Pellint — A Performance Lint Tool for Pellet

Harris T. Lin, Evren Sirin

Clark & Parsia LLC, Washington, DC, USA
{hlin,evren}@clarkparsia.com

Abstract. Predicting the performance of a tableau reasoner for an OWL ontology is generally hard. It is even harder for users who are not familiar with the details of tableau algorithms. In this paper we present Pellint, a lint tool that reports and potentially repairs modeling constructs that are known to have bad performance characteristics for Pellet. We describe the collection of modeling constructs that are currently detected by Pellint and explain why these patterns typically cause performance problems. Finally we note that our implementation allows easy extension of patterns to be detected, and we encourage feedback and contribution of more problematic modeling constructs from the community.

1 Introduction

OWL reasoners are most useful when they have predictable performance, and predicting performance for an ontology is not an easy task [3, Chap. 3, 9], especially because the data are not usually correlated with their reasoning performance. There are many sources of complexities that are known to have an impact on the reasoning performance, however most of them are hardly visible to OWL ontology engineers due to complex interactions between various modeling constructs. Inspired by traditional lint tools [6, 4], we have developed Pellint, a lint tool that reports and potentially repairs modeling constructs that are known to have bad performance characteristics for Pellet, whose reasoning is based on tableau algorithms. The results Pellint produces would mostly be applicable to other tableau reasoners though not completely because the optimizations implemented in reasoners differ.

The goal of this tool is to help ontology engineers pinpoint, and potentially remove, the source of reasoning complexities in their ontologies. Unlike traditional lint tools, however, the problems found by Pellint do not necessarily indicate a modeling error in the ontology. It just means that Pellet is probably going to be relatively slow when reasoning with that ontology. Therefore, there might not be an appropriate solution for some of the reported problems, i.e. the only options available might be removing some of the modeling constructs.

Section 2 explains the reasoning complexities in more detail. Section 3 lists some of the nontrivial modeling constructs (patterns) detected by Pellint. Section 4 gives a functional description of Pellint.

2 Reasoning Complexities

In this section, we provide a very high-level description of the tableau algorithm for DLs and explain the main sources of its reasoning complexities. We refer the reader to [3] for a more detailed and accurate description of the tableau algorithm.

In the tableau algorithm all the reasoning services can be reduced to consistency checking, which is done by building a completion graph. The nodes in the completion graph intuitively stand for individuals and literals. Each node is associated with its corresponding types. Property-value assertions are represented as directed edges between nodes. If we are checking the consistency of an ontology the initial completion graph is built from the asserted facts in the ontology. If we are checking the satisfiability of a class the initial graph contains a single node whose type is that concept. The reasoner repeatedly applies the tableau expansion rules until a clash (i.e. a contradiction) is detected in the label of a node, or until a clash-free graph is found to which no more rules are applicable.

In the process of tableau completion, there are two main sources of complexities: (1) nondeterminism in "completing" the graph; and (2) the size of the graph built.

2.1 Nondeterminism

Building the completion graph is nondeterministic due to disjunctions which are expressed with the `UnionOf` construct in OWL. The instances of the union class must be an instance of at least one of the union elements. The reasoner will do a case-by-case analysis to figure out if that is possible. A nondeterministic choice made because of a union class will have an impact on the completion graph, e.g. new edges may be added because of that choice. In the event that we made a wrong choice (which will be indicated by an inconsistency in the later stages of completion process) we have to backtrack and undo all the changes made to the graph. This effect multiplies if we have many choices to make.

There are many optimization algorithms implemented in Pellet to reduce the effects of nondeterminism, but it is not hard to see how a very large number of union classes will adversely affect reasoning time.

2.2 Completion graph size

The size of the completion graph depends on the size of the initial graph (i.e., the asserted instances), but also on the use of existential restrictions. Constructs like `SomeValuesFrom`, `MinCardinality`, and `ExactCardinality` will cause the tableau algorithm to create new nodes in the completion graph. Applying the tableau completion rules to new nodes will require more processing time and possibly increase the nondeterminism involved because there might be new nondeterministic choices made for these new nodes.

Predicting the exact size of the completion graph (without actually building the graph) is not possible, but Pellint uses some heuristics and graph analysis techniques to compute an approximation of this size.

3 Patterns Detected

In this section we describe some nontrivial patterns Pellint currently detects, and we explain why they may impact on the reasoning performance. For a complete list please see the Pellint distribution [2].

– **Explicit GCI** A subclass axiom with a complex concept expression on the left hand side is called a General Concept Inclusion (GCI) axiom. A tableau-based reasoner deals with GCI axioms by converting them into a standard form. For example, `SubClassOf(C D)` is converted into the axiom `SubClassOf(owl:Thing UnionOf(ComplementOf(C) D))` where `C` and `D` can be arbitrary concepts. Since every individual is an instance of `owl:Thing`, the reasoner then applies the converted axiom to every individual. We observe that every convertion produces a nondeterministic choice due to the `UnionOf`, which is then applied to every individual. Hence GCI axioms are extremely expensive and reported by Pellint.

– **Implicit GCI** Most of the time ontologies do not have explicit GCI axioms. However, there are cases where the interaction between different axioms create implicit GCIs. For example, if a concept is equivalent to a class expression and is also a subclass of another expression, then it implicitly defines a subclass axiom whose left hand side is complex. There are optimization techniques, e.g. absorption [3, Chap. 9], that minimizes the effect of such GCIs. However, these techniques still introduce additional nondeterminism that slows down the overall reasoning process.

– **Existential Explosion** An existential restriction, e.g. `SomeValuesFrom` or `MinCardinality` construct, causes the tableau reasoner to create new nodes in the completion graph. Many existential restrictions may interact in a complex manner and may generate an intractable number of nodes in the completion graph. Pellint estimates the total number of individuals generated by such restrictions and reports as a lint if it exceeds a configurable number. Pellint currently does not offer any repairs for this pattern.

– **Equivalence to AllValuesFrom** An `AllValuesFrom` restriction does not require the instance to have a value for the referred property but only restricts the values for existing property values. This means any concept not

Pattern Name	Example	Repair
Explicit GCI	`SubClassOf(IntersectionOf(A B) C)`	None
Implicit GCI	`EquivalentClasses(C IntersectionOf(A B))`, `SubClassOf(C SomeValuesFrom(R D))`	Change all to subclass axioms
Existential Explosion	Interconnected usages of `SomeValuesFrom`, `MinCardinality`, and/or `ExactCardinality`	None
Equivalence to AllValuesFrom	`EquivalentClasses(C AllValuesFrom(R D))`	Change to a subclass axiom

Table 1. Detected Patterns

having the property value, e.g. a concept that is disjoint with the domain of the property, will satisfy the `AllValuesFrom` restriction and turn out to be a subclass of the concept defined to be equivalent to the restriction (class `C` in the example from Table 1). This typically leads to unintended inferences and additional inferences may eventually slow down the reasoning performance.

4   Tool Description

The patterns described above along with some additional patterns have been implemented in Pellint. Pellint is available as an open source command line tool and can be downloaded at [2]. Pellint receives an ontology as its input, and produces a lint report summarizing all the detected lint patterns in plain text. It may also save the repaired ontology to a file, where it applies the available recommended repairs to the detected lints.

In addition, Pellint also performs syntactic checks to detect cases such as a resource being used without a type declaration. Such untyped resources are legal in OWL Full but not allowed in OWL DL. Some tools automatically detect and "patch" such syntax issues using a set of heuristics. However, these kinds of issues might indicate an error in the ontology, e.g. the most common case is a spelling error in the URI of the resource.

Pellint is designed for easy extensions of new patterns based on the OWL API library [5]. The package includes Javadoc documentation and a step-by-step guide on how to write new patterns and integrate them into Pellint. The goal is to encourage users to experiment with new patterns with their own ontologies and contribute them to the community.

5   Conclusion and Future Work

We have developed Pellint, a lint tool for ontologies that reports and potentially repairs modeling constructs that are known to have bad performance characteristics for Pellet. Pellint is designed for easy extensions of new patterns and we encourage users to contribute in the future.

As part of our future work we are considering a more interactive lint detection tool, e.g. integration with the Lint Roll Protégé plug-in [1].

References

[1] Lint Roll Protégé plug-in. `http://www.cs.man.ac.uk/~iannonel/lintRoll`.
[2] Pellint. `http://pellet.owldl.com/pellint`.
[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2007.
[4] I. F. Darwin. *Checking C Programs with Lint*. O'Reilly Media, 1988.
[5] M. Horridge and S. Bechhofer. Igniting the OWL 1.1 touch paper: The OWL API. In *In Proc. OWL-ED 2007, volume 258 of CEUR*, 2007.
[6] S. Johnson. Lint, a C program checker. Technical report, Bell Laboratories, 1977.