Knowledge Representation Report

Project 1: Comparison of the EL, the ELK and Hermit Reasoner when using the Subsumption Algorithm

Abstract

This report outlines a mini-experiment conducted as part of project 1 for the Knowledge Representation course. The focus of the project was the development and implementation of a reasoner employing the EL Subsumption Algorithm on an ontology related to a sushi restaurant menu. The report covers the design process in Protégé, the development of a Python-based EL reasoner, and an experiment comparing its performance with other existing reasoners.

Introduction

In fulfillment of the Knowledge Representation course requirements, this project delves into a Python-based reasoner utilizing the EL Subsumption Algorithm. The chosen ontology centers around the menu of a sushi restaurant, providing a rich source for testing diverse classes, relationships, and individuals. After creating the ontology in Protégé, a Python-based EL reasoner was developed based on the algorithm explained in the Knowledge Representation 2023 lecture [1]. The primary aim is to identify all subsumers for each class within the ontology. The aim of the experiment is to answer the following research question: 'How does the performance of our own EL reasoner, compare to alternative reasoners in identifying subsumption relationships within a sushi restaurant ontology?'. In the first section some of the Description Logics literature is reviewed, followed by a methodology section in which an outline of the sushi restaurant ontology and the experiment conducted to assess the reasoner's performance against other existing tools in terms of subsumer detection are explained. The final sections provide a presentation and discussion of the results of the research question.

Literature Review

The pivotal role of an EL reasoner in Knowledge Representation becomes evident when examining subsumption relationships within ontologies. The term "EL" designates Description Logics (DLs) characterized by restricted expressiveness, specifically tailored for simpler ontologies that boast greater computational tractability [2]. At its core, an EL reasoner is designed to perform different tasks, among which an important one is to decipher subsumption relationships between classes within an ontology. An ontology contains certain axioms from which knowledge can be extracted. In order to extract implicit facts from this knowledge a reasoner can be used [1]. The purpose of the reasoner is specifically figuring out whether a certain axiom is entailed by the ontology. In this case, specifically the subsumption axiom.

The EL ontologies facilitate more efficient reasoning processes, making them particularly adept for large-scale ontologies where computational efficiency is a critical factor. There are some reasoning tasks that are not interesting in EL reasoning. Operators like top, bottom and negation are usually neglected. It is not possible to create contradictions with this type of reasoning and axioms containing equivalence are also not reasoned with.

In contrast, ALS (Attributive Language with Complements) reasoners contain more expressive ontologies, allowing for a broader spectrum of constructs and axioms. While ALS reasoners can adeptly handle richer knowledge representations, they often incur a higher computational cost. The differentiation between EL and ALS reasoners hinges on their expressiveness and computational complexity, with EL reasoners prioritizing simplicity and efficiency, while ALS reasoners accommodate the complexities of more intricate ontologies at the cost of increased computational demands [2].

In the context of subsumption, EL reasoners typically employ specific algorithms, such as the EL Subsumption Algorithm, to distinguish relationships between classes. This algorithm simplifies the axioms in the Tbox, which is the part of the ontology which contains terminological axioms [3], by focusing on terminological axioms and excluding equivalence axioms, thereby rendering the reasoning process more manageable. The EL subsumption algorithm asserts a set of specific rules [1] on the axioms in order to determine the entailed subsumption axioms within the ontology, revealing the subsumers of each class, in other words:

$$O \models C \sqsubseteq D \quad \text{the ontology is entailed by all C are D}$$

Appreciating the distinctions between EL and ALS reasoners is imperative when selecting the appropriate tool for a given knowledge representation task. EL reasoners, characterized by their efficiency and suitability for simpler

ontologies, prove invaluable in scenarios where computational resources are a critical consideration. In contrast, ALS reasoners offer a more expressive framework for navigating complex knowledge structures, with the trade-off of heightened computational demands.

Methods

For this research we have created an ontology regarding a Sushi Restaurant. The Sushi Restaurant ontology contains 42 classes, 3 object properties and 24 individuals.
The class hierarchy of the ontology is depicted in Image 1.

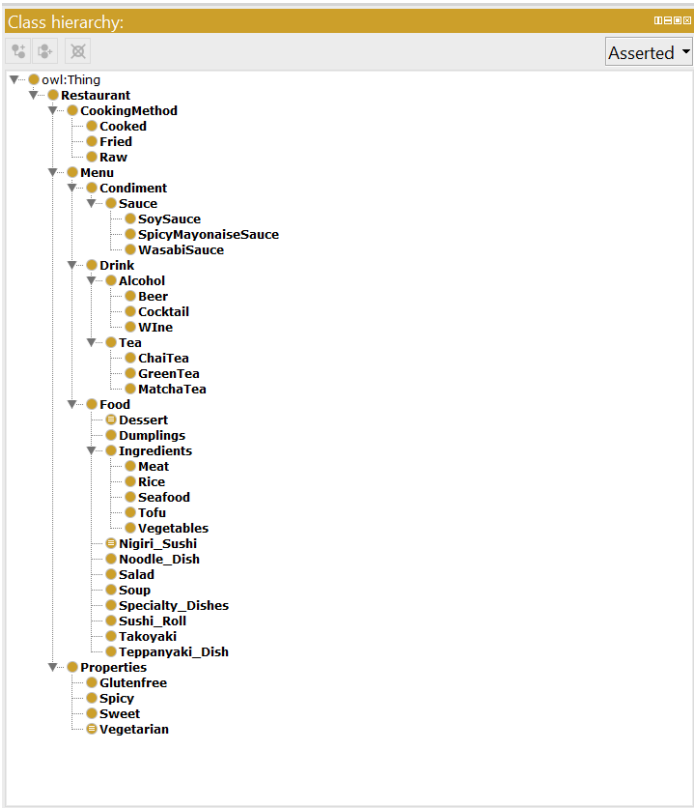

Image 1 Class hierarchy of the sushi ontology

Image 2 showcases the Restaurant class, containing the subclasses CookingMethod, Properties and Menu. As can be seen, there exists tree object properties in the ontology. These are ‘hasProperty’,’hasCookingMethod’ and ‘hasIngredient’. By creating axioms which express the internal structure of the ontology, using these classes, object properties and individuals, the practical application is that rich knowledge is retrievable from the menu to provide customers exactly with all the necessary information they need to make a well informed consumption choice.

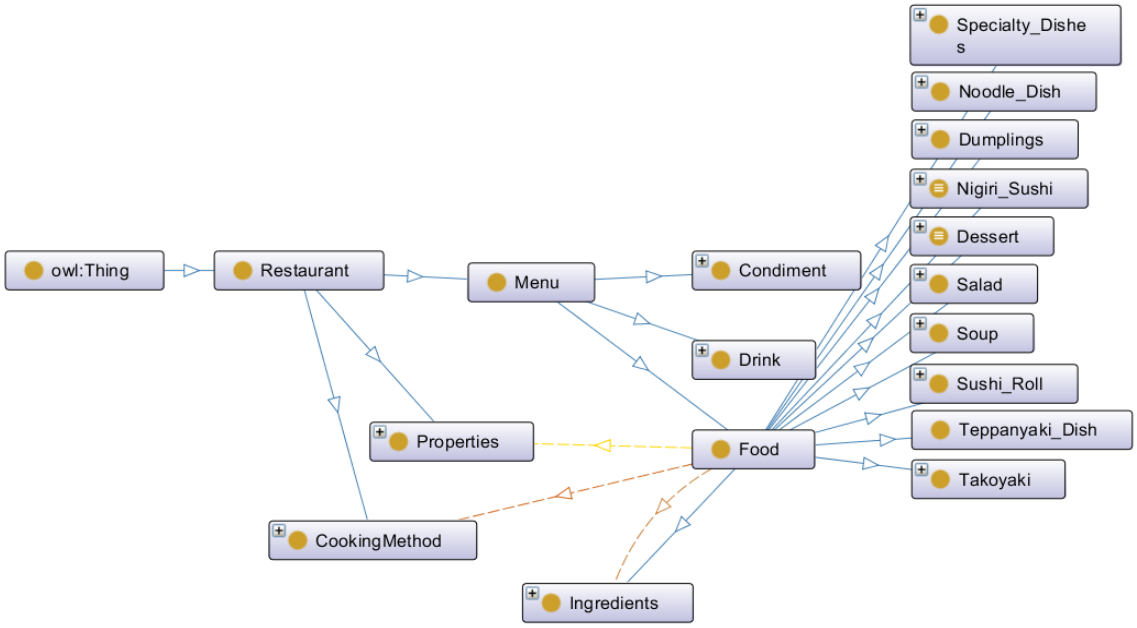

Image 2 Broad overview of Sushi Restaurant ontology

The ontology contains several complex class expressions, from which an example is shown in Image 3for the class Nigiri sushi.

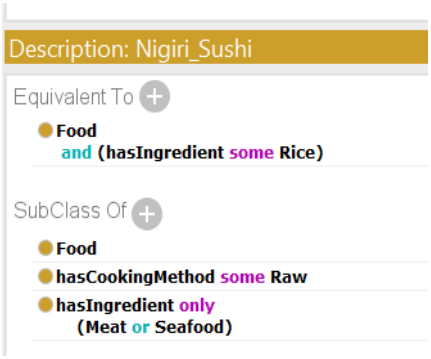

Image 3 Some class expressions for class 'Nigiri Sushi'

In order to carry out the research, a set of EL rules have been defined in the EL reasoner code. First we start with selecting an element d_0 based on the assumption $O \models C_0 \sqsubseteq D_0$, where C_0 is initially assigned to d_0. The rules aim to analyze whether the following rules can be applied to elements d to eventually assign D_0 to d_0:

The $\top$-rule involves the inclusion of $\top$ to d. For the $\sqcap$-rule 1, if d already possesses an assignment for $C \sqcap D$, C and D are additionally assigned to d. $\sqcap$-rule 2 states that if d already has assignments for both C and D, $C \sqcap D$ is further assigned to d.

Moving on to $\exists$-rule 1, in cases where d holds an assignment for $\exists r.C$ and lacks an r-successor with an assignment for C, a new r-successor is introduced to d with an assignment for C. $\exists$-rule 2, if d has an r-successor with an assignment for C, $\exists r.C$ is added to d.

Finally, $\sqsubseteq$-rule states that if d is assigned C and $C \sqsubseteq D$ exists in the TBOX T, then D is also assigned to d.
The aforementioned rules were executed by using the Python code as depicted below.

Whenever the reasoner walks over an axiom containing a negation $\neg$ or bottom $\bot$ or an existential quantifier, the reasoner will not include that axiom in the reasoning.

```
def el_completion_algorithm(input_class_name, tbox):
    # Check if the TBox class names are in quotes
    if are_class_names_quoted(tbox):
        class_name = f'"{input_class_name}"'
    else:
        class_name = input_class_name

    # Start with the initial concept and add the top concept (⊤)
    current_interpretation = {class_name: {'⊤', class_name}}
    # Dictionary to keep track of the r-successors for each individual
    r_successors = {}
    # Set of subsumers for the initial concept
    subsumers = set()
    changed = True

    while changed:
        changed = False

        # Apply rules on each element in the interpretation
        for d, concepts in list(current_interpretation.items()):
            # Apply ⊤-rule: Add ⊤ to any individual
            if '⊤' not in concepts:
                current_interpretation[d].add('⊤')
                changed = True

            # Apply ⊓-rule: If d has C ⊓ D assigned, also assign C and D to d
            for concept in list(concepts):
                if '⊓' in concept:
                    c1, c2 = concept.split('⊓')
                    # Skip negations and universal quantifiers
                    if '¬' in c1 or '∀' in c1 or '¬' in c2 or '∀' in c2 or
'∃' in c1 or '∃' in c2:
                        continue
                    if c1 not in concepts:
                        current_interpretation[d].add(c1)
                        changed = True
                    if c2 not in concepts:
                        current_interpretation[d].add(c2)
                        changed = True

            # Apply ∃-rule 1: If d has ∃r.C assigned, ensure there is an
r-successor with C
            for concept in list(concepts):
```

```
                if concept.startswith('∃'):
                    _, r, c = concept.split('.')
                    # Skip if the filler has negations or universal quantifiers
                    if '¬' in c or '∀' in c or '∃' in c:
                        continue
                    r_successor_exists = False
                    for existing_r_successor, existing_r in r_successors.get(d,
[]):
                        if c == existing_r_successor and r == existing_r:
                            r_successor_exists = True
                            break
                    if not r_successor_exists:
                        new_individual = f'new_{len(current_interpretation)}'
                        current_interpretation[new_individual] = {'⊤', c}
                        r_successors.setdefault(d, set()).add((new_individual,
r))
                        changed = True

            # Apply ∃-rule 2: If d has an r-successor with C, add ∃r.C to d
            for r_successor, r in r_successors.get(d, []):
                for c in current_interpretation[r_successor]:
                    existential_concept = f'∃{r}.{c}'
                    # Skip if the filler has negations or universal quantifiers
                    if '¬' in c or '∀' in c or '∃' in c:
                        continue
                    if existential_concept not in concepts:
                        current_interpretation[d].add(existential_concept)
                        changed = True


            # Apply ⊑-rule: If d has C assigned and C ⊑ D is in T, then also
assign D to d
            for axiom in tbox:
                if axiom.getClass().getSimpleName() ==
"GeneralConceptInclusion":
                    lhs_formatted = formatter.format(axiom.lhs())
                    rhs_formatted = formatter.format(axiom.rhs())
                    if lhs_formatted in concepts and rhs_formatted not in
concepts:
                        # Skip negations and universal quantifiers in GCIs
                        if '¬' in lhs_formatted or '∀' in lhs_formatted or '∃'
in lhs_formatted or '¬' in rhs_formatted or '∀' in rhs_formatted or '∃' in
rhs_formatted or '⩽' in lhs_formatted or '⩽' in rhs_formatted:
                            continue
                        current_interpretation[d].add(rhs_formatted)
                        changed = True

            # Collect subsumers for the initial concept
            if d == class_name:
```

```
                subsumers.update(current_interpretation[d])

    return subsumers
```

As mentioned, the EL reasoner also contains a few additional rules. Whenever the code comes across an axiom with the Equivalence property, the axiom should be replaced by two transitive general concept inclusions. Meaning,

if $C \equiv D$, then $C \sqsubseteq D$ and $D \sqsubseteq C$

```
def new_tbox(tbox):
    new_tbox = []
    axioms = tbox.getAxioms()

    for axiom in axioms:
        axiomType = axiom.getClass().getSimpleName()
        if axiomType == "EquivalenceAxiom":
            concepts = list(axiom.getConcepts())
            if len(concepts) == 2:
                left_concept = concepts[0]
                right_concept = concepts[1]
                gci1 = elFactory.getGCI(left_concept, right_concept)
                gci2 = elFactory.getGCI(right_concept, left_concept)
                new_tbox.append(gci1)
                new_tbox.append(gci2)
        else:
            new_tbox.append(axiom)
    return new_tbox
```

The full code for the EL Reasoner can be found in the accompanying EL Reasoner Python file.

For the experiment, we conducted a comparison of our EL reasoner with the ELK reasoner and HermiT reasoner to evaluate potential differences in subsumption reasoning across these tools. To highlight the differences between these reasoners: ELK is designed for EL-based profiles, with a priority for computational efficiency in large-scale ontologies. On the other hand, HermiT supports the expressiveness of OWL, e.g. inferring, but suffering high computational costs especially for large and complex ontologies.

Our evaluation includes two distinct ontologies: the sushi restaurant ontology, featuring five few complex expressions, and the pizza ontology, with a more complex and comprehensive structure. The experiment was based on formulating a targeted query aimed to retrieve all subsumers that contain the class ‘Margharita’ for the pizza ontology and the class ‘Food’ for the sushi restaurant ontology. This approach allowed us to assess the reasoning capabilities of each reasoner and examine the extent to which they could infer relationships, by comparing the outcomes of this query for all reasoners.

This experimental design aims to provide insight into the comparison in expressiveness of these three reasoners. We aim to examine the extent of inference within these ontologies and assess whether the variations are noteworthy across the two distinct and complex ontological structures.

Results

For the class name “Food” in our implemented sushi ontology we can see that all 3 reasoners -our EL reasoner, the ELK reasoner and the hermiT reasoner, get the same subsumers. In fact, this is the case for all classes in our ontology for the sushi restaurant. Results are shown in the table below.

Our EL reasoner	**ELK**	**hermiT**
Food	Food	Food
Restaurant	Restaurant	Restaurant
Menu	Menu	Menu

However when using the provided pizza ontology, and prompting the different algorithms to check for all subsumers for the class name “Margherita”, differences arise. Results can be seen in the table below.

Our EL reasoner	**ELK**	**hermiT**
Pizza	Pizza	Pizza
Margherita	Margherita	Margherita
NamedPizza	NamedPizza	NamedPizza
Food	Food	Food
DomainThing	DomainThing	DomainThing
	CheesyPizza	VegetarianPizza
		VegetarianPizza1
		VegetarianPizza2

Discussion

In the context of ontology reasoning and the task of inferring subsumers for a given class name, such as "Margherita" in a pizza ontology in this case, the differences between a simple EL reasoner, the ELK reasoner, and the HermiT reasoner become quite evident through their respective results and underlying reasoning mechanisms.

The simple EL reasoner, designed primarily for lightweight ontology languages, offers basic reasoning capabilities that focus on capturing essential hierarchical and conceptual relationships. Its result set for "Margherita" - consisting of Pizza, Margherita, NamedPizza, Food, and DomainThing - reflects a straightforward hierarchical inference, tracing the class up through its most direct superclasses. This approach, while efficient, reveals its limitations in dealing with more complex relationships or rules, as it doesn’t seem to infer beyond direct superclass relationships or to consider more nuanced ontology axioms.

In contrast, the ELK reasoner, which is well-optimized and known for its high performance in large ontologies, demonstrates a more sophisticated approach. It not only identifies the classes found by the simple EL reasoner but also includes "CheesyPizza" as a subsumer. This suggests that ELK is capable of recognizing more complex relationships, possibly identifying "Margherita" as a type of "CheesyPizza" through property restrictions or class expressions not captured by the simple EL reasoner. This balance between complexity and performance makes ELK suitable for moderately complex ontologies.

Lastly, the HermiT reasoner supports a wider range of OWL features and uses a more comprehensive approach to reasoning. It can handle complex logical inferences, as reflected in its identification of "VegetarianPizza," "VegetarianPizza1," and "VegetarianPizza2" in addition to the classes identified by the previous reasoners. This indicates an ability to infer deeper hierarchical structures and perhaps to identify "Margherita" as part of more specific subclasses within the ontology.

While HermiT provides a more thorough and complex reasoning capability, it can be more resource-intensive and slower, particularly in large ontologies. Comparatively, we think that the choice of reasoner depends on the specific needs of the ontology, the complexity of relationships it needs to capture, and the performance considerations. The simple EL reasoner is suited for less complex ontologies, such as our sushi restaurant ontology, where basic but fast reasoning is required. For ontologies requiring more nuanced inferences, ELK or HermiT would be more appropriate, with ELK providing a balance between complexity and performance and HermiT offering the deepest level of inference but at the cost of efficiency. The variance in the results from querying "Margherita" in a pizza ontology demonstrates the varying capabilities and suitability of each reasoner for different ontology reasoning tasks. The selection should be guided by the complexity of the ontology, the depth of inference required, and performance considerations, highlighting the need to match the reasoner to the specific characteristics of the ontology at hand.

Also, we are well-aware that our EL implementation is not fully-correct. In fact, we know that there are rules that are not applying and that the rules should be looping over “all_concepts” variable obtained using the “getSubConcepts()” method as demonstrated in the example.py file. When trying to do that we got run-time errors and we weren’t able to not even get 1 subsumer for a given class name hence, we decided to go with our current approach. This is a limitation that should be overcome in further research.

Conclusion

The project successfully demonstrated the efficacy of the EL Subsumption Algorithm in identifying subsumers within the sushi restaurant ontology, providing valuable insights into the application of reasoners in knowledge representation. It was observed that the developed EL Reasoner, executing the subsumption algorithm, effectively handled general concept inclusion. However, a comparison with the ELK reasoner highlighted certain limitations in our reasoner, particularly in terms of lacking a few essential rules.

Specifically, the absence of functionalities such as the existential rule and the transformation of equivalence to inclusion steps posed challenges in achieving parity with the advanced capabilities of the ELK reasoner. Additionally, the project revealed that the simplicity of our EL reasoner made it unsuitable for direct comparison with the HermiT reasoner, which predominantly relies on more advanced reasoning capabilities.

In conclusion, while our EL Reasoner demonstrated proficiency in general concept inclusion, it did not surpass the performance of ELK or HermiT due to the absence of crucial functionalities. Notably, the lack of the existential rule and transformation of equivalence to inclusion steps underscored the limited scope of our reasoner, emphasizing its

role as a straightforward tool designed primarily to showcase general concept inclusion rather than to compete with more advanced reasoners.

References

1. Koopmann, Patrick. (2023, November 10). *Lecture 5 Knowledge Representation: Practical Reasoning with EL* [Powerpoint Slides].
2. Baader, Franz & Calvanese, Diego & Mcguinness, Deborah & Nardi, Daniele & Patel-Schneider, Peter. (2007). The Description Logic Handbook: Theory, Implementation, and Applications.
3. Koopmann, Patrick. (2023, November 10). *Lecture 4 Knowledge Representation: Description Logix Axioms, Reasoning and OWL* [Powerpoint Slides].