [border]4 Origin: Relator27[0.25] (34/1 * 1.0)
Property : [flowsThrough]1 Origin: Relator27[0.25] (34/1 * 1.0)
Property : [name]1 Origin: GetIdentifierNode110[1.0] (34/1 * 1.0)
Property : [name]6 Origin: IdentifierNode12[1.0] (34/1 * 1.0)
Constant : New Mexico (Constant) Origin: IdentifierNode12 (34/1 * 1.0)
Triple : Triple ?New_Mexico6 (Object [State]) Origin: IdentifierNode12[1.0] :[name]6 Origin: IdentifierNode12[1.0] ?name6 (Literal [string]) Origin:
Triple : Triple ?River1 (Object [River]) Origin: ClassVariableGenerator11[1.0] :[flowsThrough]1 Origin: Relator27[0.25] ?State4 (Object [State]) Or
Triple : Triple ?River1 (Object [River]) Origin: ClassVariableGenerator11[1.0] :[name]1 Origin: GetIdentifierNode110[1.0] ?name1 (Literal [string])
Triple : Triple ?State4 (Object [State]) Origin: ClassVariableGenerator11[1.0] :[border]4 Origin: Relator27[0.25] ?New_Mexico6 (Object [State]) O
Condition : ?name6 (Literal [string]) Origin: IdentifierNode12[1.0] Operator'='(6) Origin: IdentifierNode12 New Mexico (Constant) Origin: Identifier
Operator : Operator'='(6) Origin: IdentifierNode12 (34/1 * 1.0)

GeoBaseQuery44 (50%)
what States border Florida?
Selection : Selection(name3) Origin: SelectionNode111 (41/1 * 0.3)
ClassVariable : ?Florida3 (Object [State]) Origin: IdentifierNode12[1.0] (13/3 * 1.0)
ClassVariable : ?name3 (Literal [string]) Origin: Relator27[1.0] (41/1 * 1.0)
Property : [name]1 Origin: GetIdentifierNode110[1.0] (41/1 * 1.0)
Constant : Florida (Constant) Origin: IdentifierNode12 (41/1 * 1.0)
Triple : Triple ?Florida3 (Object [State]) Origin: unknown[0.0] :[border]2 Origin: unknown[0.0] ?State1 (Object [State]) Origin: unknown[0.0] Origin
Triple : Triple ?Florida3 (Object [State]) Origin: IdentifierNode12[1.0] :[name]1 Origin: GetIdentifierNode110[1.0] ?name3 (Literal [string]) Origin: R
Triple : Triple ?State1 (Object [State]) Origin: unknown[0.0] :[name]1 Origin: unknown[0.0] ?name1 (Literal [string]) Origin: unknown[0.0] Origin:
Condition : ?name3 (Literal [string]) Origin: IdentifierNode12[1.0] Operator'='(3) Origin: IdentifierNode12 Florida (Constant) Origin: IdentifierNode1
Operator : Operator'='(3) Origin: IdentifierNode12 (41/1 * 1.0)

Figure 5.5: Agent stat list after stating two questions

In this panel (shown in Figure 5.5) general data about the agent is displayed. First its name, its score, its number of nodes and connections, at which run it was created and how many generations came before it. After that comes a rating for each question asked. Of course, this only works if an appropriate test set with solutions has been defined. Depending on the correctness, the respective element is displayed in a color between red and green tones. Missing elements have a red background instead.

5.2.5 Node Panel

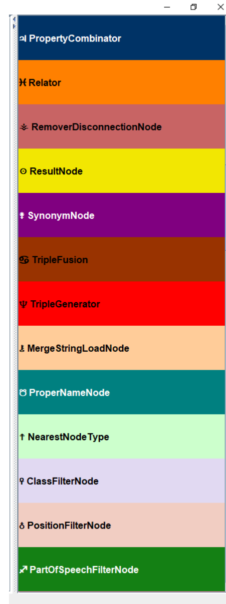


Figure 5.6: Excerpt from the list of available nodes in *Agent Creator Panel*

The Node Panel provides a list of the available node types. The node types are displayed in their specific color and with their Unicode symbol as seen in Figure 5.6. Newly created node classes are added to this list automatically. A node instance is created by a click and can be connected in the agent panel by connections.

5.2.6 Operation Buttons

Operation buttons are placed at the bottom bar of the panel. These give access to various basic operations, such as saving and loading agents, creating new ones, executing the whole test set or a specific question, executing the next question in a test set or executing a *production line* mutation (see Section 3.3.4 for more information). Furthermore, the optimization can be started. Random mutations are then performed as in the training phase and if the absolute score ($absolute\ Score = (\sum_{t \in T} eval) * (\prod_{t \in T} penalty(a, P_{a,t}, t))$ (see Equation 3.1 and Equation 3.5) basically the score if there where no other agents) of the mutated agent is higher than that of the current agent, it is replaced by this version. Unlike the automatic training, the user can intervene at any time and also make changes.

5.3 Training Environment

The *training environment* has two major parts. One is the overview shown in Figure 5.7, where the options for a training run can be set, and an overview of the progress of the different *environments* is given. The other part are the individual views of each *environment*. In the latter, the current agents, the course of development and the structure of the agents can be checked.

5.3.1 Training Overview

Figure 5.7: Example of the Training's Environment Panel with Multiple Environments

In this panel, the basic settings for a training session can be defined in Section A and the progress of the different environments can be observed in Section B. The options for setting changes are listed and explained in Table 4.6.

5.3.2 Environment View

The *Environment View* has four main components (shown in Figure 5.8) which are explained below.

Agent Panel The agent panel (A in Figure 5.8) is the same as in Section 5.2.4 but not changeable by the user and shows the agent with the current highest absolute score.

Statistics Panel The statistics panel (B in Figure 5.8) shows graphs of the progress within the environment, with four tabs plotting different values and one showing a pie chart.

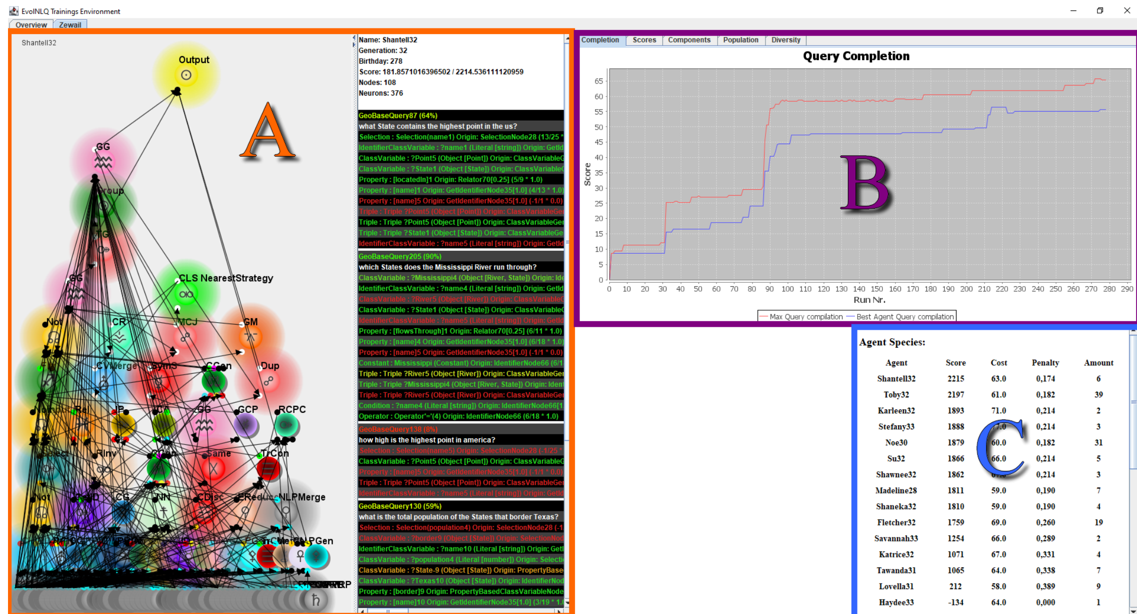


Figure 5.8: Example of the Environment View with labeled components. A Agent Panel, B Score Statistics, C Population overview

The first graph shows the percentage of the correctly computed products of the test set for each run. It shows both the value of the single best agent (blue) and the "team value" (considering all products, where maybe other agents have better solutions) in the whole run in this environment (red).

The second graph shows the development of the agents, in relation to the achieved absolute score (i.e. the "claim" as defined in Equation 3.4) and to the relative score (i.e. fitness function defined by the energy distribution as defined in Eq. 3.6) of the best agent and the average of all agents.

The third graph shows the number of nodes and the number of connections for the best agent, and the average for both in the entire environment.

The fourth graph shows the population in the respective environment for each run. It shows the total number of agents as well as the number of agents belonging to a new species.

The fifth graph is a pie chart that shows the relative distribution of agents by their respective species. However, species consisting of only one individual are filtered out. These are mostly just mutated and usually not viable. Only if an agent has produced offspring, it is listed here, otherwise this diagram would be overloaded with agents that do not survive the current run.

Population Overview The Population Overview (C in Figure 5.8) shows, as the name suggests, an overview of the current agent species and the agent distributions. The table contains the name of the species, its absolute score (i.e. the fitness function), the creation costs for an agent, the penalty

on the score caused by wrong answers, and how many agents of this species are currently present in the environment.

Run Panel The Run Progress panel shows the progress in the current run (for more details about the run process see Section 3.6). It shows how many questions each species has answered, the general progress of all species, and the progress of the subsequent calculation of relative scores. Only new species are displayed, since old species do not need to recalculate their answers and their absolute score, as it remains unchanged as long as the environment does not change, whereas the relative score for each species must be recalculated for every run, since they depend on comparing the absolute scores amongst all competing agents.

5.4 Implementation

EvolNLQ is designed to be as extensible as possible when it comes to nodes and products, so this section will primarily focus on how new nodes and products can be created after giving a brief overview. Things like synchronization or the graphical user interface will not be discussed further, because no special approaches were used here and if implemented correctly new nodes and products fit seamlessly and no special precautions are required.

5.4.1 Product Classes

While in theory each node class can inherit from multiple superclasses, in practice this is not intended or wanted in Java, so all abstract classes from Figure 4.3 except the Product class itself are interfaces in the implementation.

Each interface contains either already implemented methods if possible, or abstract methods for more specific methods. A new product class must be extended from Product or an already existing concrete Product class. The other abstract subclasses can be implemented in any combination. With each additional interface that is used by product classes, additional requirements have to be fulfilled as listed below.

- Product
  - `public ProductOrder getOrderPosition()`
    returns a predefined constant to order the products for better readability (not functional)
  - `public Product copy()`
    returns a new product of this type with the same values as the calling product, but a different reference id
  - `public int compareTo(Product o)`
    standard function for comparing products

- Compound Product
  - List<Product> getCompoundProducts()
    returns a list of all compoundProducts
  - default List<Product> getDifference(CompoundProduct compoundProduct)
    returns a list of contained products which are not part of the argument product
  - void replace(Product originalProduct, Product replacement)
    replaces the given product inside of the compound product
  - default boolean contains(Product product)
    checks if the given product is in the getCompoundProducts list
  - public int getWordCountRange()
    Difference from max to min position of all components.
- Positionable Product
  - public int getMinPosition()
    minimal pseudocount of this product
  - public int getMaxPosition()
    maximal pseudocount of this product
- Auxilary Product does not have methods
- SPARQLing
  - public QueryPosition getSPARQLPosition()
    returns a constant defined by the Jena SPARQL Builder, to which structural part of a query this product is added
  - public SelectBuilder addToBuilder(SelectBuilder sb, SelectBuilder mainsb)
    Implementation of the ToSPARQL method described in Section 4.1 for each product directly with a SelectBuilder object, and if it is a subquery with the main query Selectbuilder
  - public String getSPARQLName()
    returns the name of this product for using it in a SPARQL query
- SubSPARQLing
  - public SelectBuilder addToBuilder(SelectBuilder sb, SelectBuilder mainsb, List<Product> products)
    like the same methog of SPARQLling

- Modifier
  - `public abstract ClassVariable getContent()`
    returns the modified ClassVariable product
  - `public abstract void setContent(ClassVariable content)`
    sets the modified ClassVariable product

5.4.2 Node Classes

To create new node classes, the class only needs to inherit from Node and override the `getNodeID`, `init` and `processProduct` methods. So in the end, each node only needs to specify what it is called in the node ID, what its in and output conduits are in the init method, and to implement the operation itself.

A `NodeID` is, as the name suggests, primarily the identifier, but they also populate the library of node classes. Once entered there, the node is then automatically inserted into the GUI for manual creation and can be used by agents through mutation. A `NodeId` has the form `[name] (abbreviation, uniCode of the symbol, color, class)`. Since the node is not created by a fixed call, but is created dynamically from its class reference, only the default constructor is ever used, that is, the one with no further arguments. This constructor must at least contain the `init()` method to create the conduits for the node.

In the `init` method, new conduits can be added using the `AddInput` or `AddOutput` command with the product class they are to accept as a parameter. The order determines the index of the conduits.

The operation of the node is implemented by overriding the method `processInput`. This function is executed when the conditions of the `hasNeededProducts` method are met. Normally this is the case if there is at least one product waiting in each input. By overriding the method, e.g., when every individual product can be processed wrt. the internal storage, this criterion can be changed.

After executing the `processInput` method, `clearInputs` is executed, in its default variant, it empties the input conduits. If the behavior is to be changed, this method must be modified. To access the inputs `getInput(index).getReceivedProducts()` is used, where the index corresponds to the order of the `AddInputs` from the `init()` method.

To give one or more products to the output conduits, `addProductQueue(Product[])` is used, if according to the rules the products should be distributed (note that the index of the intended conduit is not to be given, since they act like a coin sorter as described in Section 3.3.2), or sent `sendDirectly(Index, Product)` is used to assign it directly to a conduit. To act as an `InputNode` or as a `CollectorNode`, it is necessary to inherit from the corresponding class instead of Node.

Chapter 6

Evaluation

This section evaluates different aspects of EvolNLQ, and by this also of the evolutionary dataflow agents framework. First, several facets and settings of the evolutionary dataflow agents framework are investigated, mainly wrt. qualitative and quantitative performance. Second, some learning test sets and a benchmark comparison are described.

6.1 Settings

In this section, different settings and modes of the EvolNLQ Framework are evaluated and demonstrated for their effects and whether they bring a demonstrable advantage or for which setting the effects are best-suited.

One thing to keep in mind when dealing with a –in the wider sense– search algorithm like an *evolutionary algorithm* is that achieving a certain goal is mainly a matter of time, when appropriate basic functionality is provided since an arbitrary selection method with arbitrary mutation sizes, will eventually find the optimal solution for given operations. Even a random agent, under these conditions would eventually find the optimal solution. Therefore, with these methods it is essential, *when* "eventually" can be expected.

6.1.1 Coverage and Recall

For the evaluation of the agents during the training, an evaluation according to correctly answered questions and incorrect questions is too coarse. A completely correct answer, even for a very simple question, already requires a complex and sophisticated agent. Therefore, the fitness function does not evaluate the agents by this measure but, instead, divides each reference solution (which is the intended the result of each task) into as many small subgoals as possible, resulting in a fine-grained notion of coverage of the generated answer queries wrt. the reference solutions. This is used in the following to illustrate the learning progress of agents, since it is the decisive criterion for

the algorithm, according to which the evolution of agents is controlled and the progress is better recognizable than in discrete steps of completely correctly translated questions.

On the other hand, finally, the *recall*, i.e., the percentage of *tasks* solved perfectly, and not only to 96% correctly, is the relevant quality. So this is also investigated.

6.1.2 Selection Methods

In this experiment, the *energy-based* selection method (as presented in Section 3.6.2.1) and the classical $\frac{n}{2}$ method, in which the better half of the agents are included in the next generation and each has a (probably mutant) offspring are compared. Both used a training set of 20 tasks; the evaluation of the $\frac{n}{2}$-strategy is based on 20 test runs, and the evaluation of the energy-based strategy is based on all test runs under the same, straightforward setting during the whole time.

As can be seen in Figure 6.1, both methods develop towards a better solution, but the energy-based method is clearly better in all cases after only a few runs. However, it can happen that during the first few runs neither of the two methods finds any solution if the initial agents are mutated in an extremely unfavorable way or the test set is is not solvable for the initial agent without a lucky mutation. As the completeness of the generated queries increases, the gain for both methods decreases, however, the maximum of $\frac{n}{2}$-strategy is significantly lower, reaching a coverage of 52.00% (i.e., 52.00% of the subgoals over all tasks are correctly found) after 5000 runs – the same percentage as achieved by the energy-based strategy after only 300. Overall, the energy-based approach reaches a coverage of 78.00%. As shown in the lower part of Figure 6.1, with the energy-based strategy, the population succeeds in perfectly solving in the average 15 of the 20 questions in 5000 runs, while the $\frac{n}{2}$ solves only 4.

Especially, it turns out that The $\frac{n}{2}$ method does not foster the evolution of highly skilled agents that are able to solve tasks completely perfect. Instead, it seems to conserve mediocrity. The reason might be that this method is not designed for multiple objective orientation, it cannot pursue multiple approaches to find a solution, which ultimately leads to the fact that certain solutions are not found and the agents are quickly stuck in a local maximum, without a realistic chance to still evolve decisively.

6.1.3 Changing energy values

For this experiment, different amounts of energy $en(t)$ per task are used and the effects were compared. For this purpose, test runs with low ($en(t) = 100$), medium ($en(t) = 1,000$) and high ($en(t) = 10,000$) energy were performed and the evolution processes were compared.

As shown in Figure 6.2, it can be seen that the more energy each task provide, the better the coverage, and the more time is needed. Looking at the final agents in detail showed an obvious effect for evolution: the evolution with low energy values develops simple agents that can only solve the simple tasks that are solvable by individual nodes. The agents remain at a lower level

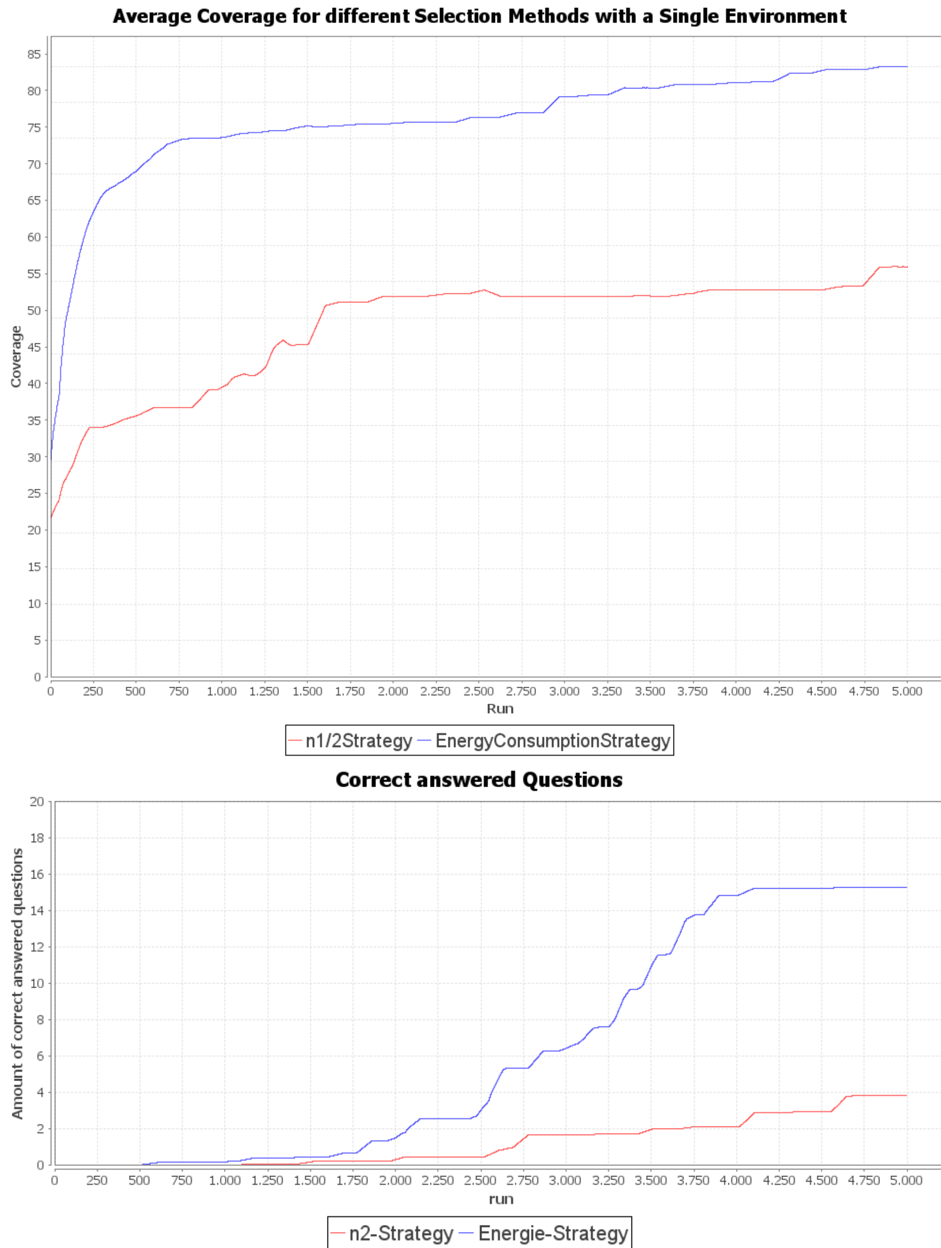


Figure 6.1: Comparison of selection methods
Coverage (upper part) and number of perfectly solved tasks (lower part) for different selection methods

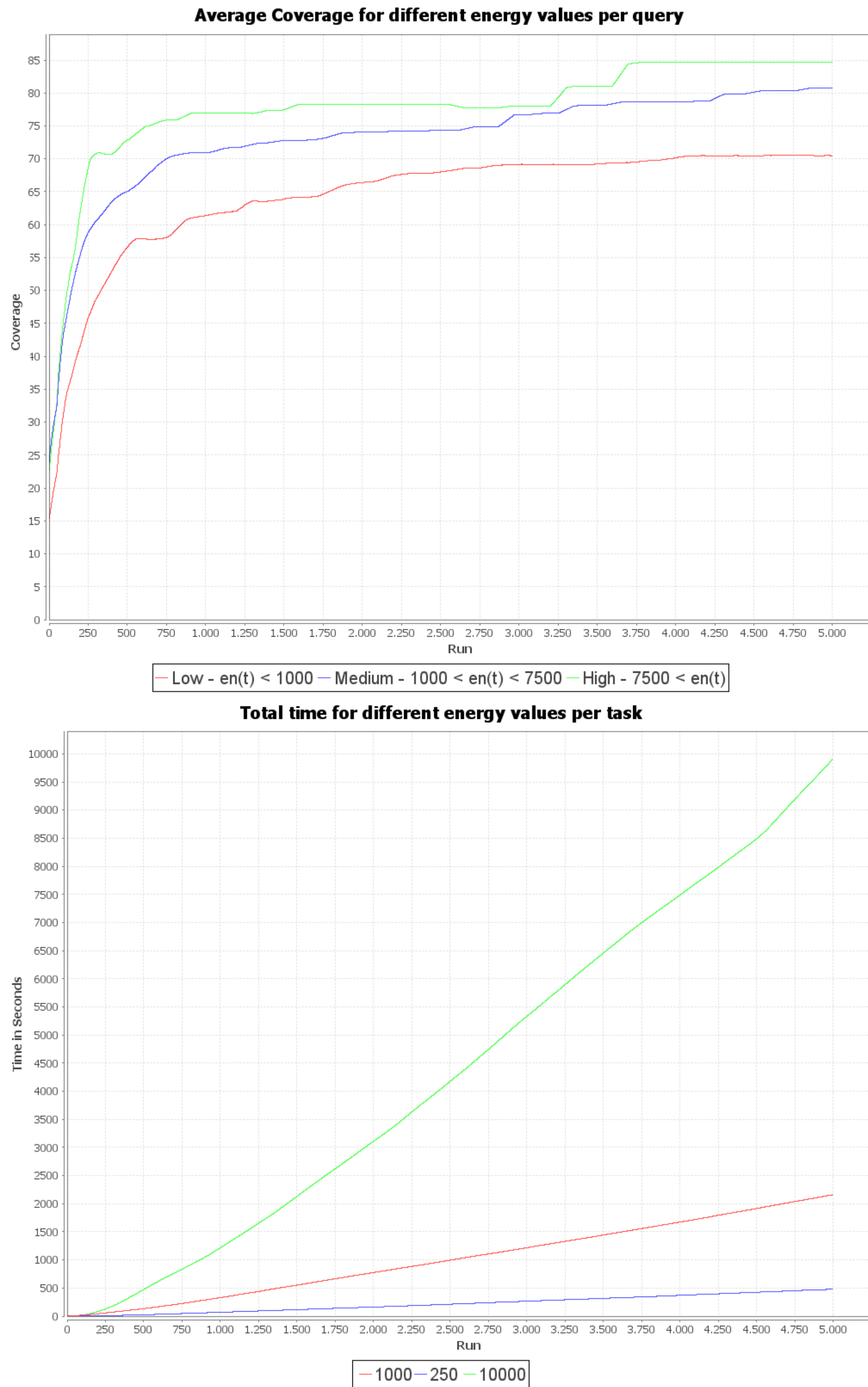


Figure 6.2: Learning performance depending on available energy.
The upper graph shows learning curves with different energy values per task, while the lower graph shows the corresponding computation time for those.

of completeness, because the energy for larger investments, and the *sustain energy* for complex creatures is missing (as in nature, in poor environments there are protozoa and other simple life forms).

At the medium and high values, it looks like the higher setting is still superior. It should be noted that at the same time that since more energy is available, the total number of agents in the environment is also significantly higher, so there is also a much higher diversity. This fosters the possibility for exceptional creatures to emerge. On the other hand, the computational time of each single (note: still training) run also increases dramatically (see Figure 6.2 lower graph). Given the same amount of total learning computation time, the medium energy level reaches about the same coverage.

6.1.4 Multiple Environments

This experiment investigates whether there is an advantage in splitting the set of training tasks among different environments or if it is better to have them all in one. To do this, 20 experimental runs are performed with one environment with 40 training queries, and then 20 with five environments with 8 queries each, where agent relocation between environments or/and environment fusion is also allowed. As shown in Figure 6.3, the group of environments evolves faster and more successfully on average.

In phases after a relocation, when different environments host instances of the same agent species, and a relocated agent species turns out to be successful, it can be extremely successful at the beginning, claiming a lot of energy from tasks that were unsolved before. Then, the species produces more offspring, often mutating. By this, that offspring evolve apart again, and often unlearn abilities to solve tasks from the original environment where the species came from, and which are not represented in the tasks of their new environment. Thus, the relocated agents do indeed make a difference, however, their performance when applied to the previous questions is, as expected, worse than it were if all happened in a single environment.

Therefore, a third experiment is performed where the environments are fused during training and they are able to exchange agents. Otherwise, the starting conditions are the same as for the other runs in which 5 environments are used, each with 8 questions. As can be seen in Figure 6.3, this results in a higher diversity of agents and better results than with fixed environments.

6.1.5 Sustain Energy Evaluation

The two schemes for defining the *sustain energy* of an agent introduced in Section 3.6.2.1 (page 63) are investigated in this section. Normally all runs were made with the mode with the "zero-cost fixed-all input" scheme as defined in Equation 3.3.

As shown in Figure 6.4, in the long-term evolution, "zero-cost fixed-all" that provides all possible inputs in advance is quite advantageous for the convergence and the coverage. This mainly seems

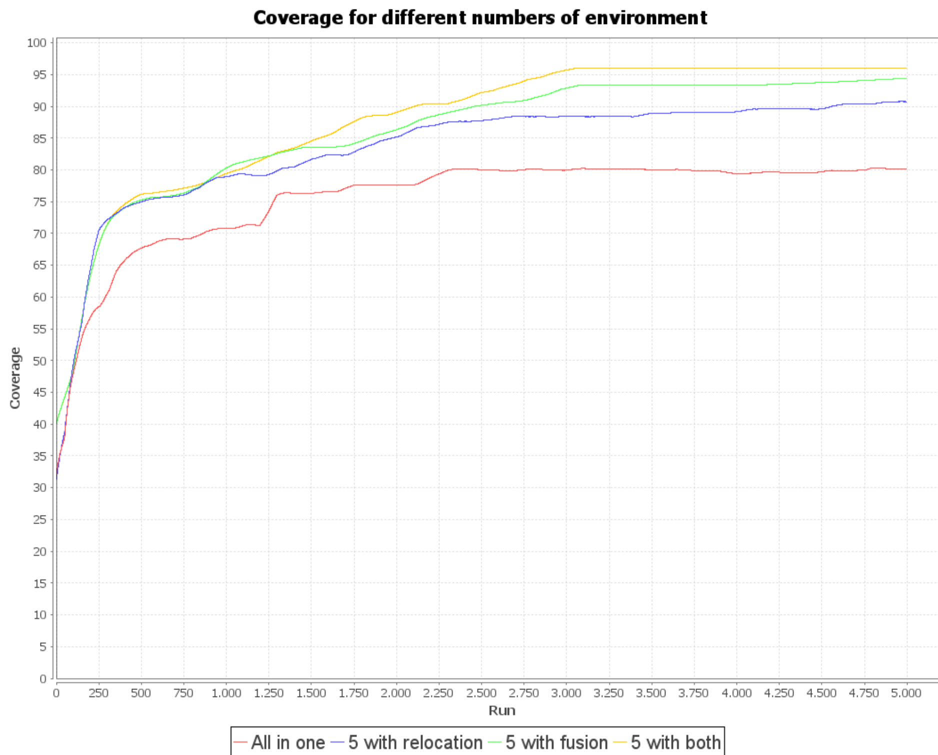


Figure 6.3: Coverage depending on number of environments.
Average coverage for a single environment with 20 tasks or for 5 environments with 4 tasks each; agent relocation or/and envorinment fusion activated.

to be due to the fact that when a mutation creates a node somewhere in an agent that uses a input Part Of Speech tag that was not used by this agent before, the mutation (that automatically adds appropriate connections feeding all its input conduits) can exploit the already existing input nodes. In contrast, with the "non-zero-cost adjustable" cost scheme, not providing pre-prepared input nodes, first a node that creates the NLPData product must have been created before (by another mutation). But, as long as such an input node only created, but connected to a wrong conduit it causes extra sustain costs and in the worst case causes even damage; in both cases the so mutated agent is in a disadvantaged position. So, the combination of such mutations happens much more unlikely. Regarding the computation times per run, both modes do not differ, as expected.

Example 30 *When writing this section, a four-eared cat was reported:* `https://www.instagram.com/midas_x24/`*. Considering the "zero-cost fixed-all input" scheme, every animal species would have received exactly one eye, ear, nose, tongue etc. Obviously, the "non-zero-cost adjustable input" scheme lead in nature to the development of keeping one tongue, one nose (with doubled input conduits), but -usually- two eyes (with very different abilities when considering eagles or dragonflies), but sometimes four, e.g.,*

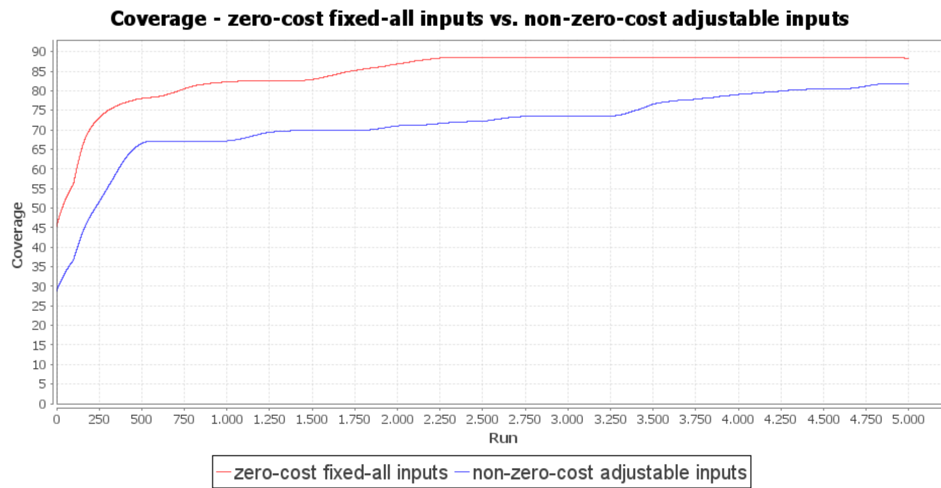


Figure 6.4: Analysis of effects of input cost
Comparison between of the coverage for "zero-cost fixed-all" vs. "non-zero-cost adjustable" input node sustain energy schemes

Anablepsi *that support to watch above the water and in it in parallel, and -usually- two ears. So, a flexible cost scheme for input organs proved useful there. Evolution will show whether it also turns out to be successful for cats to have enhanced listening abilities to cover more independent audio input directions.*

6.1.6 Forward-only Agents

Considering computability theory, sequential processing, variable assignments and branching in conditions are provided by the agents model. Turing-completeness would also require loops. Before the discussion, note that there are many specialised programming concepts in computer science that are not Turing-complete, like the relational algebra and SQL. They do not have loops, and without extensions, they can e.g. not compute the transitive closure of a relation. Internally, the nodes allow practically any functionality, as some of the graph nodes described in Section 4.2.7 show. So, any classical algorithms *can* be encapsulated into the nodes, and the contribution of the evolutionary dataflow agents framework is to allow to solve problems for which no classical algorithm can be given.

By allowing loops, the approach could be given the possibility to perform arbitrarily long iterations based on the input or ontology knowledge, but there is the inherent danger that this creates endless loops. Detecting this in the general case amounts to the halting problem. For a given application, the situation can be better.

The practical view. This can be partially mitigated by, for example, keeping lists of products that have already been processed and not processing them again. Most cases for endless loops can be

caught this way, but depending on the size of the loop, this can still mean a lot of computational effort if it is done thousands of times as part of the training, and nevertheless, in the general case there are recursion and transitivity (which are basically the same thing) where a different product, or a slightly modified variant of it, is created each time.

In contrast, there is a simple possibility of prohibiting nodes from having connections to nodes that are at a lower level in the static stratification of the agent.[1] To do this, from the input nodes, a stratification level is assigned as follows in a cycle-free agent: Each node gets the highest level of its input nodes plus one (the condition for cycle-freeness is only that it must be on a higher level), with the output node in the highest level. New nodes may then only have input nodes that have a lower level than themselves (depending on the data flow, it might also be possible to modify the stratification assignment by moving groups of nodes upwards). While loops are not possible this way, structures must be duplicated for multiple, but finite, iterations through the same nodes, if they are needed at all. Depending on the size of the structure and the number of iterations, this can become very costly for an agent.

The performance of the approach and the usefulness of loops for the quality of the results has been investigated by running 20 test runs for 3 hours in each setting and then 3 more for 24 hours each. Here, real time is used for the determination, since the number of runs is not affected by loops, but in case of infinite loops, the environment must step in: the approach can handle non-terminating agents by giving them only a certain maximum computation time, but this is much higher than the average computation time, since certain requests can actually take much longer than normal even with a very efficient agent.

In order to determine whether the Forward Only restriction has a benefit, first of all the achieved coverage is compared: here, no large differences can to be recognized.

Even though the best agent that has been created up to now (which also serves as guinea pig), Ellyn2554[2] which is described in detail below, contains loops, it is not completely clear whether this has any benefit at all. Nonetheless, it cannot be said with absolute certainty whether there is not a disadvantage in prohibiting loops. In the end, it probably depends on the type of nodes and how powerful the functions are. However, evaluating the progress per minute of the learning process, it turns out that learning agents that can have loops takes by far more time to reach the same coverage. Also, when not allowing loops, a significantly better coverage can be achieved within 3 hours. So the price of dealing with timeouts for infinite loops does not pay in terms of better evolution. However, this is put into perspective again, if the agents are given a large amount of time, since greater progress is rarely made in later developments; this is how Ellyn2554 was created; additionally, Ellyn2554 was tuned manually afterwards using the Agent Creator Panel

[1] note that collection nodes serve for the *operational* stratification of the *computation* (see Section 3.4.2), where here a simple architectural/anatomic stratification like first level, second level, third level etc. as used in the graphical representation of agents.

[2] all agents get not only a number, but also a name from a list of names for better recognizability.

described in Section 5.2.

So from the practical point of view, it doesn't matter whether backward connections are allowed or not, if enough time is available. However, for this application also no structure could be found, which draws clear advantages from it in terms of (i) no specific such substructure of an agent, and (ii) Ellyn2554 is not far better than well-trained cycle-free agents.

For future refinements it can be considered to introduce an advanced mode, which forbids backward connections in the early stages of the learning process and permits them only, if for a long time no progress has been made and the suspicion exists that loops could help. But for the basic structure and for the time being such connections to forbidden. But since, as said, no structures were found that promise an advantage if they have backward connections, this type of approach could not be tested in a meaningful way.

The theoretical view. Recursion and transitivity exist. In many applications. Detecting them by the learning framework would mean to go on the meta level, and to analyse the application and to have an introspective view on itself. This is something, it cannot do. Instead, this is the task of the designer of the application, i.e., who designs the products and the nodes.

The concrete view. For a concrete application, the design of products and nodes can in general solve this problem. In general it has to be analysed where recursion/transitivity occurs in the application, and to identify it conceptually.

For the case of EvolNLQ, one can systematically analyse the possible sources of recursion/transitivity:

- The input is a finite sentence of *natural* language which should not be recursive to an arbitrary depth. There can be nested negations. "The names of all countries such that there is no city which is not located at a river that does not flow into a sea that is not ...". With appropriate products (negation with a scope), this can be flattened.
- The output is always a SPARQL query which is in fact a simple algebra tree. Again, it might be arbitrarily deep. But, it is generated from a query graph product that contains all necessary products, by the *toSPARQL* method of the SPARQLing product types, which is implemented in Java. This shows how the problem of Turing-completeness can in this case be delegated into the nodes.
- There is also the ontology. It can –like *Mondial*– contain recursive (and symmetric) properties. These can be a source of infinitely expanding transitive patterns by creating and connecting arbitrary many new ClassVariable products products. Here, the current design of the node types restricts that all triple-generating nodes (that could try to generate such infinite chains) require that at least one of their inputs has a high specifity and confidence that it is *actually* needed. They cannot expand deeper.

- so only the *usage* of a transitive property in a query, like "all rivers whose water flows finally in the North Sea" could *force* that the *resulting query* deals with transitivity. Note that this cannot be hardcoded by joins 1,2,3,4,... into a query, because the pattern must be infinitely long (the typical problem of the *relational completeness* of the relational algebra and SQL). So the solution is to design a specialized node that detects transitivity/recursivity of a property, and an appropriate product type. Again, the mapping into the generated SPARQL query is delegated to the *toSPARQL* method of the SPARQLing product types, which is implemented in Java. Having SPARQL as target language, this will just result in *writing* SPARQL triple patterns like (?X :flowsInto+ ?Y . ?Y :name "NorthSea"). Note that with SQL as target language this needs to create a "CONNECT BY ..." clause, which is a tedious work since it needs to pull one or more relations in from the surrounding query.

6.1.7 Changing Rewards

To investigate whether changing the rewards of subtasks has an effect, 20 simulations with 5000 runs each were made, with and without variable rewards. However, no significant difference could be found in achieving coverage or perfectly solved tasks. Also there is no significant difference in diversity, only that after a successful mutation, in the following runs, as expected, the new species has a larger number of agents. Thus this species gets relatively more computing time, which should cause a faster evolution. However, this increase of the computing time due to this advantage seems to be almost completely offset by the reduction of computing time due to the disadvantage of the "old" species getting less computing time as their tasks become less valuable, and they have less offspring.

6.1.8 Influence of penalties

That penalties are a fundamental rule for any *evolutionary algorithm*. There, they are a summand of the fitness function. For evolutionary dataflow agent, the design incorporates them as a factor, which, to stay analogously, must in this case not be too low. To find a good reference value, 5 test runs were performed with different weights, with otherwise the same default settings. The results are shown in Figure 6.5. It was found that very low penalties (0.99 to 0.8 since penalties are multiplied) give the best result and the effectiveness of the agents decreases with stronger penalties. Agents must be significantly conservative in this case, especially with a penalty = 0, a single error would be directly associated with extinction. However, the simulation runs worst without penalties, ultimately the value given in the figure is not properly applicable because the simulation became so slow that no run could be completed without penalties and only the value up to practical standstill could be used. As penalties are mainly for exceed products in the resulting query graph product, this shows that penalties avoid kind of denial of service attacks by agents generating excess products.

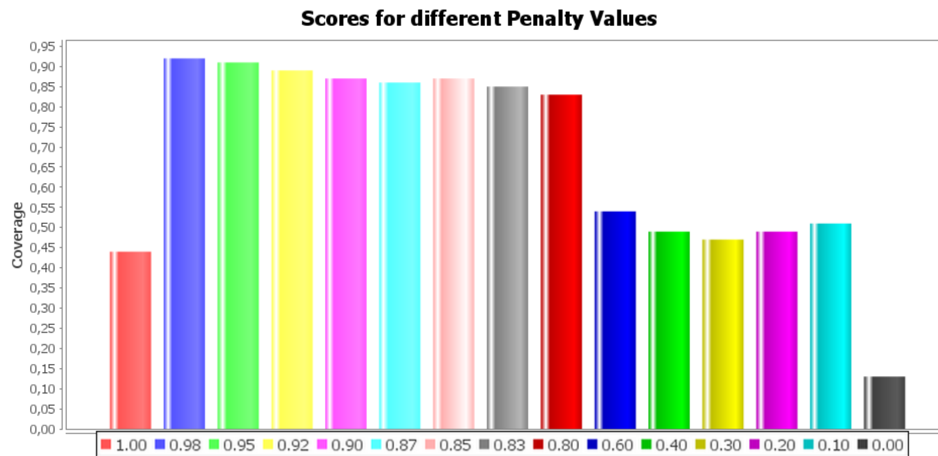


Figure 6.5: Influence of the penalty value
Average coverage after 5000 runs for different penalty values

6.1.9 Final Agent

The final evaluation uses the most successful settings, giving the test run enough time with all 60 tasks of the *Mondial* learning set (described below), distributed over 15 environments. The individual agents (more concisely: one of each species) that resulted from the learning process were collected by a swarm fusion (cf. Section sec:SwarmFusion) into a single agent. Since then, this agent serves as a guinea pig. This agent was further manually edited to include things like the ability to understand the notion of density by adding a Custom Aggregation node (cf. Section 4.2.5.3) which was not learnt because the learning set did not contain such queries. After that, it was successfully again placed in learning environments, but the coverage did not increase any more. It was also used to evaluate the benchmarks that are described in Section 6.2.

The name of this agent (species) has the designation *Ellyn255452*554 (or short Ellyn). It consists of 151 Nodes and 337 Connections. Like all agents species, this one has a designation consisting of a randomly chosen name followed by a number indicating how often the entire list of names was exhausted (recall that agents live only for one run and the offspring gets a new identifier, where "free" names are reused). The agents species are named to make it easier to refer and distinguish agent specific configurations. So while the name suggest that it is referring to a specific individual, actually only the species has a name, since the member of the species are not distinguishable anyways.

As can be seen, even Ellyn is not able to successfully solve all tasks. However, it answered 87.30% of the training set correctly with a coverage of the individual subgoals of 96.67% see Figure 6.6). The evaluation of Ellyn against benchmarks is dealt with in the following section.

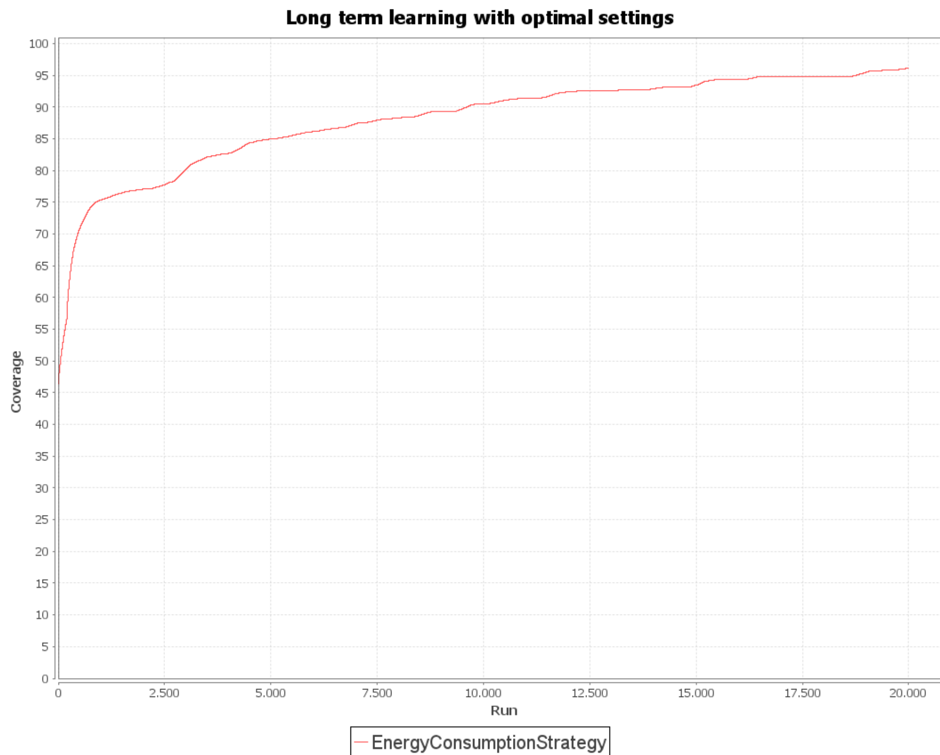


Figure 6.6: Development of the final agent

6.2 Question Test Sets and Benchmarking

> One should only ask those questions whose worst possible answer one can bear.
>
> Thomas Schlapp (translated from German)

Creating a good set of test and learning questions is not a trivial task. Even with good knowledge of the query language, so that the reference solutions actually deliver the expected result, it is still labor-intensive to create the solutions. But also the selection of the questions is not easy, here must always be decided whether one orients oneself more at the practical case, but takes into account that the queries, without context are actually not correctly answerable, or whether one orients oneself at the solvability, in order to obtain questions which can also have a solution, but do not correspond to the choice of the use cases to be tested.

The following three test sets were chosen because they satisfy both aspects to some degree. The Mondial test set was created with its focus on solvability and with certain challenges in mind and

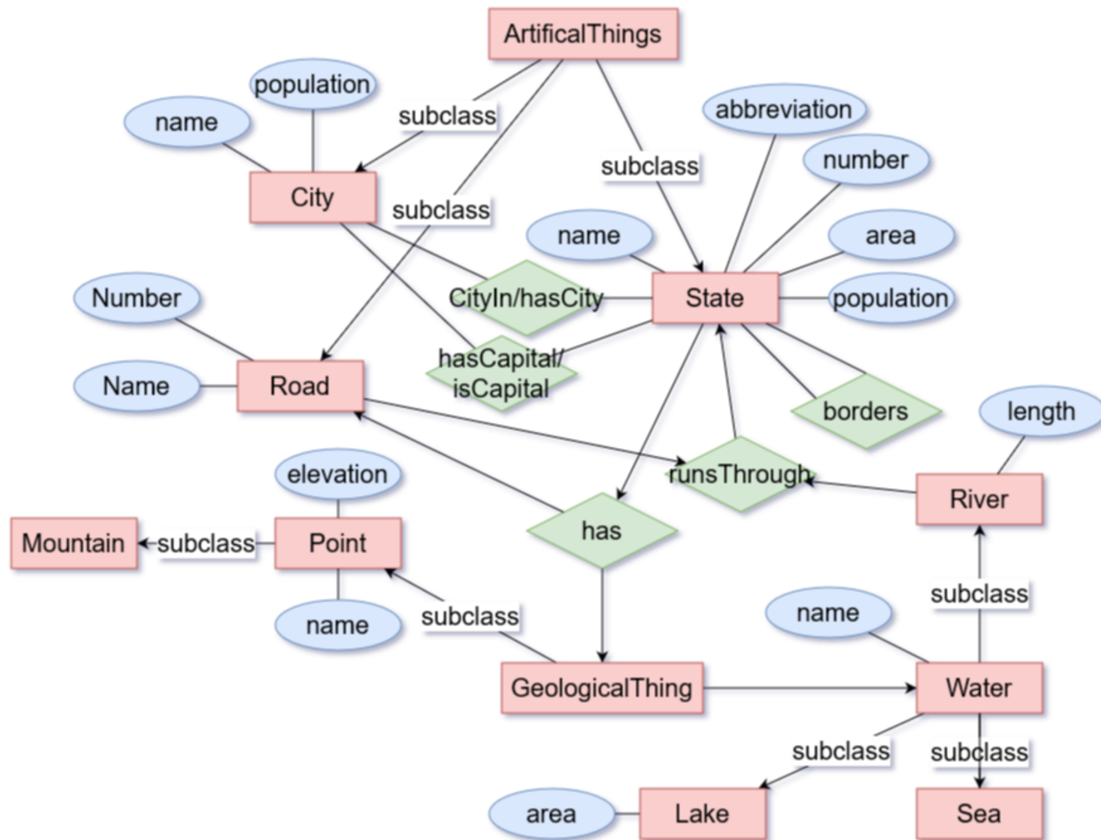


Figure 6.7: Visual representation of the RDF/OWL ontology based on the Datalog Data set. The presentation is based on ER models [81] but with directed edges if a property only exists in one direction. Red boxes are classes, blue eclipses are data properties and green rhombus are object properties

thus rarely has unintended random obstacles, but systematically approaches certain issues (like, e.g. reified properties).

The *Geobase* test set, contains both types of questions and in particular many question patterns in an unambiguous and an ambiguous variant. For example, "How many people live in Alaska" and "How many people live in New York", where Alaska is unambiguous, while New York could be a city or a state.

The *Movie* test set consists almost entirely of questions that are phrased in general language and are usually full of inaccuracies and often contain too little information.

6.2.1 GeoBase

The Geobase Query Set includes 250 queries in natural language and their solutions. It refers to a data set of the same name. This data set describes the states of the USA and the US American cities, lakes, rivers and roads. The data set is far from being exhaustive, and was written just for this benchmark. It itself is relatively old and has been used for a long time for training and evaluating NLP applications. The data set and the solutions to the queries are written in Datalog [30].

This test set was used as part of the evaluation of this thesis but also to illustrate certain aspects more explicit reference is made to individual queries from this training set.

Since EvolNLQ does not have a Datalog translation and *RDF2SQL* does not understand Datalog either, an RDF ontology was designed for GeoBase as part of this work. Where possible, this was done in an attempt not to reduce complexity. In fact, some items have become rather more complex as a result of the design. Since it is orgiginally in Datalog, and Datalog uses the relational model, most values are literals. Additionally, the number of instances is limited, e.g., each state has exactly 4 cities stored in its tuple, sorted by number of inhabitants. City1 is therefore always also the one with the highest population. In the ontology, an object-valued relation has been defined in such cases, which is not sorted since the RDF data model (without RDF containers, which cannot be handled by SPARQL) is an unsorted graph-based data model. Also, new classes have been introduced such as *Point* and *Mountain*. These are otherwise stored in the *state*'s tuple as *HighPoint* and *HighPointValue*, the two fields contain the name of a mountain and its elevation. With the new classes, these values are mapped to objects of the *Mountain* class, a subclass of *Point*. To the class *Point* also includes the points resulting from the values of *LowPoint* and *LowPointValue*, but it is not possible to say reliably whether they are valleys, lakes, riverbeds, or seas. This makes queries for the lowest or highest points in states more difficult, since an aggregation over the points in a state must now be made instead of just looking up a literal value.

When creating the ontology, a conscious effort was made not to express non-synonymous terms by equivalent classes or properties in order to create additional difficulty. Furthermore, attempts were also made, when possible, to give each property a domain and a range as broad as possible (to form a counterpart to the *Mondial* test set where both are as concise as possible), especially when it is detrimental to the query.

6.2.1.1 Results

The entire set of GeoBase questions is shown in Appendix A.2 along with the coverage achieved for each question by the agent Ellyn (see Section 6.1.9).

In total, 96.00% (240 of 250) of the queries have been answered correctly. As mentioned in the introduction, this approach is directly compared to Athena

The set was given to agents partially over many iterations and the other part was used for

evaluation. The agents from different runs were partially merged or reused. Therefore, this is not to be considered a "fair" benchmark, since training set and test set were not strictly separated from each other, but rather to determine whether this approach is at all capable of answering these kinds of questions. During the development of the nodes, it was necessary to check every now and then which operations were still needed. Nevertheless, in the following table the results are listed, but they are not to values that this approach is clearly superior. A better relativization of this approach is given by the benchmark in the next section.

Approach	Correct Queries	Recall
EvolNLQ	240	96.00%
Athena [6]	218	87.20%
PRECISE [3]	194	77.60%

While most questions are relatively basic, there are also some more complex ones that can be answered successfully, like GeobaseQuery215 in Example 31 or GeobaseQuery243 in Example 32. But there are also still a total of 10 questions that could not be answered correctly. In the following a brief reason, why those questions could not be solved is given.

Example 31 *Question Geobase215: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase215 are depicted in Figure 6.8.*

Example 32 *Question Geobase243: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Geobase243 are depicted in Figure 6.9.*

GeoBaseQuery62 *"how many citizens live in California?"* and **GeoBaseQuery90** *"how many citizens in Alabama?"* have the problem that population and citizens are not synonyms and thus has too little information to not also offer all other numeric properties of California or Alabama. This result is considered too uncertain by the agent itself and is discarded.

GeoBaseQuery104 *"how many square kilometers in the us?"* is not really a meaningful question, moreover the database (and the user) does not know whether the area is given in square kilometers.

GeoBaseQuery173 *"what capital is the largest in the US?"* and **GeoBaseQuery225** *"what is the largest state capital in population?"* causes similar problems for EvolNLQ as for Athena. The lack of context that the entire database refers to the US makes this question so speculative that it is not answered by EvolNLQ's top agent Ellyn (see Section 6.1.9). Even if the solution is found within the agent, all information is so uncertain that it is discarded just before the calculation is finished. When changing manually, the confidence threshold or the evaluation of the individual elements, wrong results arise with quite a few other questions, therefore it must be accepted here probably that there is no possibility with the current functionality of EvolNLQ to answer such questions.

GeoBaseQuery193 *"how many major cities are in Florida?"* here the problem lies in the distinction between Capital and City. When adding Capital, which is also a city, it is counted twice and the result is wrong.

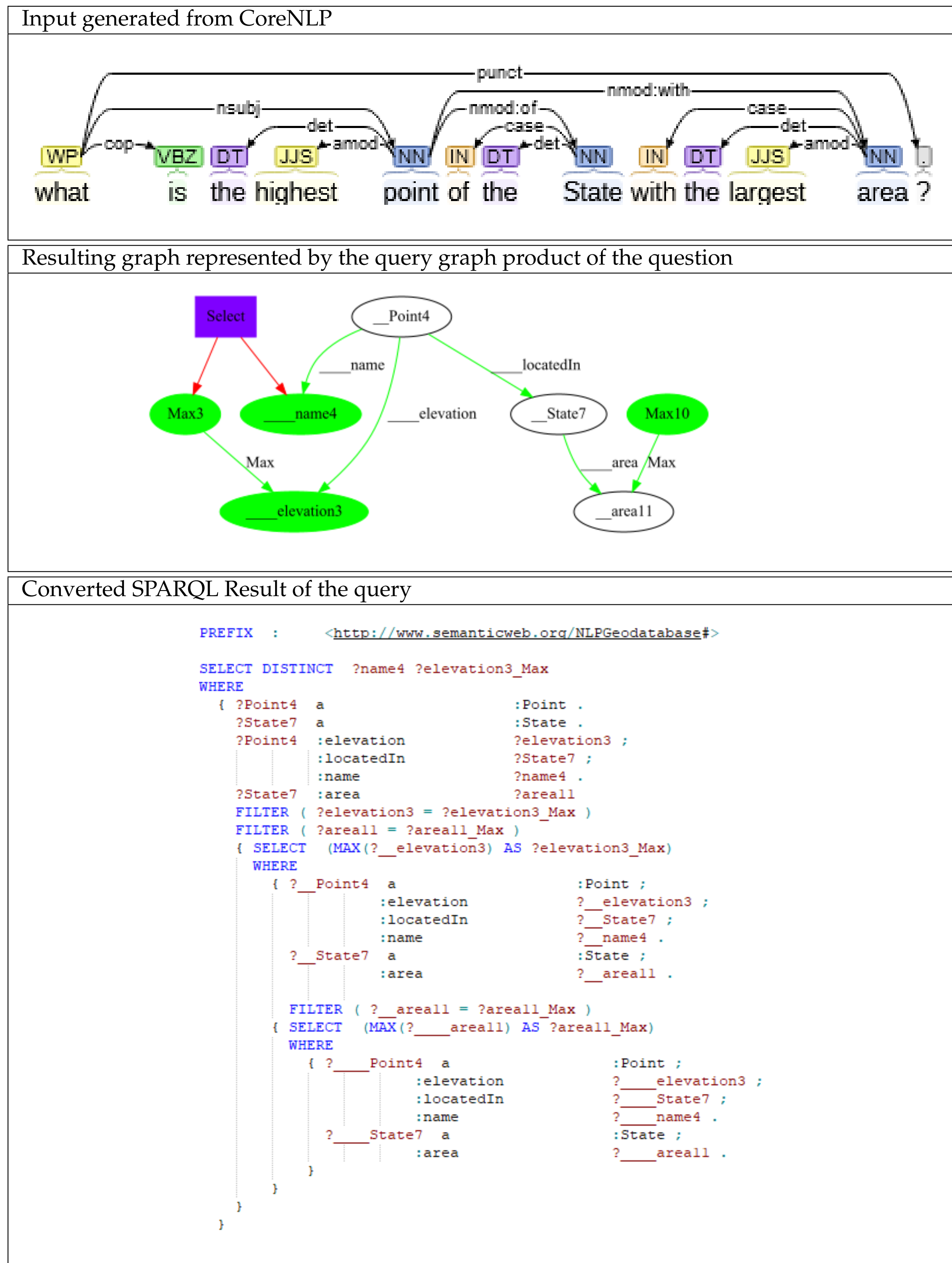


Figure 6.8: Question *Geobase215* with CoreNLP anntotations, query graph, and resulting SPARQL query

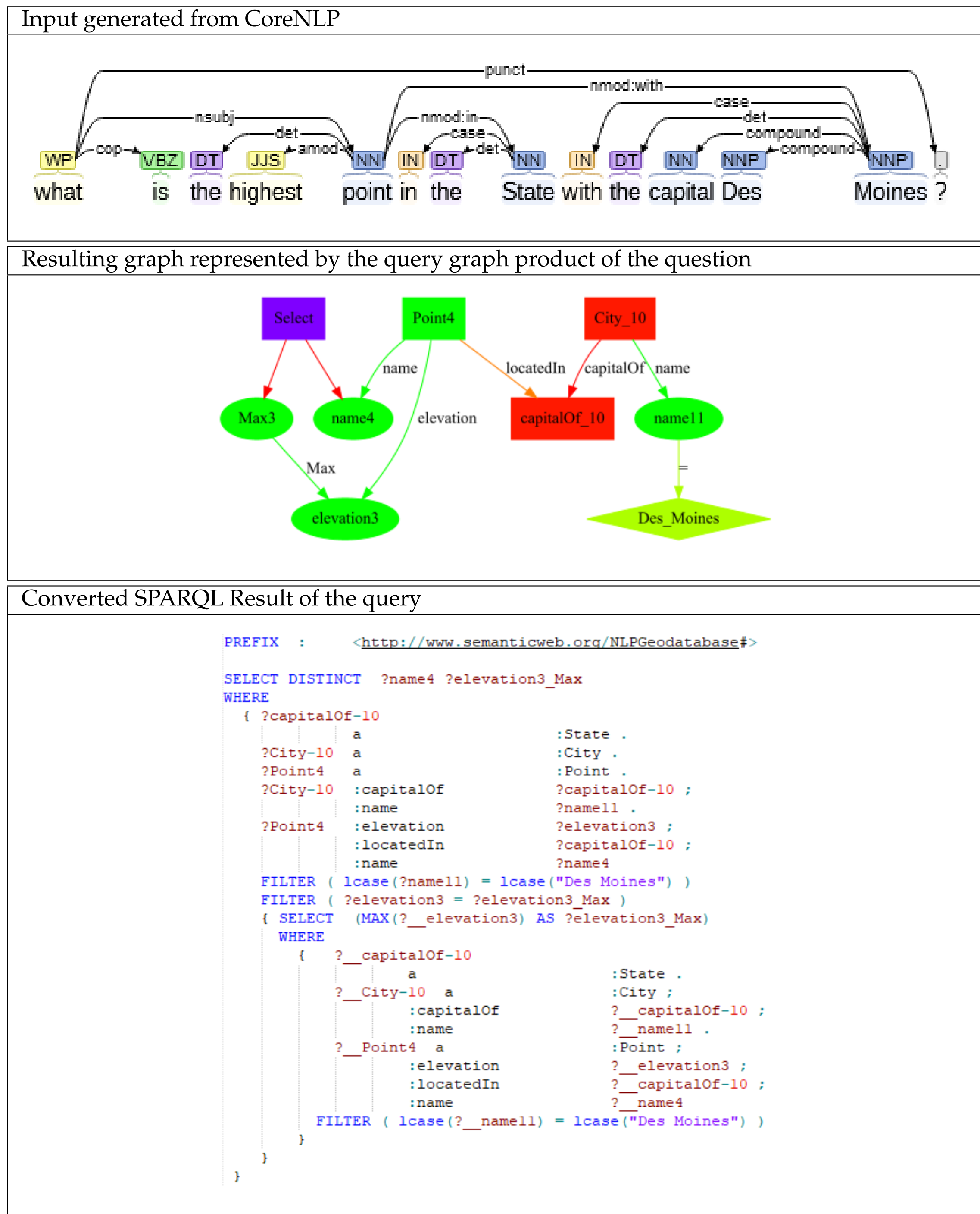


Figure 6.9: Question *Geobase243* with CoreNLP anntotations, query graph, and resulting SPARQL query

GeoBaseQuery206 *"Which states capital is Dover?"* has no particular reason, but has a problem with sameAs which might be a code-related problem.

GeoBaseQuery211 *"What is the total area of the USA?"* and **GeobaseQuery247** *"what is the combined area of all 50 States?"* calculates all areas together, not only the area of all states but also the area of lakes and gets a wrong result.

GeobaseQuery230 *"what are the populations of states through which the Mississippi River runs?"* while not too complicated it seems that the question contains too many ambiguities with Mississippi as a river and state, leading to a completely incorrect answer, while similar question do not pose a problem.

6.2.1.2 Disadvantages of the Geobase Set

Overall, it must unfortunately be said about this test set that it does not really come close to a real application due to its low complexity. In particular, the absolut focus on the "State" class usually makes queries much easier, since no connection between two objects does not pass through "State". The age of the set is also noticeable in the simplicity questions as such. The possibility to ask cross-class queries is never used. For example, one could ask for all the routes that go through a state and expect the union of rivers and roads. Other easy multi-class queries would be like the *names of all relevant things in an area*, etc. In fact, more than one literal is never asked for in the questions either. The main difficulties encountered in this test set are ambiguities in names and the use of non-synonymous terms. For example, the names of rivers, cities and states are often the same. This often leads to questions where even a human can only guess what is meant, e.g., *GeoBaseQuery175 "How many people live in New York?"*. Here it cannot be said whether the solution means the city or the state. In such cases, the solution has been adapted so that the user gets both results presented and to which instance they belong.

In the case of *Athena* [6], such questions were counted as correct if the same confidence applied to both *State* and *city*, even if the randomly chosen result was not the intended one. This is effectively the same, but less practically useful.

6.2.2 Mondial

Although much larger and more complex, Mondial [87] and the Geobase dataset are very similar in their application domain. The ER diagram of Mondial is shown in Figure 6.10. In fact, the Geobase test set can almost be seen as a very restricted subset of Mondial (only the numbers and abbreviations of states (what are provinces in Mondial), low points and roads are not contained in Mondial). However, since care was taken when creating the operations of EvolNLQ that the vocabulary of the domain does not matter, the agents do not really have an advantage for being trained in a similar domain. Unlike the Geobase test set, Mondial offers significantly more opportunities to state questions, but also to make mistakes. It covers many modeling issues, e.g.,

as attributed relationships that must be mapped to reified classes in RDF (and in several other data models), transitive and symmetric relationships.

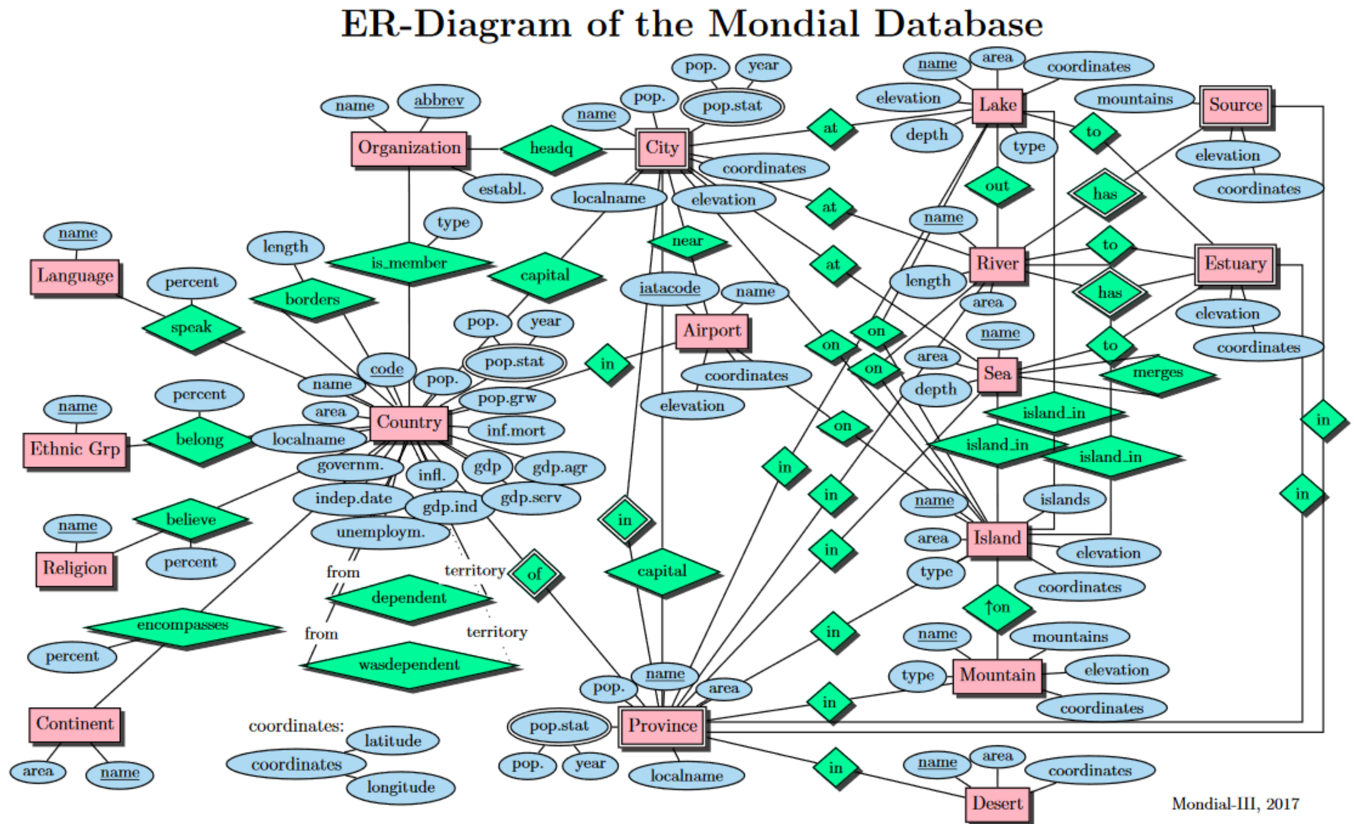


Figure 6.10: ER Model of the *Mondial* database [87]

The big disadvantage is that so far no test set for Natural Language Queries has been created for this set and therefore it is especially not used by other approaches to make a qualified statement about the performance. In course of this work such a test set was therefore created. In the questions created, care was taken to ensure that each is unique to some degree and introduces another relevant aspect; repetition was deliberately avoided. Overall, this set of questions is intended to be able to show the limitations of the approach, therefore it is expected that all approaches will have significantly lower success rates than with simpler benchmarks (such as the Geobase set or Movie database). The entire set of questions against Mondial consists of 63 questions and can be found in Appendix A.3 along with the partial coverage achieved. Of those queries, EvolNLQ was able to achieve a coverage of 96.91% or more importantly a recall of 87.30%, i.e., 55 of 63 questions were

perfectly solved.

6.2.3 Movie Benchmark

The Geobase and *Mondial* sets have been used in parts for training and for testing. The Move benchmark databse described in this section has then be used only for benchmarking.

The paper *A comparative survey of recent natural language interfaces for databases* [16], a comprehensive investigation has been done, which possibilities which NLQ approach offers. For this purpose, a new database has been created, which contains movies and persons; its ER model is shown in Figure 6.11.

For this database, 10 questions were then developed (*Q1* to *Q10*) these are supposed to map certain aspects that are typical for NLQ.

In addition to the specific question, each Q is equipped with certain tags. These tags refer to the algebraic constructs that are used/needed in the resulting query. Thus, they are somewhat SQL-specific, having a rather indirect correspondence to the issues in NLQ understanding. The article mentions that due to the fact that almost none of these approaches is actually freely available, the authors relied on statements in the literature to determine where the limitations of an approach are. The comparison is thus based on comparing examples with other questions from publications of the authors of the respective systems, and mapping these according to the tag system. If an approach states to be able to handle all associated tags of a Q_i, this question is considered correctly answered.

Name	Query	Tags	Reference
Q1	Who's is the director of 'Inglourious Basterds'?	J, F(s)	Example 33
Q2	All movies with a rating higher than 9.	J, F(r)	Example 34
Q3	All movies starring Brad Pitt from 2000 until 2010.	J, F(d)	Example 35
Q4	Which movie has grossed most?	J, O	Example 37
Q5	Show me all drama and comedy movies.	J, U	Example 38
Q6	List all great movies.	J, A	Example 40
Q7	What was the best movie of each genre?	J, A	Example 41
Q8	List all non-Japanese horror movies.	J, F(n)	Example 43
Q9	All movies with rating higher than the rating of 'Sin City'.	J, S	Example 45
Q10	All movies with the same genres as 'Sin City'.	J, 2×S	Example 46

Table 6.1: Q1 to Q10 from [16] and the assigned tags. (**J**oin; **F**ilter (**s**tring, **r**ange, **d**ate, **n**egation); **A**ggregation; **O**rdering, **U**nion; **S**ubquery)

The authors thereby consider their small number of questions rather as being typical representatives of the problems associated with their tags. The assumption is thus that the ability of any approach can be evaluated based on answering queries of a similar type, and a larger variety of the "same" question types would not be necessary. This contradicts the experience made in the context of this work, because a full sentence cannot isolate a particular feature, but always includes other

aspects as well. Even just changing instances can make a difference, depending on ambiguities of the instances involved. Therefore usually the interaction of the different components leads to more problems than the sum of problems of their individual parts.

The approaches with more than half of the results correct have been compared with EvolNLQ in Table 6.2. Here, ✓ stands for either successfully tested or found in the literature that the approach should be able to do so. ✗ stands for not implemented or unsuccessfully tested, ˜ for not fully applicable, and "?" for no information found in the literature and no way to test it.

While the literature-based results should be taken with a grain of salt, just being able to test EvolNLQ on a completely different (albeit very simple) database is helpful in assessing the quality of the approach and comparing it to others.

In the following, the results of EvolNLQ for *Q1* to *Q10* are presented for the individual queries from Table 6.1. The results shown in Table 6.2 are divided into two evaluation methods for EvolNLQ, the *test-based* and the *tag-based* one. The *test-based* variant of EvolNLQ is tested against the actual queries Q_i, to have a comparison to the approaches that have actually been tested by [16] on the Q_i. The *tag-based* results should be comparable to the literature-based evaluations, i.e., using another, own, question that satisfies the same tags. Therefore in case of failure for an original Q_i, a more specific variation of the question was created, which contains the same tags and issues, but not as much imprecision was tested also to have a valuation comparative to the literature-based testing.

Name	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	$\sum$ ✓
NLQ/A [88]	✓	✓	?	✓	✓	✓	✓	?	?	?	6
Google Assistant	✓	˜	✗	✓	✓	˜	˜	✓	✓	✓	6
Athena [6]	✓	✓	✓	✓	✓	✓	˜	✗	✓	✗	7
SPARKLIS [89]	✓	✓	✓	˜	✓	?	˜	✓	✓	✓	7
NaLIR [2]	✓	✓	?	✓	✗	✓	✓	✓	✓	✓	8
EvolNLQ (test - based)	✓	✓	✗	✗	✓	✓	✗	✗	✓	✗	5
EvolNLQ (tag - based)	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	8

Table 6.2: Selection of the results of Affolter et al. [16] and comparison to this approach.

In the following, each question is discussed individually and what additional challenges each of the questions still poses. In fact, the difficulties of each query are usually not related to the problems indicated by the tags, but are mostly in inaccuracies or other not tag-related challenges in the question. For the additional challenges, new tags are defined and are given listed below under *Additional Challenges* and are explained in more detail later in the text, which also states how far EvolNLQ solves the task.

6.2.3.1 Q1: Who's is the director of 'Inglourious Basterds'? (sic!)

Example 33 *Question Movie1: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie1 are depicted in Figure 6.12.*

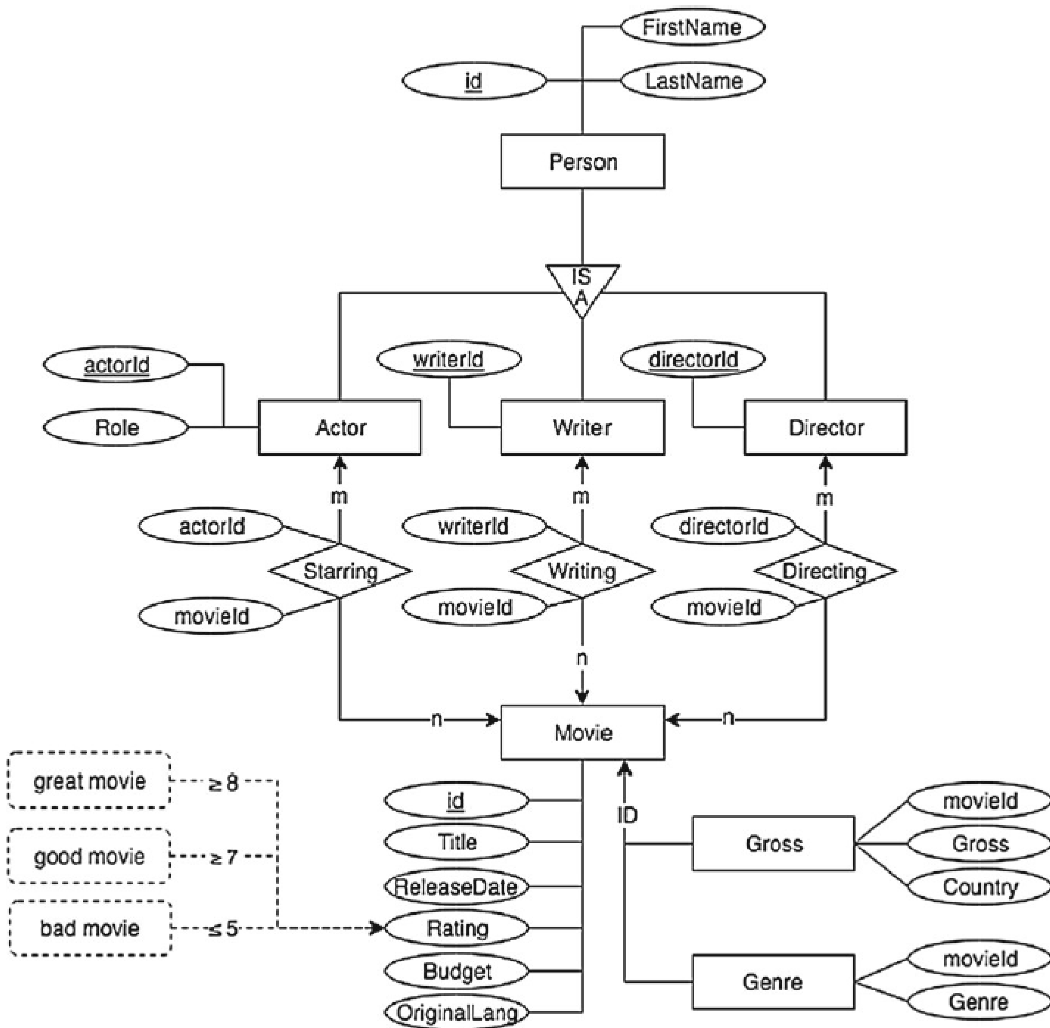


Figure 6.11: "Ontology of the sample World "movie database" from [16]

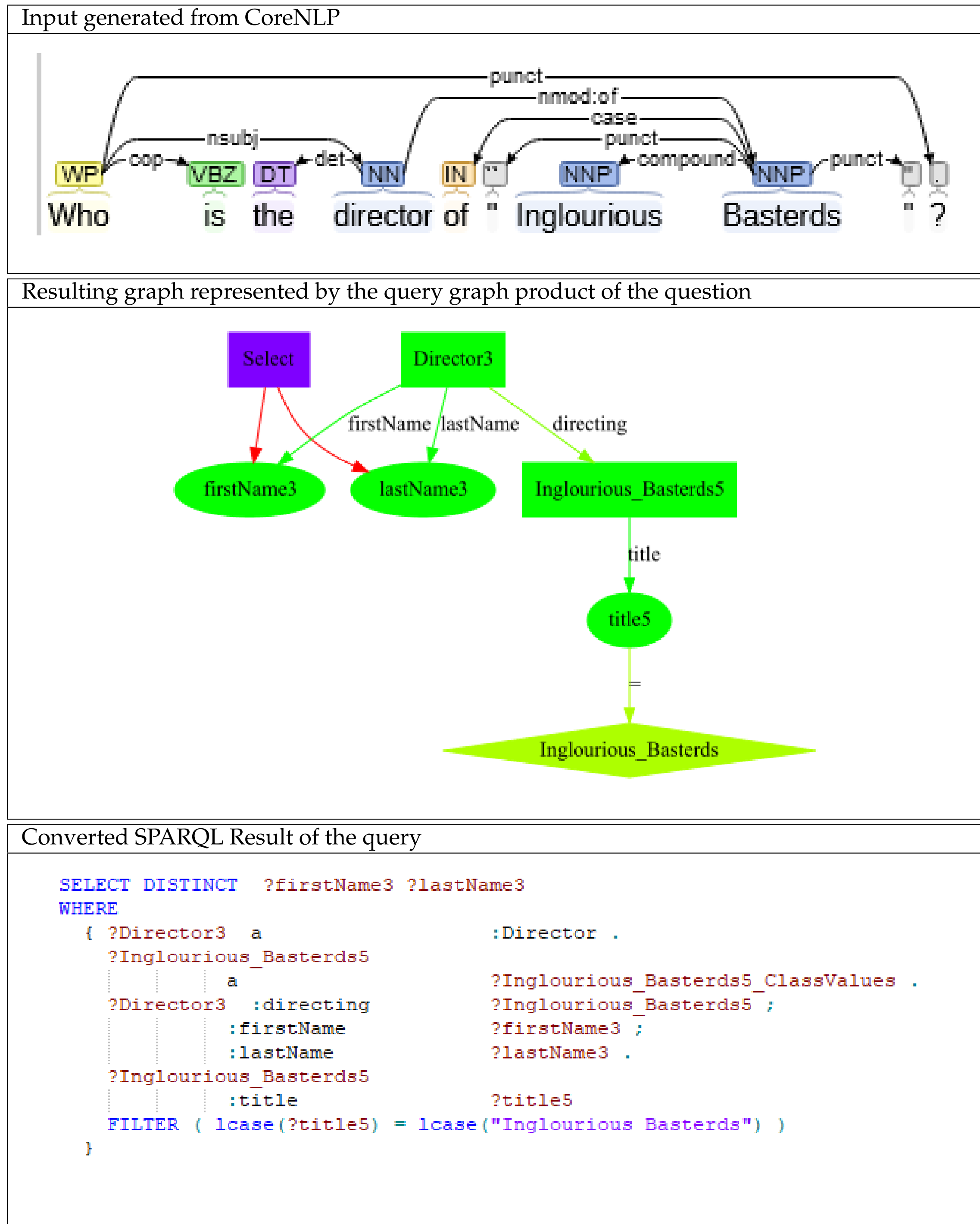


Figure 6.12: Question *Movie1* with CoreNLP anntotations, query graph, and resulting SPARQL query

Assigned Tags: Join, String Filter

Additional challenges: Proper Name, Instance recognition, Class hierarchy, identifying properties

While joins are much more challenging in SQL (using the keys instead of object variables) than in SPARQL, the challenge of this question is rather elsewhere: Q1 contains a compound proper name, especially one that is still plural. Approaches that use a porter stemmer [85] or similar in this case will get "basterd" without "s" according to the stemmer rules. The quotation marks might be a help depending on the approach. EvolNLQ's Ellyn2554 (see Section 6.1.9) was trained with and without quotation marks in questions and has learned to disregard them, since they are not reliable.

Once *Inglourius Basterds* is identified as the name of an entity, the approach still needs to figure out what this entity actually is, which means a mapping of a data value to a class or table of the database has to be done.

Finally the *Director* itself would be in RDF just an IRI which is normally not what a user expects as a result. Therefore an identifying literal is needed, which in case of *Director* are *ID, first name*, and *last name*. The query returns first name and last name. Original Q1 solved.

6.2.3.2 Q2: All movies with a rating higher than 9

Tags: Join, Range filter

Additional challenges: Single identifier property, Numeric comparison

Example 34 *Question Movie2: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie2 are depicted in Figure 6.13.*

This question can be answered very easily, there are no instances and the names of the class and the property are specified correctly, so this question is not a challenge for almost any approach that can handle simple filters and numerics. Original Q2 solved.

6.2.3.3 Q3: All movies starring Brad Pitt from 2000 until 2010.

Example 35 *Question Movie3: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie3 are depicted in Figure 6.14.*

Tags: Join, Filter Date

Additional challenges: Between, Instance identifying with multiple identifiers, Matching numeric value to certain property

This query has the goal to check whether filtering is done for dates. Ultimately, however, it must be said that there is very little evidence in the question that this is a date, since only two numerical values occur that were not specified in more detail. Also, the ER model does not identify the datatypes of attributes. Concluding that any four-digit number is automatically a year leads to

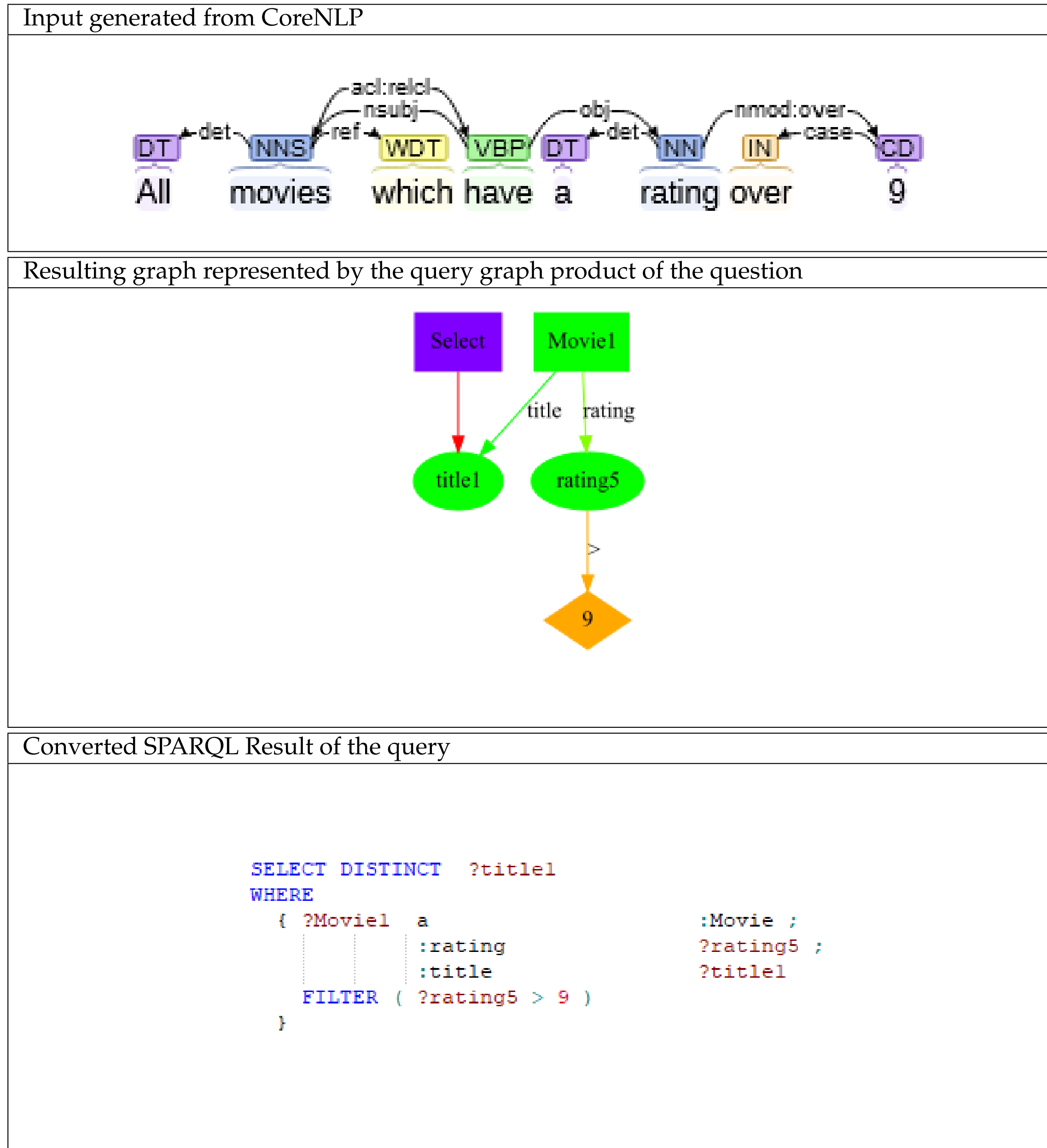


Figure 6.13: Question *Movie2* with CoreNLP anntotations, query graph, and resulting SPARQL query

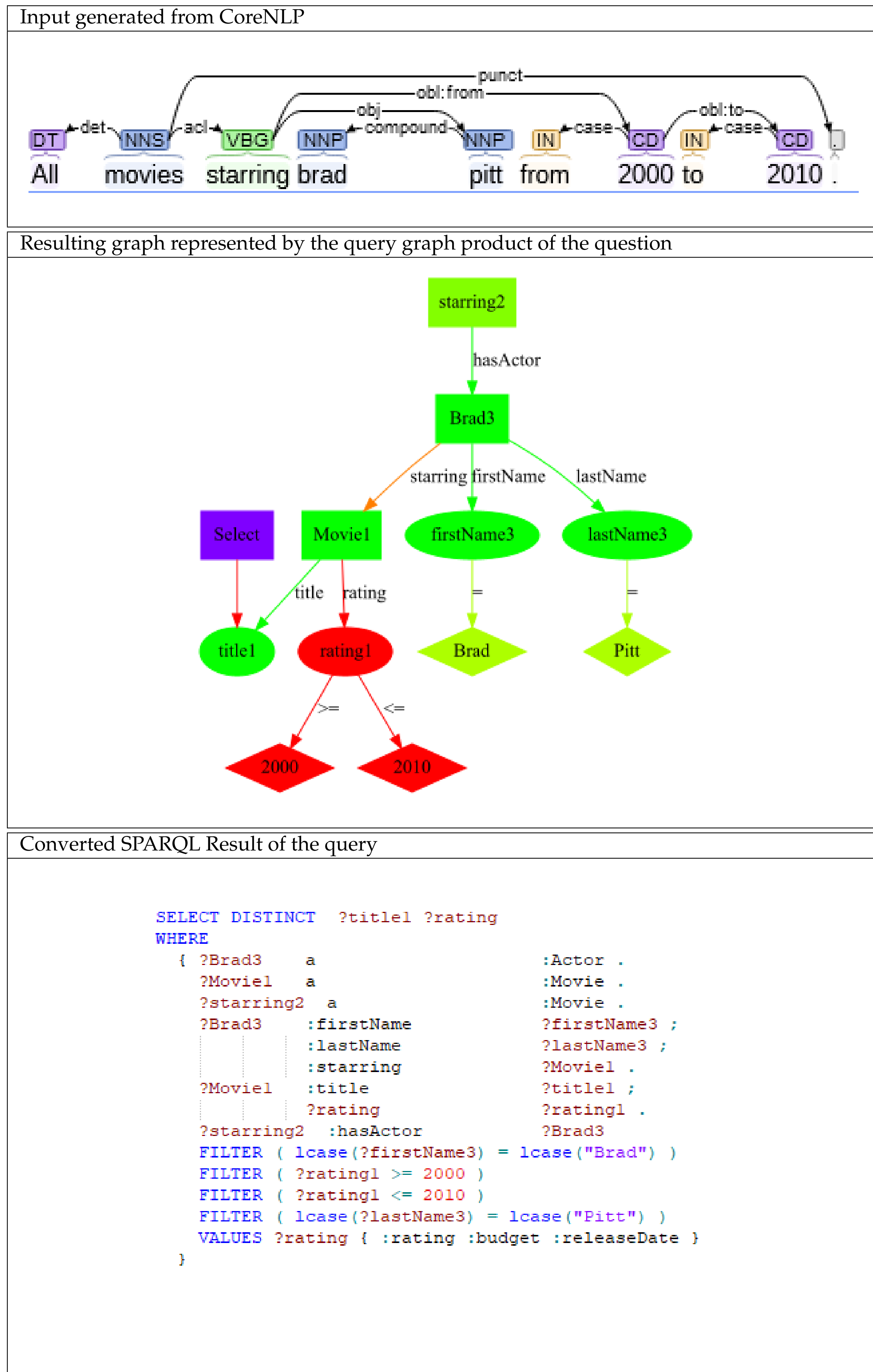


Figure 6.14: Question *Movie3* with CoreNLP anntotations, query graph, and resulting SPARQL query

too many false positives in general. Therefore EvolNLQ cannot conclude from four-digit numbers alone that they are years. In this special case where two literals and the keyword "until" is used, it can be concluded that the date datatype is meant. Otherwise, besides the *ReleaseDate*, *Movies* also have the numeric literals *Rating* and *Budget* and as a human user one can probably already conclude that if *great movies* start at > 9, 2000 to 2010 is too high and for a *budget* of 2000$ would be very low for a *movie* with *Brad Pitt*, but the value could possibly not be in dollar but for example in 1k dollar increments and therefore a $2000 * 1000\$ = 2,000,000\$$ and be much more reasonable. Nevertheless an interpretation of such a number without data would only be applicable for specific systems, with deep domain knowledge or for systems with broad general knowledge and much higher complexity.

However, since this database has a minimal data set and apart from *instance identifiers* (see Section 4.2.2.4 and Table 4.3), EvolNLQ does not use any data, it is not able to decide which value is the correct one and it offers all numeric values and the user has to decide which one they actually meant. Even if not according to the tag, this request is still a good BETWEEN request, with the additional challenge of also using multiple identifiers via first and last name.

Nevertheless, the difficulty here is again not in the tags specified, so it is not easy to say whether EvolNLQ can answer questions of type Q3. In the concrete query, the filtering of the values is done properly, but it is not recognized that it is about years, so the union of all numeric values is requested. However, with enough information, as for example in the alternative Movie Query 3b in Example 36 created in the context of this work, the filter can be applied in this way without any problems, even if there is technically no difference to non-date values, as long as the date is only stored as a year and not as a complex time like for example the XML Schema datatype xsd:datetime [82].

The resulting SPARQL queries, while correct, are not optimal since ?starring2 is ultimately redundant. This leads to the fact that all combinations of films of Brad Pitt with all movies of him in the period from 2000 to 2010 are bound, but since only the title is selected, and by default each query makes a DISTINCT selection, the result set is correct, but ineffectively determined. For Q3, the tag-equivalent Query Q3b is solved.

Example 36 *Question Movie3b: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie3b are depicted in Figure 6.15.*

6.2.3.4 Q4: Which movie has grossed most?

Example 37 *Question Movie4: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie4 are depicted in Figure 6.16.*

Tags: Join, Ordering

Additional challenges: Max aggregation (more difficult than plain ordering which has already

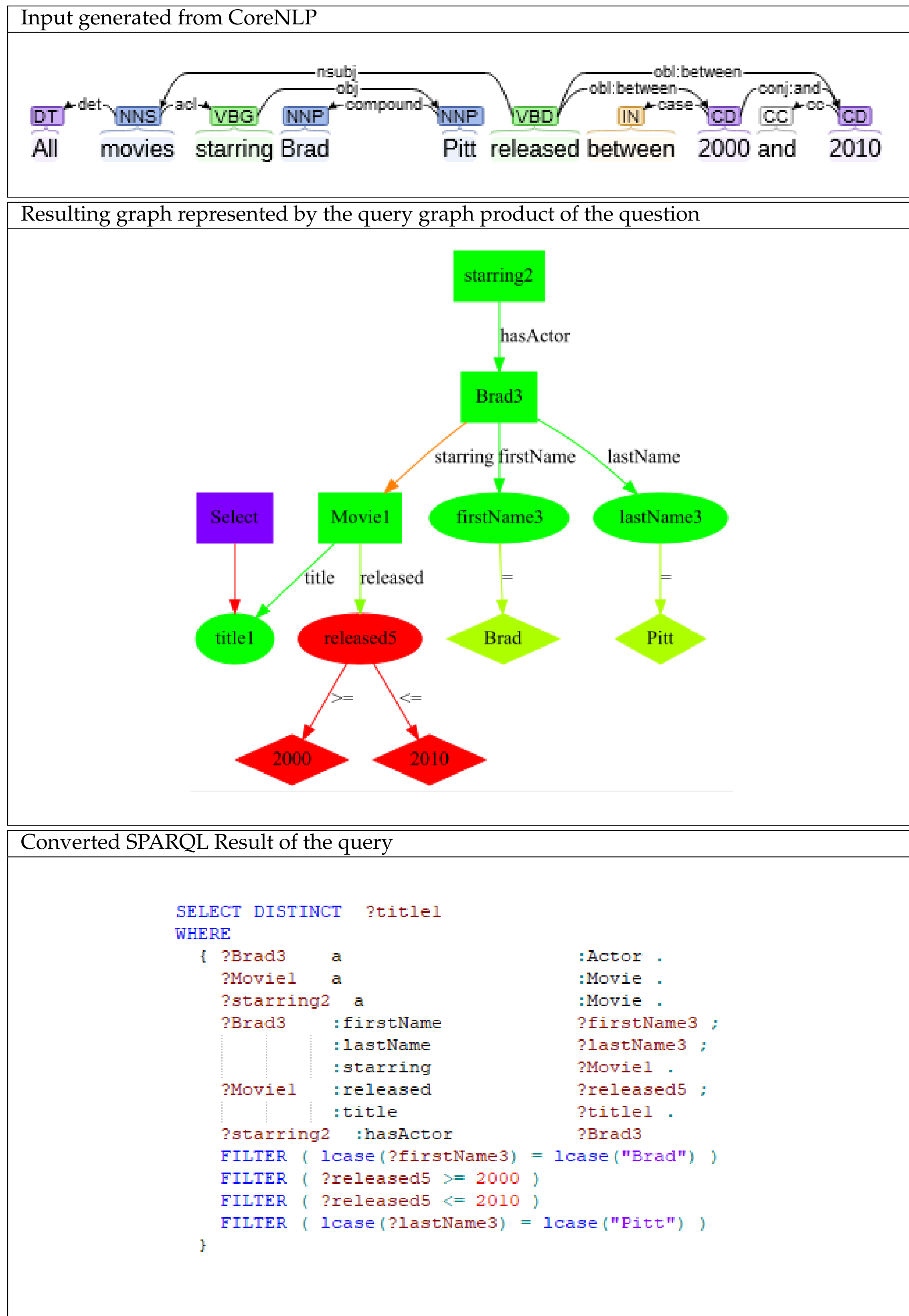


Figure 6.15: Question *Movie3b* with CoreNLP anntotations, query graph, and resulting SPARQL query

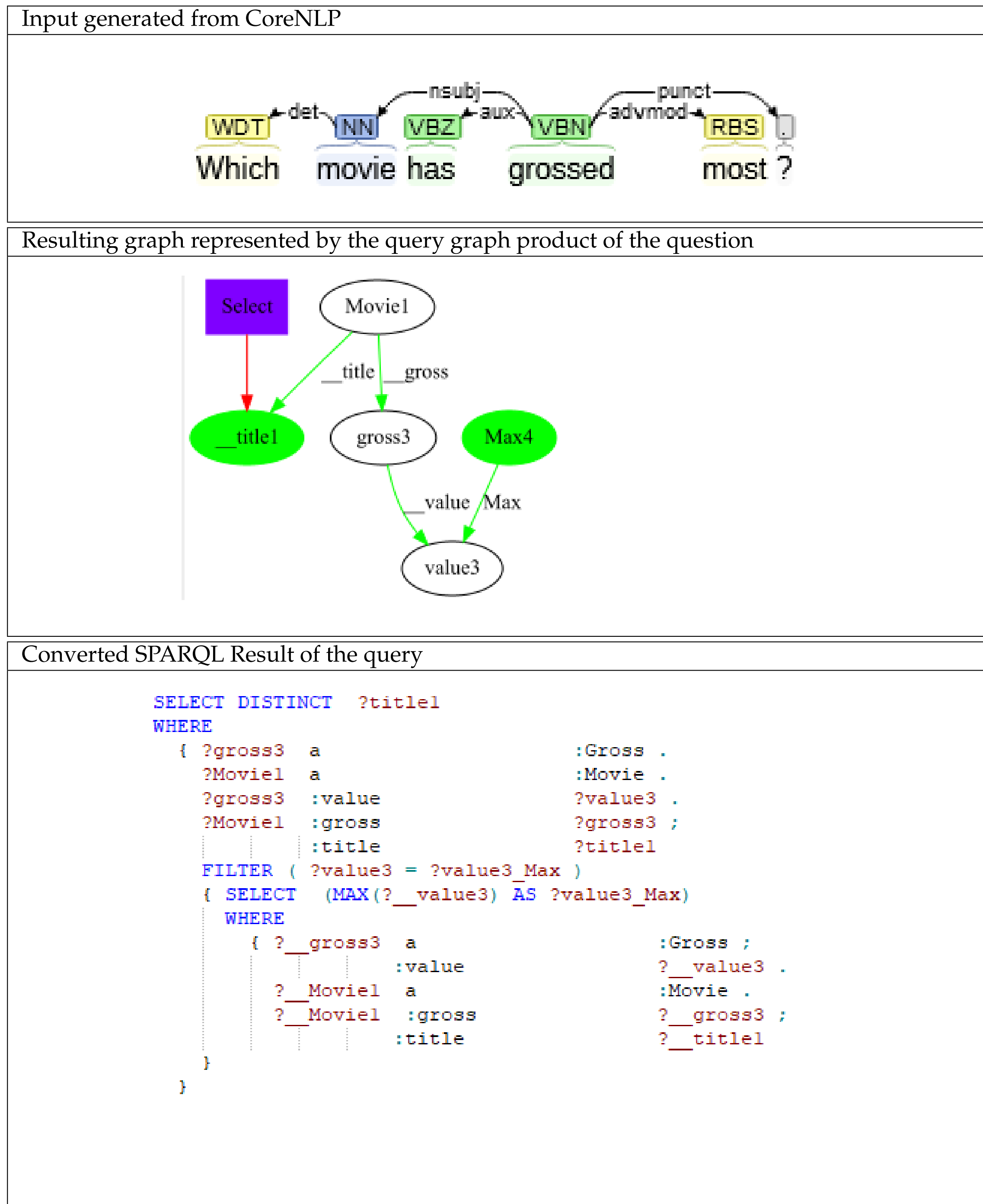


Figure 6.16: Question *Movie4* with CoreNLP anntotations, query graph, and resulting SPARQL query

been used with the Range filter tag in Q2), Implicit group by, Reified multivalued complex attribute

In this question, it should be noted that a property is used which is in the database associated to the "Gross" class which actually is a *reified* multivalued complex attribute (note the special, rather non-standard notation in the ER diagram). Also, the implicit meaning of "most" must be recognized, not to mean the highest value, but the highest sum aggregation over all countries, grouped by movie. So far, the question is much more complicated than denoted by its tags. EvolNLQ did not solve Q4, because "most" is only translated to "max", not to "sum(max)".

6.2.3.5 Q5 Show me all drama and comedy movies

Example 38 *Question Movie5: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie5 are depicted in Figure 6.17.*

Tag: Join, Union

Additional challenges: For this question is already said in the publication: "The query can either be interpreted as 'movies that have both genres' (intersection) or 'movies with at least one of those genres' (union). The expected answer is based on the union interpretation" [16]. EvolNLQ interprets the "and" literally and corresponds to the intersection variant with the original question. However, since a union is actually desired here, the question in 5b was changed to use an "or" instead of the "and", which produces the desired result using a union. Original (ambiguous) question solved, disambiguated tag solved.

Example 39 *Question Movie5b: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie5b are depicted in Figure 6.18.*

6.2.3.6 Q6: List all great movies

Example 40 *Question Movie6: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie6 are depicted in Figure 6.19.*

Tags: Join, Aggregation

Additional challenges: Reasoning

This query mainly tests the reasoner (that [16] assumes to be contained in any such system) to deal with the rules that define "great movie". With EvolNLQ, the class hierarchy itself is contained and specified in the OWL ontology, and the SPARQL query must then be evaluated using an OWL reasoner of the database. So, the reasoning itself is delegated back to the database, hence this query is very small in SPARQL. The only challenge is to recognize that this is a class composed of two words and one of these words could be another class. Original Q6 solved.

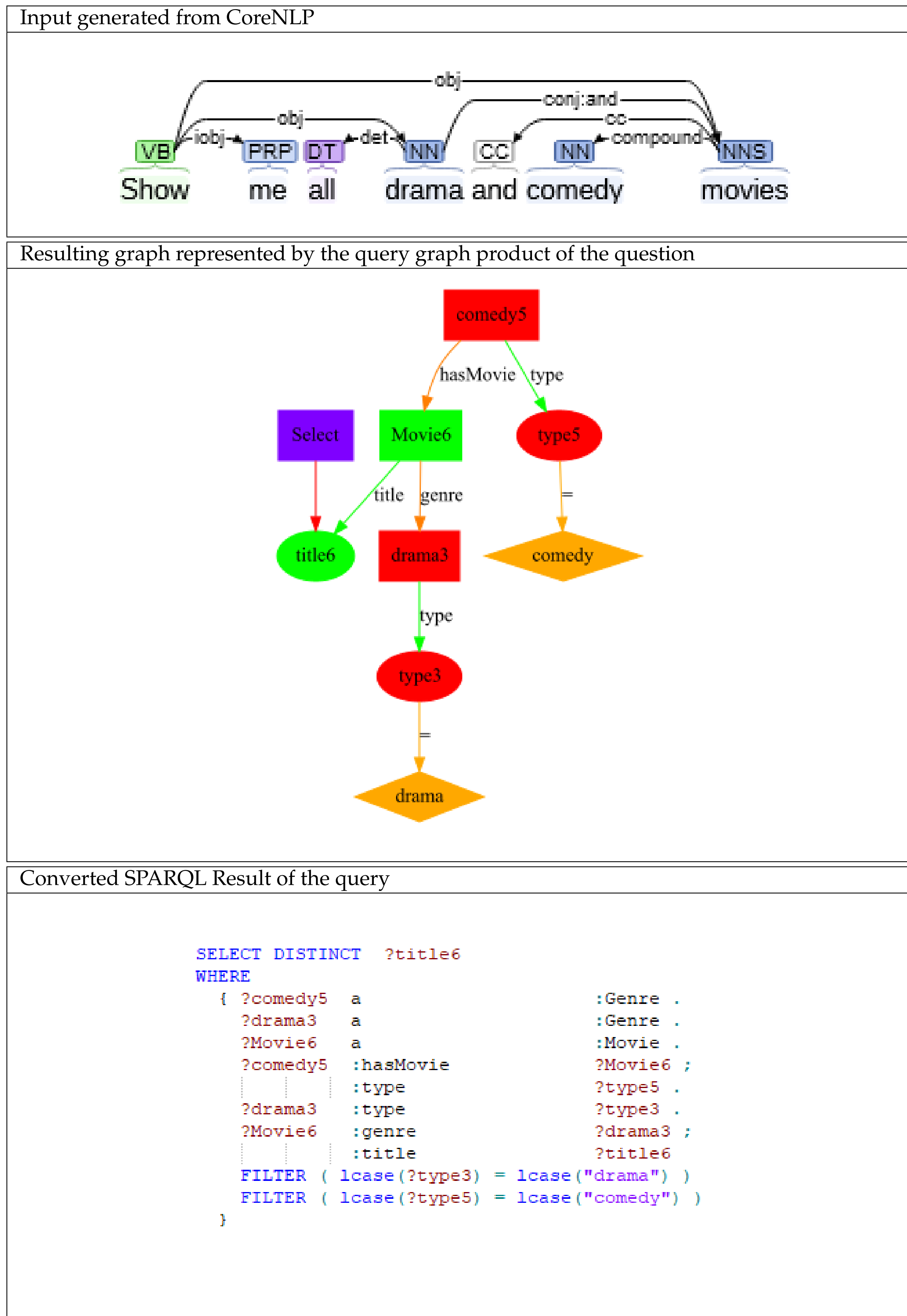


Figure 6.17: Question *Movie5* with CoreNLP anntotations, query graph, and resulting SPARQL query

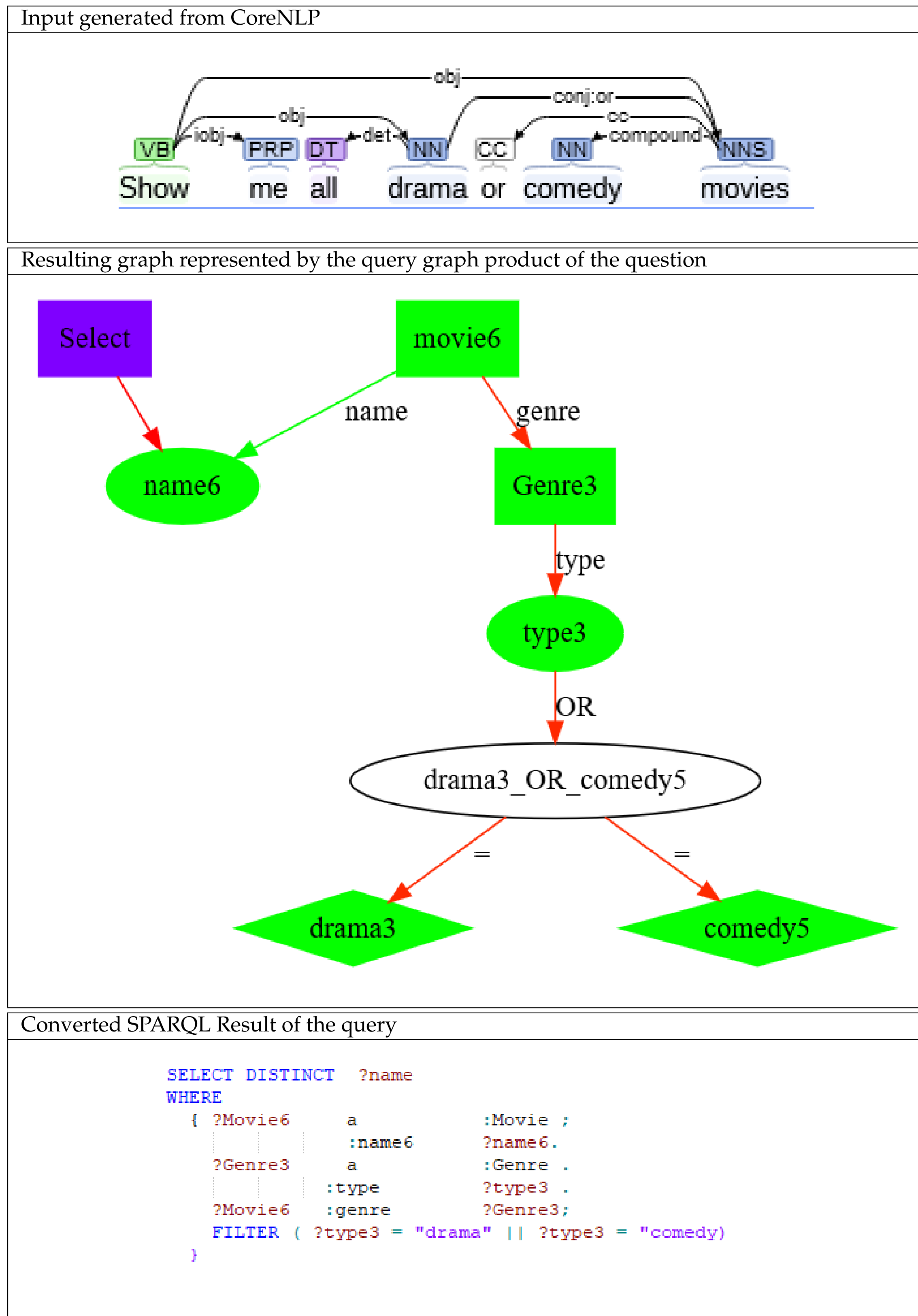


Figure 6.18: Question *Movie5b* with CoreNLP anntotations, query graph, and resulting SPARQL query

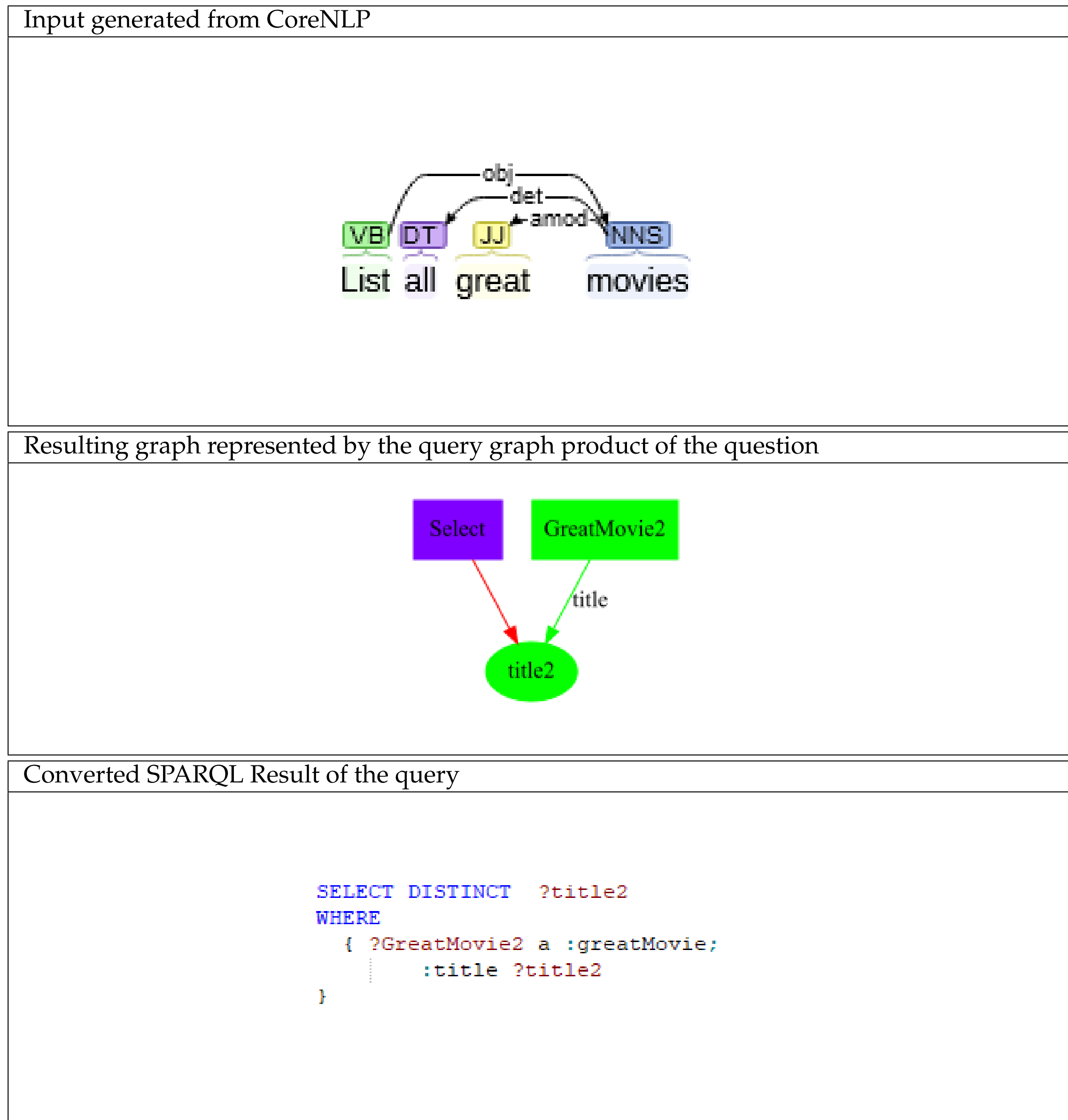


Figure 6.19: Question *Movie6* with CoreNLP anntotations, query graph, and resulting SPARQL query

6.2.3.7 Q7: What was the best movie of each genre?

Example 41 *Question Movie7: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie7 are depicted in Figure 6.20.*

Tags: Join, Aggregation

Additional challenges: Group by, Understanding quality-related properties

This question is answered incorrectly in two aspects. First, EvolNLQ does not correctly recognize that grouping by genre is necessary, and second, it is not clear what makes a movie good. Since there are two numerical values for Movie, Rating and Budget, EvolNLQ is confused.

Question 7B clarifies the question a little, limiting "best" to the rating property. Then, EvolNLQ solves the GroupBy correctly.

Example 42 *Question Movie7b: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie7b are depicted in Figure 6.21.*

6.2.3.8 Q8: List all non-Japanese horror movies

Example 43 *Question Movie8: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie8 are depicted in Figure 6.22.*

Tags: Join, Filter Negation

Additional challenges: Non-identifier identification, Composed negation

Question 8 is not a good representative of negations. Although this question has a negation, it is the recognition of "non-Japanese" that is the problem. First of all CoreNLP does not recognize the word correctly as two different entities, further the word refers to a value without any context. So this can only be solved by searching the whole database for this value, finding where it is used, and excluding all corresponding properties. With a larger database this procedure would generate an enormous number of false positives. So the fact that this value is still negated is not the real problem.

While concluding from "non-Japanese" that the original language of a film is not allowed to be Japanese, is understandable for a human, but this question needs a very high level of reasoning and context to be answerable. Because this question is so difficult, it is suspected that programs that pass this test stated alternative, simpler negation tasks in the literature as answerable. For EvolNLQ, a similar but better formulated question is presented in Q8b. For Q8, the tag-equivalent question Q8b is solved.

Example 44 *Question Movie8b: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie8b are depicted in Figure 6.23.*

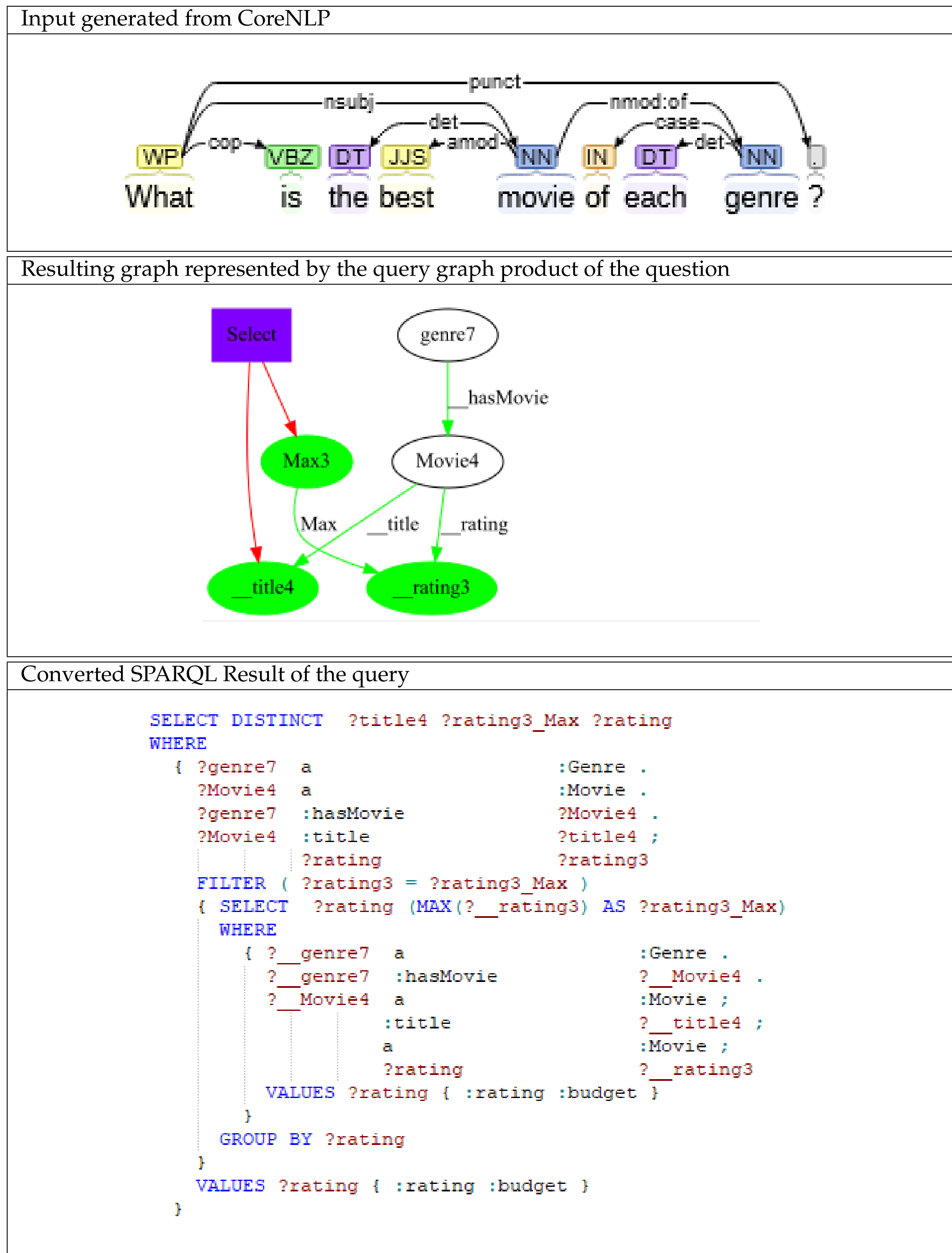


Figure 6.20: Question *Movie7* with CoreNLP anntotations, query graph, and resulting SPARQL query

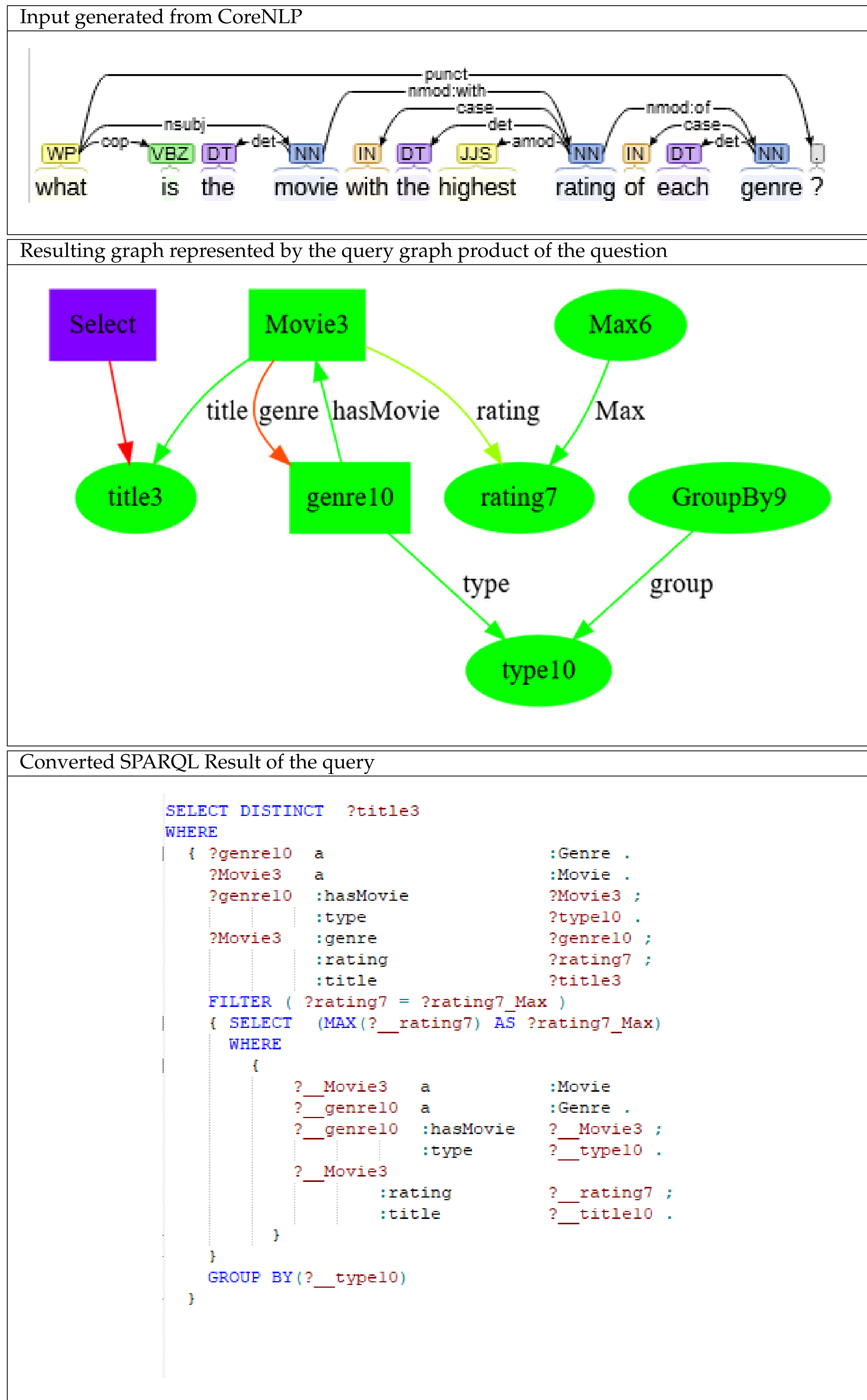


Figure 6.21: Question *Movie7b* with CoreNLP anntotations, query graph, and resulting SPARQL query

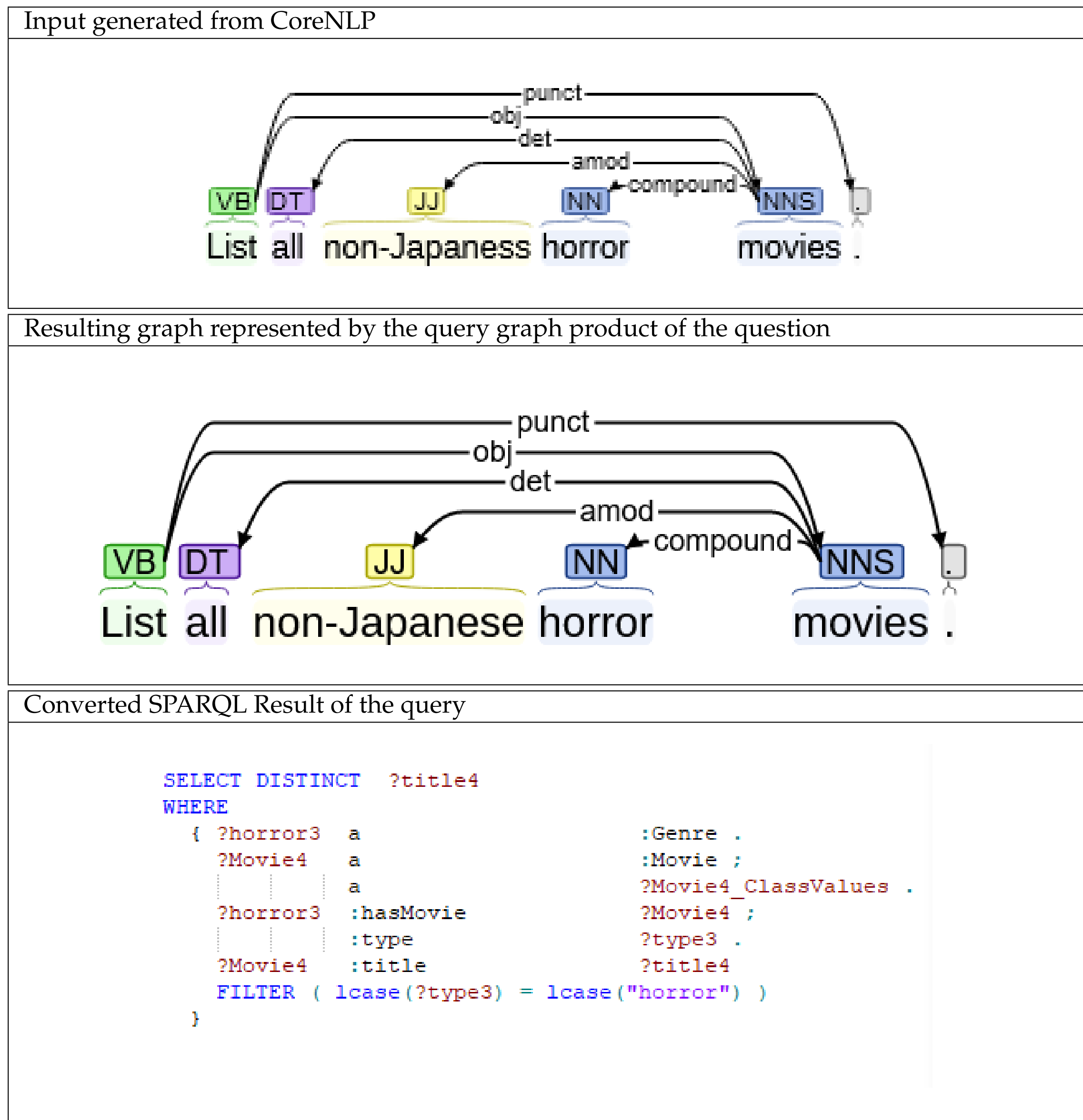


Figure 6.22: Question *Movie8* with CoreNLP anntotations, query graph, and resulting SPARQL query

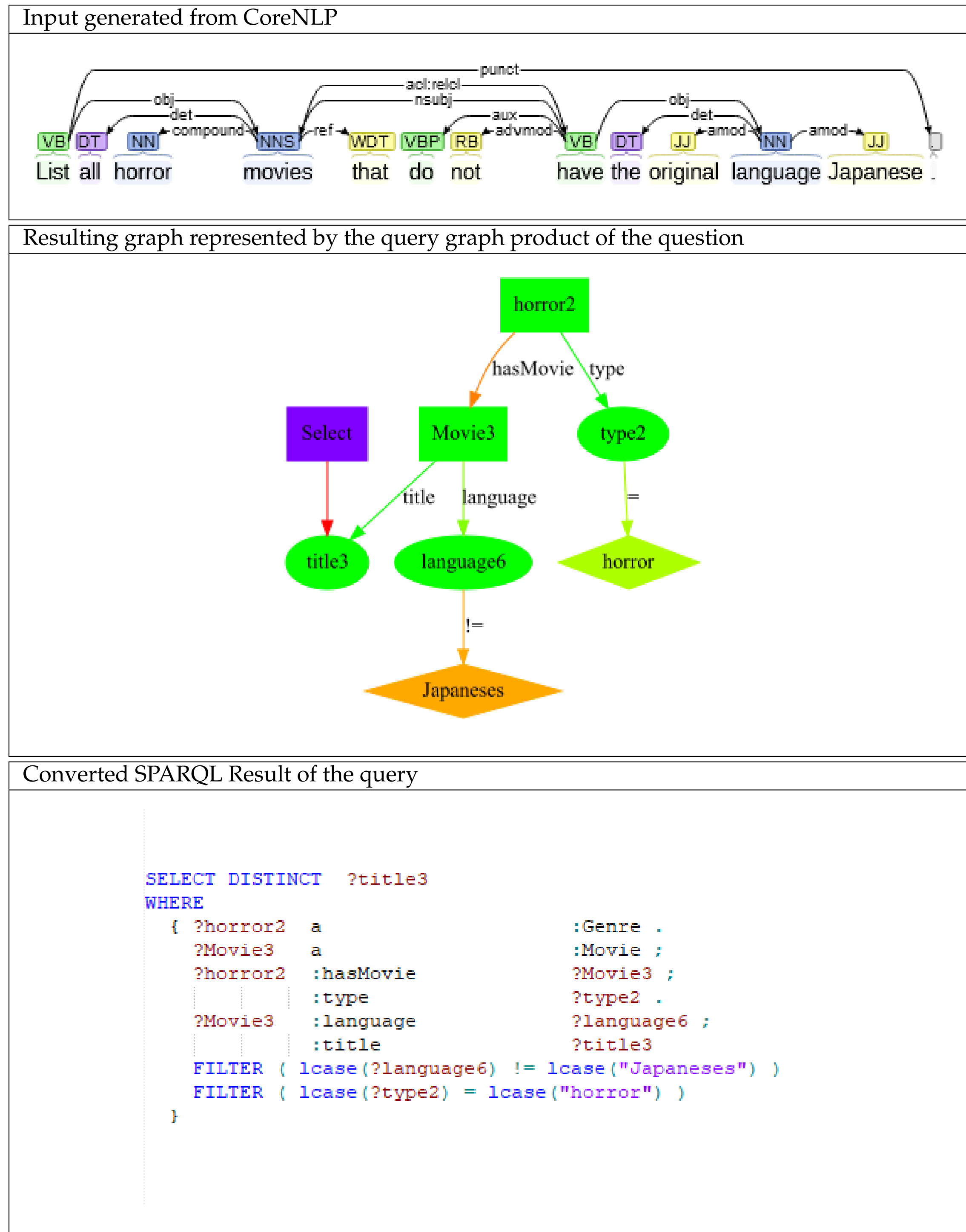


Figure 6.23: Question *Movie8b* with CoreNLP anntotations, query graph, and resulting SPARQL query

6.2.3.9 Q9: All movies with rating higher than the rating of 'Sin City'

Example 45 *Question Movie9: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie9 are depicted in Figure 6.24.*

Tags: Join, Subquery

Additional challenges: Self-join, Instance comparison

Question 9 is a good comparison question in combination with instance recognition. For both instances there is once used the word Rating. Rating is also the correct name of the property being compared. The only non-trivial task is to create a self join which has to be aliased correctly in SQL – which is not a problem when translating to SPARQL. For being labeled as the second most difficult question, this was rather easy. Actually, it does not even need a subquery (although it is tagged so). Original Q9 solved.

6.2.3.10 Q10: All movies with the same genres as 'Sin City'

Example 46 *Question Movie10: The question text with CoreNLP annotations, query graph, and resulting SPARQL query for Movie10 are depicted in Figure 6.25.*

Additional challenges Join, equality of sets, 2 $\times$ Subquery

Note that this question actually does not mean "which movie has at least one genre that also "Sin City" has", which would be another simple self-join as in Q9. It seems that the intention (tagged with "2 subqueries") is clearly that the authors intend a relational division, i.e., movies that have *at least* those genres that "Sin City" has (all movies such that there is no genre that Sin City does not have, i.e., two nested subqueries). Note that "the same ... as" literally means also to compare the other direction, which would require four subqueries. There may be doubts whether the other systems that are claimed to solve this question actually did it in that right way. EvolNLQ did not solve Q10.

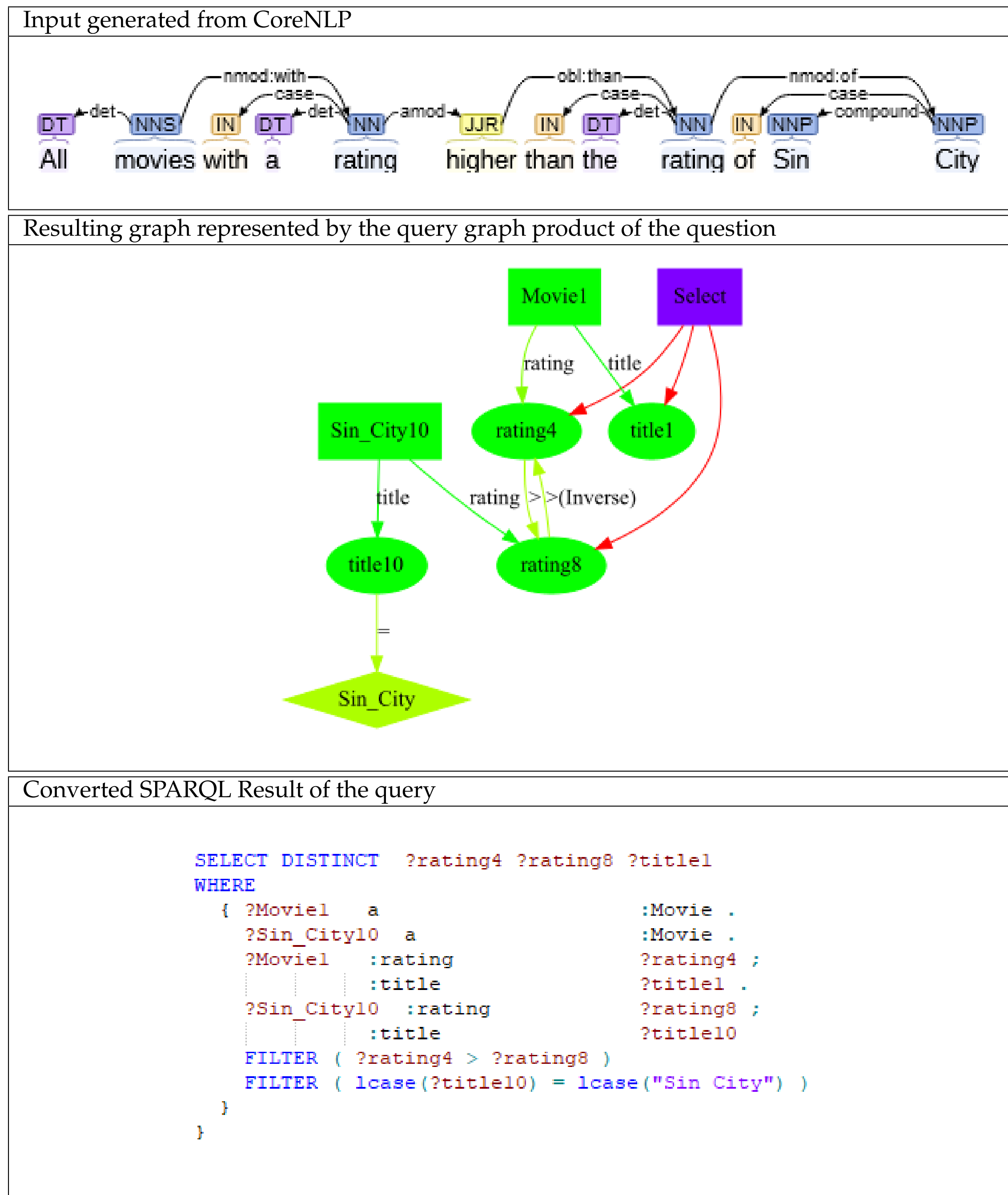


Figure 6.24: Question *Movie9* with CoreNLP anntotations, query graph, and resulting SPARQL query

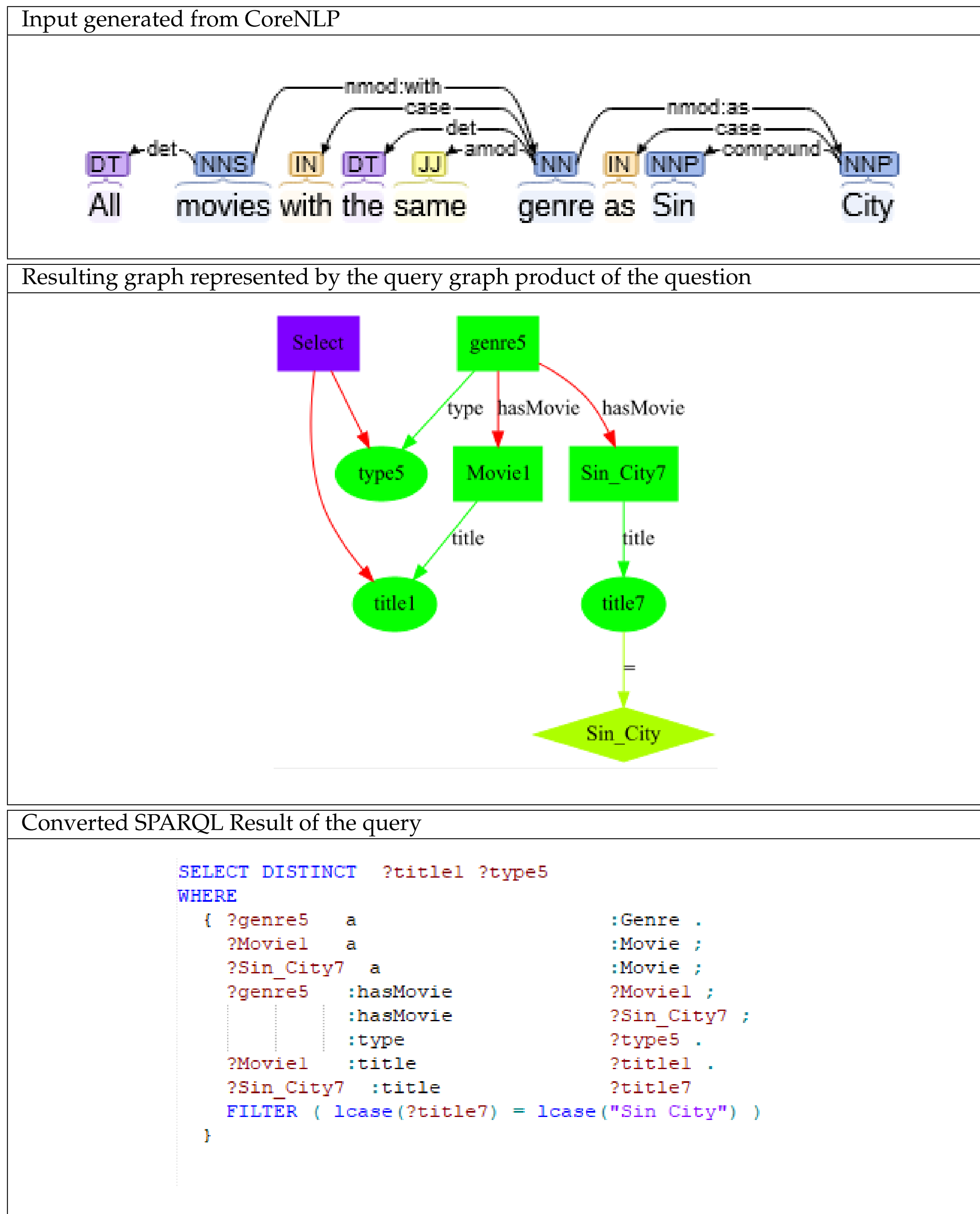


Figure 6.25: Question *Movie10* with CoreNLP anntotations, query graph, and resulting SPARQL query

Chapter 7

Conclusion

In this work, a framework has been developed that can be used for training and improving agents based on evolutionary programming. It allows easy and fast addition of new methods and training of agents for a specific type of tasks. For this purpose, an energy-based selection procedure was developed, following artificial life methods, to generate a wider variety of agents. The framework was used to develop an application that translates natural language queries into correct SPARQL queries wrt. a given ontology.

Logical, graph-theoretic, and ontology-based methods have been developed for this purpose. Agents were then trained to combine these methods to answer the posed training queries. The restriction to datasets with an ontology and the preparation of the metadata by RDF2SQL gives the possibility to effectively use the metadata from the ontology and provides a much larger amount of information than for example a pure SQL database. Due to the formal correctness of the ontology this data is much more reliable than for example huge knowledge bases like DBPedia.

Thus, on the one hand, it was possible to define powerful graph operations in the nodes that extensively and correctly use metadata of the ontology, on the other hand, evolutionary programming could take care that the many variants, inaccuracies and special cases due to the usage of natural language could be handled. The resulting mix of intrinsically reasonable and correct operations combined with the experience-based structures of the agents has proven to be a successful method for translating natural language into SPARQL. Using test sets, besides the obvious purpose for agent learning, has the added advantage that problems in unsolved tasks can be easily found and analyzed using the graphical user interface provided by the framework.

Overall, this combines approach turns out to be a mixture between *black-box* and *white-box* reasoning an learning: Since all nodes perform comprehensible operations, it is also possible to detect what is missing, incorrectly assigned, or not considered, so that it is possible to create more nodes for these cases, as presented in the example of composite filters. This iterative process is supported by the framework in which both by hand and by learning these new nodes can be integrated and tested.

On the other hand, the exact structure of the developed agents is hard to analyse, to understand, and to debug. In this way, a collaboration between humans and algorithms is created which, as shown in this approach, yields results that can compete with current state of the art approaches.

This was demonstrated on three different ontologies and benchmarks. Furthermore, agent creation was tested under different conditions and with or without the methods developed in this thesis or compared to commonly used methods for the same tasks. Here it could be shown that a selection method specifically used for this method performs better for this application than traditional and/or commonly used approaches. Nevertheless, it must be said that this work will not conclude the topic of natural language database interfaces. As with other state of the art approaches, EvolNLQ still does not allow for unrestricted communication with a database as a human expert would, nor does it provide the same range of capabilities that a professional database user would have. Still, it can be a great tool for anyone who wants to use a database without knowing its structure or the database language to get information from simple to moderately difficult queries.

Complex database queries in natural language and with common vocabulary are too often too demanding (as a rule of thumb, anything that requires complex concepts (e.g., nearest neighbor, standard deviation, etc.) or need more than a sentence or two to be described), often requiring more general knowledge and intuition than this kind of system can ever have.

In addition, more complicated queries, for example for statistics, are often easier to describe formally than in natural language, and there is no real advantage in the natural language interface anymore, on the contrary it makes the task more difficult and can be an additional source of errors.

Ultimately, anyone who has ever taught a database query language will attest that correctly answering a complex question remains a challenge even for trained humans, from that perspective EvolNLQ can by all means be seen as a successful intermediate step towards a fully satisfactory solution.

Future Work Several paths could be pursued here, one being to further expand the training set and spend more time training agents. This will probably be accompanied by the development of new node types and product types and thus further developing the overall application along the paths already taken. In particular, concepts that have not been considered so far can be introduced. For example, nodes that can handle the transitive closure would be useful here, and perhaps more urgently nodes that can restrict the range of literals based on certain words, as has been shown for Q3, for example. Furthermore, of course, the framework can be applied to other applications.

Glossary

Agent An agent is a set of nodes and connections in a defined layout. Agent evolve through the training's phase of the EVolNLQ programm to optimize their problem solving. See Section 3.4 & 3.3 for more information. 8, 23–25, 27–29, 31–36, 38, 39, 45–47, 49, 50, 52–57, 59–71, 75, 77, 79, 82, 107, 108, 110, 128, 133, 134, 148, 154, 161, 164, 166, 167, 169, 174, 176–178, 184–186, 189, 190, 197, 198, 201, 204, 207, 210, 211, 214, 243–249

Agent Creator Panel A graphical user interface to analyse, manipulate and test agents. 183–185, 187, 189

Aggregation Product Modifies a ClassVariable product to execute a aggregation function over the value of the ClassVariable product. 92, 105, 106, 123–125, 127, 146, 167, 168

Atomic Product Products which do not contain any other product in their data tuple p_d. Opposite of compound product. 48, 49, 60, 85, 135, 150, 244, 248

Auxiliary Product A product which does not occur in the final result set, but can provide additional information for other nodes. More details see Section 4.1.21. 49, 107–110, 143, 246

Bloating Is a typical problem in *evolutionary algorithm*, which occurs when agents have an opportunity to grew, with out decreasing their score. In those cases the agents grow ever larger and slow down the computation of the simulation up to a point where the program runs out of memory or at least slows down to unacceptable degree. More information [13]. 64

claim Claim that an agent makes to the reward of a task by correct solutions. This is determined by the similarity of his solution with the sample solution. 65

ClassVariable Product A object or literal-Valued variable with a domain. More details see Section 4.1.13. 87, 88, 92, 94, 96–98, 100, 101, 103, 105, 106, 108, 113, 115, 119, 121–125, 127, 128, 130–134, 136, 137, 141, 143–147, 150, 151, 153–156, 158, 159, 161, 164, 168, 169, 173, 174, 179, 181, 188, 194, 205, 243, 244, 246

Collector Node Are a type of node, who stores incoming products instead of processing and forwarding them and has a learned/random ordering number. Only if there is no other activity in the agent and no other *collector node* with a lower order has anything to forward, a *collector node* releases their products. More details see Section 3.3.3.2. 142, 147, 149, 169

Comparison Product compound product which consists of three products to form a comparison. See Section 4.1.11 for more information. 85, 92, 96, 97, 119, 132–134, 153, 155, 247

Compound Product Products which contain at least one other product in their data tuple p_d. Opposite of atomic product. 48, 49, 60, 85, 86, 90–92, 94, 96, 100, 103, 105, 107–109, 141, 146, 148–152, 243, 244, 246, 248, 249

CompoundClassVariable Product A variable compound from two other ClassVariable products and an operator product, it is used to compute values. Section 4.1.15. 103, 136, 137

Conduit A Conduits is the anchor point for connection between nodes. Conduits receive or send products via connections to/from other conduits between two Runs. During their processing nodes access the received products of their conduits and afterwards they divide them between them for further distribution. Also see Section 3.3.2. 48, 51, 55, 56, 142, 148, 186, 187, 244

Confidence Graph A weighted directed graph consisting of the meta data of an ontology and the products of an query graph product used to derive missing information to close the graph of query. 154–157, 159, 161

Confidence Value Degree of confidence of an agent in its own product to fulfil a given task. 49, 50, 53, 55, 66, 135, 147, 160, 161

Configuration A configuration describes the specific components and their layout of an agent. 23, 54

Connection A connection facilitates the data flow between two conduits from two different nodes. For more details see Section 3.3.2 . 45, 48, 51, 52, 56, 63, 71, 184, 186–189, 244, 249

Constant TriplePart product with a value attribute which contains a static literal value, like a string or a number. 94, 96, 97, 104, 111, 113, 117–119, 151, 154, 155, 179

CoreNLP A natural language annotator developed by the Stanford University [44]. 19, 20, 40, 41, 45, 49, 51–53, 75, 76, 79–82, 87–89, 91–93, 95, 98–102, 110, 112, 116–118, 120, 125–127, 137–140, 165, 172, 178, 183, 211–213, 217, 219–237, 246, 247

Coverage Proportion of subtasks filled.. 197–199

Distinctness Product A distinctness product has a set of TriplePart products which are mutually different from each other and describe not the same entity. More details see Section 4.1.23. 106–109, 140, 141, 143, 169

Ellyn25545 Ellyn25545 is the most successful agent species so far. All benchmarks are executed with Ellyn. The XML version of Ellyn can be found in the Appendix A.1 Like all agents this agent has a designation composite of a randomly chosen name followed by how often the entire list of names was exhausted. 207, 210, 211, 259

Environment An environment contains a set of tasks and agents. It is responsible for organizing the training process and evolution of its agents (see Section 3.6). 39, 60, 61, 63, 69–72, 248, 249

EvolNLQ EvolNLQ is the program which has been created in the process of this theses. It uses evolutionary programming to translate natural language queries into SPARQL queries. ix, 4, 8, 11, 13, 17–19, 22, 25, 34, 40, 41, 45, 46, 48, 51–53, 57–59, 61, 79, 81, 82, 84–86, 103, 110, 114, 116, 117, 123, 125, 149, 163, 166, 168, 176, 179, 183, 184, 187, 197, 205, 210, 211, 214, 215, 217, 220, 223, 226, 230, 235, 240, 245

Evolutionary Algorithm A programming technique, which uses a greedy algorithm to develop an optimal solution for an optimization problem over several runs, evaluations, changing and reevaluation. For more details see Section 2.2.2.3 . 2, 3, 8, 23, 24, 26, 27, 33–37, 39, 45, 54, 58, 67, 77, 176, 197, 206, 243, 245, 248

Evolutionary dataflow agent Agents which are created from EvolNLQ they differ from generic agents in their reproduction. ix, 3, 4, 45, 47, 48, 51, 55, 57, 67, 77, 79, 197, 203, 206

Evolutionary programming A subclass of *evolutionary algorithm* in which a set of domain-specific methods is provided. [59] . 25, 30–32, 46, 47

Evolutionary strategies A subclass of *evolutionary algorithm* in which parameter of a function are changed to converge to a optimal solution. [49] . 25, 27, 46

Except Product Defines an interval $[p_{min}, p_{max}]$, all products which are at least partially inside of this interval then become part of an except statement in the final SPARQL query. More details see Section 4.1.8. 84, 91, 92, 94, 122, 123

Fitness Function Calculates the degree of success of an agent in solving a *task* in regard to solution of t. 23–25, 33, 34, 38, 60–62, 66, 77, 178, 245, 248

Fitness Score Value an agent got assigned from the fitness function. This score represents the degree of success an agent had with its task. It can be seen as the inverted error of the agent. 23

Genetic Algorithm *Evolutionary algorithm* which uses a genetic code for agents creation and crossovers as it major source of optimization [13] . 25, 27–29, 32, 33, 45, 46

Genetic Programming *Evolutionary algorithm* which creates a tree of operations as agents and uses primarily crossovers to evole. [51]. 25, 27, 45–47

Geobase A geographical database for natural language queries with a benchmark and training's set of questions and solutions.. 19, 149, 210, 215, 216

Global Fusion Likelihood Global value for the likelihood of an environmental fusion. 69, 70, 72

Global Relocation Likelihood Global value for the likelihood of a environmental relocation of an agent. 69, 70

Grammatical Relationship Encapsulating Product for the output of CoreNLP data regarding the relationship between two Part Of Speech tags. 19, 20, 80, 112, 113, 125, 127, 136, 146–148, 164–167

Grammatical Relationship Product Auxiliary product to describe the grammatical relationship between two positionable product. 109, 125

GraphViz "Graphviz is open source graph visualization software. It has several main graph layout programs. See the gallery for some sample layouts. It also has web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings." [86]. 185, 187

Group By Product Consists of a aggregation and TriplePart products which the grouping should be done by. 106, 167

IdentifierClassVariable Product A special form of the ClassVariable product. It refers to an specific individual instead of a set. See for more details Section 4.1.14. 100, 101, 119

International Resource Identifier Internationalized form of the Uniform Resource Identifier (URI). IRIs extend the allowed characters set to most characters of the Universal Coded Character Set (Unicode/ISO 10646). 13, 220

Jdom XML manipulation tool for Java [74]. 41

Jena "Apache Jena (or Jena in short) is a free and open source Java framework for building semantic web and Linked Data applications" [73]. 41

JFreeChart A free Java chart library [75]. 41

Literal Data Types Set of all literal datatypes as specified by XML Schema [82]. 80, 98, 104

Looking-For-Replacement product A compound product which stores a statement consisting of a subject, a predicate and an object .More details see Section 4.1.10. 109, 124, 125, 127, 131, 132, 142, 143, 145, 150, 162, 163

Mapping Dictionary A systematic storage of metadata for an ontology. [15] . 17, 79, 80, 82, 98, 103, 113, 115, 116, 119, 121, 124, 130–132, 137, 151, 152, 173, 174, 248, 249

Metadata Information about a data structure. 11, 12

Mondial A geographical database for learning and testing database related topics. Mondial is available in XML, RDF and SQL, among others. [87] . 14, 80, 154, 156, 166, 205, 207, 210, 215, 216, 259

Multi-Objective-Optimisation "In a multi-objective optimisation (MOO) problem, one optimises with respect to multiple goals or fitness functions $f1, f2, \ldots$ The task of a MOO algorithm is to find solutions that are optimal, or at least acceptable, according to all the criteria simultaneously. In most cases changing an algorithm from" [13] . 34

N3 Format for RDF data, which is easier to read and write then *RDF/XML*. 13, 14, 249

NLP Data Encapsulating Product for the output of CoreNLP data regarding a single Part Of Speech tag. 88, 90, 110, 112, 115–120, 122–125, 131, 132, 135–137, 140–142, 147, 202

Node Element of the graph structure of an agent which receives products, executes an operation on them and sends the resulting products to other nodes. 45–57, 62–64, 71, 79, 80, 84–86, 106–110, 112, 113, 120, 123, 125, 128, 131, 133–135, 140, 143, 144, 146, 154, 160–162, 164, 168, 169, 174, 184–189, 244, 247–249

Ontology "An ontology is a collection of terms used to describe and represent an area of concern" [32]. 41, 155, 247, 248

Operator Product A TriplePart product which contains a single mathematical operator, only used in comparison products. 96, 97, 103–105, 122, 132, 134, 179, 244

OWL Is an extension of the RDF vocabulary to provide more possibility to describe metadata. It is understood by reasoners and can be used to infer information of an ontology. 13, 15, 209

Part Of Speech Tag Part of speech tags are used to group lexical items with identical or at least very similar syntactic behavior (E.g. noun, verb, adjective, adverb, pronoun, preposition, conjunction, interjection, numeral, article, or determine). In modern linguistics the terms *word class*, *lexical class*, or *lexical category* are preferred used. But since CoreNLP uses the term *Part Of Speech*, it has been also used here. 53, 75, 80, 88, 112, 134, 141, 146–148, 166, 202, 246, 247

Path Network Collection Product Collection of the k-shortest paths sequences to turn a disconnected query graph product into a connected one. The path costs are derived from the number of passed edges and their respective certainty. More details see Section 4.1.27. 110, 158–160

Positionable Product is a product which can be located at a specific position or over a specific interval in the input query. 82, 87, 88, 91, 92, 94, 96–98, 100, 103–106, 110–113, 146–148, 167, 246

Product Products are the general term for the encapsulation of information, which are sent between different nodes. Also see Section 4.1.1. 46–50, 52, 53, 55–57, 59, 60, 62, 79, 80, 82–88, 90, 92, 97, 106–109, 112, 122, 128, 131, 134, 135, 141, 142, 144–146, 148–156, 168, 169, 174, 177–179, 183, 187, 188, 244, 245, 247, 249

Projection Product A *projection product* is a compound product which contains a TriplePart product which should be part of the selection clause in the final SPARQL Query. More details in Section 4.1.22. 85, 86, 107, 120, 128, 168

Property Product A TriplePart product which can only be at the predicate positions of a Statement. Contains one or more properties from the Mapping Dictionary. More details see Section 4.1.16. 80, 87, 88, 94, 100, 103, 104, 108, 115, 116, 119, 124, 125, 127, 128, 130–132, 136, 137, 146, 150–153, 155, 159, 162, 173, 174, 179

Query Graph Product Product which contains a collection of products. Interprets atomic products as nodes or edges and compound products as relations between them. 49, 75, 76, 86, 89, 91–96, 99, 100, 102, 105, 106, 108–110, 126, 127, 134, 138, 139, 142, 144, 146, 148–150, 152–154, 156–163, 165, 167, 168, 171–174, 177, 185, 187, 205, 206, 212, 213, 219, 221, 222, 224, 225, 227–229, 231–234, 236, 237, 244, 247

RDF The recommend language to describe data in the semantic web [33]. 12–17, 209, 247–249

RDF/XML Format for RDF data, which is also a valid XML document. 13, 14, 247

RDF2SQL Framework for metadata extraction and conversion from SPARQL queries to SQL as well as from an ontology to a relational database [15]. 17, 40, 81, 82, 210, 248

RDFS Extension of RDF to added the concepts of a class hierarchy and domain and range. 13, 15

Run A run involves the following steps for a environment. First give each agent computation time to solve the given *task*s, second evaluate each agent according to its solution with the fitness function. Third decide which agents may reproduce and assemble the next set of agents for the next run. Forth, rest the environment and rewards and other clean up. 244

SameAs A SameAs product has a set of TriplePart products which all describe the same entity and can be replaced with one another. More details see Section 4.1.24. 106–109, 141–143, 150

Selection Method Is a procedure to select the next generation of an *evolutionary algorithm*. 61, 67

Semantic Data Dictionary Dictionary with meta data information about an ontology generated by *RDF2SQL*. 17, 18, 183

Set of application classes An application domain describes a set of classes. Individuals which are considered in an application domain must be at least of one of the domain classes. $\mathcal{CLS}$ is the

set of all classes in the Mapping Dictionary specified, thus all classes mentioned explicitly in the ontology (e.g., as domain or range of properties). 80, 98, 115, 122, 171

Simulation A simulation is the entirety of all training runs to train a set of environments. Usually, for a simulation, the settings are set and not changed during the run. 61, 69, 71, 72, 183, 243, 249

SPARQL "SPARQL Protocol and RDF Query Language is an RDF query language—that is, a semantic query language for databases—able to retrieve and manipulate data stored in Resource Description Framework (RDF) format" [90]. 12, 13, 17, 41, 57, 58, 76, 79, 82, 84, 89, 90, 93, 95, 96, 99, 102, 123, 126, 138, 139, 172, 178, 212, 213, 219, 221, 222, 224–229, 231–234, 236, 237, 245, 249

SPARQLing product Are products which are possible to be translated into SPARQL statements (more details in Section 4.1.6). 82, 84, 90, 92, 97, 106, 107, 205, 206

Statement Product Abstract product which consists of a three-part statement over an ontology. 86, 92, 94, 96, 97, 132, 146, 149, 154, 155, 161, 162, 168

SubSPARQLing Are products which are possible to be translated into SPARQL statements, but need a subquery to be realized in SPARQL (more details in Section 4.1.6). 90, 91, 105

Sustain Energy Amount of energy necessary for an agent to survive a round of a simulation. The energy needed for that depends on the nodes and connections an agent consists of. More details in Section 3.6.2.1. 63–67

Task A task consists of input data and the intention to reach a certain goal. For training purposes a task always comes with a solution how the intended goal is reached.. 24, 50, 56–58, 61–63, 65, 67–72, 198, 202, 245

Triple A compound product which stores a statement consisting of a subject, a predicate and an object. More details see Section 4.1.10. 85, 91, 92, 94, 96, 97, 100, 109, 110, 119, 121, 124, 125, 127, 130–132, 136, 137, 145, 151–153, 156, 158, 159, 162–164, 169, 171, 173, 174

TriplePart Product Every kind of product which could be part of SPARQL Statement. 86, 90, 94, 97, 98, 100, 103–105, 107–109, 123, 128–132, 140, 146, 150–153, 162, 165, 167–171, 177, 244, 246–248

Turtle format Format for RDF data, which is derived from *N3* with some extensions manly for more convenience.. 13, 14

WordNet "WordNet® is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept" [72]. 41, 81, 82

XML Extensible Markup Language (XML) is a semi structured data storage document format. [38] [39] . 14

Bibliography

[1] W. Woods, R. Kaplan, and B. Webber, “The lunar sciences natural language information system,” 07 1972.

[2] F. Li and H. V. Jagadish, “Constructing an interactive natural language interface for relational databases,” *Proc. VLDB Endow.*, vol. 8, no. 1, p. 73–84, Sep. 2014. [Online]. Available: https://doi.org/10.14778/2735461.2735468

[3] A.-M. Popescu, O. Etzioni, and H. Kautz., “Towards a theory of natural language interfaces to databases,” *Intelligent User Interfaces*, pp. 149–157, 2003.

[4] N. Steinmetz, A. Arning, and K. Sattler, “From natural language questions to sparql queries: A pattern-based approach.” *BTW*, pp. 289–308, 2019.

[5] S. Hu, L. Zou, and X. Zhang., “A state transition framework to answer complex questions over knowledge base.” *EMNLP*, pp. 2098–2108, 2018.

[6] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan, “Athena: An ontology-driven system for natural language querying over relational data stores.” *VLDB*, vol. 9, pp. 1209–1220, 2016.

[7] P. Cimiano, V. Lopez, C. Unger, E. Cabrio, A.-C. Ngonga Ngomo, and S. Walter, “Multilingual question answering over linked data (qald-3): Lab overview,” pp. 321–332, 2013.

[8] C. Unger, “Multilingual question answering over linked data: Qald-4 dataset,” 2014.

[9] R. Usbeck, A.-C. N. Ngomo, F. Conrads, M. Röder, and G. Napolitano, “8th challenge on question answering over linked data (qald-8) (invited paper),” in *Semdeep/NLIWoD@ISWC*, 2018.

[10] R. Usbeck, R. Gusmita, M. Saleem, and A.-C. Ngonga Ngomo, “9th challenge on question answering over linked data (qald-9),” 11 2018.

[11] L. Zou, R. Huang, H. Wang, J. Yu, W. He, and D. Zhao, “Natural language question answering over rdf - a graph data driven approach,” *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 06 2014.

[12] K. S. D. Ishwari, A. K. R. R. Aneeze, S. Sudheesan, H. J. D. A. Karunaratne, A. Nugaliyadde, and Y. Mallawarrachchi, "Advances in natural language question answering: A review," 2019.

[13] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza). [Online]. Available: http://www.gp-field-guide.org.uk

[14] H. Schulz and S. Behnke, "Deep learning: Layer-wise learning of feature hierarchies," *Künstliche Intelligenz*, vol. 26, 01 2012.

[15] L. Runge, S. Schrage, and W. May, "Systematical representation of rdf-to-relational mappings for ontology-based data access." 2017, https://www.dbis.informatik.uni-goettingen.de/Publics/17/odbase17.html.

[16] K. Affolter, K. Stockinger, and A. Bernstein, "A comparative survey of recent natural language interfaces for databases," *VLDB J.*, vol. 28, no. 5, pp. 793–819, 2019. [Online]. Available: https://doi.org/10.1007/s00778-019-00567-8

[17] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger, "Soda: Generating sql for business users," *Proc. VLDB Endow.*, vol. 5, pp. 932–943, 2012.

[18] S. Shekarpour, E. Marx, A.-C. Ngonga Ngomo, and S. Auer, "Sina: Semantic interpretation of user queries for question answering on interlinked data," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 30, 06 2014.

[19] D. Damljanovic, V. Tablan, and K. Bontcheva, "A text-based query interface to OWL ontologies," in *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*. Marrakech, Morocco: European Language Resources Association (ELRA), May 2008. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2008/pdf/64_paper.pdf

[20] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, and J. Harrison, "Tr discover: A natural language interface for querying and analyzing interlinked datasets," 10 2015.

[21] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162

[22] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 963–973. [Online]. Available: https://aclanthology.org/P17-1089

[23] V. Zhong, C. Xiong, and R. Socher, “Seq2sql: Generating structured queries from natural language using reinforcement learning,” 2017.

[24] X. Xu, C. Liu, and D. Song, “Sqlnet: Generating structured queries from natural language without reinforcement learning,” 2017.

[25] S. Yavuz, I. Gur, Y. Su, and X. Yan, “What it takes to achieve 100% condition accuracy on WikiSQL,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 1702–1711. [Online]. Available: https://aclanthology.org/D18-1197

[26] D. D. Chamberlin and R. F. Boyce, “Sequel: A structured english query language,” in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET ’74. New York, NY, USA: Association for Computing Machinery, 1974, p. 249–264. [Online]. Available: https://doi.org/10.1145/800296.811515

[27] D. Chamberlin, M. Carey, G. Ghelli, D. Kossmann, and J. Robie, “XQueryP: An XML application development language,” 01 2006.

[28] E. Prud’hommeaux and A. Seaborne, “SPARQL query language for RDF,” W3C, W3C Recommendation, Jan. 2008, https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[29] A. Seaborne and S. Harris, “SPARQL 1.1 query language,” W3C, W3C Recommendation, Mar. 2013, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[30] H. Gallaire and J. Minker, *Logic and data bases*. New York : Plenum Press, 1978.

[31] D. Wood, M. Lanthaler, and R. Cyganiak, “RDF 1.1 concepts and abstract syntax,” W3C, W3C Recommendation, Feb. 2014, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[32] B. "Keio, “Ontologies - w3c,” 2015, https://www.w3.org/standards/semanticweb/ontology.

[33] D. Brickley and R. Guha, “RDF schema 1.1,” W3C, W3C Recommendation, Feb. 2014, https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[34] “OWL 2 web ontology language document overview (second edition),” W3C, W3C Recommendation, Dec. 2012, https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

[35] “An Introduction to Multilingual Web Addresses,” 05 2008. [Online]. Available: https://www.w3.org/International/articles/idn-and-iri/

[36] G. Carothers and E. Prud’hommeaux, “RDF 1.1 turtle,” W3C, W3C Recommendation, Feb. 2014, https://www.w3.org/TR/2014/REC-turtle-20140225/.

[37] T. Berners-Lee, D. Connolly, and S. Hawke, “Semantic web tutorial using n3,” 01 2000.

[38] M. Sperberg-McQueen, J. Paoli, F. Yergeau, T. Bray, and E. Maler, "Extensible markup language (XML) 1.0 (fifth edition)," W3C, W3C Recommendation, Nov. 2008, https://www.w3.org/TR/2008/REC-xml-20081126/.

[39] J. Paoli, E. Maler, M. Sperberg-McQueen, T. Bray, and F. Yergeau, "Extensible markup language (XML) 1.0 (fourth edition)," W3C, W3C Recommendation, Aug. 2006, https://www.w3.org/TR/2006/REC-xml-20060816/.

[40] "Naming of OWL Re: NAME: SWOL versus WOL," 12 2001. [Online]. Available: https://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html

[41] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, "Pellet: a practical owl-dl reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, pp. 51–53, 06 2007.

[42] B. Motik, R. Shearer, and I. Horrocks, "A Hypertableau Calculus for SHIQ," in *Proc. of the 20th Int. Workshop on Description Logics (DL 2007)*, D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, S. Tessaris, and A.-Y. Turhan, Eds. Brixen/Bressanone, Italy: Bozen/Bolzano University Press, June 8–10 2007, pp. 419–426.

[43] M. Khodadadi, R. Schmidt, and D. Tishkovsky, "An abstract tableau calculus for the description logic shoi using unrestricted blocking and rewriting," *CEUR Workshop Proceedings*, vol. 846, 01 2012.

[44] C. D. Manning, M. Surdeanu, and J. B. et al., "The Stanford CoreNLP natural language processing toolkit," *ACL*, pp. 55–60, 2014.

[45] V. Agel, L. M. Eichinger, H.-W. Eroms, P. Hellwig, H. J. Heringer, and H. Lobin, *Dependenz und Valenz*. de Gruyter, 2003.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[47] J. Holland, "Outline for a logical theory of adaptive systems," *Journal of the Association for Computing Machinery 9*, p. 297–314, 1962.

[48] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[49] W. Vent, "Rechenberg, ingo, evolutionsstrategie — optimierung technischer systeme nach prinzipien der biologischen evolution. 170 s. mit 36 abb. frommann-holzboog-verlag. stuttgart 1973. broschiert," *Feddes Repertorium*, vol. 86, no. 5, pp. 337–337, 1975. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/fedr.19750860506

[50] P. A. Vikhar, "Evolutionary algorithms: A critical review and its future prospects." *ICGTSPICC*, pp. 261–265, 2016.

[51] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. USA: L. Erlbaum Associates Inc., 1994, p. 87–112.

[52] Y.-L. Li, Y.-R. Zhou, Z.-H. Zhan, and J. Zhang, "A primary theoretical study on decomposition based multiobjective evolutionary algorithm," *IEEE*, vol. 20, no. 4, pp. 563–576, 2015.

[53] C. A. Coello Coello, "An introduction to evolutionary algorithms and their applications," in *Advanced Distributed Systems*, F. F. Ramos, V. Larios Rosillo, and H. Unger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 425–442.

[54] H. Schwefel, "Numerical optimization of computer models," *Annals of Operations Research*, 1981.

[55] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems." *Journal of the Association for Computing Machinery 9*, p. 297–314, 1975.

[56] L. Booker, "Intelligent behavior as an adaptation to the task environment ; part i." 02 2006.

[57] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. USA: L. Erlbaum Associates Inc., 1987, p. 14–21.

[58] ——, "Adaptive selection methods for genetic algorithms," in *Proceedings of the 1st International Conference on Genetic Algorithms*. USA: L. Erlbaum Associates Inc., 1985, p. 101–111.

[59] D. Fogel, *Evolutionary Computation: The Fossil Record*. Wiley, 1998. [Online]. Available: https://books.google.de/books?id=e1Q_AQAAIAAJ

[60] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 1990.

[61] P. Garg, "A comparison between memetic algorithm and genetic algorithm for the cryptanalysis of simplified data encryption standard algorithm," *CoRR*, vol. abs/1004.0574, 2010. [Online]. Available: http://arxiv.org/abs/1004.0574

[62] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, p. 768–774.

[63] B.-T. Zhang and H. Mühlenbein, "Evolving optimal neural networks using genetic algorithms with occam's razor," *Complex Systems*, vol. 7, 02 1995.

[64] W. Langdon, S. Barrett, and B. Buxton, "Predicting biochemical interactions - human p450 2d6 enzyme inhibition," in *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, vol. 2, 2003, pp. 807–814 Vol.2.

[65] G. Turk, "Sticky feet: Evolution in a multi-creature physical simulation." *InALife MITPress*, vol. 12, pp. 496–503, 2010.

[66] T. Hoverd and S. Stepney, "Energy as a driver of diversity in open-ended evolution," *ECAL 2011*, pp. 356–363, 2011.

[67] G. Turk, "Sticky feet - evolution in a multi-creature physical simulation," in *ALIFE*, 2010.

[68] "Website CoreNLP," 2021. [Online]. Available: https://stanfordnlp.github.io/CoreNLP/

[69] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*, ser. EMNLP '00. USA: Association for Computational Linguistics, 2000, p. 63–70. [Online]. Available: https://doi.org/10.3115/1117794.1117802

[70] D. Chen and C. Manning, "A fast and accurate dependency parser using neural networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 740–750. [Online]. Available: https://aclanthology.org/D14-1082

[71] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by Gibbs sampling," in *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Ann Arbor, Michigan: Association for Computational Linguistics, Jun. 2005, pp. 363–370. [Online]. Available: https://aclanthology.org/P05-1045

[72] C. Fellbaum, "Wordnet: An electronic lexical database," *MIT Press*, 1998.

[73] "Website Apache Jena," 2021. [Online]. Available: https://jena.apache.org/

[74] "Website JDOM," 2021. [Online]. Available: http://jdom.org/

[75] "Website JFreeChart," 2021. [Online]. Available: https://www.jfree.org/index.html

[76] C. Fellbaum, "On the shortest spanning subtree and the traveling salesman problem," *Proceedings of the American Mathematical Society, 7*, p. 48–50, 1956.

[77] J. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quarterly of Applied Mathematics*, 1970.

[78] B. Smock, "k-shortest-paths," https://github.com/bsmock/k-shortest-paths.git, 2017.

[79] A. Veen, "Dataflow machine architecture." *ACM Comput. Surv.*, vol. 18, pp. 365–396, 12 1986.

[80] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, 01 1995.

[81] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, p. 9–36, Mar. 1976. [Online]. Available: https://doi.org/10.1145/320434.320440

[82] M. Sperberg-McQueen, H. Thompson, M. Maloney, D. Beech, N. Mendelsohn, and S. Gao, "W3C xml schema definition language (XSD) 1.1 part 1: Structures," W3C, W3C Recommendation, Apr. 2012, https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[83] "Jena Query Builder - A query builder for Jena." 2021. [Online]. Available: https://jena.apache.org/documentation/extras/querybuilder/index.html

[84] N. Nakashole, G. Weikum, and F. Suchanek, "PATTY: A taxonomy of relational patterns with semantic types," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Jeju Island, Korea: Association for Computational Linguistics, Jul. 2012, pp. 1135–1145. [Online]. Available: https://aclanthology.org/D12-1104

[85] M. F. Porter, "An algorithm for suffix stripping," *Program: electronic library and information systems Vol. 40 No. 3*, pp. 211–218, 1980.

[86] Gansner, E. R., North, and S. C., "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[87] W. May, "The MONDIAL database," 1999–2017, http://dbis.informatik.uni-goettingen.de/Mondial/.

[88] W. Zheng, H. Cheng, L. Zou, J. X. Yu, and K. Zhao, *Natural Language Question/Answering: Let Users Talk With The Knowledge Graph*. New York, NY, USA: Association for Computing Machinery, 2017, p. 217–226. [Online]. Available: https://doi.org/10.1145/3132847.3132977

[89] S. Ferré, "An expressive query builder for sparql endpoints with guidance in natural language," in *Open J Semant Web*, 10 2017.

[90] J. Rapoza, "Sparql will make the web shine," *eWeek*, 2006.

Appendix A

Agents Species and Test Sets

A.1 Agent Species Ellyn25545

The most successful agent species, is named Ellyn. The name is automatically generated from a list of names to make it easier to keep track of the development of agents during training. Ellyn was created during a training run (see Section 6.1.9) on the *Mondial* training set (see Appendix A.3) and augmented with manual changes, such as the custom aggregation nodes.

Ellyn2554 consists of 151 Nodes and 337 Connections.

It has been used for evaluating the Geobase Query Set in A.2 and the *Mondial* testset (A.3).

The XML serialization of the agent species Ellyn2554 is shown below. All nodes are listed, and the list of connections is shortened.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<AgentCollection>
  <Agent Name="Ellyn2554">
    <Node ID="0" Type="PartOfSpeech">
      <Parameter Value="NNS" />
    </Node>
    <Node ID="1" Type="PartOfSpeech">
      <Parameter Value="NN" />
    </Node>
    <Node ID="2" Type="PartOfSpeech">
      <Parameter Value="VBN" />
    </Node>
    <Node ID="3" Type="PartOfSpeech">
      <Parameter Value="NNP" />
    </Node>
    <Node ID="4" Type="PartOfSpeech">
```

```
    <Parameter Value="IN" />
  </Node>
  <Node ID="5" Type="PartOfSpeech">
    <Parameter Value="RB" />
  </Node>
  <Node ID="6" Type="PartOfSpeech">
    <Parameter Value="CD" />
  </Node>
  <Node ID="7" Type="PartOfSpeech">
    <Parameter Value="JJ" />
  </Node>
  <Node ID="8" Type="PartOfSpeech">
    <Parameter Value="VBD" />
  </Node>
  <Node ID="9" Type="PartOfSpeech">
    <Parameter Value="NNPS" />
  </Node>
  <Node ID="10" Type="PartOfSpeech">
    <Parameter Value="VBP" />
  </Node>
  <Node ID="11" Type="CVGenerator" />
  <Node ID="12" Type="Identifier" />
  <Node ID="13" Type="ProperName" />
  <Node ID="14" Type="Synonym">
    <Parameter Value="attribute" />
  </Node>
  <Node ID="15" Type="Database">
    <Parameter Value="Class" />
  </Node>
  <Node ID="16" Type="CVGenerator" />
  <Node ID="17" Type="Identifier" />
  <Node ID="18" Type="ProperName" />
  <Node ID="19" Type="Synonym">
    <Parameter Value="attribute" />
  </Node>
  <Node ID="20" Type="Database">
    <Parameter Value="Lit" />
  </Node>
  <Node ID="21" Type="PropertyCombinator" />
  <Node ID="22" Type="PredicateGenerator" />
  <Node ID="23" Type="Synonym">
    <Parameter Value="attribute" />
    <Parameter Value="HYPERNYM" />
  </Node>
  <Node ID="24" Type="AnyGenerator" />
  <Node ID="25" Type="ConstantGenerator">
    <Parameter Value="percent" />
  </Node>
  <Node ID="26" Type="ExceptNode">
```

```
    <Parameter Value="except" />
    <Parameter Value="without" />
    <Parameter Value="not" />
  </Node>
  <Node ID="27" Type="Relator" />
  <Node ID="28" Type="OperatorGenerator" />
  <Node ID="29" Type="ConditionGenerator" />
  <Node ID="30" Type="GraphGeneratorNode" />
  <Node ID="31" Type="ViewBroker" />
  <Node ID="32" Type="Relator" />
  <Node ID="33" Type="PartOfSpeech">
    <Parameter Value="JJR" />
  </Node>
  <Node ID="34" Type="ConflictedLiteralsSolver">
    <Parameter Value="GrammaticalRelationRemovalStrategy" />
  </Node>
  <Node ID="35" Type="PathChooser" />
  <Node ID="36" Type="GraphReducer" />
  <Node ID="37" Type="KSPNode" />
  <Node ID="38" Type="EvidenceReducer" />
  <Node ID="39" Type="DuplicateFilterNode" />
  <Node ID="40" Type="GrAggregationNode">
    <Parameter Value="Avg" />
    <Parameter Value="average" />
    <Parameter Value="mean" />
  </Node>
  <Node ID="41" Type="AggregationNode">
    <Parameter Value="Sum" />
    <Parameter Value="how large" />
    <Parameter Value="sum" />
    <Parameter Value="combined" />
    <Parameter Value="total" />
  </Node>
  <Node ID="42" Type="TripleSplitter" />
  <Node ID="43" Type="Database">
    <Parameter Value="Lit" />
  </Node>
  <Node ID="44" Type="GrAggregationNode">
    <Parameter Value="Count" />
    <Parameter Value="how many" />
    <Parameter Value="How many" />
  </Node>
  <Node ID="45" Type="PartOfSpeech">
    <Parameter Value="WRB" />
  </Node>
  <Node ID="46" Type="NLPMerge" />
  <Node ID="47" Type="PartOfSpeech">
    <Parameter Value="JJS" />
  </Node>
```

```
<Node ID="48" Type="GrAggregationNode">
  <Parameter Value="Max" />
  <Parameter Value="maximal" />
  <Parameter Value="max" />
  <Parameter Value="highest" />
  <Parameter Value="most" />
  <Parameter Value="biggest" />
  <Parameter Value="most" />
  <Parameter Value="longest" />
  <Parameter Value="largest" />
  <Parameter Value="greatest" />
</Node>
<Node ID="49" Type="PartOfSpeech">
  <Parameter Value="RBR" />
</Node>
<Node ID="50" Type="ConflictedLiteralsSolver">
  <Parameter Value="NearestStrategy" />
</Node>
<Node ID="51" Type="PCV" />
<Node ID="52" Type="CVMerge" />
<Node ID="53" Type="Corrector" />
<Node ID="54" Type="GraphGeneratorNode" />
<Node ID="55" Type="ComparativeCondition" />
<Node ID="56" Type="PartOfSpeech">
  <Parameter Value="PRP" />
</Node>
<Node ID="57" Type="GrammaticalRelationFilter">
  <Parameter Value="amod" />
  <Parameter Value="none" />
  <Parameter Value="nsubj" />
  <Parameter Value="none" />
  <Parameter Value="compound" />
  <Parameter Value="none" />
</Node>
<Node ID="58" Type="PartOfSpeech">
  <Parameter Value="PRP$" />
</Node>
<Node ID="59" Type="GrammaticalRelationFilter">
  <Parameter Value="nmod:poss" />
  <Parameter Value="none" />
</Node>
<Node ID="60" Type="ReplaceSLNode" />
<Node ID="61" Type="ReplaceTPNode" />
<Node ID="62" Type="ConfidenceImprover">
  <Parameter Value="0.5" />
</Node>
<Node ID="63" Type="ConfidenceImprover">
  <Parameter Value="1.0" />
</Node>
```

```
<Node ID="64" Type="GrammaticalRelationFilter">
  <Parameter Value="conj" />
  <Parameter Value="none" />
</Node>
<Node ID="65" Type="ConfidenceImprover">
  <Parameter Value="-0.5" />
</Node>
<Node ID="66" Type="GrammaticalCartesianProduct" />
<Node ID="67" Type="ReplaceCompoundProductComponent" />
<Node ID="68" Type="Relator" />
<Node ID="69" Type="TripleFusion" />
<Node ID="70" Type="GrammaticalRelationFilter">
  <Parameter Value="nmod" />
  <Parameter Value="of" />
</Node>
<Node ID="71" Type="DisjunctionDetector" />
<Node ID="72" Type="Logical">
  <Parameter Value="Not" />
</Node>
<Node ID="73" Type="Database">
  <Parameter Value="NonPredicateBased" />
</Node>
<Node ID="74" Type="IndividualPreciser" />
<Node ID="75" Type="GrammaticalRelationFilter">
  <Parameter Value="nmod" />
  <Parameter Value="of" />
</Node>
<Node ID="76" Type="GrammaticalRelationFilter">
  <Parameter Value="compound" />
  <Parameter Value="none" />
</Node>
<Node ID="77" Type="CVGenerator" />
<Node ID="78" Type="Database">
  <Parameter Value="PositionBased" />
</Node>
<Node ID="79" Type="Database">
  <Parameter Value="Lit" />
</Node>
<Node ID="80" Type="Inverter" />
<Node ID="81" Type="GrAggregationNode">
  <Parameter Value="Min" />
  <Parameter Value="minimum" />
  <Parameter Value="min" />
  <Parameter Value="smallest" />
  <Parameter Value="less" />
  <Parameter Value="lowest" />
  <Parameter Value="cheapest" />
  <Parameter Value="worst" />
  <Parameter Value="shortest" />
```

```
    <Parameter Value="least" />
  </Node>
  <Node ID="82" Type="GrammaticalRelationFilter">
    <Parameter Value="nmod" />
    <Parameter Value="none" />
  </Node>
  <Node ID="83" Type="SameAsNode" />
  <Node ID="84" Type="GraphGeneratorNode" />
  <Node ID="85" Type="GrAggregationNode">
    <Parameter Value="Max Count" />
    <Parameter Value="most" />
  </Node>
  <Node ID="86" Type="PartOfSpeech">
    <Parameter Value="RBS" />
  </Node>
  <Node ID="87" Type="NearestNode">
    <Parameter Value="0" />
    <Parameter Value="2" />
  </Node>
  <Node ID="88" Type="Database">
    <Parameter Value="Class" />
  </Node>
  <Node ID="10" Type="PartOfSpeech">
    <Parameter Value="VBP" />
  </Node>
  <Node ID="90" Type="Database">
    <Parameter Value="Asymmetrical" />
  </Node>
  <Node ID="7" Type="PartOfSpeech">
    <Parameter Value="JJ" />
  </Node>
  <Node ID="92" Type="ConflictedLiteralsSolver">
    <Parameter Value="CompoundDistanceFirst" />
  </Node>
  <Node ID="93" Type="IndividualPreciser" />
  <Node ID="94" Type="PartOfSpeech">
    <Parameter Value="FW" />
  </Node>
  <Node ID="95" Type="GraphMergingNode" />
  <Node ID="96" Type="Corrector" />
  <Node ID="97" Type="Corrector" />
  <Node ID="98" Type="Disjunction" />
  <Node ID="99" Type="Database">
    <Parameter Value="NonReflexive" />
  </Node>
  <Node ID="100" Type="SubGraphMerger" />
  <Node ID="101" Type="EstablishLookingFor" />
  <Node ID="102" Type="PartOfSpeech">
    <Parameter Value="VBZ" />
```

```
</Node>
<Node ID="103" Type="CustomAggregation" />
<Node ID="104" Type="Logical">
  <Parameter Value="NotSameClass" />
</Node>
<Node ID="105" Type="Logical">
  <Parameter Value="ContainsProduct" />
</Node>
<Node ID="106" Type="StemmerNode" />
<Node ID="107" Type="GrammaticalRelationFilter">
  <Parameter Value="advmod" />
  <Parameter Value="none" />
</Node>
<Node ID="108" Type="Logical">
  <Parameter Value="NoInput" />
</Node>
<Node ID="109" Type="Database">
  <Parameter Value="NonNegative" />
</Node>
<Node ID="110" Type="GetIdentifier" />
<Node ID="111" Type="SelectionNode" />
<Node ID="112" Type="GraphGeneratorNode" />
<Node ID="113" Type="CVMerge" />
<Node ID="114" Type="ConflictedLiteralsSolver">
  <Parameter Value="CompetingObjectsStrategy" />
</Node>
<Node ID="115" Type="Database">
  <Parameter Value="NonNegative" />
</Node>
<Node ID="116" Type="CVMerge" />
<Node ID="117" Type="Logical">
  <Parameter Value="Not" />
</Node>
<Node ID="118" Type="Logical">
  <Parameter Value="Delay" />
</Node>
<Node ID="119" Type="DisjunctionDetector" />
<Node ID="120" Type="SeverWeakLinks">
  <Parameter Value="0.25" />
</Node>
<Node ID="121" Type="Logical">
  <Parameter Value="NotContainsProduct" />
</Node>
<Node ID="122" Type="Logical">
  <Parameter Value="Delay" />
</Node>
<Node ID="123" Type="LiteralRename">
  <Parameter Value="withAggregationSelection" />
</Node>
```

```
<Node ID="124" Type="SymmetricalRelation" />
<Node ID="125" Type="Logical">
  <Parameter Value="Not" />
</Node>
<Node ID="126" Type="Logical">
  <Parameter Value="Delay" />
</Node>
<Node ID="127" Type="GrammaticalRelationFilter">
  <Parameter Value="nsubj" />
  <Parameter Value="none" />
</Node>
<Node ID="128" Type="PartOfSpeech">
  <Parameter Value="WP" />
</Node>
<Node ID="5" Type="PartOfSpeech">
  <Parameter Value="RB" />
</Node>
<Node ID="130" Type="GrammaticalRelationFilter">
  <Parameter Value="advmod" />
  <Parameter Value="none" />
  <Parameter Value="dep" />
  <Parameter Value="none" />
</Node>
<Node ID="131" Type="Logical">
  <Parameter Value="NoInput" />
</Node>
<Node ID="132" Type="Logical">
  <Parameter Value="ContainsProduct" />
</Node>
<Node ID="133" Type="TrimGraph" />
<Node ID="134" Type="NLPMerge" />
<Node ID="7" Type="PartOfSpeech">
  <Parameter Value="JJ" />
</Node>
<Node ID="136" Type="GrammaticalRelationFilter">
  <Parameter Value="advmod" />
  <Parameter Value="none" />
</Node>
<Node ID="128" Type="PartOfSpeech">
  <Parameter Value="WP" />
</Node>
<Node ID="138" Type="Database">
  <Parameter Value="Lit" />
</Node>
<Node ID="139" Type="Database">
  <Parameter Value="Aggregation" />
</Node>
<Node ID="140" Type="Database">
  <Parameter Value="Aggregation" />
```

```
    </Node>
    <Node ID="141" Type="Logical">
      <Parameter Value="NoInput" />
    </Node>
    <Node ID="142" Type="Logical">
      <Parameter Value="And" />
    </Node>
    <Node ID="143" Type="GrammaticalRelationFilter">
      <Parameter Value="advmod" />
      <Parameter Value="none" />
    </Node>
    <Node ID="144" Type="LiteralTriple">
      <Parameter Value="big" />
      <Parameter Value="short" />
      <Parameter Value="heavy" />
      <Parameter Value="high" />
      <Parameter Value="low" />
      <Parameter Value="long" />
      <Parameter Value="large" />
    </Node>
    <Node ID="145" Type="PartOfSpeech">
      <Parameter Value="VBG" />
    </Node>
    <Node ID="146" Type="Database">
      <Parameter Value="Lit" />
    </Node>
    <Node ID="147" Type="GrammaticalRelationFilter">
      <Parameter Value="nmod" />
      <Parameter Value="none" />
    </Node>
    <Node ID="148" Type="Database">
      <Parameter Value="Lit" />
    </Node>
    <Node ID="149" Type="GrammaticalDisjunction" />
    <Node ID="150" Type="ReplaceInverse" />
    <Connection FromID="13" FromConduitID="2" ToID="11" ToConduitID="0" />
    <Connection FromID="7" FromConduitID="0" ToID="41" ToConduitID="0" />
    <Connection FromID="13" FromConduitID="1" ToID="12" ToConduitID="0" />
    <Connection FromID="11" FromConduitID="1" ToID="15" ToConduitID="0" />
    <Connection FromID="12" FromConduitID="2" ToID="15" ToConduitID="0" />
    <Connection FromID="0" FromConduitID="0" ToID="14" ToConduitID="0" />
    <Connection FromID="1" FromConduitID="0" ToID="14" ToConduitID="0" />
    <Connection FromID="2" FromConduitID="0" ToID="14" ToConduitID="0" />
    <-- To save space, the Connections have been shortened as they add little value to
        the reader.-->
  </Agent>
</AgentCollection>
```

A.2 Geobase250 Query Set

Query name	Coverage	Query Text
GeoBaseQuery1	1.0	which rivers run through States bordering New Mexico?
GeoBaseQuery2	1.0	what is the highest point in Montana?
GeoBaseQuery3	1.0	what is the most populated state bordering Oklahoma?
GeoBaseQuery4	1.0	through which States does the Mississippi run?
GeoBaseQuery5	1.0	what is the longest River?
GeoBaseQuery6	1.0	how long is the Mississippi?
GeoBaseQuery7	1.0	which State has the smallest population density?
GeoBaseQuery8	1.0	what is the area of Wisconsin?
GeoBaseQuery9	1.0	what is the lowest point of the State with the largest area?
GeoBaseQuery10	1.0	what is the longest River in Mississippi?
GeoBaseQuery11	1.0	what states border Montana?
GeoBaseQuery12	1.0	what States border New Jersey?
GeoBaseQuery13	1.0	which State has the longest River?
GeoBaseQuery14	1.0	name the rivers in Arkansas.
GeoBaseQuery15	1.0	which States have points higher than the highest point in Colorado?
GeoBaseQuery16	1.0	how many people live in the capital of Texas?
GeoBaseQuery17	1.0	how long is the Delaware River?
GeoBaseQuery18	1.0	what is the smallest city in the usa?
GeoBaseQuery19	1.0	what States border Georgia?
GeoBaseQuery20	1.0	what is the smallest State by area?
GeoBaseQuery21	1.0	how long is the Mississippi River?
GeoBaseQuery22	1.0	what States border Delaware?
GeoBaseQuery23	1.0	what is the shortest River in the usa?
GeoBaseQuery24	1.0	what States have cities named Plano?
GeoBaseQuery25	1.0	how many rivers does Colorado have?
GeoBaseQuery26	1.0	what is the biggest city in Georgia?
GeoBaseQuery27	1.0	what States border Hawaii?
GeoBaseQuery28	1.0	what is the capital of the state with the highest point?
GeoBaseQuery29	1.0	what State has the highest population?
GeoBaseQuery30	1.0	what is the capital of Maine?

Query name	Coverage	Query Text
GeoBaseQuery31	1.0	which state border Florida?
GeoBaseQuery32	1.0	what State has highest elevation?
GeoBaseQuery33	1.0	what rivers run through the States that border the State with the capital Atlanta?
GeoBaseQuery34	1.0	what is the biggest city in Oregon?
GeoBaseQuery35	1.0	what is the lowest point of the us?
GeoBaseQuery36	1.0	which State border Hawaii?
GeoBaseQuery37	1.0	what are the major cities in Ohio?
GeoBaseQuery38	1.0	what is the population of Springfield Missouri?
GeoBaseQuery39	1.0	how many people live in California?
GeoBaseQuery40	1.0	where is the highest point in Montana?
GeoBaseQuery41	1.0	what are the major cities in Alaska?
GeoBaseQuery42	1.0	what are the major cities in Kansas?
GeoBaseQuery43	1.0	which State has the highest point?
GeoBaseQuery44	1.0	what States border Florida?
GeoBaseQuery45	1.0	what States does the Ohio River go through?
GeoBaseQuery46	1.0	what is the largest city in Minnesota by population?
GeoBaseQuery47	1.0	how many rivers are there in Idaho?
GeoBaseQuery48	1.0	how high is the highest point in Montana?
GeoBaseQuery49	1.0	what is the lowest point in California?
GeoBaseQuery50	1.0	what is the capital of Georgia?
GeoBaseQuery51	1.0	how big is Texas?
GeoBaseQuery52	1.0	what is the highest point in Nevada in meters?
GeoBaseQuery53	1.0	how many people live in Minneapolis Minnesota?
GeoBaseQuery54	1.0	what is the area of Maine?
GeoBaseQuery55	1.0	what is the lowest point in Oregon?
GeoBaseQuery56	1.0	what State has the city Flint?
GeoBaseQuery57	1.0	give me the largest State?
GeoBaseQuery58	1.0	how many States does the Colorado River run through?
GeoBaseQuery59	1.0	what is the area of South Carolina?
GeoBaseQuery60	1.0	which State has the highest elevation?

Query name	Coverage	Query Text
GeoBaseQuery61	1.0	how large is Alaska?
GeoBaseQuery62	0.06	how many citizens live in California?
GeoBaseQuery63	1.0	what is the biggest city in Wyoming?
GeoBaseQuery64	1.0	which States border South Dakota?
GeoBaseQuery65	1.0	what State has the largest population density?
GeoBaseQuery66	1.0	what is the population of Utah?
GeoBaseQuery67	1.0	how many people live in Rhode Island?
GeoBaseQuery68	1.0	what is the population of New York city?
GeoBaseQuery69	1.0	which States border Texas?
GeoBaseQuery70	1.0	what is the population of Seattle Washington?
GeoBaseQuery71	1.0	what is the highest point in Colorado?
GeoBaseQuery72	1.0	how large is the largest city in Alaska?
GeoBaseQuery73	1.0	what is the longest River in the us?
GeoBaseQuery74	1.0	how many States does the Mississippi River run through?
GeoBaseQuery75	1.0	what are the highest points of States surrounding Mississippi?
GeoBaseQuery76	1.0	what is the highest point of the usa?
GeoBaseQuery77	1.0	what is the largest River in Washington State?
GeoBaseQuery78	1.0	what is the population of Illinois?
GeoBaseQuery79	1.0	which State border the most States?
GeoBaseQuery80	1.0	which rivers flow through Alaska?
GeoBaseQuery81	1.0	what city has the most people?
GeoBaseQuery82	1.0	which States does the Mississippi run through?
GeoBaseQuery83	1.0	what is the capital of Washington?
GeoBaseQuery84	1.0	what is the smallest city in the us?
GeoBaseQuery85	1.0	what are the major cities in Texas?
GeoBaseQuery86	1.0	which State has the highest population density?
GeoBaseQuery87	1.0	what State contains the highest point in the us?
GeoBaseQuery88	1.0	what States does the Delaware River run through?
GeoBaseQuery89	1.0	which States capital city is the largest?
GeoBaseQuery90	0.06	how many citizens in Alabama?

Query name	Coverage	Query Text
GeoBaseQuery91	1.0	what is the highest point in States bordering Georgia?
GeoBaseQuery92	1.0	what rivers are in Utah?
GeoBaseQuery93	1.0	what is the area of the largest State?
GeoBaseQuery94	1.0	what are all the rivers in Texas?
GeoBaseQuery95	1.0	what is the population density of Wyoming?
GeoBaseQuery96	1.0	what is the capital of New Jersey?
GeoBaseQuery97	1.0	what is the lowest point in nebraska in meters?
GeoBaseQuery98	1.0	what major rivers run through Illinois?
GeoBaseQuery99	1.0	what is the capital of New Hampshire?
GeoBaseQuery100	1.0	what is the lowest point in Massachusetts?
GeoBaseQuery101	1.0	what is the largest city in States that border California?
GeoBaseQuery102	1.0	what States border Indiana?
GeoBaseQuery103	1.0	where is the lowest spot in Iowa?
GeoBaseQuery104	0.0	how many square kilometers in the us?
GeoBaseQuery105	1.0	what is the highest point in Rhode Island?
GeoBaseQuery106	1.0	what are the major cities in Rhode Island?
GeoBaseQuery107	1.0	what States border Arkansas?
GeoBaseQuery108	1.0	where is the lowest point in the us?
GeoBaseQuery109	1.0	rivers in New York?
GeoBaseQuery110	1.0	what is the population density of Maine?
GeoBaseQuery111	1.0	what is the lowest point in the State of California?
GeoBaseQuery112	1.0	what is the highest point in the us?
GeoBaseQuery113	1.0	how long is the Colorado River?
GeoBaseQuery114	1.0	how long is the North platte River?
GeoBaseQuery115	1.0	how large is Texas?
GeoBaseQuery116	1.0	which States border Colorado?
GeoBaseQuery117	1.0	what is the lowest point in louisiana?
GeoBaseQuery118	1.0	what is the population of Dallas?
GeoBaseQuery119	1.0	what is the population of Tempe Arizona?
GeoBaseQuery120	1.0	how many rivers in Washington?

Query name	Coverage	Query Text
GeoBaseQuery121	1.0	what is the shortest River in the us?
GeoBaseQuery122	1.0	what are the major cities of Texas?
GeoBaseQuery123	1.0	how many people live in Kalamazoo?
GeoBaseQuery124	1.0	how many rivers does Alaska have?
GeoBaseQuery125	1.0	what rivers run through Colorado?
GeoBaseQuery126	1.0	what is the length of the Colorado River?
GeoBaseQuery127	1.0	what is the State with the lowest population?
GeoBaseQuery128	1.0	what States border Rhode Island?
GeoBaseQuery129	1.0	how many rivers are in Colorado?
GeoBaseQuery130	1.0	what is the total population of the States that border Texas?
GeoBaseQuery131	1.0	what is the length of the Mississippi River?
GeoBaseQuery132	1.0	what is the population of Oregon?
GeoBaseQuery133	1.0	how many cities are there in the US?
GeoBaseQuery134	1.0	what is the area of Alaska?
GeoBaseQuery135	1.0	how many people live in Spokane Washington?
GeoBaseQuery136	1.0	what is the combined population of all 50 States?
GeoBaseQuery137	1.0	what State has the capital Salem?
GeoBaseQuery138	1.0	how high is the highest point in america?
GeoBaseQuery139	1.0	what is the biggest city in the US?
GeoBaseQuery140	1.0	what is the smallest city in Alaska?
GeoBaseQuery141	1.0	how long is the shortest River in the usa?
GeoBaseQuery142	1.0	what States have cities named Dallas?
GeoBaseQuery143	1.0	what is the biggest River in Illinois?
GeoBaseQuery144	1.0	what is the capital of Iowa?
GeoBaseQuery145	1.0	what is the highest point in Iowa?
GeoBaseQuery146	1.0	what is the population density of Texas?
GeoBaseQuery147	1.0	what is the longest River in Florida?
GeoBaseQuery148	1.0	what is the population of Hawaii?
GeoBaseQuery149	1.0	what is the smallest city in Washington?
GeoBaseQuery150	1.0	what are the major cities in Oklahoma?

Query name	Coverage	Query Text
GeoBaseQuery151	1.0	what State is Des Moines located in?
GeoBaseQuery152	1.0	what is the highest point in the country?
GeoBaseQuery153	1.0	what State border Michigan?
GeoBaseQuery154	1.0	what States border New Hampshire?
GeoBaseQuery155	1.0	what is the lowest point in the united States?
GeoBaseQuery156	1.0	how long is the Rio Grande River?
GeoBaseQuery157	1.0	what are the major rivers in Ohio?
GeoBaseQuery158	1.0	what is the capital of North Dakota?
GeoBaseQuery159	1.0	what is the largest city in Rhode Island?
GeoBaseQuery160	1.0	what is the population of the capital of the smallest state?
GeoBaseQuery161	1.0	what is the most populous State?
GeoBaseQuery162	1.0	what is the largest city in Wisconsin?
GeoBaseQuery163	1.0	what is the population of the major cities in Wisconsin?
GeoBaseQuery164	1.0	give me the cities in Virginia?
GeoBaseQuery165	1.0	which States have cities named Austin?
GeoBaseQuery166	1.0	what State is Columbus the capital of?
GeoBaseQuery167	1.0	what is the city with the smallest population?
GeoBaseQuery168	1.0	what States does the Missouri run through?
GeoBaseQuery169	1.0	what is the longest River in the united States?
GeoBaseQuery170	1.0	how many cities are in Montana?
GeoBaseQuery171	1.0	what is the highest elevation in New Mexico?
GeoBaseQuery172	1.0	how long is the Missouri River?
GeoBaseQuery173	0.0	what capital is the largest in the US?
GeoBaseQuery174	1.0	what is the population of South Dakota?
GeoBaseQuery175	1.0	how many people live in New York?
GeoBaseQuery176	1.0	what is the population of San Antonio?
GeoBaseQuery177	1.0	what are the major cities in California?
GeoBaseQuery178	1.0	what State has the greatest population density?
GeoBaseQuery179	1.0	which River runs through the most States?
GeoBaseQuery180	1.0	which States does the Missouri River run through?

Query name	Coverage	Query Text
GeoBaseQuery181	1.0	which State has the highest peak in the country?
GeoBaseQuery182	1.0	what is the biggest city in Arizona?
GeoBaseQuery183	1.0	what is the lowest point in the State of Texas?
GeoBaseQuery184	1.0	which State is the city Denver located in?
GeoBaseQuery185	1.0	what is the lowest point in Arkansas?
GeoBaseQuery186	1.0	what is the biggest city in Texas?
GeoBaseQuery187	1.0	what is the biggest city in the USA?
GeoBaseQuery188	1.0	which State has the largest city?
GeoBaseQuery189	1.0	how many rivers are in New York?
GeoBaseQuery190	1.0	what is the lowest point in Texas?
GeoBaseQuery191	1.0	which States border Kentucky?
GeoBaseQuery192	1.0	which State border most States?
GeoBaseQuery193	0.95	how many major cities are in Florida?
GeoBaseQuery194	1.0	what are the major cities in Wyoming?
GeoBaseQuery195	1.0	what is the highest point in the USA?
GeoBaseQuery196	1.0	what is the population density of the smallest State?
GeoBaseQuery197	1.0	name all the rivers in Colorado?
GeoBaseQuery198	1.0	what is the capital of Vermont?
GeoBaseQuery199	1.0	what is the population of Tucson?
GeoBaseQuery200	1.0	what is the highest mountain in the US?
GeoBaseQuery201	1.0	what is the capital of Utah?
GeoBaseQuery202	1.0	how long is the Ohio River?
GeoBaseQuery203	1.0	what rivers do not run through Tennessee?
GeoBaseQuery204	1.0	what is the highest point in Wyoming?
GeoBaseQuery205	1.0	which States does the Mississippi River run through?
GeoBaseQuery206	0.5	what States capital is Dover?
GeoBaseQuery207	1.0	what is the population of Arizona?
GeoBaseQuery208	1.0	whats the largest city?
GeoBaseQuery209	1.0	what is the biggest city in Louisiana?
GeoBaseQuery210	1.0	how many people live in Austin?

Query name	Coverage	Query Text
GeoBaseQuery211	0.38	what is the total area of the usa?
GeoBaseQuery212	1.0	what is the highest point in Kansas?
GeoBaseQuery213	1.0	which States border New York?
GeoBaseQuery214	1.0	what State has the highest elevation?
GeoBaseQuery215	1.0	what is the highest point of the State with the largest area?
GeoBaseQuery216	1.0	how many people live in Washington?
GeoBaseQuery217	1.0	how many people live in Hawaii?
GeoBaseQuery218	0.82	what rivers run through New York?
GeoBaseQuery219	1.0	how many people live in Riverside?
GeoBaseQuery220	1.0	what is the population of Texas?
GeoBaseQuery221	1.0	which States border Arizona?
GeoBaseQuery222	1.0	what is the area of the smallest State?
GeoBaseQuery223	1.0	which State border Kentucky?
GeoBaseQuery224	1.0	what States border Kentucky?
GeoBaseQuery225	0.07	what is the largest State capital in population?
GeoBaseQuery226	1.0	what is the smallest State in the usa?
GeoBaseQuery227	1.0	where is the highest point in Hawaii?
GeoBaseQuery228	1.0	what is the smallest city in Hawaii?
GeoBaseQuery229	1.0	what is the population of Portland Maine?
GeoBaseQuery230	0.5	what are the populations of States through which the Mississippi River runs?
GeoBaseQuery231	1.0	what is the shortest River?
GeoBaseQuery232	1.0	what is the population of Idaho?
GeoBaseQuery233	1.0	what is the population of Erie Pennsylvania?
GeoBaseQuery234	1.0	how many major rivers cross Ohio?
GeoBaseQuery235	1.0	what is the population of Montana?
GeoBaseQuery236	1.0	which State is Kalamazoo in?
GeoBaseQuery237	1.0	what are the rivers in Alaska?
GeoBaseQuery238	1.0	which State is the smallest?
GeoBaseQuery239	1.0	what States surround Kentucky?
GeoBaseQuery240	1.0	which State has the greatest population?

Query name	Coverage	Query Text
GeoBaseQuery241	1.0	what is the area of Idaho?
GeoBaseQuery242	1.0	what rivers run through west Virginia?
GeoBaseQuery243	1.0	what is the highest point in the State with the capital Des Moines?
GeoBaseQuery244	1.0	what length is the Mississippi?
GeoBaseQuery245	1.0	what is the shortest River in Iowa?
GeoBaseQuery246	1.0	what States border Ohio?
GeoBaseQuery247	0.82	what is the combined area of all 50 States?
GeoBaseQuery248	1.0	what is the longest River in Texas?
GeoBaseQuery249	1.0	what is the population of boston Massachusetts?
GeoBaseQuery250	1.0	what is the capital of the State with the largest population?

A.3 Mondial Query Set

Name	Coverage	Query Text
Types of Membership	1.0	Tell me of which type each country is in each organization
Depth Persian Gulf	1.0	Give me the depth of the Persian Gulf
Percent of Sikh	1.0	How many percent of India are Sikh?
OrganizationsOfGermany	1.0	Give me the name and the abbrev of all organizations where Germany is member in.
CitiesOfGermany	1.0	Give me all cities in Germany
Countries encompassed by Asia	1.0	Which countries are encompassed of Asia
Deepness	1.0	Give me the deepness of all seas.
RODI 112c	0.77	Give me the name and population of all capitals
RODI 112b	1.0	What is the population of the capital of each country
CitiesNotRiver	1.0	Give me all cities that have more than 1000000 inhabitants, and not located at any river that is more than 1000 km long
CountryMultiLiteral	1.0	Give me the name, capital, gdpAgri, populationGrowth, and the type of government of each country
Everything in Asia	1.0	Give me everything located in Asia
All Names except for Deserts	1.0	Give me all names except for deserts
Less than 100 Percent	1.0	Give me all countries which are encompassed by a continent with less than 100 percent
Depth of the Sea of Japan	1.0	Give me the depth of the Sea of Japan
Lake Literals	1.0	Give me the name and the type of each lake
OrganiSations	1.0	Give me the name of all organisations
Short Rivers	1.0	Give me all rivers with a length shorter than 100 kilometers
WaterName	1.0	Give me all waters with their name
More Cities than Russia	0.71	Which countries have more cities than Russia?
City highest Population	0.77	What is the city with the highest population?
Lakes in France	1.0	Which Lakes are located in France
RODI 113	0.77	Give me the name and the othername of all cities with a population over 1000000
RODI 111	1.0	Which countries have a population over 10000000
More Muslim than Jewish	0.8	Is the percentage of Muslim greater than the percentage of Jewish in Luxembourg

Name	Coverage	Query Text
RODI 116a	1.0	Give me all cities which are located at a water
Islands	1.0	Give me all islands with their name and area
Anything with a Name	1.0	Give me anything with a name
Neighbors of Germany	1.0	Give me the border length from Germany and its neighbors
CityNames	1.0	Give me all city names
ReligionNames	1.0	Give me the names of all Religions
Portugal Member Nato	1.0	Is Portugal one of the members of the NATO
NeighboursOfFrance	1.0	Give me the neighbours of France
LocatedAt	1.0	Give me all cities located at a river and their country
City Population in China	1.0	How large is the population of all cities in China?
Membership	1.0	Give me all organizations and the countries who are member of it.
CitiesCountKorea	1.0	How many cities are there in South Korea?
Countries without Christians	1.0	Give me the name of all countries without christians
Area of all Seas	0.77	How large is the area of all seas?
States of Europe	1.0	Give me all states in Europe
Lat equals Long	1.0	Is there a city where the latitude and longitude are equal
CopenhagenDenmark	1.0	Is Copenhagen in the Denmark
Bigger Than Capital	1.0	Give me all cities where the population is greater then the population of the capital of their country
Eu Member Capitals	1.0	Give me all the capitals of the EU members.
Salt Lake City	1.0	What is the population of Salt Lake City?
Language Spoken In	1.0	Give me all languages and the countries they are spoken in.
Smallest Country	0.72	Which country has the smallest area?
River Through Countries	1.0	Which rivers flow through the Netherlands, Switzerland and Germany?
EU Capitals	1.0	What are the capitals of the members of the EU
Country equals Capital	1.0	Give me all countries where the name is equal to the capitals name
Cities Of Europe	1.0	Give me all cities which are in Europe
CityLiterals	1.0	Give me the name , population and elevation of each city
Mountain Height	1.0	Give me the height of each mount
Chicago in USA	1.0	Is Chicago in the USA
Higher Population Than Capital	1.0	Which cities have a larger population than the capital of their countries

Name	Coverage	Query Text
More Turkish than Croat in Austria	1.0	Is the percentage of Turkish people greater than the percentage of Croat people in Austria
Average City size	0.74	What is the average population of a city?
River in Germany and Poland	1.0	Which rivers are located in Poland and Germany
Countries NATO	1.0	List the countries of the North Atlantic Treaty Organization
Everything at Pacific	1.0	Give me everything located at the Pacific
Bigger Capital	1.0	Which cities have more population than the capital of their country
Greater	1.0	Give me all cities which have a population over 1000000
All States Except Africa	1.0	Give me the name of each state except it is encompassed from Africa