Knowledge Representation

Project 1

November 2023

1 Abstract

This study focuses on the development and evaluation of an of simple EL reasoner in python for ontology reasoning. The project involved, the development of a detailed ontology for a Turkish restaurant specializing in kebabs, the implementation of an EL reasoner in Python, and a comparative performance analysis between Python and Java implementations. The ontology, encompassing 36 intricate concepts, aims to support complex queries about menu items, ingredients, and dietary preferences. The EL reasoner, developed in Python, is evaluated against its Java counterpart to assess execution speed and efficiency.

2 Introduction

Description Logics, a family of knowledge representation languages, provide a formal framework for encoding knowledge about the world and reasoning about it. They are instrumental in developing ontologies, which are explicit formal specifications of the terms in the domain and relations among them. Rooted in formal semantics, DLs enable precise specification of the meaning of ontological constructs, facilitating unambiguous knowledge exchange and logical deduction for additional information inference.

EL family of DLs, characterized by its computational efficiency, plays a crucial role in managing, structuring large ontologies/knowledge, particularly in AI systems. This efficiency is achieved by focusing on existential quantifiers and concept intersections, while deliberately avoiding more complex constructs like universal quantifiers, thereby enabling polynomial-time reasoning tasks.

The OWL Web Ontology Language, standardized by the W3C, is built upon the principles of DLs. It translates DL constructs into web-compatible formats, making DLs instrumental in developing and manipulating web-based ontologies. The EL family's principles, in particular, have directly influenced the design of OWL, especially in the form of OWL EL, one of its subsets optimized for large-scale ontologies.

This project explores the utility of DLs, particularly the EL family, in the development of an ontology and the implementation of an EL reasoner. The project's aim is twofold: firstly, to develop OWL ontology using EL family

of DL principles. Secondly, to evaluate the effectiveness of the EL reasoning algorithms, particularly ELK [3] and HermiT [2] in process and reason over large and from complex OWL EL ontologies.

2.1 The Research Question

This study aims to investigate the performance implications of reimplementing the EL reasoner in Python, with a specific focus on execution speed, and to compare it against its Java-based implementation. The research will first involve the development of an EL reasoner in Python, utilizing string operators for core functionalities. A comprehensive testing environment will then be established to methodically evaluate and compare the performance metrics of both implementations.

Additionally, the study will extend to assessing the performance of established EL reasoners, namely ELK and HermiT, on complex ontologies. This comparative analysis aims to provide insights into the efficiency and applicability of these reasoners in handling ontologies of varying complexity and structure. The objective is to understanding of how different implementation languages and reasoning algorithms impact the execution speed and overall performance in practical ontology reasoning tasks.

3 Ontology Development

We developed an ontology for a Turkish restaurant, focusing on a variety of kebabs and other dishes, using Protege [**?**] in the OWL Web Ontology Language. The ontology comprises 36 intricate concepts, meticulously designed to capture the diversity of culinary elements in the restaurant. This design facilitates intelligent querying about menu items, ingredient specifics, and customer dietary preferences.

3.1 Details of Key Concepts

In the context of evaluating EL reasoners, our ontology is structured around the following primary categories.

- **Food:** This top level category covers a spectrum of items, from main dishes to side accompaniments, enabling detailed queries about various food offerings.
- **Kebab:** As the core category, it includes diverse subclasses representing kebabs with distinct flavors and cooking styles, crucial for nuanced culinary queries.
- **Meat:** This category encompasses various meats like beef, lamb, and chicken, integral to many dishes, and allows for queries based on specific meat types.

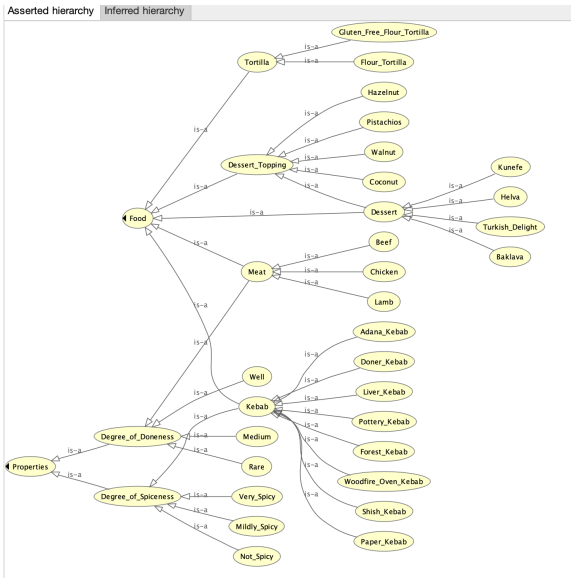

Figure 1: Overview of the ontology hierarchy, illustrating the complex relationships and categories within the Turkish restaurant domain.

- **Dessert:** It includes both traditional and modern desserts, such as Baklava and Turkish Delight, catering to queries about sweet dishes.
- **Tortilla:** This category offers a range of tortillas, from traditional to gluten-free options, pertinent for dietary-based queries.

Properties linked with these categories, such as the Degree of Spiciness and Degree of Doneness, are integral for tailoring dishes to customer preferences. These elements are important in evaluating the query-handling capabilities of EL reasoners like ELK and HermiT.

3.2 Category Relationships and Properties

Our ontology is structured to clearly define relationships and properties between classes.

3.2.1 Disjoint Classes

Disjoint classes are used to separate major categories, such as `disjoint(Food, Properties)`, to avoid overlap between food items and their properties, ensuring a clear ontology structure.

3.2.2 Domain and Range Properties

Specific domains and ranges for properties are as follows:

- *Domain of degreeOfDoneness:* We specified the domain of the degreeOfDoneness property as Meat, indicating that this attribute of doneness is solely applicable to meat products.
- *Domain and Range of spiciness:* The domain of the spiciness property is defined as Kebab, and its range as Degree of Spiciness, signifying that this characteristic of spiciness is specifically used to describe various kebabs.
- *Domain and Range of hasTopping:* For the Dessert category, we defined the hasTopping property, with its domain as Dessert and its range as Dessert Topping, indicating that desserts can have various toppings like nuts or jams.

3.2.3 Disjoint Properties

To avoid conflict and ambiguity between properties, some properties are defined as disjoint, such as `degreeOfDoneness` and `degreeOfSpiciness`, ensuring these attributes are not used in inapplicable categories.

3.3 Logical Constructors and Complex Expressions

The ontology leverages logical constructors and complex class expressions, crucial for enhancing query accuray and aligning with DL and EL reasoning capabilities.

3.3.1 Complex Class Expressions

Complex class expressions are used to model detailed relationships. For example, the expression "Kebab $\sqsubseteq \exists$spiciness.Degree_of_Spiceness" is utilised to denote that all kebab variants possess a certain spiciness level. Such expressions allow the system to respond precisely to specific customer queries, such as preferences for spiciness. This enables testing of EL reasoning efficiency

3.3.2 Use of Logical Constructors

The ontology's logical constructors include combinations of various categories and properties. Existential quantifiers are prominently used to define properties of concepts. For example, we used existential quantifiers (like $\exists$) to express the concept of "having certain properties." This is particularly important in describing scenarios like "all desserts that offer specific toppings" (Dessert $\sqsubseteq$ $\exists$hasTopping.Dessert_Topping) or "all meats with a certain degree of doneness" (Meat $\sqsubseteq \exists$degreeOfDonenes.Degree_of_Doneness).

3.3.3 Combination of Property and Category Hierarchies

The ontology integrates property and category hierarchies to form comprehensive relationships. For instance, we defined multiple subclasses under the Meat

category (such as Beef, Lamb, Chicken) and specified particular degrees of doneness and flavor properties for these subclasses.

These categories, relationships and logical constructors, ensure that our ontology not only accurately describes the restaurant's menu and services but also supports complex customer's queries, personalisation needs and preferences. Such structural complexities are important in evaluating the effectiveness of the EL reasoner, particularly in its ability to manage and interpret intricate hierarchical relationships within the ontology.

4 EL Reasoner Implementation

For the process extracting information from the *.owl* files, we employ the *py4j* library[1] as our primary tool. This library is essential for facilitating the interoperability between Java frameworks and Python, enabling us to efficiently extract axioms and concept names from the ontology files. Apart from this integration, the EL reasoner is entirely developed in Python, with a focus on utilizing string operations in line with our research objectives.

In the development of our EL reasoner, the algorithm is structured around the implementation of six distinct rules. These rules are systematically applied to each element within the current interpretation, exhaustively exploring all possible applications to ensure comprehensive reasoning.

Further, the algorithm is designed to update the system's state whenever a new element is introduced or an existing concept is assigned a new property. This dynamic nature of the algorithm is reflected in the design of the rule functions, each of which returns a `boolean` indicator denoting whether there has been a change in the system state.

Our Description Logic (DL) reasoner architecture relies on node objects to structure ontology elements. Each node retains references to its predecessor and assigned concepts, facilitating relationship and hierarchy tracking. The graph-based structure not only aids in the graphical representation of elements but also streamlines rule application, enhancing the reasoning process. This design aligns with the intricate nature of DL and EL reasoning tasks.

Algorithm Implementation:

The algorithm starts by preparing the ontology for Python using `prepare_ontology()` and `get_axioms()` functions, leveraging a Java framework for ontology parsing. Subsequently, the initial node, d_0, is created, and the queried class is added. The algorithm operates within a while loop, checking for changes in the system state. If a new element or concept assignment occurs, the loop continues, ensuring an efficient reasoning process. The pseudo-code encapsulates these steps:

The first loop iterates through all nodes. $rule\_one()$ is executed before the second loop because it needs to check every node instead of the every element in each node. The first rule checks if T is assigned to the node or not. If not,

it assigns T and changes the system state. Then, the algorithm moves into the second loop through every element in the iterated node.

Inside the second loop, all remaining rules are executed one by one over all elements of the iterated node. First, $rule\_two()$ checks if $\cap$ exists in the given element. If so, the function splits the formula into 2 element by considering $\cap$. Then, the function checks whether the axioms list includes the splitted parts. If any of the splitted elements is in the axioms list somehow and it is not appended to the elements of the node before, the function appends it and changes the system state to True. Otherwise, the function does nothing.

When it comes to the $rule\_three()$, it gets all combinations of the iterated element with all elements in iterated node so that it could check whether the $element_1 \cap element_2$ exists in the axioms list. If it exists in axioms list and it is not appended to the elements of iterated node before, the function appends $element_1 \cap element_2$ and changes the system state to True. Otherwise, the function does nothing.

The $rule\_four()$ is activated if the element starts with the symbol of $\exists$. It is the most complicated operator among all rules. The function splits the formula into 2 element by considering $\exists$. It searches for the second part of the splitted formula inside the elements of all other nodes. If it catches a match, it makes the founded node the successor of the iterated node considering the first part of the splitted formula. If it cannot find such a node, it creates a new node successor to iterated node. Finally, it changes the system state to True.

If the iterated node has a successor, the $rule\_five()$ is executed. It basically assigns to the successor a new element with the $\exists$ rule if the element does not exist in the successor node. Finally, $rule\_six()$ is finds all axioms that are like $A \subseteq B$ and assigns both A and B to the nodes that have either A and B. Then, it changes the system state to true.

Algorithm 1 Our Reasoner - Pseudo-Code

```
1:  while system state = True do
2:      Set system state = False
3:      while i < length(nodes list) do
4:          Execute rule_one()
5:          while j < length(elements of i^th node) do
6:              Execute system state = rule_two()
7:              Execute system state = rule_three()
8:              Execute system state = rule_four()
9:              Execute system state = rule_five()
10:             Execute system state = rule_six()
11:         end while
12:     end while
13: end while
```

Table 1: Test Environment

Ontology	n Axioms	n Concepts	Size
Our Ontology	37	36	S
Pizza Ontology	266	99	M
Ore Ontology 10516	450	294	L

4.1 Experimental Setup

To maintain consistency across all experiments and ensure reliable comparisons, all tests are conducted on the same computing environment. This approach eliminates variations in hardware and software configurations that could potentially influence the performance outcomes, thereby providing a standardized basis for evaluating the efficiency of our EL reasoner implementation.

5 Results

To test our research question, we build a test environment comprising of 3 different ontologies, each varying in size as seen in the Table 1. Our objective is to compare the execution time of two algorithms across the ontologies by selecting disparate concepts at random from each ontology. These concepts serve as inputs for the algorithms. In order to gather the performance data, we run the algorithms for each class 25 times. This methodology enables us to collect comprehensive runtime metrics, which are essential for a robust evaluation of the hypothesis at hand.

Figure 2 illustrates the results of Our Ontology. Upon examination of the figure 2, it is evident that the performance of the algorithm employing Java significantly exceeds that of the algorithm using string operators in terms of speed. Execution of a two-sided t-test for the constituent subexperiments yields p-values nearly 0. Based on these outcomes, we can confidently reject the null hypothesis of t test asserting that the mean execution speeds of the two algorithms are equivalent.

The observations noted for Experiment 2, involving the Pizza Ontology, and Experiment 3, involving the the Ore Ontology, align with those from the preceding trials. In each instance, the EL algorithm implemented in Java demonstrated superior performance in terms of speed when compared to its counterpart. The significant disparity observed in the mean processing times precludes the necessity for a two-sided t-test in these cases, as the difference in performance is markedly evident.

6 Discussion

In addressing the performance differences between our Python-based EL reasoner and the Java-based ELK and HermiT reasoners, it's essential to consider

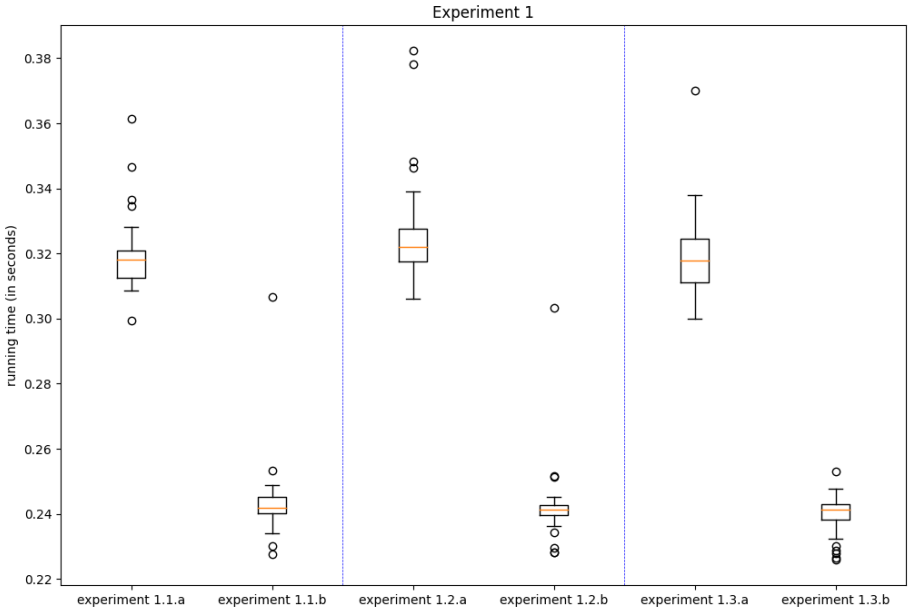

Figure 2: Experiment 1 Results

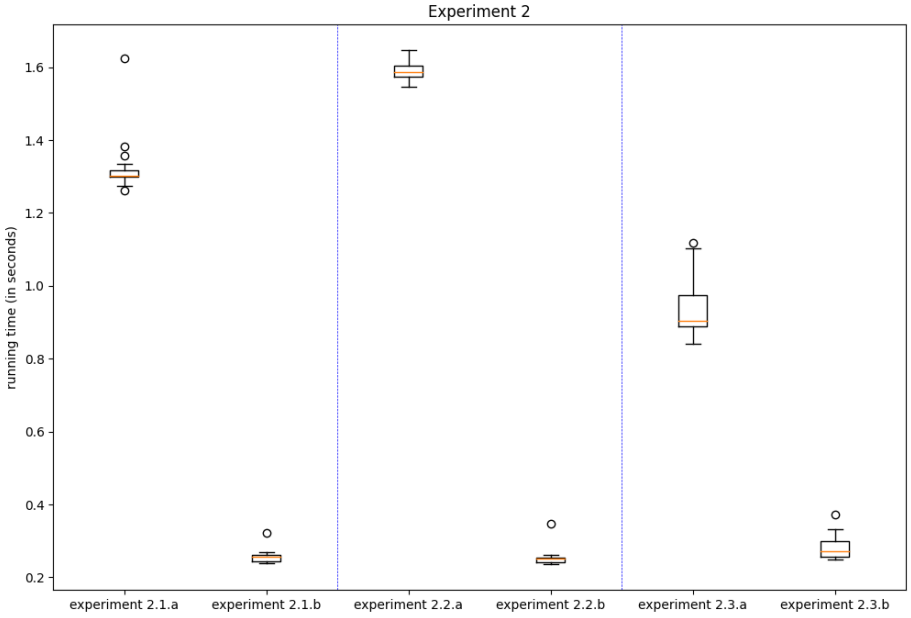

Figure 3: Experiment 2 Results

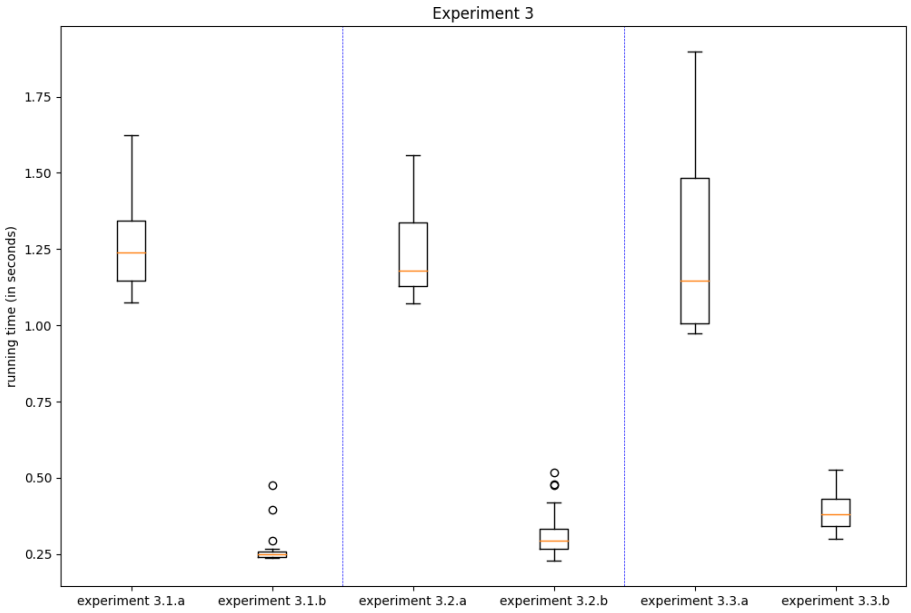

Figure 4: Experiment 3 Results

the inherent characteristics and optimizations present in these tools. Notably, ELK and HermiT are recognized for their efficient handling of ontologies within the realm of Description Logics (DL).

Description Logics, a family of knowledge representation languages, provide a formal framework for encoding knowledge about the world and reasoning about it. They are instrumental in developing ontologies, which are explicit formal specifications of the terms in the domain and relations among them. Ontologies play a crucial role in structuring and managing knowledge, particularly in AI systems.

ELK is optimized for the ELH profile of the OWL (Web Ontology Language), which is based on the ELHO variant of DL. ELK is specifically designed for classifying large terminological boxes (TBoxes), which are collections of class and property descriptions.[3] On the other hand, HermiT is based on the SROIQ(D) DL, which is more expressive and is the foundation of OWL DL.[2] OWL DL offers a balance between expressiveness and computational decidability, allowing for a broad range of ontological modeling while ensuring that reasoning processes can be completed in finite time.

The reason for the superior performance of ELK and HermiT in certain scenarios lies in their specialized nature. ELK is fine-tuned for handling large TBoxes efficiently, making it ideal for scenarios where rapid classification of large terminologies is needed. HermiT, while potentially slower due to its broader expressivity, is capable of handling more complex ontologies with a richer set of

logical constructs.

In contrast, our Python-based implementation, though it successfully implements the EL reasoning algorithm, does not possess the same level of optimization and specialization as ELK and HermiT. While it provides the basic functionality of an EL reasoner, it lacks the advanced optimizations and tailored algorithms that contribute to the high performance of ELK and HermiT. This difference in design and optimization is a primary factor in the observed performance gap between our implementation and these established reasoners.

Furthermore, the choice of implementation language and specific programming techniques plays a significant role. The use of Python and its string operations, while providing a flexible and accessible development environment, does not offer the same computational efficiency as Java, especially in the context of complex reasoning tasks. This difference is crucial in understanding why ELK and HermiT outperform our Python-based reasoner in terms of speed and efficiency.

7 Conclusion

In conclusion, the observed performance differences can be largely attributed to the specialized nature and advanced optimizations of ELK and HermiT, as well as the inherent characteristics of the languages used for their implementation. Our Python-based reasoner, while effective in implementing the EL reasoning algorithm, does not match the performance of these established tools due to its lack of similar optimizations and the inherent limitations of the chosen programming language and techniques.

References

[1] Barthélémy Dagenais. Py4J - A Bridge between Python and Java, 2023. Accessed: November 25, 2023.

[2] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: an owl 2 reasoner. *Journal of automated reasoning*, 53:245–269, 2014.

[3] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. Elk reasoner: architecture and evaluation. In *ORE*. Citeseer, 2012.