Optimizing EL Reasoning: Combined Rule Application and Queue Strategy

November 25, 2023

Abstract

Description Logic (DL), as a part of knowledge representation, is widely in ontology building. Based on previous study, there already exist a number of researches that build a well functioning ontology that could offer information inference and recommendation based on present knowledge. With such convenient property, we decide to build our own ontology for a sushi restaurant as well as an EL reasoner algorithm. At the end of this research, an optimization of our EL reasoner algorithm is done to make a comparison of execution time and numbers of iterations, between two algorithms.

1 Introduction

As people increasingly prioritize their health, food is also considered as an indispensable part. By the knowledge we learned from course Knowledge Representation, it is possible and reasonable to build an intelligent system for a restaurant to recommend customers personalized dishes if they have allergies or diet requirements. For our group, we choose a sushi restaurant to start with. Sushi, as common knowledge, has a lot of different food ingredients, ranging from a big amount of seafood to various types of vegetables. Our intelligent system has an overall knowledge of the ingredients as well as dishes, and it could customize some recommendations based on customers tastes and dietary requirements. A striking feature could be that some customers, that are not comfortable with raw fish, want to ask the system which dishes they should avoid when taking order, the system will select all the dishes with raw fish in it, in such way that the dietary requirements of customers could be easily fulfilled.

After the construction of ontology, the knowledge base is done. Then we will start with our reasoner algorithm. We choose EL reasoner, which is a member in Description Logic (DL) family. EL reasoner algorithm is one of the most doable reasoner in DL, it can do some reasoning within tolerable time.

In recent researches, some authors also put a great value on the possibilities that ontologies as well as EL reasoner could bring.

In their research, IJntema et al. (2010) compared the outcomes of both ontology-based system and traditional term-based algorithms. They demonstrated that the ontology-based recommend system, outperforms traditional term-based recommend systems in terms of accuracy and precision. This research result gives us a high motivation to develop our own ontology-based algorithm to accomplish recommendation tasks in a sushi restaurant.

Horrocks and Sattler (2001) built an ontology with description logic SHOQ(D) with a sound and complete decision procedure for concept satisfiability and subsumption. In our project we use EL instead of SHOQ(D). The main reason is that despite SHOQ(D) provides more expressive modeling capabilities, the computational complexity increases, while EL sacrifices some expressiveness for computational tractability. The authors followed the SHOQ(D) rules step by step, which is inspiring for our own EL reasoner build-up, since the rule-applying stages are quite similar.

Kazakov et al. (2012) took a step in-depth by solely researching the properties of ELK reasoner. They came to conclusion that ELK stands out as a dedicated reasoner designed for the streamlined ontology language OWL EL. ELK is practical in the sense that its fusion of superior performance with extensive coverage of language features. Their conclusion validates a good reason to develop an EL reasoner because of its praticality.

Ontology reasoning is dominant in many other fields. With the same idea to put ontology system into usage in a restaurant, Snae and Bruckner (2008) developed a complete ontology system which can not only individualize the menu based on the customers' preference, but also store the customers' previous preference and ask them to change when they come back to the restaurant. On the other hand, ontology works well also in health information system, which is comprised of cases to recommend health information based on patients' context using instances of ontology. Lee and Kim (2015) made a comparison between two different systems, one of which is ontology-based, in eHealth recommendation. Kang and Choi (2011) dived into the potential to use ontology in pandemic context, and they also concluded that ontology enhances customization precision and boosts the flexibility, extensibility, and reusability of learning objects. Ontologies also perform well in context model and reasoning. Wang et al. (2004) showed that ontology based context model is feasible and necessary for supporting context modeling and reasoning in pervasive computing environments. Their researches all point to a high utility of ontology-based system.

In this study, we propose addressing the following main research question: *To what extent does the adjustment to the $\mathcal{EL}$ reasoning algorithm, achieved by consolidating $\sqcap$-rule-1 and $\exists$-rule-1 in a unified loop, and incorporating a double-ended queue for element processing, influence the algorithm's computational efficiency, specifically in terms of execution time and iteration count?*.

An optimized version of the $\mathcal{EL}$-Completion Algorithm has been devised to enhance efficiency and computational performance. This refined algorithm addresses two critical aspects of the original implementation. Firstly, combining $\sqcap$-rule-1 and $\exists$-rule-1 in a single loop streamlines rule application, reducing redundancy and improving computational efficiency. Secondly, utilizing a double-ended queue for element processing introduces a structured approach, potentially enhancing convergence to a stable state. These modifications aim to optimize the reasoning process, improving its effectiveness in ontology interpretation.

2 Algorithm Description

In this section, we will introduce and elaborate on two implemented algorithms, setting the stage for a comparative analysis in the subsequent chapters. Both algorithms can be applied to a sushi restaurant's ontology, which has hierarchy of classes representing different types of food, ingredients, sushi rice, and specialness, described in Section 3.1.2.

2.1 $\mathcal{EL}$-Completion Algorithm

1. **Initial Setup**: We always begin with an initial element d_0 and assign it to an initial concept C_0. The algorithm aims to compute all subsuming concept names of the selected concept name C_0.
2. **Iterative Completion**: Iteratively apply completion rules to ontology elements. A 'changed' flag indicates if any changes occur during an iteration. The iteration stops when no changes are made.
3. **Applying Rules**: Use completion rules ($\sqsubseteq$-rules and $\exists$-rules, among others) to expand the set of concepts for each individual based on the ontology's defined relationships and constraints. The details of the rule application process, along with specific rules, are not described in this section, as they are extensively covered in Lecture 5.
4. **Termination**: Upon completion of the reasoning process, the final set of subsumers is derived if no new concepts were assigned and no additional elements were created.

2.2 Optimization of the $\mathcal{EL}$-Completion Algorithm

Below we describe a modified version of the initial reasoning algorithm, aiming to enhance its efficiency and optimize its processing. Two key adjustments have been implemented, each addressing specific aspects of the original algorithm.

2.2.1 Consolidating Rule Applications

In our modified algorithm, we made a key change by combining two rules, $\sqcap$-rule-1 and $\exists$-rule-1, that originally required separate passes through the concepts of the current element. This adjustment simplifies the algorithm, reducing repeated iterations and boosting computational efficiency. Our goal is to streamline the process and save time by addressing both rules in a single loop that goes through the concepts of the current element.

2.2.2 Element Processing Using a Double-Ended Queue

The second major modification involves a strategic change in how elements are processed. Instead of using while loop to iterate through all elements in each cycle, the modified algorithm utilizes a double-ended queue for element processing.

Algorithmic Workflow:

- The initial element is added to the queue, and the number of iterations without change is set to 0.
- Elements are dequeued from the right for processing.
- Rules are applied to the current element.
- If new elements are generated, they are added to the right of the queue.
- If no new concepts are assigned to the current element, it is added to the left of the queue; otherwise, it is added to the right.
- If no new concepts or elements are created during an iteration, the number of iterations without change increases by 1. This modified approach aims to leverage the inherent characteristics of a double-ended queue, potentially reducing the overall execution time.

3 Experimental Setup

In this section, we discuss the implementation of the algorithms previously described. Additionally, we take a detailed examination of the ontology we have been worked on. Since the ontology is where the knowledge is stored, having an overall view of it could help the users, not necessarily the customers but the restaurant managers, to understand what the system is capable of.

3.1 Sushi Ontology

Our Sushi Ontology contains mostly about different types of dishes and the ingredients contained in the dishes, without considering drinks and desserts, as customers care more about the dietary requirements in sushi, which is the result of sushi's various ingredients. However, the ontology could be easily extended to contain these menus as well.

3.1.1 Class Hierarchy

The sushi ontology we have has two general classes, namely "Food" and "Specialness". "Food" class is used to define all kinds of dishes as well as ingredients, while "Specialness" class is used to define possible dietary requirements, such as raw or cooked dishes, as well as taste preferences like spiciness.

First we look at "Food" class, where there are three sub-classes - "Sushi", "SushiIngredient", and "SushiRice". Both "SushiIngredient" and "SushiRice" are served to classify sub-classes in "Sushi".

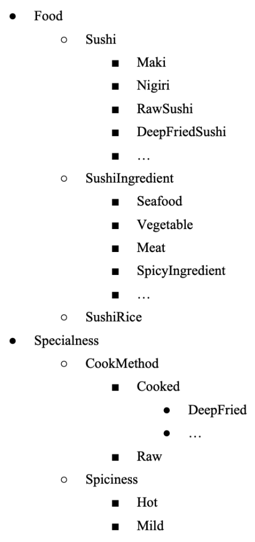
Under "Sushi" we have several sub-classes, most of which denote the property of a dish, such as "SpicySushi" and "RawSushi". By defining these sushi, at later steps we can easily infer if a sushi belongs to spicy one or if it has raw fish as its ingredient, while "Maki" and "Nigiri" contains the dishes we are going to reason about.

Under "SushiIngredient", several well-classified ingredients could be found. Worth-noting is that "CookedIngredient", "RawIngredient", "SpicyIngredient" and "Vegetarian" are also used to infer if the ingredients belong to such sub-classes, which facilitates our final reasoning.

Now we turn into another general class "Specialness". In "Specialness", there are two sub-classes "CookMethod" and "Spiciness", whose meaning could be easily inferred from the name.

Under "CookMethod", we made a distinction between "Raw" and "Cooked", which could be one of the most highly concerned dietary requirements when it comes to sushi, while inside "Cooked", we made another distinction between several cook method.

Then finally under "Spiciness", we have two different types spicy, with "Hot" denoting very spicy while "Mild" not that spicy.



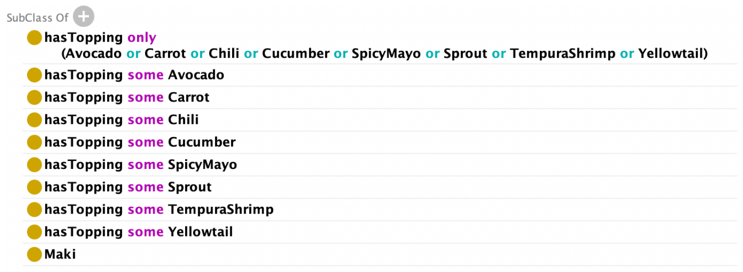
- Food
    - Sushi
        - Maki
        - Nigiri
        - RawSushi
        - DeepFriedSushi
        - ...
    - SushiIngredient
        - Seafood
        - Vegetable
        - Meat
        - SpicyIngredient
        - ...
    - SushiRice
- Specialness
    - CookMethod
        - Cooked
            - DeepFried
            - ...
        - Raw
    - Spiciness
        - Hot
        - Mild

Figure 1: Class Hierarchy

3.1.2 Complex Class Expressions

There are some complex class expressions we want to point out, since they contain specific information and lead to an entailment. For example, for "DynamiteRoll", we have the following: These existential restrictions together serve as a closure axiom, to put it in natural language, it means "A dynamite roll only has toppings chili, carrot, spicy mayo, sprout, avocado, cucumber, tempura shrimp as well as yellow tail and nothing else."

If we look at ∃hasTopping.SpicyMayo, and wrap it down, we can see that class "SpicyMayo" has a relationship "hasSpiciness" to "Hot", which is equivalent to "SpicyTopping". Therefore, "SpicyMayo" is reasoned as a sub-class of "SpicyTopping". Since "DynamiteRoll" has a "SpicyTopping" ("SpicyMayo") as an ingredient, which is equivalent to "SpicySushi", it is reasoned as a sub-class


SubClass Of

● hasTopping only
(Avocado or Carrot or Chili or Cucumber or SpicyMayo or Sprout or TempuraShrimp or Yellowtail)
● hasTopping some Avocado
● hasTopping some Carrot
● hasTopping some Chili
● hasTopping some Cucumber
● hasTopping some SpicyMayo
● hasTopping some Sprout
● hasTopping some TempuraShrimp
● hasTopping some Yellowtail
● Maki

Figure 2: Class Expression of "Dynimate Roll"

of "SpicySushi".

3.1.3 Relationships

Since we are just reasoning about diet requirements and taste preferences, we do not need complicated relationships base. We have three general classes of relationships “hasIngredient”, “hasSpecialness”, and “isIngredientOf”, where “isIngredientOf” is simply the inverse of “hasIngredient”.

3.2 Reasoners Implementation

3.2.1 Implementation Details: Algorithm 1

The implementation of this algorithm can be found in the "el_reasoner.py" file.

In the implementation of the algorithm, dictionary was used to efficiently store essential information for each element, such as initial concepts, all concepts, and successors. The values associated with initial concepts, all concepts, and successors are stored as sets within the dictionary. The choice of using dictionaries and sets offers several advantages: dictionaries facilitate quick retrieval of information related to each element, optimizing the overall algorithmic efficiency; and sets ensure uniqueness, preventing redundancy in concept assignments.

Prior to the application of rules, equivalence axioms are replaced by two General Concept Inclusion axioms as described in Lecture 5.

The current interpretation is updated after each rule, meaning that the order of rule application is crucial. Additionally, an initial concept is identified if it has not been assigned to any element previously.

To address issues with standard dictionary copying methods, dict.copy() and copy.deepcopy(dict), a custom function for copying the dictionary was implemented. These difficulties could potentially arise due to potential the format of elements stored in sets.

To prevent missing conjunctions, both versions of the same conjunction are checked during $\sqcap$-rule-22 (as conjunction is commutative) to ensure their presence in the input concepts.

The reasoner is designed to be invoked from the command line using the format: python PROGRAM_NAME ONTOLOGY_FILE CLASS_NAME. Subsumers of the specified class are displayed, with one class name per line.

During testing, specific modes can be activated using the "mode" parameter in the main function; one of the possible values for mode is "lecture_example", which prints intermediate steps, class name, subsumers, and their total number for an example from Lecture 5 (slide 13). To obtain input concepts for this example, a custom function was implemented.

It is also important to note a limitation when testing the pizza ontology via the command line. The reasoner does not compute subsumers correctly, returning only one subsumer due to a specific format issue with class names in this ontology.

3.2.2 Implementation Details: Algorithm 2

The implementation of this algorithm can be found in the "el_reasoner_second.py" file.

The second algorithm builds upon the foundation of the first, with two notable exceptions previously discussed. It introduces an approach that combines $\sqcap$-rule-1 and $\exists$-rule-1 in a single loop, where the concept type for each concept of the current element is determined, and the corresponding rule is applied. For the sake of convenience, functions for executing these rules were incorporated.

To ensure immediate application of rules to concepts added during the same iteration, newly assigned concepts resulting from $\sqcap$-rule-1 are stored, and $\exists$-rule-1 is applied after the loop.

The concept of using a double-ended queue is maintained, employing the built-in deque from the collections module to facilitate efficient element processing.

The implementation reuses the copy_elements function, as well as the Ontology and ELReasoner classes from the first algorithm, promoting code modularity and reuse.

This algorithm, as well as the first one, supports only $\sqcap$, $\exists$, $\top$, and $\equiv$ axioms.

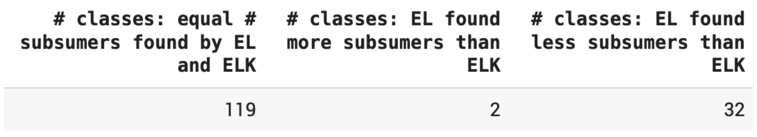
This refined algorithm aims to achieve a more streamlined and efficient execution, building on the principles of the initial approach while incorporating optimizations to enhance its overall effectiveness.

4 Experimental Results

For comprehensive experimental results, detailed information is available in the "results.csv" file, accompanied by data analysis and visualization in the "analysis.ipynb" notebook. In total, our algorithms underwent testing across 153 named classes sourced from 8 distinct ontologies (including sushi ontology), designated as "Reasoner 1" for the original algorithm and "Reasoner 2" for the modified version.

4.1 Comparing the Implemented $\mathcal{EL}$ Reasoner with ELK

To evaluate the performance of our implemented algorithm in comparison to the ELK built-in algorithm, we calculated the number of classes for which the subsumer counts were equal, greater, or fewer between the two reasoners. Figure 3 visually represents these comparisons, revealing that, with the exception of 2 classes in the pizza ontology (which is probably a mistake related to the format of concept names in this ontology), the two algorithms generally exhibited comparable results.



# classes: equal # subsumers found by EL and ELK	# classes: EL found more subsumers than ELK	# classes: EL found less subsumers than ELK
119	2	32

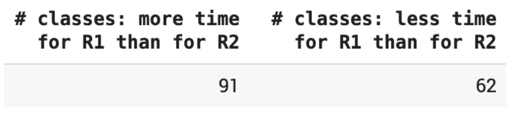
Figure 3: # of classes for which # subsumers were =, >, < for 2 reasoners

4.2 Evaluation and Comparison of two Implemented Algorithms

Moving on to the comparison between the two implemented algorithms to assess optimization possibilities, we evaluated the execution time and the number of iterations required for completion.

4.2.1 Comparison of Execution Time

Figure 4 illustrates that Reasoner 2 outperformed Reasoner 1 in terms of speed for the majority of classes (91 out of 153). Figure 5 provides a closer look, showing instances where Reasoner 1 took more time, though the differences were sometimes marginal, warranting a closer examination of the number of iterations.



# classes: more time for R1 than for R2	# classes: less time for R1 than for R2
91	62

Figure 4: Execution Time Comparison Table

4.2.2 Comparison of Numbers of Iterations

Concerning iterations, Figure 6 indicates that both Reasoners performed similarly overall. Intriguingly, Figure 7 demonstrates that for ontologies requiring more time for both reasoners, Reasoner 2 exhibited significantly better performance, sometimes requiring almost half the iterations. This notable improvement was particularly evident for classes such as "MerkelDisc," "Langerhan-

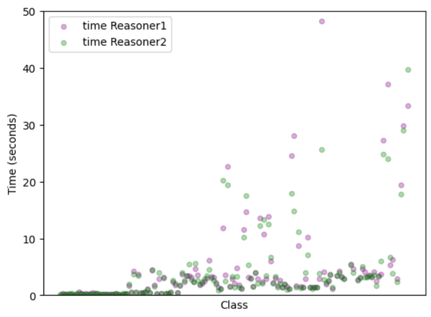
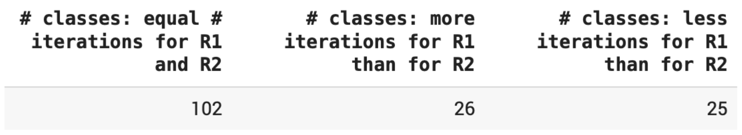


Figure 5: Execution Time Comparison Graph

sCell," "MerkelCell," and "StratumBasale" in the "Skin Physiology Ontology 2.0.owl."



# classes: equal # iterations for R1 and R2	# classes: more iterations for R1 than for R2	# classes: less iterations for R1 than for R2
102	26	25

Figure 6: Iterations Comparison Table

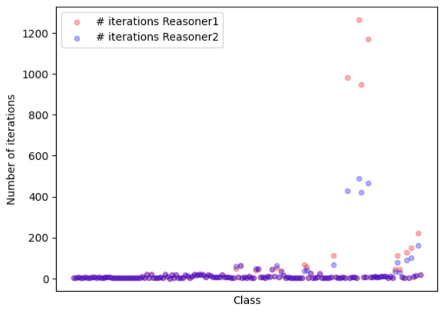


Figure 7: Iterations Comparison Graph

5 Conclusions

In conclusion, our research develops an intelligent system for a sushi restaurant, with an ontology representing diverse dishes, ingredients, and dietary specifications to facilitate personalized recommendations. Leveraging the $\mathcal{EL}$ reasoner algorithm, we optimize computational efficiency through consolidated rule applications and the introduction of a double-ended queue for element processing. Testing and comparing the original and optimized algorithms across various classes reveal that combining $\sqsubseteq$-rules and $\exists$-rules in a unified loop, along with a double-ended queue, can significantly enhance computational efficiency in specific ontologies, leading to reduced execution time and fewer iterations.

References

Horrocks, I. and Sattler, U. (2001). Ontology reasoning in the shoq (d) description logic. In *IJCAI*, volume 1, pages 199–204.

IJntema, W., Goossen, F., Frasincar, F., and Hogenboom, F. (2010). Ontology-based news recommendation. In *Proceedings of the 2010 EDBT/ICDT Workshops*, pages 1–6.

Kang, J. and Choi, J. (2011). An ontology-based recommendation system using long-term and short-term preferences. In *2011 International Conference on Information Science and Applications*, pages 1–8. IEEE.

Kazakov, Y., Krötzsch, M., and Simančík, F. (2012). ELK: a reasoner for OWL EL ontologies. System description, University of Oxford. available from `http://code.google.com/p/elk-reasoner/wiki/Publications`.

Lee, H. J. and Kim, H. S. (2015). ehealth recommendation service system using ontology and case-based reasoning. In *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 1108–1113.

Snae, C. and Bruckner, M. (2008). Foods: A food-oriented ontology-driven system. In *2008 2nd IEEE International Conference on Digital Ecosystems and Technologies*, pages 168–176.

Wang, X., Zhang, D., Gu, T., and Pung, H. (2004). Ontology based context modeling and reasoning using owl. In *IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second*, pages 18–22.