FACILITATING DL REASONERS THROUGH ONTOLOGY PARTITIONING

by

SADIQ CHARANIYA

(Under the Direction of John Miller)

ABSTRACT

Ontologies are used by domain experts for knowledge representation in an unambiguous manner. After the World Wide Web Consortium (W3C) has recommended Web Ontology Language (OWL) for representing ontologies on the web, a large number of domain ontologies have been developed. This represented knowledge in the form of OWL can not only be used by other people but also by software applications. Although representing knowledge in a structured form like OWL is important, reasoning with OWL ontologies holds the key to many applications. Several OWL reasoners (Racer, FaCT++, Pellet, etc.) have been developed to serve the purpose. However, reasoning with some ontologies can be computational so expensive that reasoners practically becomes unusable. The thesis discusses facilitating reasoners by finding the smallest construct of an ontology that causes reasoners to be unusable.

INDEX WORDS: Ontology, OWL, reasoners, description logic, tableau algorithm.

FACILITATING DL REASONERS THROUGH ONTOLOGY PARTITIONING

by

SADIQ CHARANIYA

B.E., Nagpur University, India, 2006

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

FACILITATING DL REASONERS THROUGH ONTOLOGY PARTITIONING

by

SADIQ CHARANIYA

Major Professor:	John Miller
Committee:	Krzysztof J Kochut William York

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2010

DEDICATION

To my parents and sister.

ACKNOWLEDGEMENTS

I would like to thank Dr. John Miller, my major advisor, for all the guidance and support provided to me in the course of this thesis for the two years that I spent at the LSDIS lab and at the same time I would like to thank Dr. Krys Kochut and Dr. William York for their valuable suggestions and time to my research work. I would also like to thank my colleagues who have always offered help whenever I needed it. Lastly but not the least I would like to thank my family and friends for giving me much needed morale boosts from time to time.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 Ontologies

Ontologies are used to represent knowledge of a domain in an unambiguous manner in such a way that both humans and machines can understand it and so it can be used for communication. An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms with some specification of their meaning, thereby it provides a good means of capturing terminological knowledge of a domain. According to [1] an ontology “is an explicit specification of a conceptualization", it not only explicitly defines concepts of a particular domain, but it also defines relationship between those concepts.

Ontologies can unambiguously define a domain knowledge which helps the domain experts to share knowledge of a domain in a convenient way. Ontologies not only facilitate sharing common understanding between people, but also make domain assumptions explicit. These domain assumptions can be used by other people to analyze a domain or they can even extend it to make it more specific to their application [2].

1.2 OWL

Ontologies can be represented in various forms, but after the World Wide Web Consortium (W3C) has recommended Web Ontology Language (OWL) for representing ontologies on Web, OWL has been the most widely language used for representing ontologies. The major advantage of using OWL is it can be easily integrated in Web applications (as it is written in XML) and more importantly, considerable work is being done to provide ontology editors which can be used to develop and maintain OWL ontologies, one of the most popular ontology editor is Protégé [3]. Apart from building good ontology editors, work is being done to build well documented Application Programmer Interfaces (APIs) which facilitate the use of these OWL ontologies in applications. Some popular APIs for OWL are Protégé-OWL API [4], the Jena API [5] and the Manchester-OWL API [6]. Today, OWL ontologies are being widely built and used by researches and domain experts to represent domain knowledge which can be used and reused by other people and machines for various purposes. OWL being machine interpretable plays an important role in interoperability between various applications. Search engines have also been built to find ontologies that have been uploaded on Web, SWOOGLE is one of the well known OWL search engines.

OWL provides three sublanguages to represent ontologies [2]

- OWL-Lite - It is the least expressive of OWL languages and is used if user's primary goal is to just define a concept hierarchy of a domain with simple constraint features.
- OWL-DL – As the name depicts OWL-DL is based on Description Logic (DL); it provides rich expressiveness with a guarantee of computational completeness (all entailments are guaranteed to be computed) of reasoning systems. Extensive research has

been done in description logics in the past few years and OWL-DL incorporates this research in its language.

- OWL-Full – It is the most expressive of OWL languages; users of this language will get maximum expressivity, while representing domain knowledge but the biggest drawback of this language it that it does not give any computational guaranties.

Out of the above three languages, OWL-DL is the most widely used, as it give high expressivity while still being decidable [9]. Hence, throughout this thesis we are going to work with OWL-DL ontologies.

1.3 Inference Services

In OWL-DL ontologies, knowledge of a domain is represented using classes, properties, instances and axioms. However, apart from knowledge representation, making inferences from these OWL-DL ontologies is also important. There is a suite of inference services (Consistency checking, Concept Satisfiability, Classification and Realization) provided by the description logic community [9], which hold a key to infering knowledge that can be semantically derived from explicitly represented knowledge. Due to the importance of inference with these ontologies various reasoners have been developed to serve the purpose. Some of the most widely used reasoners are PELLET [9], FACT++ [18] and RACERPRO [22]. However, reasoning with large and complex domain ontologies usually has less favorable computational complexity, because of the size and complexity of ontologies that tend to increase as the creator of ontology tries to make it more expressive. Although most modern reasoners use various optimization techniques [11], still with some ontologies reasoners takes too long to give results. Even though code for

these reasoners is provided to the users, it is often difficult to identify components of the ontology that are computationally expensive for the reasoner.

This thesis presents ways to facilitate the reasoning process via different approaches for finding ontology constructs that are the cause of making reasoners computationally expensive. We have developed a tool that takes a computationally expensive ontology and reports ontology constructs (concepts, properties and axioms) that cause the ontology to be computationally expensive.

The rest of this thesis is organized as follows: chapter 2 discusses the journal paper that we have written about the approaches we took; it also discusses the case studies, chapter 3 concludes the thesis and discusses future work.

CHAPTER 2

FINDING PERFORMANCE BOTTLENECKS FOR DL REASONERS THROUGH ONTOLOGY PARTITIONING [1]

[1] Sadiq Charaniya, John Miller, William York, Krys Kochut. To be submitted to IJSWIS.

2.1 Abstract

The Web Ontology Language (OWL) has become the most widely used representation format for ontologies. Although checking the consistency of OWL ontologies is an important step in the ontology development life cycle, reasoning with these ontologies holds the key to many applications. Several OWL Description Logic (DL) reasoners (Racer, FaCT++, Pellet, etc.) have been developed for reasoning with ontologies. However, because of the size and structural characteristics of some ontologies, the DL reasoners may exhibit unacceptably long runtimes in certain cases or may run out of memory. Our approach to deal with this problem is to use different methods to partition the ontology in a recursive manner until we find concepts, properties and axioms that cause reasoners to be slow and memory intensive.

2.2 Introduction

An Ontology formally represents domain knowledge in an unambiguous manner by explicitly defining concepts in a particular domain and relationship between these concepts. Ontologies are widely used to share domain knowledge not only between people, but also between software agents by representing it in a machine interpretable format and thereby encouraging automated processes. After the World Wide Web Consortium (W3C) recommended OWL for representing ontologies on the Semantic Web [19], a large number of ontologies have been developed in this format for representing knowledge in different domains. OWL comes in three "flavors": OWL-Lite, OWL-DL and OWL-Full [19]. For the purposes of this paper, we focus only on OWL-DL,

which provides a high-level of expressiveness without sacrificing computational completeness and decidability [3].

OWL-DL is based on SHOIN (D) [3] Description Logic (DL) and hence an OWL-DL ontology corresponds to a SHOIN (D) knowledge base. The key advantage of OWL-DL ontologies is that many DL reasoners have been developed with the help of which we can not only check whether there are some inconsistencies in an ontology, but also perform various inference services. Inconsistency in an ontology can occur for several reasons including modeling errors, migration from other formalisms, merging ontologies and ontology evolution [4]. However, it is important to check consistency of an ontology. According to [3] and the DL community there are several useful inference services (consistency checking, concept satisfiability, classification and realization) that are key to many applications or knowledge engineering efforts. There are many OWL-DL reasoners developed for this purpose that offer services to reason about the stored terminologies and assertions. Some examples of reasoners are Pellet [3], FaCT++ [12], RacerPro [16], KAON [20], and Hoolet [13]. Of these OWL-DL reasoners, Pellet, FaCT++, and Racer are based on tableau algorithms [3] and are most widely used for reasoning with OWL-DL ontologies.

Many domain ontologies are large and complex, as they attempt to express a comprehensive subset of the knowledge in a particular domain. However, reasoning with such ontologies may be unacceptably slow in certain cases with runtimes measured in days, even though typical performance is often acceptable (measured in seconds or minutes). One of the challenges faced by OWL-DL reasoners is scalability, because of the high computational complexity of reasoning in the worst case. The tableaux algorithm used by OWL-DL reasoners is known to be NexpTime-Complete [8]. To deal with this issue, modern reasoners (Pellet, Fact++,

Racer) use various optimization techniques [5] such as lazy unfolding, dependency direct back jumping, semantic branching, early blocking, strategies for transformation of axioms, General Concept Inclusion (GCIs), model caching, etc. Even though modern reasoners use these optimization techniques, still with some ontologies these reasoners have very long runtimes and consume a great deal of memory and may well run out of memory, in rest of the paper, we have termed such ontologies as "computationally expensive ontologies". As many of the users of these reasoners do not have knowledge about how these reasoners work, they either start randomly removing axioms/concepts from the ontology by blindly guessing and trying their luck to catch the axioms/concepts that degrade performance or they try to contact the reasoners implementers. Although reasoner implementers have general ideas about types of axioms that can induce computational problems, it is difficult for them to pinpoint a specific problem. In [5], researchers have built a partially automated tool for finding axioms that may lead to such computational problems.

Our approach to deal with this problem is to partition computationally expensive ontologies with different techniques to assist reasoners in finding the classes/properties/axioms that cause these ontologies to be computationally expensive. For partitioning, we have used an Incremental Approach and a Graph Partition Approach. Both the Incremental Approach and the Graph Partition Approach have their pros and cons; hence, we also implemented a Hybrid Approach that incorporates the advantages of the former two. Here, we would like to point out that we are not primarily interested in profiling the reasoners, but rather in developing a tool-supported methodology for making small adjustments to ontologies to improve the performance of the reasoners. Based on our approach, we have developed a tool that finds concepts/properties/axioms in ontologies that causes reasoners (Pellet, Fact++ and Racer) to be

computational expensive. From here after we will term these concepts/properties/axioms as "expensive concepts"/ "expensive properties"/ "expensive axioms" respectively.

The rest of the paper is organized as follows: In Section 2, we discuss related work and background on Description Logic (DL). Based on this overview of DL, in Section 3, we briefly look at tableau algorithms used in OWL-DL reasoners (e.g., Pellet, Fact++ and Racer). Our own approaches for ontology partitioning are discussed in Section 4. Section 5 discusses Fault Isolation (i.e., finding axioms that causes OWL-DL ontologies to be computationally expensive). In Section 6, we report on our preliminary experiments performed in our empirical evaluation. Finally, we discuss our conclusions and possible future work in Section 7.

2.3 Related Work

To the best of our knowledge, making an automated tool for finding computationally expensive concepts (classes in OWL)/properties/axioms, to date, has been unexplored. However in [5], researchers have built a tool which gives a detail view of performance statistics of a reasoner where their tool gave Sat. Time (Time for checking satisfiability of a class), #Clashes (numbers of clashes, i.e. contradictory statements, encountered during satisfiability check), Model Depth (depth of the completion graph), Model Size (size of the completion graph), etc. Also, partitioning ontologies into sub-ontologies to facilitating DL-reasoners has been done independently by many researchers. In [7], they have facilitated a DL-reasoner by partitioning ontology into smaller parts to find and rank inconsistencies in ontology. In [8], researches have partitioned the ontology to be able to apply distributed reasoning on it. In [4], the authors have

also divided the ontology into sub-ontologies to deal with different issues such as ontology maintenance, validation, publishing, and processing.

2.4 Background

A knowledge representation system stores information about the real world in a knowledge base. Description logics (DLs) are a family of knowledge representation languages which can be used to represent such information for an application domain in the form of a terminological knowledge [1]. In DLs, terminologies for a particular domain are represented in a structured way, such that both humans and applications can understand these terminologies without any ambiguity. In DLs, these terminologies are represented in the form of concepts.

A minimum propositionally closed DL is ALC which allows the specification of concepts in a particular domain, individuals that are instances of these concepts, and roles, which are interpreted as relationships between pairs of individuals. There are two types of restrictions on roles, value restriction (denoted by symbol $\forall$) and existential restriction (denoted by symbol $\exists$). Finally, there are axioms that make statement about how concepts and roles are related amongst themselves. Axioms are used to represent assertions or facts of a particular domain.

Concept can be constructed by concept expression:

$C \mid \top \mid \bot \mid \neg C \mid C \sqcup D \mid C \sqcap D \mid \exists R.C \mid \forall R.C$

where C is a concept name, C and D are concept expressions, and R is a role name. $\top$ and $\bot$ are top level and bottom level concept, respectively [1]. There are several class axioms and property axioms in SHION(D) description logic [9]. They can be represented using OWL the Abstract Syntax as follows.

Class Axioms:

- SubClassOf (C D), i.e., concept D is subsumed by C.
- EquivalentClass (D_1 …D_n), i.e., all concepts D_1 …D_n are equivalent to each other.
- DisjointClass (D_1 …D_n), i.e., all concepts D_1 …D_n are disjoint to each other.
- Class (C partial D_1 …D_n), i.e., a class expression (D_1 …D_n) is subsumed by C.
- Class (C complete D_1 …D_n), i.e., a class expression (D_1 …D_n) is equivalent to C.
- EnumeratedClass(O_1 …O_n), i.e., a class that contains exactly the enumerated individuals.

Property Axioms:

- SubPropertyOf(P_1 P_2), i.e., property P_2 is sub-property of P_1.
- EquivalentProperties (P_1 …P_n), i.e., all properties P_1 …P_n are equivalent to each other.
- ObjectProperties (domain (D_1 …D_n) range (D_1 …D_n)), i.e., both domain and range are concepts.
- DataTypeProperties (domain (D_1 …D_n) range (E_1 …E_n)), i.e., domain is concept but range is XSD data types.

DL concept expressions and axioms are together used to store information of a domain. In DLs, a distinction is drawn between the so-called TBox (terminological box) and the ABox (assertional box). In general, TBox uses the DL to formalise the terminological domain knowledge and Abox formalizes relationship between individual and concepts.

TBox Example:

Mother is a woman who should have a child who is a person.

DL representation : Mother ≡ Woman ⊓ ∃hasChild.Person

ABox Example:

Mary is a Mother.

DL representation : Mary : Mother

In the DL family 'S' is often used for ALC with transitive roles and additional letter indicates other extension to it.

'H'	for role hierarchy
'O'	for nominals/singleton classes
'I'	for inverse role
'N'	for number restriction
'D'	Data type restriction

Table 2.1 ALC extensions

The underlying Description Logic for OWL DL is SHOIN (D) [3]. For OWL 2 DL, it is SROIQ (D) [17].

2.5 OWL-DL Reasoners

In OWL-DL, knowledge of a particular domain is stored in terms of terminologies and assertions/facts/axioms and an OWL-DL reasoner is a piece of software that is able to infer/reason logical consequences from a set of asserted facts or axioms. Some of the reasoning tasks related to TBox are; classification, which checks whether a concept is more general than another, concept satisfiability, which checks whether a concept is meaningful, i.e., whether it can

have any individuals. Other reasoning tasks related to Abox are; realization where we check whether an individual is a member of a concept and consistency checking which checks if an instance satisfies constraints of its class.

All the reasoning tasks can be reduced to subsumption reasoning and subsumption can be reduced to concept satisfiability: $D \sqsubseteq C$ by showing $D \sqcap \neg C$ is unsatisfiable.

Most popular DL-reasoners (Pellet, Fact++ and Racer) use a tableau algorithm to check concept satisfiability. A Tableau algorithm is a decision procedure which is used to solve the problem of satisfiability. It has a set of transformational rules which are used to incrementally decompose the formula in a top down manner by applying these rules. It tries to apply all possible rules to decompose formula and formula is satisfiable if no more rule can be applied and no clash (contradiction in formulas) is detected in at least one of paths of that was taken during formula decomposition [2].

Using a tableau algorithm for satisfiablity, which has high worst-case complexity, may result in long run times for some complex ontologies. In the next two sections, we discuss how we have used different approaches for ontology partitioning to deal with this problem.

2.6 Ontology Partitioning

Whenever we find that reasoning on a particular ontology is computationally expensive, we partition the ontology into two parts. We have used two approaches to partition an ontology: an Incremental Approach, which we will discuss in a later section, and a Graph Partitioning Approach, which is done in three steps as explained below.

Step 1: Weighted Graph Construction

We first parse the ontology source files to create a weighted dependency graph. We have used the Manchester-OWL API [18] to read OWL files and output it in a graph format G (V, E) where 'V' is set of vertices and 'E' is set of edges in graph G. The concepts in OWL are represented as vertices and the relationship between them are represented as edges in the graph. The edges in the graph are weighted edges. The relationships between concepts form edges representing subclass relationships, equivalence relationship, object type properties (domain-range relationship) and disjoint relationships. We have set the default for subclass relationship edge to have a higher weight, but the user can also give different weights to different types of edges. The intuition behind giving subclass relationships higher weight is to avoid cutting these edges out while partitioning the ontology, because a subclass inherits properties from its superclass. However, if during partitioning, edges corresponding to subclass relationships are cut, then we add all the properties of the superclass to its subclass so that the integrity of that class, even after partitioning (i.e., without its superclass), is maintained. We also implemented the Heiner Stuckenschmidt and Michel Klein's [4] approach to determine the weights of dependencies between concepts (edge weights in graphs) in an ontology. They have used results from social networking theory by computing the proportional strength network for the dependency graph. However, this approach did not make much difference to our partitioning, so we kept it as an optional step.

Step 2: Partitioning the Weighted Graph

The second step is quite important in finding expensive axioms in the ontology, as it might be performed many times depending on the ontology. Hence the graph partitioning algorithm should be fast and balanced (equal size sub-graphs). If the cut is efficient, by efficient we mean that the cut size (sum of edge weights lost during partition) should is minimal, it is an added advantage, but our priority is to complete the partitioning step as soon as possible, hence in our software, quick partitioning comes before efficient partitioning.

We initially used JGAP [14], which is a Genetic Algorithms and Genetic Programming Java framework, for graph partitioning, but as the size of the graph increases, performance of partitioning through JGAP decreases. Hence, in order to make our software more scalable, we used METIS [15], which is a family of well known programs for partitioning unstructured graphs into k equal size parts and with reasonable efficiency. METIS is multilevel partitioning algorithms. Multilevel partitioning algorithms want to "reduce the size of the graph by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph" [15]. METIS was not only quick in partitioning the graph, but also gave smaller cut sizes as compared to the partitioning program developed by us using JGAP.

Step 3: Create Ontology from Graphs

Based on the two sub-graphs that we get after partitioning, the ontology is divided into two sub-ontologies. The division is accomplished by deleting concepts from the original ontology that correspond to the second sub-graph to obtain the first sub-ontology and then deleting concepts

from the original ontology that correspond to the first sub-graph to obtain the second sub-ontology. While creating two sub-ontologies, axioms/relationships between the concepts of two sub-ontologies are deleted; hence, the two sub-ontologies are not same as they were in the context of the original ontology, but they still may contain not the same axiom(s) that were causing the reasoners to be computationally expensive. For example, if an ontology contains 9 axioms, let their name be A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9 and let us assume that A_5 is deleted which causes the ontology to be partitioned into two parts, hence the first sub-ontologies (O_1) will contains axioms A_1, A_2, A_3, A_4 and the second sub-ontology (O_2) will contain A_6, A_7, A_8, A_9. Here three cases arise, where the expensive axiom might be present in O_1, O_2 or the third possibility that the axiom is deleted (e.g., axiom A_5). All these cases are addressed in next section.

2.7 Isolating Expensive Classes/ Expensive Properties/ Expensive Axioms

The process of isolating expensive classes/properties/axioms is done in two steps: In the first step we try to find the set of classes/properties/axioms that are suspect of being expensive classes/expensive properties/expensive axioms and from here on we will call these axioms as suspect classes/ suspect properties/ suspect axioms. In the second step we pinpoint the expensive classes/expensive properties/expensive axioms.

2.7.1 Graph Partition Approach

The first approach for finding suspect classes/ suspect properties/ suspect axioms is graph based partitioning. In the Graph Partition Approach, the following constructs are found on top-down manner.

1) Suspect Axioms

2) Expensive Axioms

3) Suspect Properties

4) Expensive Properties

5) Suspect Class

6) Expensive Class

In this approach, after dividing the ontology into two sub-ontologies through the process of graph partitioning as explained in the previous section, we check which of these two sub-ontologies are computationally expensive. There are in general two possibilities;

Case I- at least one of the two sub-ontologies contains expensive axiom(s) or

Case II - neither of them contains expensive axiom(s).

Case I occurs when the expensive axioms are carried on to either of the sub-ontologies. We can find this by checking the newly created two sub-ontologies and if any one of them is still computationally expensive that means expensive axioms have been carried forward to that sub-ontology. In this case, we further divide the sub-ontology that carries the expensive axioms recursively until we derive a set of sub-ontologies that do not contain expensive axioms, i.e., until Case II is achieved. This will occur only when we delete the expensive axiom, while ontology partitioning. Every time we divide the ontology, the axioms that were lost during division are recorded as suspect axioms.

Finally, when none of the sub-ontologies contains an expensive axiom, we stop further partitioning of ontology. At this point we have set of suspect axioms which contain expensive axioms. To have a better understanding of this, consider Figure 2.1 which is a graph where nodes represents concepts and edges represent relationships between concepts in an ontology.

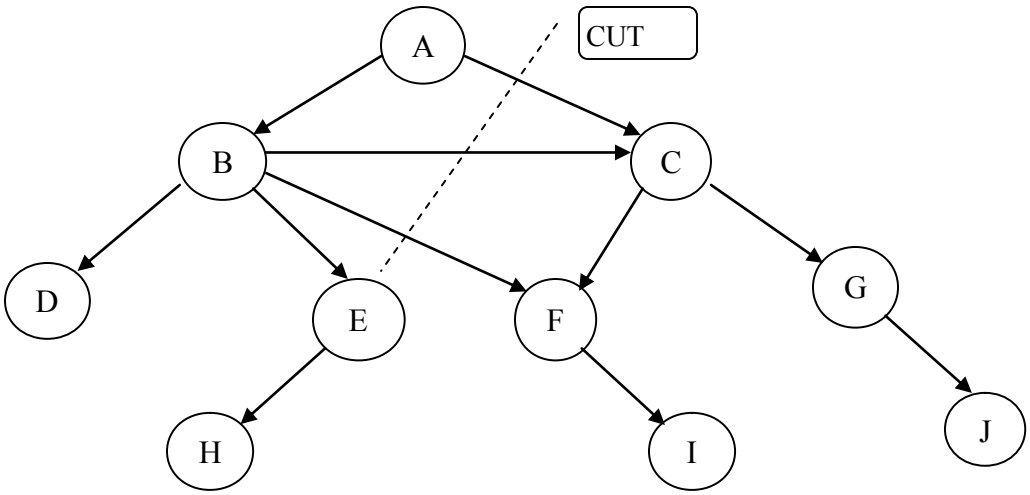

Figure 2.1: Partitioning through Graph Partition Approach.

Let us assume that after partitioning, the two sub-ontologies are O_1, containing concepts ABDEH, and O_2, containing concepts CFGIJ. In this case, during graph partitioning edges AC, BC and BF are lost. As these edges represent axioms, we record these axioms as suspect axioms, however, it might be possible that the axiom represented by edge BC is an expensive axiom and axioms represented by edge AC and BF are not.

After getting suspect axioms, we remove all the suspect axioms from the ontology and add one suspect axiom at a time. After adding each axiom, we check ontology for being computationally expensive, if the ontology is not computationally expensive than the added axiom is not expensive; hence, that axiom is kept added in the ontology and then we check other axioms from the suspect axioms list. Now, if the ontology becomes computationally expensive after adding a suspect axiom then we remove that axiom from the ontology, record it as an expensive axiom. We keep doing these steps until all suspect axioms are checked. When all axioms are checked, we get the list of expensive axioms. We have named this step as "Axiom Pin-Pointing".

A similar approach has been used to find expensive concepts from suspect concepts and expensive properties from suspect properties and we have named those approaches as "Concept Pin-Pointing" and "Property Pin-Pointing", respectively.

2.7.2 Incremental Partition Approach

The second approach for finding suspect axioms is the Incremental Approach where we first find concepts related to expensive axioms (we term it as expensive concepts) and then actual expensive axioms. In this approach, the following constructs are found on top-down manner.

1) Suspect Classes
2) Expensive Classes
3) Suspect Axioms
4) Expensive Axioms
5) Suspect Properties
6) Expensive Properties

In an Incremental partition, rather than using a graph partitioning algorithm to divide the ontology, we divide the ontology by deleting concepts and its sub-concept(s), recursively. We start concept deletion from the second-level concept(s) in the ontology hierarchy (subclasses of Thing) and traverse down in a breadth-first manner until we find an expensive concept at the lowest level, or a minimal set of expensive concepts. When a concept is deleted from an ontology, all of its asserted descendent concepts are also deleted. After deleting a concept and its descendents, we recheck the ontology to determine if it is computationally expensive, and if the answer is yes, then we can reasonably assume the deleted concept and its descendants do not contain expensive concepts. Hence, we add that concept and its descendants back to the ontology and mark that concept and its descendants as non expensive and move on to another concept in a breadth-first traversal order.

However, if we find that a concept is expensive, we do not further traverse its siblings. Instead, we continue the breadth-first traversal from its descendants. We continue moving down the hierarchy until we reach the lowest possible level for finding expensive concept(s) of the ontology. The expensive concept(s) reported by this approach is the lowest possible expensive concept(s), none of its children are expensive.

In order get a better understanding of the Incremental Approach, consider the ontology in Figure 2.2. First, we delete each of the second level concepts (i.e., subclass of T or Thing in Protege), one at a time, to find out which concept and its descendents contain expensive concepts. In Figure 2.2, let us assume that even after deleting concept K and its descendants, i.e., LM, the ontology is still computationally expensive. This means that concepts KLM are not expensive concepts and we mark them as non expensive and move on to the descendants of concept A in a breadth-first manner. Now, we delete concept B and its descendants DEH from the ontology and check whether the ontology is still computationally expensive.

Here, two cases arise:

- If the ontology is still computationally expensive, then sub-ontology BDEH does not contain expensive concepts, so now the sub-ontology of interest for finding expensive concepts will be CFGIJ.
- If the ontology is not computationally expensive, which implies that sub-ontology CFGIJ does not contain any expensive concepts, so we will mark CFGIJ as non expensive and will further examine, in a breadth-first manner, sub-ontology BDEH for expensive concepts.

We continue the above process until we find out that we have reached the lowest possible expensive concept in the Hierarchy that can be deleted.

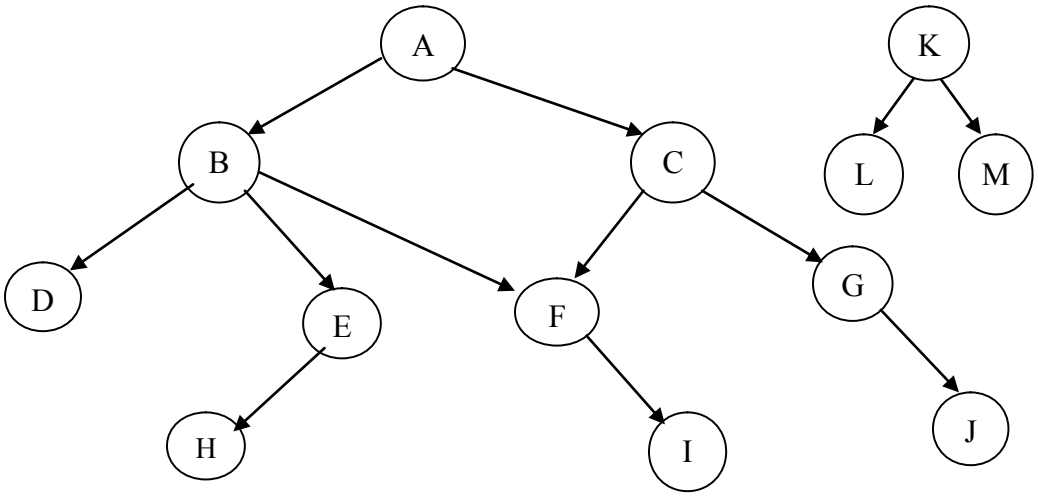

Figure 2.2: Partitioning through Incremental Approach.

When we reach the lowest possible expensive concepts, we record axioms related to that concepts as suspect axioms, as these axioms will contain both expensive axioms and other axioms related to those concepts. Now, we can use our axiom pin-pointing approach to find expensive axioms. From expensive axioms, we can get suspect properties and to get expensive properties from suspect properties we use property pin-pointing.

2.7.3 Graph Partition Approach Vs Incremental Partition Approach

The axiom(s) reported as expensive axioms cause the ontology to be computationally expensive. However, both the Graph Partition Approach and the Incremental Approach have their advantages and disadvantages.

The Graph Partition Approach is relatively fast in reporting expensive axioms. However, it might report axioms that are higher in the hierarchy. For example, in Figure 2.3 axioms related to edges HF and BF are expensive axioms and axioms related to edges DF and EF are inherited axioms from BF. Assuming these axioms are expensive only when they both are present in the

ontology, we can either delete the axiom related to HF or the axiom related to BF to make the ontology computationally inexpensive. However, our goal is to find an expensive axiom(s) which is at the lowest possible level in the hierarchy of ontology, so that deleting that axiom will have the least possible negative effect on the ontology. For example, by deleting axiom BF will also delete axioms DF and EF possibly causing a more negative effect on the ontology but by deleting axiom HF no other axioms are deleted as it is in the lowest possible level in the hierarchy of the ontology, thereby having less negative effects on to the ontology.

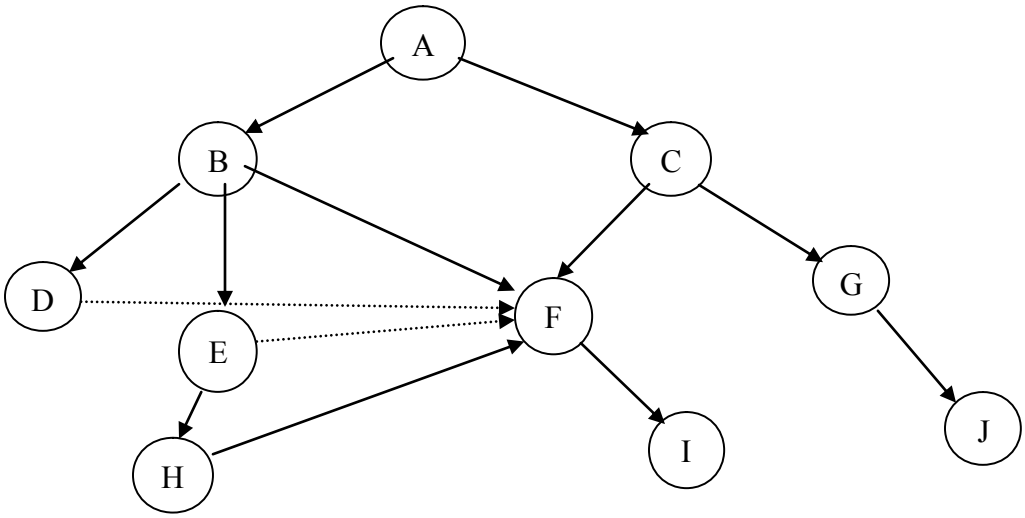

Figure 2.3: Inherited Relationships.

On the other hand, the Incremental Partitioning Approach always finds the lowest-level expensive axioms as it first finds these axioms from lowest-level expensive concepts, but this method is very slow, because the Incremental Approach performs partitioning and checking of the ontology many more number of times than the Graph Partition Approach. The only case in which the Incremental Approach will be as fast as the Graph Partition Approach is when the

ontology is in the form of a binary tree, which is a very rare case. In case of the ontology being in the form of a binary tree, there is always a guarantee that the partition is going to be balanced, however, in case of the Graph Partition Approach a balanced partition is always guaranteed.

2.7.4 Hybrid Approach

In order to take advantages from both the Graph Partition Approach and the Incremental Partition Approach, we implemented a Hybrid Approach. In the Hybrid Approach, we first run the Graph Partition Approach to get expensive axioms and from these axioms, we get the set of concepts used in these axioms. Now, instead of running the Incremental Approach on the entire ontology, we run it on only this set of concepts to get lowest level expensive axioms in the ontology. The Hybrid Approach takes the advantages of speed from the Graph Partition Approach and quality of Incremental Approach in that it finds the lowest level expensive axioms in the ontology. Hence, the Hybrid Approach can be quick and at the same time it finds the lowest level expensive axioms in the ontology.

The performance of the Hybrid Approach depends on the axioms found by the Graph Partition Approach. The best case for the Hybrid Approach is when Graph Partitioning reports axioms related to the lowest level concept in the ontology because in such cases the Incremental Approach will have to check for the least number of concepts and their sub-concepts. On the other hand, the worst case is when the axiom found by the Graph Partition Approach is related to high-level concepts, because in such cases the Incremental Approach will check the most number of concepts and their sub-concepts. The evaluation explains more about the behavior of the Hybrid Approach.

Based on our approaches, we were able to make a software tool that is able to find expensive classes / expensive properties / expensive axioms within computationally expensive ontologies. Evaluations and experimental results on an ontology based on our approach are shown in next section.

2.8 Evaluation

In the evaluation, we examine the Glycomics Ontology GlycO [6] which focuses on the glycoproteomics domain to model the structure and functions of glycans and glycoconjugates, the enzymes involved in their biosynthesis and modification, and the metabolic pathways in which they participate. GlycO is intended to provide both a schema and a sufficiently large knowledge base, which will allow classification of concepts commonly encountered in the field of glycobiology in order to facilitate automated reasoning and information analysis in this domain. Another ontology that we examine is Enzyme Ontology EnzyO [18] which has been populated with rich descriptions of enzyme structures and reactions, thus embodying a deep knowledge of the domain. The version of EnzyO that was analyzed has a total of over 8000 enzyme and reaction classes. The GlycO and EnzyO ontologies exploit the expressiveness of OWL-DL to describe its domain knowledge. However, when we tried to classify GlycO and EnzyO ontologies, the Pellet, Racer and FACT++ reasoners all ran out of memory and on increasing the memory size it took too long to classify.

2.8.1 Experiments with GlycO

The version of GlycO that was analyzed contains 347 concepts, 25 data type properties, 62 object type properties and 464 individuals (we are in the process of adding more); its DL expressivity is SHION(D). Even though GlycO is only of moderate size, reasoning on it was surprising slow and memory intensive.

In our first experiment with GlycO, we initially did not increase the memory size (80 MB) and allowed the reasoners to run out of memory and termed the ontology as computationally expensive when reasoners run out of memory. When GlycO was run with our tool it returned 2 expensive concepts, 1 expensive object property and 2 expensive axioms.

In the second experiment with GlycO, we increased the memory size (30 GB) to keep Pellet from running out of memory. It did not run out of memory, but it took 107 hours and 24 minutes to classify GlycO. When GlycO was run on our tool with increased memory and a time constraint of 15 seconds, it returned the same 2 expensive concepts, 1 expensive object property and 2 expensive axioms. Where a time constraint of 15 seconds means we allow the reasoner to classify ontology for 15 seconds and if the classification process is not finished within 15 seconds then we stop the classification process and report the ontology as computationally expensive.

We have given only 15 seconds to the classification process, because our tool will classify the ontology many times. It runs the classification process whenever the following changes are made to the ontology while running one of our approaches.

- Ontology is divided into two sub-ontologies.
- Addition/deletion of axiom(s) form ontology.
- Addition/deletion of concept(s) from ontology.
- Addition/deletion of property(s) from ontology.

After each of the above changes we run the classification process to check if it is still computationally expensive, hence the classification time is limited to allow the overall tool to complete in a reasonable amount of time. Results from our tool when run on GlycO are shown in Table 2.2.

Expensive Classes	1. N-glycan 2. O-glycan
Expensive Object Properties	1. has_carbohydrate_moiety
Expensive	1. EquivalentClasses(N-glyco_oligopeptide ObjectAllValuesFrom(has_carbohydrate_ moiety N-glycan)) 2. EquivalentClasses(O-glyco_oligopeptide ObjectAllValuesFrom(has_carbohydrate_ moiety O-glycan))

Table 2.2: GlycO Expensive Constructs

The performance of all the three approaches on GlycO with respect to the JVM default of 80MB of memory and a time constraint of 15 seconds with 1.5 GB of memory is given in Table 2.3

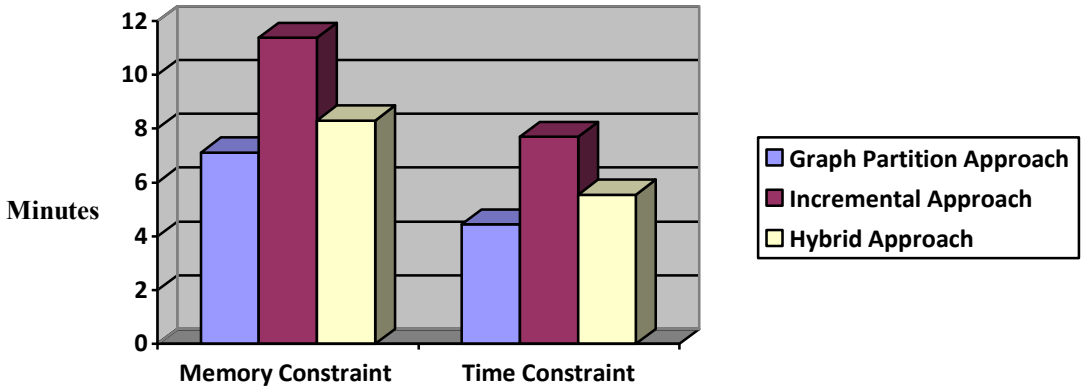

Table 2.3: Performance of three different Approaches with GlycO

After deleting the two expensive axioms or the two expensive concepts or the one expensive property reported by our tool, the GlycO ontology did not run out of memory even without increasing memory size (80 MB) and classification was completed in approximately 4 seconds.

2.8.2 Experiments with EnzyO

The version of EnzyO that was analyzed contains 8223 concepts, 28 data type properties and 2 object type properties; its DL expressivity is ALUN (D), where AL is attribute language, U is concept union, N if cardinality restriction and D is use of data type properties. Although reasoning with EnzyO was not memory intensive, it was slow.

As EnzyO was not running out of memory, we used the version of our tool that check for slow classification, i.e., using a time constraint. When EnzyO was run on our tool with a time constraint of 60 seconds (reason for increased time constraint is the size of EnzyO which is much larger than GlycO), it gave different partial results for different approaches. The Graph Partition Approach was not able to report expensive axioms/ expensive concepts/expensive properties; this case generally arises when expensive axioms are inferred axioms. As our tool records only asserted axioms, inferred expensive axioms cannot be detected. However, even in the case of EnzyO where expensive axioms are inferred axioms, our tool was able to detect a sub-ontology (513 concepts) that was computationally expensive. The Incremental Approach was again not able to detect expensive axioms, but as it starts by finding expensive concepts first, it was able to detect 3 concepts (+ all their sub-concepts) that were expensive and when we analyzed properties related to the reported 3 concepts we were able to find 1 expensive property. We were not able to run the Hybrid Approach as it requires Graph Partition Approach to report expensive axiom(s) first, which was not possible in the case of EnzyO. The timing results of the two approaches that were able to run on EnzyO are shown in Table 2.4 and Table 2.5 shows expensive concepts and expensive properties found out by our tool using the Incremental Approach.

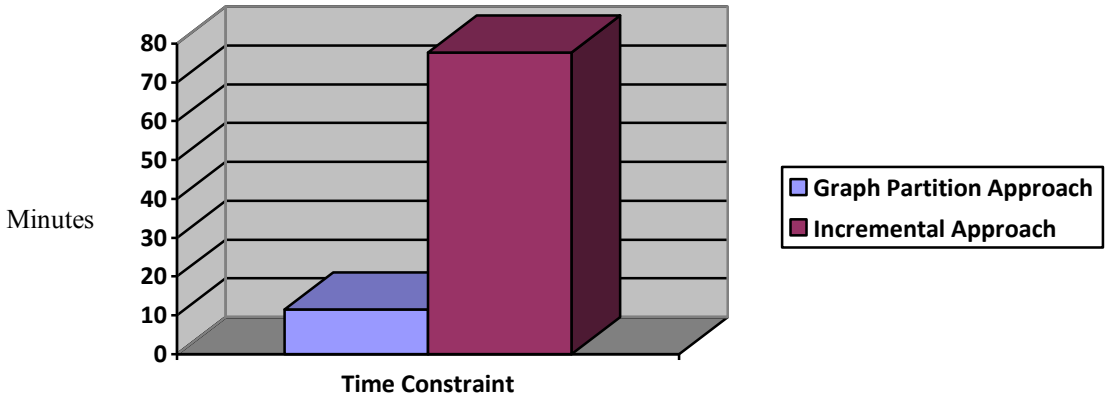

Table 2.4: Performance of two different Approaches with EnzyO

Expensive Classes	1. reaction_ec_1 2. reaction_ec_2 3. reaction_ec_3
Expensive Object Properties	1. catalyzes_reaction

Table 2.5: EnzyO Expensive Constructs

Finally, Table 2.6 shows a summary of different expensive constructs that our tool was able to find with different approaches on both GlycO and EnzyO. The G.P.A stands for Graph Partition Approach, I.A stands for Incremental Approach, H.A stands for Hybrid Approach. Runtime (TC, min) means, total time (in minutes) taken by our tool when run with time constraint on the classification process and Runtime (MC, min) means, total time (in minutes) taken by our tool when run with memory constraint on the classification process. With GlycO ontology our tool was able to report all types of expensive constructs, it was able to find expensive sub-ontology, expensive concepts, expensive properties and as we can see from the time taken by each approach, that the Graph Partition Approach is the fastest of all, which is followed by the Hybrid Approach and as expected the Incremental Approach was the slowest. The 2 expensive axioms of GlycO that were reported were asserted axioms and after deleting those axioms GlycO took less than 80 MB of memory and less than 2 seconds to complete the classification process. However, our tool did not give complete results for EnzyO, the Graph Partition Approach was able to find sub-ontology but could not find more than that because it was not able to find expensive axioms as they were inferred axioms. The Iterative approach was able to find concept that were expensive, but again we were not able to find expensive axioms.

When we deleted all the asserted axioms related to expensive concepts, the ontology was still expensive, which means that some axioms were being inferred by these concepts that we were not able to delete. When we analyzed properties related to those concepts we were able to find 1 expensive property. As Graph Partition Approach did not reported any expensive concepts we could not run the Hybrid Approach.

Expensive Components	**GlycO**			**EnzyO**		
	G.P.A	I.A	H.A	G.P.A	I.A	H.A
Sub-Ontology (concepts #)	43	5	5	513	4022	X
Concepts	2	2	2	X	2	X
Properties	1	1	1	X	1	X
Axioms	2	2	2	X	X	X
Runtime (TC, min)	4.44	7.7	5.55	11.52	77.7	X
Runtime (MC, min)	7.12	11.38	8.30	X	X	X

Table 2.6: Summary of Expensive Construct for GlycO and EnzyO.

2.9 Conclusions and Future Work

We were able to develop three approaches to find expensive constructs (classes/properties/axioms) from a computationally expensive ontology, also based on our approach we were able to build a tool which implements all our approaches to find expensive constructs from an ontology. As we have seen through our case study of the GlycO ontology, the

classes/properties/axioms found by our program were causing ontologies to be computationally expensive, however, we are not stating that these classes/properties/axioms are the defects in the ontology, we are stating that if they are included in the ontology, then reasoning with the ontology may be computationally expensive depending on the queries.. Even if the user of the ontology does not want to delete or modify these classes/properties/axioms, it is useful to know which constructs of the ontology will make the reasoning computationally expensive.

One limitation is expensive axioms cannot be reported if the expensive axioms are inferred axioms, because to find these inferred axioms we need to run reasoners which take us back to our problem of slow and memory intensive reasoning. Our experiment with EnzyO shows that even in the case when the expensive axioms are inferred axioms our tool is able to give partial expensive constructs of ontology.

As of today, our approach finds the expensive ontology constructs (classes/properties/axioms), hence deleting these axioms is the only way to speed up reasoning with these ontologies, but deleting these axioms will change the ontology as a whole. In the future, it would be interesting to see how we can just modify the axioms rather than completely deleting them. We would also like to explore a more efficient and faster way to partition the ontology, as the speed of our process depends on how quickly the ontology partition is done.

2.10 References

1. Baader F, & Sattler U. (2001). An Overview of Tableau Algorithms for Description Logics. Studia Logica 69(1): 5-40, Springer Netherlands.

2. Baader, F., & Nutt, W., (2003). Basic description logics, in: Baader F., Calvanese D., McGuinness , Nardi D., & Patel-Schneider P.F. (Eds.). The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, pp. 43–95.
3. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A. , & Katz, Y. (2004). Pellet: A practical OWL-DL reasoner, from http://www.mindswap.org/papers/PelletJWS.pdf
4. Stuckenschmidt, H., & Klei, M. Structure-Based Partitioning of Large Concept Hierarchies. ISWC 2004. In Lecture Notes in Computer Science 3298, pp. 289–303, Springer, Heidelberg (2004)
5. Wang, T., & Parsia, B. (2007). Ontology Performance Profiling and Model Examination: First Steps. The Semantic Web. In Lecture Notes in Computer Science, pp. 595-608, Springer Berlin, Heidelberg.
6. Thomas, C., J., Sheth, A., & W. S. York, W. (2006). Modular Ontology Design Using Canonical Building Blocks in the Biochemistry Domain. In Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS). Fairfax, Virginia: IOS Press.
7. Lam, S., Pan, J., Sleeman, D., & Vasconcelos, W. (2006). Ontology Inconsistency Handling: Ranking and Rewriting Axioms. Technical report aucs/tr0603, University of Aberdeen.
8. Leon, A., & Dumontier A. A platform for distributing and reasoning with OWL-EL knowledge bases in a Peer-to-Peer environment. Carleton University, Ottawa, Canada. Retrieved from http://www.webont.org/owled/2009/papers/owled2009_submission_34.pdf
9. Horrocks, I, & Patel-Schneider, P. (2004). Reducing OWL entailment to description logic satisfiability, J. of Web Semantics 1 (4) 345–357.

10. Hai W., Matthew H., Alan R., Nick D., & Julian S. (2005). Debugging OWL-DL Ontologies: A Heuristic Approach. ISWC 2005, In Lecture Notes in Computer Science, 3729, pp. 745–757, Springer, Heidelberg.
11. Parsia, B., Sirin, E., Kalyanpur, A. (2005). Debugging OWL Ontologies, International World Wide Web Conference, 633 – 640, ACM New York, NY
12. Tsarkov, D., & Horrocks, I. FaCT++ description logic reasoner: System description. In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006), volume 4130 of Lecture Notes in Artificial Intelligence, pages 292–297, 2006
13. Hoolet: http://owl.man.ac.uk/hoolet/
14. Meffert, K., Rotstan,N., Knowles, C., Sangiorgi, U. JGAP –Java Genetic Algorithms and Genetic Programming Package, retrieve from http://jgap.sourceforge.net/
15. Karypis, G. & Kumar, V. (1998). MeTiS { A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0, University of Minnesota.
16. V. Haarslev, V., & R. Moller R. (2001). Racer system description. In International Joint Conference on Automated Reasoning, IJCAR, Siena, Italy.
17. OWL 2 Web Ontology Language Document Overview. [Online]. Available: http://www.w3.org/TR/owl2-overview/
18. EnzyO. [Online]. Available: http://lsdis.cs.uga.edu/projects/glycomics/2006/EnzyO.owl
19. OWL Web Ontology Language Guide. [Online]. Available: http://www.w3.org/TR/owl-guide/
20. Motik, B., & Studer, R. (2005) KAON2 – A Scalable Reasoning Tool for the Semantic Web. In: Proceedings of the 2nd European Semantic Web Conference, Heraklion, Greece.

CHAPTER 3

SUMMARY

In this thesis, we addressed the problem of users being clueless while reasoning with expensive ontologies. We have facilitated reasoners by reporting back the expensive constructs of an ontology to the users of the ontology. On the basis of our approach, we have built a program which takes in computationally expensive ontology and reports back its expensive constructs. However, we again want to state that we are not claiming that the expensive constructs reported are faults or defects in the ontology, although we are stating that if those expensive constructs of an ontology are included in the ontology, then reasoning with the ontology is going to be extremely slow and memory intensive. It is always helpful for the user to know which constructs in their ontology will reduce the performance of reasoning.

We have also done an evaluation on the ontologies built in our lab, i.e. the GlycO and EnzyO ontology.

In the future, we can also keep working on optimizing our tool by trying different ways of graph partitioning as our tool largely depends on how efficient and quick graph partition is, more work can also be done in building more a user-friendly interface.

REFERENCES

1. Gruber, & Thomas, R. (1993). A translation approach to portable ontology specifications. In: Knowledge Acquisition. 5: 199-199.
2. OWL Web Ontology Language Guide. [Online]. Available: http://www.w3.org/TR/owl-guide/
3. Noy, N.F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R.W. & Musen, M.A. (2001). Creating semantic web contents with protege-2000. In IEEE Intelligent Systems. 16(2). Pages 60-71
4. Knublauch, H., Fergerson, R. W., Noy, N. F., & Musen, M. A. (2004).The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In Lecture Notes in Computer Science. Published by Springer. Pages 229-243.
5. McBride, B., (2001). Jena: Implementing the RDF model and syntax specification, in: Semantic Web Workshop, WWW.
6. Bechhofer, Sean, Volz, R. & Lord, P. (2003). Cooking the Semantic Web with the OWL API. Published by Springer Berlin.
7. Baader F, & Sattler U. (2001). An Overview of Tableau Algorithms for Description Logics. Studia Logica 69(1): 5-40, Springer Netherlands.
8. Baader, F., & Nutt, W., (2003). Basic description logics, in: Baader F., Calvanese D., McGuinness , Nardi D., & Patel-Schneider P.F. (Eds.). The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, pp. 43–95.

9. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A. , & Katz, Y. (2004). Pellet: A practical OWL-DL reasoner, from http://www.mindswap.org/papers/PelletJWS.pdf
10. Thomas, C., J., Sheth, A., & W. S. York, W. (2006). Modular Ontology Design Using Canonical Building Blocks in the Biochemistry Domain. In Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS). Fairfax, Virginia: IOS Press.
11. Wang, T., & Parsia, B. (2007). Ontology Performance Profiling and Model Examination: First Steps. The Semantic Web. In Lecture Notes in Computer Science, pp. 595-608, Springer Berlin, Heidelberg.
12. Stuckenschmidt, H., & Klei, M. Structure-Based Partitioning of Large Concept Hierarchies. ISWC 2004. In Lecture Notes in Computer Science 3298, pp. 289–303, Springer, Heidelberg (2004)
13. Lam, S., Pan, J., Sleeman, D., & Vasconcelos, W. (2006). Ontology Inconsistency Handling: Ranking and Rewriting Axioms. Technical report aucs/tr0603, University of Aberdeen.
14. Leon, A., &Dumontier A. A platform for distributing and reasoning with OWL-EL knowledge bases in a Peer-to-Peer environment. Carleton University, Ottawa, Canada. Retrieved from http://www.webont.org/owled/2009/papers/owled2009_submission_34.pdf
15. Horrocks, I, & Patel-Schneider, P. (2004). Reducing OWL entailment to description logic satisfiability, J. of Web Semantics 1 (4) 345–357.
16. Hai W., Matthew H., Alan R., Nick D., & Julian S. (2005). Debugging OWL-DL Ontologies: A Heuristic Approach. ISWC 2005, In Lecture Notes in Computer Science, 3729, pp. 745–757, Springer, Heidelberg.

17. Parsia, B., Sirin, E., Kalyanpur, A. (2005). Debugging OWL Ontologies, International World Wide Web Conference, 633 – 640, ACM New York, NY
18. Tsarkov, D., & Horrocks, I. FaCT++ description logic reasoner: System description. In Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006), volume 4130 of Lecture Notes in Artificial Intelligence, pages 292–297, 2006
19. Hoolet: http://owl.man.ac.uk/hoolet/
20. Meffert, K., Rotstan,N., Knowles, C., Sangiorgi, U. JGAP –Java Genetic Algorithms and Genetic Programming Package, retrieve from http://jgap.sourceforge.net/
21. Karypis, G. & Kumar, V. (1998). MeTiS { A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0, University of Minnesota.
22. V. Haarslev, V., & R. Moller R. (2001). Racer system description. In International Joint Conference on Automated Reasoning, IJCAR, Siena, Italy.
23. OWL 2 Web Ontology Language Document Overview. [Online]. Available: http://www.w3.org/TR/owl2-overview/
24. EnzyO. [Online]. Available: http://lsdis.cs.uga.edu/projects/glycomics/2006/EnzyO.owl

APPENDIX A

Tool User's Guide

Installation Instructions

1. Install Java (Java 1.6 or Java 1.5): http://java.com

2. For ontologies that are memory intensive, we have a tool called OCT_Memory and for ontologies that are not memory intensive but take too long to classify, we have a tool called OCT_Time. Both the tools are available in oct.zip file: (current location http://cs.uga.edu/~jam/glyco/oct/oct.zip).

3. Set the path (path of project folder) in the AllPath.property file in the project OCT_Memory / OCT_Time.

Execution and Usage Instructions

For project OCT_Memory

1. Download otc.zip and unzip it.
2. The readme.txt contains instructions to run the tool. Click on one of the three approaches to find expensive classes, expensive properties and expensive axioms (shown in Figure A.1).
3. Output will be the expensive classes, expensive properties and expensive axioms (shown in Figure A.2

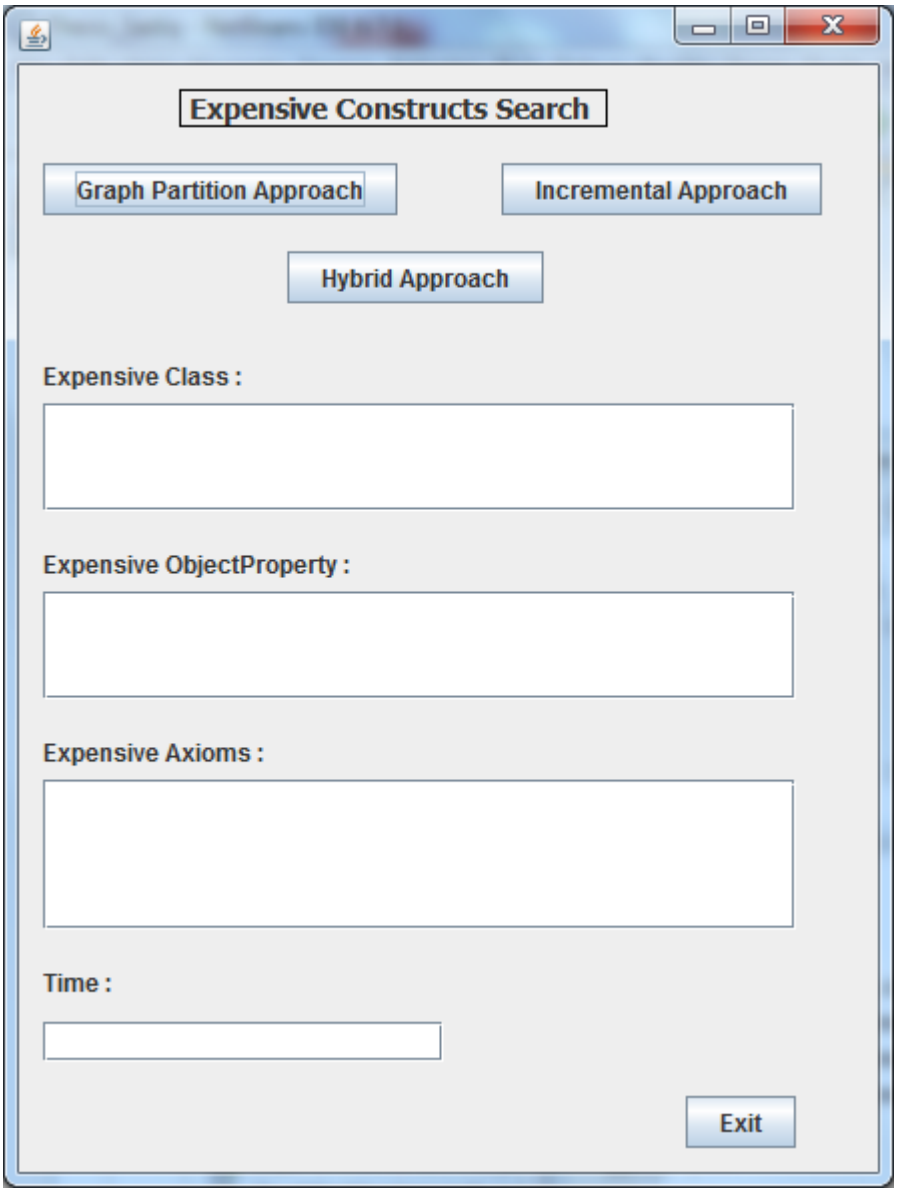

Figure A.1: Initial User Interface for project: OCT_Memory

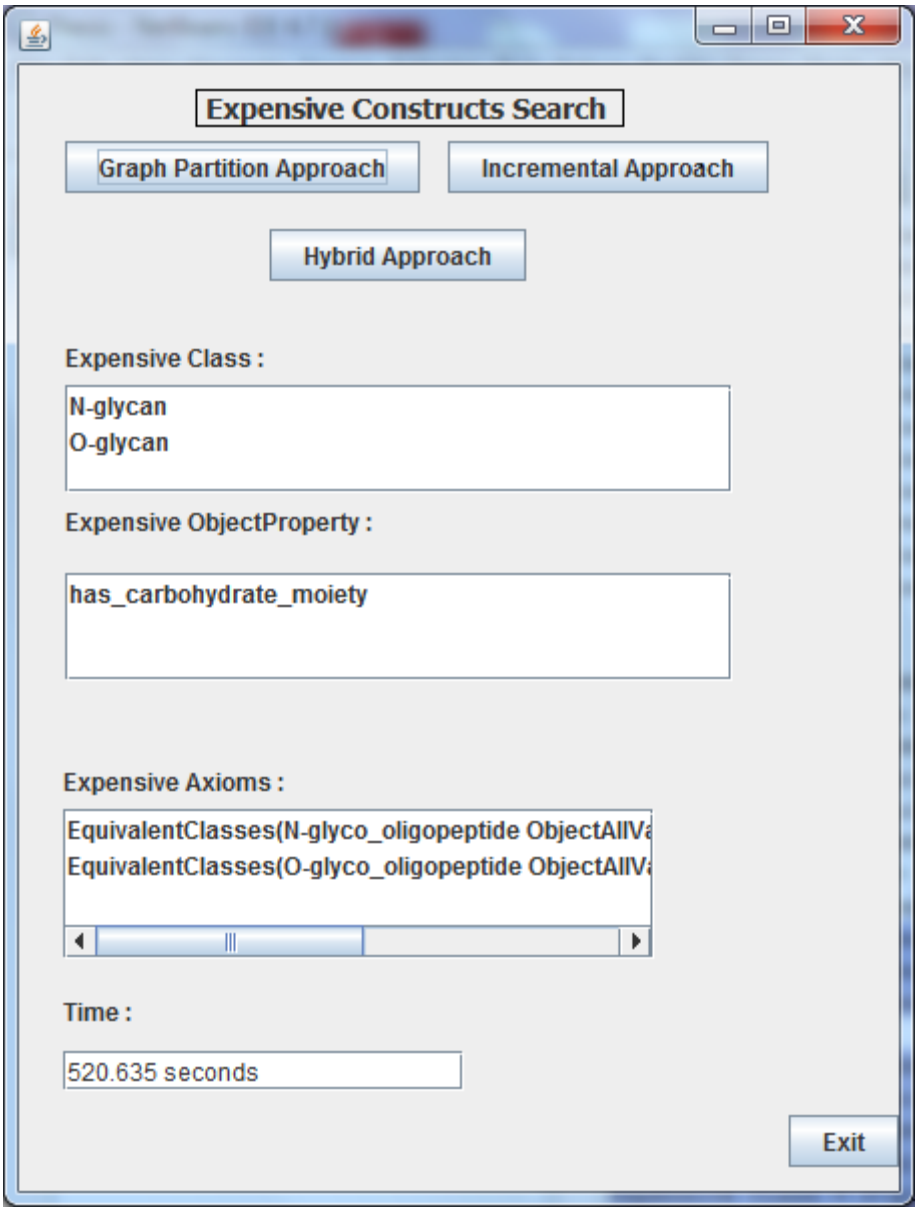

Figure A.2: Output User Interface for project: OCT_Memory

For project OCT_Time

1. Download otc.zip and unzip it.
2. The readme.txt contains instructions to run the tool. Enter time (classification time out) and click on one of the three approaches to find expensive classes, expensive properties and expensive axioms (shown in Figure A.3).
3. Output will be the expensive classes, expensive properties and expensive axioms (shown in Figure A.4).

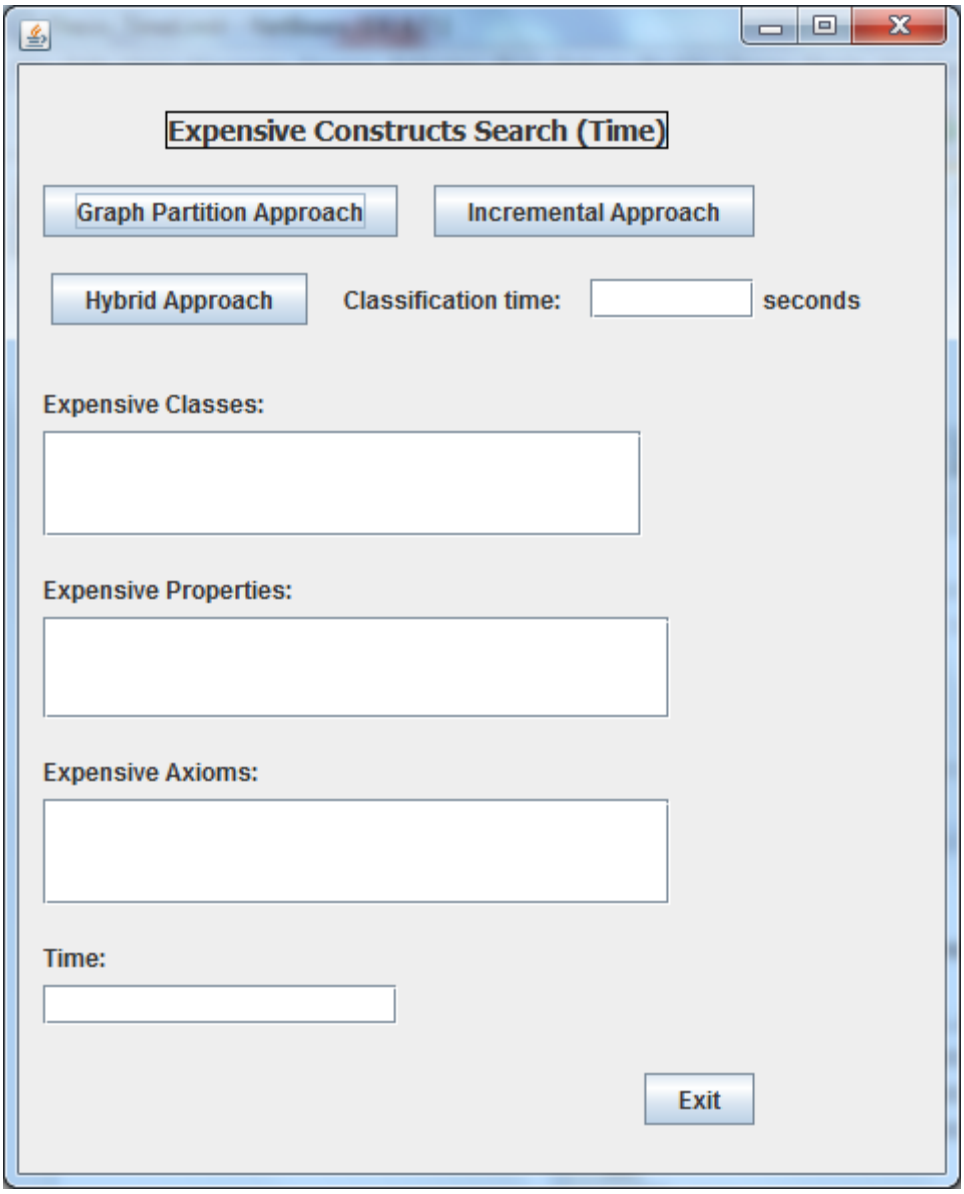

Figure A.3: Initial User Interface for project: OCT_Time

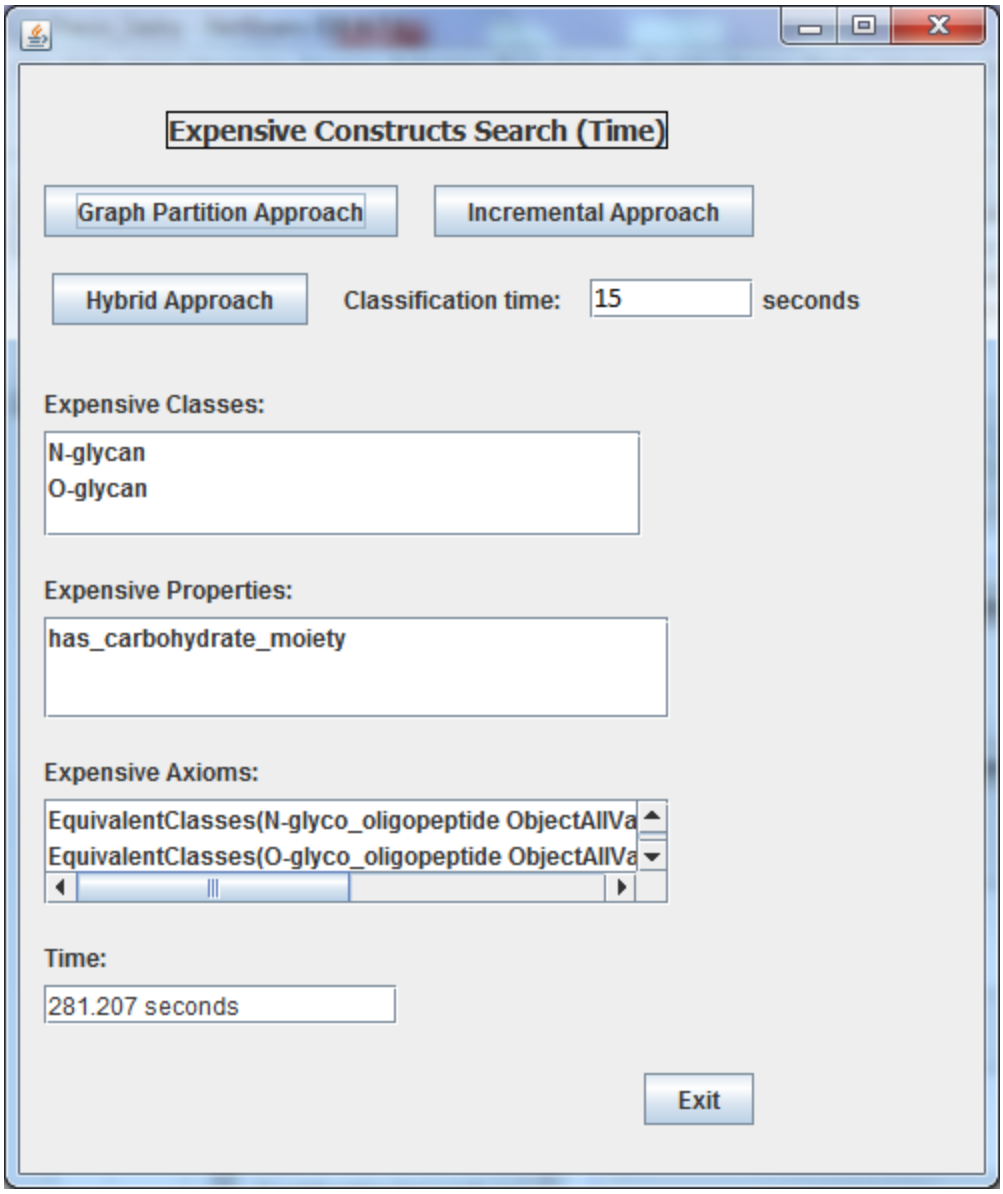

Figure A.4: Output User Interface for project: OCT_Time