NTNU
Det skapende universitet

ontoWiz

Making Tasks of Reasoning Fast

Ole Kristian Ekseth

Master i datateknikk

Innlevert: juni 2013

Hovedveileder: Pål Sætrom, IDI

Medveileder: Martin Kuiper, IBI
Jan Christian Meyer, IDI
Vladimir Mironov, IBI

Norges teknisk-naturvitenskapelige universitet

Institutt for datateknikk og informasjonsvitenskap

In gratitude to Jesus,

who pin-pointed answers
unkown to researchers.

Where then does wisdom come from?
And where is the place of understanding?
Thus it is hidden from the eyes of all living
and concealed from the birds of the sky. [...]
God understands its way, and He knows its place.
For He looks to the ends of the earth
and sees everything under the heaven.
Book of Job 28; 20-24

Sammendrag

Det finnes i dag ingen gode verktøy for søk i bioemedisinske ontologier. I denne masteroppgaven presenterer vi et verktøy kalt ontoWiz, som gir en ytelsesforbedring på mer enn seks størrelsesordner. Vår tilnærming har forsøkt å forenkle anvendelsen av verktøyet gjennom et brukervennlig grensesnitt uten å gå på bekostning av korrekthet, ytelse, eller tilpasningsevne til fremtidige krav, derav ontoWiz som er utformet for søk biomedisinske ontologier uten å være begrenset til disse.

Programvaren som er utviklet i løpet av arbeidet har blitt tilgjengeliggjort via *world wide web*, i den hensikt å fremme reproduserbarhet for andre forskere, samt å gjøre ytelsesvurderingene relaterbare til egenskaper ved praktisk anvendte ontologier.

Ytelsen i ontoWiz stammer fra vår integrasjon av algoritmer og minneaksess-teknikker. OntoWiz støtter ontologiresonnering med høy ytelse, som er oppnådd gjennom forhåndsberegning av spørringer før de blir stilt, dvs. igjennom pre-prosessering av ontologier. I denne masteroppgaven utvikler vi en algoritme for slik pre-prosessering. Algoritmen senker antallet minneaksesser vesentlig i sammenligning med tidligere tilnærminger, dvs. en betydelig ytelsesforbedring for ontologiresonnerings-oppgaver.

Grensesnittet til ontoWiz tillater brukere å tilpasse regler for pre-prosesseringen, for slik å støtte viktige oppgaver innenfor ontologiressonering. Selv om regelanvendelser ikke utgjorde del av den opprinnelige oppgaven, forsto vi underveis i vårt arbeide betydningen av denne utvidelsen, dvs. innenfor bruksområdet til vårt program. Resultatet av vårt arbeid er et program som dekker oppgaver som vanligvis regnes for uhåndterbare, slik som konstruksjon av komplekse tillukninger av ontologirelasjoner i sanntid.

Masteroppgavens Oppgavebeskrivelse

ONTO-PERL (OP) er et program skrevet i Perl, et program som kun tillater enkle søk i ontologier som har egenskapene til et partielt ordnet set. Eksempler på enkle søk som OP støtter er å finne skjæringspunktet, uionen eller korteste stier mellom et gitt antall vertexer. OP ble opprinnelig laget som et program for å konvertere mellom ulike ontologiformater, men blir nå brukt som verktøy for å søke i store ontologier. Som en konsekvens av dette går OP veldig tregt. Denne masteroppgaven vil foreslå og teste en forbedret algorithme og programvarearkitektur for søk i store ontologier. Spesifikt skal algoritmen og programmet som blir utviklet klare å svare på søk vedrørende skjæringspunkt, uion og korteste-sti i et partielt ordnet set.

Summary

The field of biomedical ontology reasoning is hampered by the limitations of slow tools. In this master thesis we present a novel tool named ontoWiz, which outperforms other software by more than six orders of magnitude. Our approach has sought to balance a user friendly interface without compromising correctness, performance or adaptability for future requirements, ontoWiz is designed for biomedical ontologies without being constrained by them.

The software developed in the course of this work has been made available on the world wide web, in the interest of enabling other researchers to reproduce our results, or relate the performance analysis to properties of real-world ontologies.

The performance of ontoWiz stems from our integration of algorithms and memory access techniques. ontoWiz supports high-performance ontology reasoning through calculation of subsets of queries before they are asked, *i.e.* through an ontology pre-processing. This thesis develops an algorithm to perform such pre-processing. The algorithm lowers the number of memory accesses significantly in comparison to previous approaches, which provides significant performance improvements for ontology reasoning tasks.

Through the application interface of ontoWiz we allow the user to configure the rules of the pre-processing, without limiting the set of pre-computed queries. Although rule application was not considered in the original task description, it was found in the course of this work that including rule applications was necessary in order to cover common tasks in the problem domain. With this extension of the scope, we have constructed a program which covers tasks commonly held to be intractable, such as the construction of complex closures of ontology relations in real-time.

The Task Description of the Master Project

ONTO-PERL (OP) is a Perl software for basic querying in ontologies that are represented as partially ordered sets. Examples of such queries are finding intersections, unions, or shortest-paths given a set of vertices. OP was originally designed for the purpose of parsing ontology specifications, but is currently used to query large ontologies. Consequently, OP currently has severe performance issues. This master project will propose and test an improved algorithm and software architecture for querying large ontologies. Specifically, the algorithm and software should handle intersections, unions, and shortest-path queries

in partially ordered sets.

Acknowledgments

The research which is presented in this master thesis cites several sources of knowledge, as seen in the bibliography. The bibliography does not contain the sources of knowledge and experience which the author of this master thesis regards as the most valuable.

Professor M. Kuiper brought his limitless hospitality and optimism into difficult days. The weekly discussions with Dr. V. M. Mironov shed light into corners of biomedical research which was not covered by earlier research, such as generalizations of rule-based queries, which made it possible to develop a tool which we expect the biomedical community to benefit from. Assisting our work in construction and evaluation of micro benchmarks, Dr. J. C. Meyer provided valuable hands on knowledge from the field. All three of them (*i.e.* M. Kuiper, V. M. Mironov and J. C. Meyer) volunteered with a joyful smile to spend their long evenings reading through the endless number of manuscripts leading to this master thesis. We also want to thank Dr. O. V. Solberg and MD. B. H. Helleberg for their contributions and support.

The work which we present has strongly benefited from the contribution of the people referred to in this section. If errors are found in this master thesis, the sole responsible is the main author of this master thesis.

12 June 2013
Ole Kristian Ekseth

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Ontologies constitute an indispensable component of modern knowledge management. An ontology holds a set of collected facts, and the relations between these facts[33]. An example of an ontology is a tree of individuals, such as human diseases or feeding patterns of animals. Ontologies are amenable to computation and thus provide a way of answering complex questions. Example of questions are:

1. What are the components of a cell that are directly involved in the cell's metabolism?
2. How many generations have elapsed on average since the first Norwegian entered Norway?
3. Does the West Nile virus[44] infect squirrels?

Answering the above questions implies an investigation of the relatedness of the individuals in the ontology. The investigation implies following all of the relations until the answer is found.

The mathematical model underlying ontologies is the graph, which is a set of relations, where a singular relation is understood as a pair of connected individuals from the ontology. A graph uses the term *vertex* to describe ontology's individuals. *Edges* in a graph represents the linkage between the *vertices*. When a direction is important to the edges, they are denoted as *arcs*. An ontology is understood as a graph with arcs, i.e. a directed graph. A properly designed ontology should not contain cycles, in this case it is a *directed acyclic graph* (DAG). , This is because cycles make inferencing and querying

hardly possible. Often we are interested in relating vertices connected through particular types of arcs. An arc is given a label in order to provide an interpretation of the relation, *e.g.* a Norwegian is a human or a Norwegian is part of humanity.

Connecting different types of arcs requires a set of connectivity rules. As various research communities are involved in both ontology engineering and exploitation, different standards expressing the connectivity rules evolved[41]. Each standard represent an ontology format, and each ontology format provides a syntax defining (among others) the arc labels. Examples of ontology formats are the Open Biological and Biomedical Ontologies Format (OBOF), Ontology Web Language (OWL) or the Resource Description Framework Schema (RDFS). .

Handling of ontologies requires awareness of:

1. the differences in the ontology structures, *e.g.* as seen for the ontologies stored in OBO, OWL or the RDFS format.

2. how ontologies are translated into graphs without loss of information.

3. the set of rules defining the connectivity between distinct arc labels.

4. knowledge of what makes the process of reasoning slow.

5. approaches for reducing the time cost of reasoning (in the ontology) and how the approaches may be translated into high performing tools.

The importance of ontologies and efficient handling thereof increases every year [15, 41]. Both the number of ontologies in use and the sizes thereof are constantly growing. The steadily growing impact of ontologies can be illustrated by the fact that the volume of genome annotations with terms from Gene Ontology, the most widely used biomedical ontology, has risen from 6.0 M in 2006 to 77.8 M in 2012 [56]. This makes highly efficient automated ontology reasoning of utmost importance.

However, currently reasoning in large and complex ontologies is often extremely time consuming. The computationally demanding parts of reasoning in ontologies stem from the complexity and number of interrelated vertices. Examples of time/memory consuming operations are:

1. finding the intersection of union of sets of vertices,

2. discovering the ancestors and descendants of a vertex,

3. finding the shortest path(s) between two arbitrary vertices.

The bottlenecks, the computationally demanding parts of ontology reasoning, make reasoning on large ontologies at best semi-tractable[39].

At the same time, applications in bioinformatics are often developed in high level languages by people without knowledge of well established algorithms. Therefore, we expect the suboptimal performance to be additionally confounded for by:

1. problems with efficient handling of big data sets, i.g. the number of relations to evaluate,

2. performance lag (i.e. overhead) due to the programming languages abstraction from the memory layout of modern computers (*e.g.* the high level implementation of associative arrays in Perl),

3. failure to use well established in the field of computer science algorithms for high performance querying.

Indeed, our preliminary observations confirm a wide spread incidence of random memory access and redundant operations in some applications (Supplementary material).

Therefore, we decided to develop a new approach to querying knowledge in (biomedical) ontologies. The objective was to build a high performance generic tool that could be plugged in a variety of exiting applications, including those developed in high level languages. This would combine very efficient querying with the ease and convenience of maintenance and further development.

1.2 Concept

The key component in our approach is an engine designed to deliver maximal performance by making use of highly optimized data structures to target the computationally most intensive elements of logical reasoning in ontologies, addressing specific cost constraints.

Naive data structure implementations are poorly suited to the hierarchical memory structure of contemporary computers, affecting the observed performance of algorithms with regard to the number of operations (*e.g.* the "Quick Sort" algorithm[34] for the task of hierarchical sorting). When working on data, each processor loads from the memory, and the rate of data transfer can affect execution speed dearly. In recent years this performance issue has grown more acute, and the problem is expected to continue for years to come [8] – since 1970s the processor speed has annually increased by a factor of 50% while the performance of memory speed has increased by only 7% [48]. To address the performance limitations of accessing large memory, modern computer architectures store

smaller subsets of data in a hierarchy of successively faster, but smaller, cache memories [1, 8]. The most immediate optimization strategy for memory intensive operations is to arrange tool's memory accesses in order to maximize the utilization of this hardware (i.e. the cache memory). One technique is to exploit the high probability of spatial proximity in hardware near-future-accesses. Applying the principle of accessing memory which has spatial proximity, implies accessing a set (*e.g.* of ontological relations) in the same order as it was stored. For operations on data sets exceeding the cache limit, differences in the memory access patterns can account for a significant part of the run time. With appropriate spatial locality in the access pattern, it is possible to maximize the utilization of hierarchical cache memory. This implies that future requests can be pre-loaded in cache memory, from which we surmise that the design of the data structure is an important performance consideration. Fitting cache-aware access patterns into an efficient algorithm enables minimizing the number of cache misses. A cache miss is understood as a memory request which was not found in the hierarchical cache structure, and therefore is retrieved from the considerably slower main memory. Improving the memory access patterns requires systematizing of the data, which is a pre-condition for exploiting the likelihood of spatial locality.

Reduction of the cache miss frequency does not directly translate into an efficient data structure for reducing query time due to the added work of ordering the data. The tractability of an improved approach therefore depends on the total running time of the alternative to be lower, *i.e.* compared with the naive implementation. Looking at the set of operations required for our proposed cocO(*n*) structure (see below), an important operation is comparison of lists. When unordered lists are compared, both of length $|V|$ with each element $v_i \in V$ tested against $v_k \in V, i \neq k$, the approximate running time becomes $|V|^2$, corresponding to a naive algorithm performing intersection or union of related vertices on a set of vertices. Given the importance of the performance impact, a refined algorithm would address the option of list sorting at building time vs at reasoning time.

Improving the speed of calculating intersections, efficient memory representations have been proposed by Briggs *et al.*[18]: when storing the relations in a bit vector, operations of the type intersection and union are performed fast, however at the cost of high implementation complexity. Briggs *et al.* discuss the alternative of combining a matrix representation of the relations with a dense set, *i.e.* for providing fast look up to important fields in the matrix representation. The benefit of this approach is a lower operation cost for finding the intersections or unions, and finding the ancestors or descendants. The problem however with both approaches is the use of non-contiguous memory, resulting in unpredictable cache access patterns, which increases the running time of the software.

Considering the added complexity of finding the shortest/longest path between two arbitrary vertices, the need for an alternative structure is further underlined. We expect the optimized queries on such a structure would be of a complexity far lower than $|E|$ operations, *i.e.* compared to the naive implementation where in a worst case all of the E relations in the ontology must be explored before the longest path is calculated. With distinct parts for building and reasoning, the resulting implementation should offer efficient execution using known memory handling approaches[25]. By carefully building the data structure using predictable memory access patterns, a considerable speed improvement in the resulting reasoning is expected.

Thus, the speed improvement is due to a transformation of (a subset of) ontology into a special purpose data structure. The data structure is generated once by iterating through the ontology and extracting relevant properties. Afterward it could be used repeatedly to answer queries (i.e. reasoning tasks). The procedure of ontology iteration is what we call pre-processing. The pre-processing results are expected in a running time of $O(n)$ for each distinct query performed, where O is the upper bound of the asymptotic running time, and n is the number of $|V|$ vertices in the ontology.

The core engine was developed as a C++ library supporting multithreading, which we call cocO(*n*) (pronounced *kəkuːnoet*, where the letters correspond to the International Phonetic Alphabet (`http://en.wikipedia.org/wiki/International_Phonetic_Alphabet`)). To be used cocO(*n*) should be integrated into a three tiered application as exemplified below.

We have chosen the ONTO-PERL[4] library as our use case to demonstrate the validity of our approach. ONTO-PERL is a generic API for handling ontologies which provides rather comprehensive functionality for ontology engineering and also some querying capability. However, the latter suffers from the performance issues described above. We developed an application that emulates ONTO-PERL taking the advantage of cocO(*n*), whose architecture is explained in Table 1.1:

Tier	Description
1	Using the existing ONTO-PERL user interface for parsing, exporting and user interaction,
2	A middle layer designed as a loose coupling between the different layers of the user-interface and memory schemes for ontology storage.
3	The C++ core performing the time-consuming operations, named cocO(*n*).

(Continued on next page.)

Table 1.1 – continued from previous page.

Tier	Description

Table 1.1: The three tiered architecture of ontoWiz.

1.3 Outline

Below is an outline of what follows this Introduction. Chapter 2 explores the features of ontologies most relevant in the context of the concept presented above and concludes with concrete decisions on the implementation of the concept. Chapter 3 contains a brief study of applications in the field of reasoning, in order to identify algorithms and applications that seem applicable. Providing a framework for investigating these assumptions, chapter 4 provides the design of micro benchmarks measuring the effect of memory access patterns. Chapter 5 describes conclusions drawn from this, *i.e.* the description of important attributes of ontology-access-patterns.

Chapter 6 describes the cocO(n) algorithm and data structure, which is inspired by the work presented in the pre-project. Chapter 7 includes a description of ontoWiz. Highlighting the knowledge gained from the above chapters, chapter 8 summarizes the important findings of our study. The report spans the domains developments, practices, techniques, solutions, *etc*. At the end a bibliography, the list of external tools which we use is provided in Appendix D. An analysis of ONTO-PERL's implementation and functionality is provided in Appendix B. Appendix C provides a brief summary of the basics in the field of computational algorithms, before including the Index at the reports last pages.

Chapter 2

Assessment of the domain of discourse

Given the goal of this project, the development of a high performance tool for inferencing in ontologies, the domain of discourse is naturally the domain of ontologies taken as such. It is obviously mandatory to carefully explore the domain of discourse before embarking on the development. As the complete domain is really too broad it was important to choose a representative set of ontologies. We opted for ontologies developed in the area of the life sciences because of their wide spread use and diversity. More specifically we focused on ontologies which are members or candidates of the OBO Foundry[63], which mostly follow closely good principles of ontology design. The most relevant metrics obtained for these ontologies were used in the subsequent decision making. The conclusions of this analysis are expected to be broadly applicable.

Note: the definitions of ontology related terms used in this chapter can be found in the Glossary.

2.1 Ontologies and metrics

Our main objective for ontology benchmarking was to provide estimates of memory consumption. More precisely, we wanted to see if it would be possible to load all the necessary information either into the cache or at least the main memory. The estimates were made on the basis of a number of ontology metrics that were deemed most relevant with respect to memory consumption. All memory estimates presumed 4B per term and 12B per relation.

From the list of ontologies featuring on the web site of the OBO Foundry we selected 66 which are used in the Biogateway project[6], since those all passed rigorous quality control within the project. Nevertheless, 4 ontologies were found to contain cycles and thus were excluded from further analysis. The complete set of ontologies is found at `https://code.google.com/p/ontowiz/source/browse/#hg%2Fml_ontology%2Fsample_data`. A detailed study of the ontology metrics and the 128 parameters we analyze are included at `https://code.google.com/p/ontowiz/wiki/PerformanceBenchmark`. When considering the details of all the 62 analyzed ontologies, we observed several requirements regarding our implementation. Below is a brief summary of our observations.

First of all we quantitated the most basic metrics - the number of terms, relations, instances and relation types (Figure 2.1).

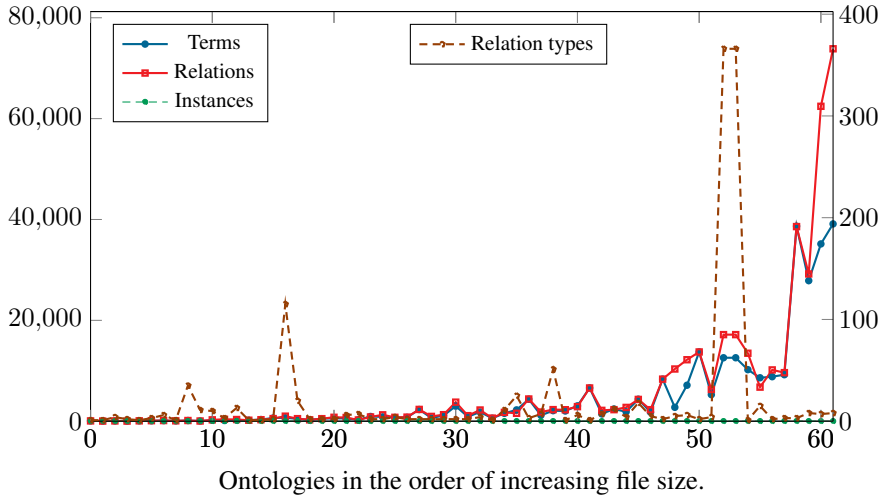

Figure 2.1: Number of terms, relation, instances (left y-axis) and relations types (right axis) per ontology.

Firstly, we observe that the number of relations relative to the number of terms is rather low. Secondly, all the ontologies are completely devoid of instances, except the Teleost Taxonomy Ontology. In the latter, for the total of 36,665 terms there exists only 8 instances (or *objects* in the owl-language). Thus, the structural evaluation of ontologies will be limited to linkage between terms (denoted as *classes* in the owl language). Finally, while the vast majority of ontologies use only a very limited number of relation types, there are just a few which use a staggeringly broad range of relation types (up to 366).

Storing the complete set of relation type transformations in memory would require a 366-dimensional matrix. The memory required for such a matrix amounts to:

$$\frac{366^{366}}{1024 * 1024}\text{MB} = 1,6 * 10^{932}\text{MB} \quad \text{(where 1MB is 1024*1024B)} \tag{2.1}$$

which is astronomical and therefore prohibitive.

Our next concern was the memory footprint of storing all the explicit relations in memory, which is addressed in Figure 2.2:

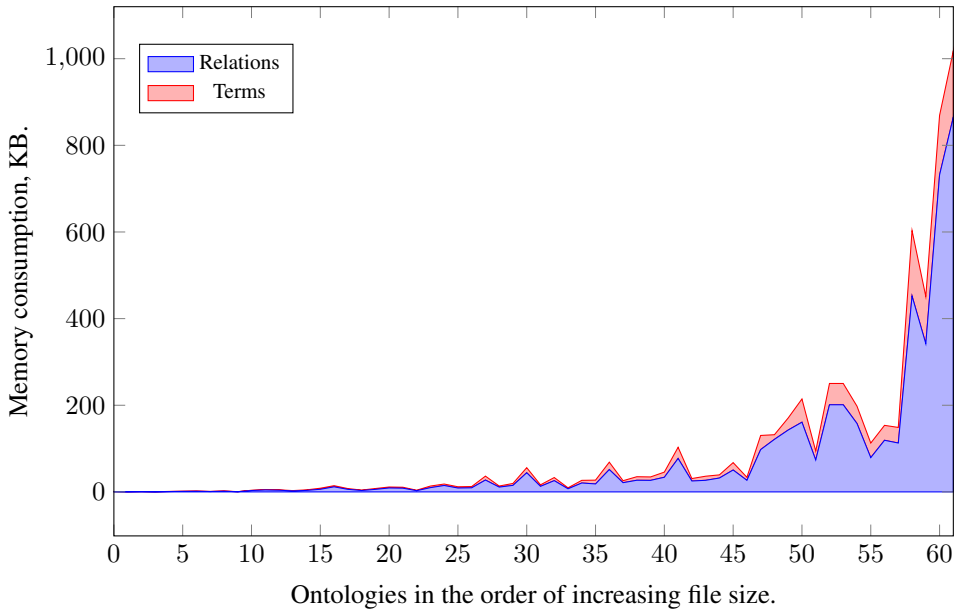

Figure 2.2: Stack plot of the size of memory required to store relations and terms.

The results presented in Figure 2.2 afford a couple of useful conclusions:

1. The complete ontology may reside in the memory, but not always in the smallest cache.

2. Memory consumption is clearly dominated by relations.

Finally, we evaluated the tractability of building the collection of all possible paths (CAPP). CAPP is calculated by first finding for each vertex all the paths in the ontology containing the vertex and then aggregating the paths for all vertices, therefore CAPP is

not a set and requires much more memory than terms and relations. Since each vertex may belong to any number of paths the total number of paths can grow very high and thus this operation may easily be the most memory intensive. If building CAPP is tractable memory-wise, then it is safe to assume that building the ancestors/descendants set is also feasible. The construction of CAPP is expected to be tractable if the memory growth is below polynomial. Polynomial growth is understood as

$$\sigma^{\eta} \tag{2.2}$$

where σ represents the average number of added paths and η the number of parent vertices having the average number of paths. We estimate the local complexity (entropy) of the paths as

$$\frac{\forall_i \cap (paths(v_i))}{\forall_i \cup (paths(v_i))} = \frac{\sum \left((v_k \in \omega) \rightarrow 1 \right), v_k \in \Phi[v_i] \ , \ \omega \in \Omega[v_j]}{\left| \Phi[v_i] \right|} \tag{2.3}$$

where

- v_i and v_k are arbitrary vertices in the ontology.
- Ω holds the set of paths to the roots and
- ω is a simple path (*i.e.* set of vertices without furcation) to the root.

For each ontology Equation (2.3) is applied to every node and the complexity of CAPP is produced by averaging over all nodes. The evolution of CAPP complexity is seen in Figure 2.3:

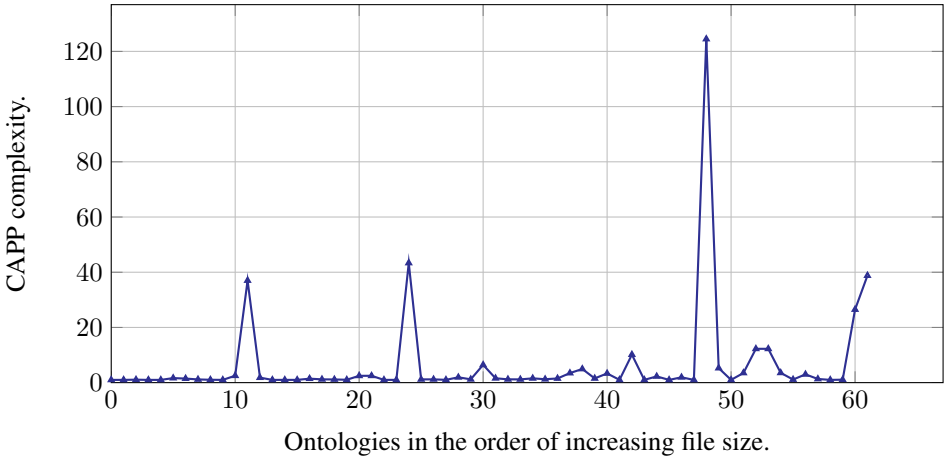

Figure 2.3: The relationship of CAPP complexity with ontology size.

Figure 2.3 clearly demonstrates the lack of polynomial growth (or any for that matter) of the paths complexity with the size of the ontology and consequently we conclude that storing all paths for each vertex in memory is likely to be possible.

2.2 Impact of Inference Rules

The ontologies in our benchmark set are identified by a set of of terms and relationships. If the semantics of relation types are properly specified it is possible to define additionally a set of inference rules, which allow to make implicit knowledge explicit. The ontologies in the previous section were evaluated without applying rules. In this section we evaluate the possible impact of application of rules on the design of cocO(n).

Our analysis capitalizes on the works by Smith *et al.*[62], Boeker *et al.*[17], Hoehndorf *et al.*[35], Aranguren *et al.*[7] and Blonde[15] and attempts to:

1. support property-based evaluation of the benchmark ontologies (*i.e.* to validate conclusions drawn in the previous sub-section),
2. enhance algorithmic design by providing concrete examples of usage,
3. define the axioms required for proofs of correctness of of rule based queries.

Below we will be using the following notation:

- Latin capital letters denote vertices (terms/classes), and

- Greek letters alphabet denote relation types.

As the assessed ontologies don't use instances (see the previous section) we consider only class-class relations. The definitions of semantics of relations are universally based on instances. In the absence of instances the semantics must be specified explicitly in the ontology, which is the case with the ontologies under investigation.

Description Logic's inferencing is inherently based on the notion of all-some semantics [62], therefore all relations in a properly designed ontology are expected to bear all-some semantics. Informally, this means that if class A relates to class B by an all-some relation R then all instances of A must relate along R to at least one instance of B (a more formal definition will be given later on in this section). Formally, the *all-some* property is understood as

$$A \xrightarrow{\beta} B \equiv a \xrightarrow{\beta,t} b, \quad \forall\{(a,t) \in A\} \wedge \exists\{b \in B\} \tag{2.4}$$

where all of vertex A's instances (*i.e. individuals* in DL) have at least one relation to B (*i.e.* the rightmost vertex) at time t.

Below we define the semantic properties that will be used for making inferences.

Definition 2.2.1 of the anti-symmetric property:

Definition 2.2.1 (the anti-symmetric property)**.**

$$\{A, B\} \in V \qquad \text{is the vertices related by} \tag{2.5a}$$
$$\{\alpha\} \quad \in R \qquad \text{with the property} \tag{2.5b}$$
$$(\alpha \cap \{\text{anti-symmetry}\}) \neq \emptyset \qquad \text{then } \alpha \text{ is anti-symmetric for} \tag{2.5c}$$
$$A \xrightarrow{\alpha} B \qquad \textit{which is not equal to} \tag{2.5d}$$
$$B \xrightarrow{\alpha} A \qquad \textit{illegal from anti-symmetry.} \tag{2.5e}$$

Rule: If vertex *A* is related to vertex *B* by an anti-symmetric relation α, and given the fact that there does not exist any relation from vertex *B* to vertex *A*, then vertex *B* is not related to vertex *A*.

Equation (2.5d) is an example of a relation. A relation is sometimes called a triplet, *i.e.* $\big(A, B, \alpha\big)$. From our study of ontologies, we know that the anti-symmetric property from Definition 2.2.1 is covered by most of the ontologies in our benchmark set: a relation type is anti-symmetric if not stated as symmetric (`http://oboedit.org/docs/html/An_Introduction_to_OBO_Ontologies.htm#symmetry`). The symmetric property is given by Definition 2.2.2:

Definition 2.2.2 (the symmetric property)**.**

$\{A, B\} \in V$	are the vertices related by	(2.6a)
$\{\alpha\} \quad \in R$	with the property	(2.6b)
$(\alpha \cap \{\text{symmetry}\}) \neq \emptyset$	then α is symmetric for	(2.6c)
$A \xrightarrow{\alpha} B$	which is equal to	(2.6d)
$B \xrightarrow{\alpha} A$	the result from symmetry.	(2.6e)

Rule: If vertex *A* is related to vertex *B* by an symmetric relation α, then vertex *B* is related to vertex *A* by relation type α.

Definition 2.2.2 implies that a symmetric relation provides the correct meaning irrespective of the interpretation order (*i.e.* the order in which the vertices are read). An example is the relation "men are *related to* squirrels", which also holds for "squirrels are *related to* men". The symmetric property describes knowledge between two entities (*i.e.* vertices). When the knowledge of a relation describes the vertex itself, the relation is said to be reflexive:

Definition 2.2.3 (the reflexive property)**.**

$\{A, B\} \in V$	is the vertices related by	(2.7a)
$\{\alpha\} \quad \in R$	given relation	(2.7b)
$A \xrightarrow{\alpha} B$	which has the property	(2.7c)
$(\alpha \cap \{\text{reflexive}\}) \neq \emptyset$	then α is reflexive, with the implication	(2.7d)
$A \xrightarrow{\alpha} A$	the result from reflexivity.	(2.7e)

Rule: If vertex *A* is related to an arbitrary vertex *B* by an reflexive relation type α, then relation type α describes a property of itself (*i.e.* vertex *A*).

The properties covering symmetry, anti-symmetry and reflexivity describe inferences regarding a single relation.

The type of inferences we are mostly interested in is generally referred to as rule chains or compositions. Here is the most generic formulation of the chain rule: Definition 2.2.4:

Definition 2.2.4 (the property of chains)**.**

$$\{\alpha, \beta, \mu\} \quad \in R \qquad \text{is the set of relation types,} \tag{2.8a}$$
$$\{\alpha, \beta\} \prec \mu \qquad \text{where } \mu \text{ holds over } \alpha \text{ and } \beta, \text{ and given relations} \tag{2.8b}$$
$$A \xrightarrow{\alpha} B \xrightarrow{\beta} C \qquad \text{then the new relation becomes} \tag{2.8c}$$
$$A \xrightarrow{\mu} C \qquad \text{where } \mu \text{ is the result from property of chains.} \tag{2.8d}$$

Rule: If *A* is related to relation type *B* by α, *B* is related to *C* by relation type β, and relation type μ holds over both α and β, then *A* is related to *C* by relation type μ.

The rule above has a number of very important sub-rules. Among those we are particularly interested in the rule known as 'priority over is_a': Definition 2.2.5:

Definition 2.2.5 (the priority over is_a rule)**.**

$$A \xrightarrow{\beta} B \xrightarrow{\zeta} C \qquad \text{are two concrete relations defined by} \tag{2.9a}$$
$$\beta \in \{\text{all-some}\} \qquad \text{where } \beta \text{ describes a set of relation types, and} \tag{2.9b}$$
$$\zeta \in \{\text{is\_a}, \text{sub-class}\} \qquad \text{the set of } \zeta \text{ relation types,} \tag{2.9c}$$
$$\zeta \prec \beta \qquad \text{where relations described by } \beta \text{ are also described by } \zeta: \tag{2.9d}$$
$$A \xrightarrow{\beta} C \qquad \text{the result from the all-some property.} \tag{2.9e}$$

Rule: If *A* is related to *B* by an *all-some* relation type β and *B* is related to *C* by the *sub-class* relation (*is_a*) ζ, then *A* is related to *C* by β.

Another very important sub-rule of the rule of chains is the transitivity rule.

Definition 2.2.6 (the transitive property)**.**

$$\{A, B, C\} \in V \qquad \text{is the vertices related by} \tag{2.10a}$$
$$\{\alpha\} \quad \in R \qquad \text{applied to the relations} \tag{2.10b}$$
$$A \xrightarrow{\alpha} B \xrightarrow{\alpha} C \qquad \text{implicates} \tag{2.10c}$$
$$A \xrightarrow{\alpha} C \qquad \text{the result of transitivity.} \tag{2.10d}$$

Rule: If *A* is related to *B* by a transitive relation α and *B* is related to *C* by a transitive relation α then *A* is related to *C* by a transitive relation α.

If the transitive relation in question is 'is_a' we have the all important class subsumption hierarchies.

Similarly to class subsumption we can define property subsumption:
Definition 2.2.7:

Definition 2.2.7 (the property subsumption)**.**

$$\beta \subset \mu \qquad \text{where } \beta \text{ is a sub-relation of } \mu, \text{ and given relation} \tag{2.11a}$$

$$A \xrightarrow{\beta} B \qquad \text{then the relation becomes} \tag{2.11b}$$

$$A \xrightarrow{\mu} B \qquad \text{where } \mu \text{ is the result from property-over-subsumption.} \tag{2.11c}$$

Rule: If vertex *A* is related to vertex *B* by a relation type β, and β is a sub-relation type of relation type μ, then vertex *A* is related to vertex *B* by the relation type μ.

The rules formulated above form a minimal set of design requirements for supporting inferencing in cocO(*n*)/ onto*Wiz*.

Now we estimate the impact of implementing rule support in cocO(*n*) on memory consumption. Finding all ancestors and descendents for a term is an essential inferencing operation which form the basis for numerous other reasoning tasks, therefore we estimated the amount of memory required to store for each term in the ontology the complete sets of ancestors and descendants (Figure 2.4:

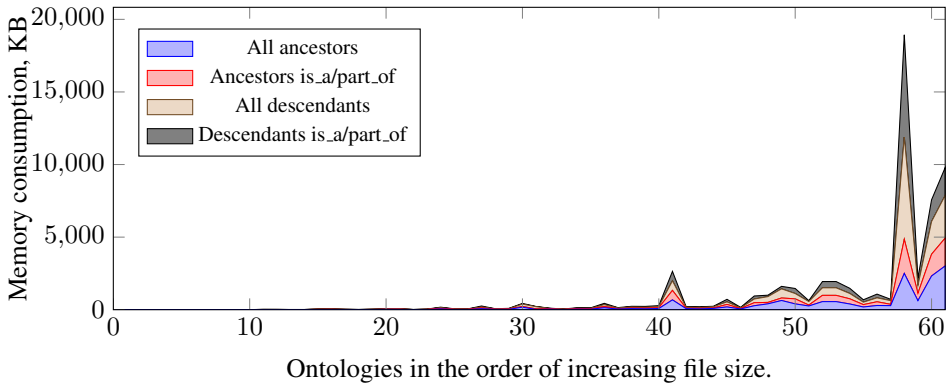

Figure 2.4: Memory requirements for storing ancestors and descendants for each term. Estimates are given for all ancestors/descendants and for those along the relations 'is_a' or 'part_of' only.

In the previous section it was shown that storing information for each of the relation types separately is out of reach. Consequently, we computed ancestors/descendants without regards to the relation type (All ancestors/descendants in Figure 2.4). Yet, it is often

desirable to restrict the ancestry to a particular relation type. Therefore, we estimated as well the additional memory cost of storing ancestors/descendants for a few selected relation types. The Figure show results for the two by far most important relation types in ontologies - 'is_a' and 'part_of', which combined contribute more that 90% of the benchmark ontology's relations. It is easy to see there are no hardware limitations for storing ancestors and descendants in memory.

The analysis above confirms the feasibility of pre-processing ontologies and storing in memory for each vertex the sets of:

1. ancestors and descendants without regard to relation type,
2. ancestor and descendant's which are connected through distinct relation types, and
3. all possible ancestor paths.

The structural evaluation of the pre-processing suggested to stored the set of all unique vertices and (non-rule-based) paths in memory, organized in contiguous memory blocks to optimize the utilization of cache. The result of the analysis of the application of rules indicates that it does not increase the memory consumption in a dramatic way, but that it does increase the complexity of the algorithm. We conclude that the analysis of our benchmark ontologies allowed us to define a practical design of cocO(*n*) (Chapter 7).

2.3 Structural requirements

The previous subsections evaluated structural features of benchmark ontologies, and implications of applying rules to the benchmark ontologies. Translating this knowledge, resulted in a set of structural requirements for onto*Wiz* and cocO(*n*), as outlined in Table 2.1:

Requirement-id	Background	Description of Requirement	Package
SR_1	Identifiers in the ontology are represented by strings.	Separate internal representation from actual implementation, *e.g.* map a Terms string identifier into a number.	onto*Wiz*

(Continued on next page.)

Table 2.1 – continued from previous page.

Requirement-id	Background	Description of Requirement	Package
SR_2	The front end ontology model is not designed for ontology reasoning.	Separate the internal ontology representation from the representation used in the task of reasoning.	cocO(*n*)
SR_3	Naive algorithms are used for ontology reasoning.	Design specific algorithms reasoning with minimal overhead, and document both their correctness and speed-improvement.	cocO(*n*)
SR_4	Implementation in our use-case (ONTO-PERL) is specifically designed for a limited set of ontologies.	Design the packages using strict separation between the layers.	ontoWiz and cocO(*n*)
SR_5	ONTO-PERL is integrated in several packages.	Provide a functional interface equal to ONTO-PERL, while not constraining future developments of the API nor its internal representation.	ontoWiz
SR_6	Build new ontologies from a rule-set.	Enable the pre-processing to build new ontologies from user-applied rule-sets.	cocO(*n*)
SR_7	Deliver high-performance reasoning-support to operations on arbitrary set of vertices, such as finding all connecting paths, intersection, union, *etc*.	Build, for each vertex, ordered sets of related vertices and paths.	cocO(*n*)

(Continued on next page.)

Table 2.1 – continued from previous page.

Requirement-id	Background	Description of Requirement	Package

Table 2.1: The structural requirements that is fulfilled during our implementation of onto*Wiz* and cocO(*n*).

Table 2.1 defines the structural requirements of our implementation, *i.e.* the blueprint for our work. All these requirements should ideally be met by onto*Wiz* and cocO(*n*). Essentially we have now defined a layered set of system requirements, but not yet the interaction between onto*Wiz* and cocO(*n*). Before we discuss that, we will first summarize what we have stated so far:

1. Getting the content of an ontology, a parser is required. As we consider ONTO-PERL efficient enough for ontology parsing, the ontology is parsed by ONTO-PERL.

2. The onto*Wiz* interface is written in Perl, while onto*Wiz*' underlying core is in C++; the Perl–C++ conversion requires special purpose code to support the communication between the programming languages.

3. The task of ontology reasoning is performed by cocO(*n*). onto*Wiz* therefore sends a simplified ontology-model to cocO(*n*), which generates an image of the ontology for efficient support of ontology reasoning.

The layered interaction (between onto*Wiz* and cocO(*n*)) is defined by the steps of building the ontology model, and retrieving results from operations of ontology reasoning. The first task is translating the data structure of the parser (*e.g.* ONTO-PERL) into accessible data for onto*Wiz*. We expect the parser to store ontologies in sets of (complex Perl) objects. Below we describe the overall details of the (back-and-forth) format conversion:

Step 1 The Perl part, *i.e.* the upper fragment of onto*Wiz*, performs a for-each call iterating through the parsed ontology-object (*i.e.* the ONTO-PERL object):

Ontology transfer: The onto*Wiz* Perl object initiates its underlying C++ data object (*i.e.* the one holding the complete ontology), by iterating through the parser's ontology object, and inserting each item in the C++ data object.

Ontology gathering: The C++ data object is accessed from Perl using shallow Perl-classes. A shallow Perl class stores only an index-reference to the underlying C++ code. Explicit item-knowledge (*e.g.* of a Terms synonyms) is retrieved using function-calls to the underlying C++-code (*i.e.* the Perl object does not store the ontology).

Step 2 The C++ part interacts directly with a cocO(*n*) object. The process of transforming the receiving Perl ontology consists of three sub-steps:

Step 2.1 Gathering data from the Perl part of onto*Wiz* and storing the objects using a special-purpose generic model. For effective look-up and higher search speed, the strings are converted into hashes, *i.e.* generation of name-index correspondence is performed.

Step 2.2 The pre-processing step is executed when the complete ontology is received, upon which a limited ontology model is sent to cocO(*n*). cocO(*n*) iterates systematically through the ontology, and for each vertex constructs and stores in memory a set of ancestor paths and a set of related ancestors/descendant vertices.

Step 2.3 Reasoning tasks are answered by a call to the cocO(*n*) object. As the glue connecting the programming languages of Perl with C++ does not support transfer of sets, set-items are transferred one-by-one (*i.e.* to the caller.) This approach makes it independent of the upper-level programming languages (supports calls made by programming-languages such as *Python*, *Java*, *C#*, *etc*.)

Step 3 At the time of query, the C++ part of onto*Wiz* returns the answer to the Perl-layer. The Perl-layer translates the returned answer into objects with functionality similar to what ONTO-PERL returns.

The above outline of the interface together with our specified requirements are sufficient to start our first steps in building a tool for fast reasoning in ontological data.

Chapter 3

Related Work

Ontology reasoning involves searches in massive graphs having on average a limited depth and broad width. Ontology reasoning poses a special case in the field of algorithmic design. Algorithmic design covers a broad field. The importance of ontology reasoning prompted the development of several applications and algorithm. A short summary of our findings from the literature review indicates that:

1. There does not yet exist an application for high-performance reasoning in biomedical ontologies.

2. Algorithms exist for general cases of biomedical ontology engineering. None of these, however, are implemented in ONTO-PERL. The main reason for this was the unawareness of the speed-impact of existing algorithms (which we know from discussions with the authors of ONTO-PERL).

3. To our knowledge, there does not exist a specific algorithms or data-structure designed to enhance the tasks encountered in ontology reasoning, for instance to perform operations of intersections, unions and longest paths.

Existing software approaches are examined (*i.e.* with regard to data structures and ability for reasoning), followed by a survey of relevant algorithms and data-structures, where parts of the survey are based on our previous work[24]. Combined with analysis of approaches for speeding up memory handling (as seen in the next sections), the content of this section forms the basis for our cocO(n) ontology reasoning algorithm.

3.1 Software for Reasoning in Biomedical Ontologies

Supporting ontology construction/modeling, the *Protege-2000* software does not focus on performance (in the context of ontology reasoning). In the work of Berry *et al.*[14], a query library for semantic graphs is presented: the library does not include data-structures for cache-efficient data-access (*i.e.* in order to maximize utilization of computers). On the other hand the HDT (Header-Dictionary-Triples) provides a tight representation of semantic RDF graphs, without adding capability for reasoning. (Description of the HDT algorithm is found at `http://www.w3.org/Submission/2011/SUBM-HDT-20110330/`).

Limiting the set of operations, Harispe *et al.*[32] provides a tool measuring the semantic relatedness between vertices. In a similar fashion Hollunder *et al.*[36] identify graph patterns using substructure-frequency-counting. At the user application level the java package *OWL API* supports operations similar to onto*Wiz*. In handling of the data, the two packages differ, as the onto*Wiz* uses a special-purpose reasoning tool for shortening the time of queries (*i.e.* the cocO(*n*) package). This is in contrast to the generic data model of the *OWL API*. Handling huge RDF graphs, Priete *et al.*[53] propose the *rdfhdt* software, but with its limited set of integrated queries, its expression-power is considerably lighter compared to both ONTO-PERL and *OWL-API*.

Optimizing the speed of reasoning without concern about memory access patterns, Haarslev *et al.*[31] presents an algorithm for rule-based calculation, *e.g.* of transitivity. The approach applies pre-processing, but only to a limited degree, by caching results for improving the query-speed (of similar queries). Extending the set of optimization-techniques to the set of sub-sumption rules, Horrocks *et al.*[38] provide a set of algorithm which deliver a substantial performance impact, without proposing nor comparing the implementation to approaches for optimal memory accesses.

Combining a tightly coupled data model with full-fledged reasoning, *Metarel*[16] is developed for ontology reasoning. Making ontologies consistent, and preparing ontologies for efficient reasoning, Metarel utilizes ONTO-PERL's functionality for transitive closures[5]. Comparing Metarel with onto*Wiz*, Metarel accepts a higher overall time and memory consumption for the benefit of allowing more specific queries. This is in contrast to onto*Wiz*' limited, but time and memory efficient, set of queries. Of special interest in this context is the choice of internal structures representing the data, where Metarel uses the RDF-triple-store and SPARQL (`http://www.w3.org/TR/sparql11-query/`) as its knowledge representation language, while onto*Wiz* uses a special-purpose structure (*i.e.* the cocO(*n*) package) and Perl function calls for the task of answering queries.

A scalable implementation of RDF inferences on a huge number of CPUs is described

by Goodman *et al.*[29]. The implementation, which is dedicated to the architecture of Cray XMT supercomputers, provides functionality similar to Metarel. The major weakness of Goodman's tool, is its focus on hiding memory latency through a large number of streams on a given processor, as commodity processors with complex core designs do not provide optimal conditions for this technique[29]. In contrast, both Metarel and our implementation does not depend on special-purpose hardware, making both tools available for integration into existing pipelines.

3.2 Algorithm and Memory Structure

We provide here an analysis of the applicability of known algorithms and data-structures in the field of computer science, given the context of biomedical ontology reasoning. For those readers un-familiar with graph-theories, we recommend reading our brief summary of algorithms and data-structures (found at page 123). But first an overview of the known facts regarding the costs and types of memory handling.

Chilimbi *et al.*[20] provides conceptualized argumentation for the weakness of pointer-based implementations of graphs in a general context. Looking at the specific systems, Bader *et al.*[9] discuss the difference of memory architecture, stating how efficient implementations depend on a given system such as the Cray MTA-2[2]. Stressing this concept, the point of sustaining the ability in increasing memory bandwidth is promoted[50]. Techniques utilizing the hardware operations for pre-fetching and caching are discussed in several papers[40, 26] and books[64, 21]. In a general case typical graph implementations have irregular structure, causing pre-fetching to not give any improvement on the performance[9].

Methods have been proposed to combine a low-latency data structure with specific algorithms for solving performance issues. An interesting alternative which partially covers our work, is described in implementations of single source shortest paths problems (page 126). The algorithm described by Bellman and Ford (BF)[13] evaluates all of the edges as an alternative to the above greedy approach, answering the problem of finding all paths given a single source. A weakness of the BF algorithm in our context is the concatenation of temporary paths, as it imposes the requirement of frequent updating for each of the vertices involved.

Single source shortest paths problems are generally described by the Bellman-Ford and Dijkstra's algorithm[22]. The former accepts negative weights on an arc connecting two vertices, giving that Dijkstra's algorithm is most similar to our work. In comparison to the others, BF does not use a greedy approach, nor does it only find the best path, but as its

approach uses concatenation of temporary paths, it imposes a requirement frequent updating for each of the vertices involved. Extending the problems size to cover all sources, the work of Floyd and Warshall[27] and Johnson[42] (*i.e. Johnson's algorithm*) suggest alternatives, but as both accept negative edges and the first expects a dense graph of vertices, an overhead compared to our problem exists: None of the above described algorithms suggest methodology for calculating paths between all vertices (and not only the shortest).

Neither the algorithms by Floyd and Warshall nor Johnson suggest methods for calculating paths between all vertices (and not only the shortest). Given the problem of finding the path of minimal length connecting vertices (s, t) in large graphs, Berry *et al.*[14] propose using the "Breadth-First-Search" (BFS)[46, 55] from both ends.

3.3 Recommendations for the Algorithmic Structure of cocO(*n*)

We have now reviewed and inspected a set of:

1. important applications,
2. techniques for memory handling and
3. algorithms for generic cases of ontology reasoning.

The result of our above analysis is a method for finding both the set of related vertices, and all paths interconnecting all sources: the set of all paths between all vertices is calculated by replacing the "best-first" approach in Dijkstra's algorithm with "evaluate all".

Using Johnson's algorithm as our model, the edge-adjusting inclusion of the BF algorithm is avoided, as the arc-weights are non-negative. Our approach is inspired by the effort by Berry *et al.* of maximizing ontology reasoning through ontology heuristics, *i.e.* by designing the ontology pre-processing for which the ontology reasoning tasks is applied to. Making the modified algorithm of Dijkstra perform the calculation of shortest paths, the result is an implementation without the memory/time cost of the non-matrix algorithms described above.

The algorithms of Dijkstra and Prim apply a *heap* as their memory storage scheme, which is a tree shaped memory storage scheme applied with properties, such as the sorted property. The cost of our approach is further reduced if we replace the heap with a dynamically sized list for storing and extracting intermediate data, which is implemented combining the BFS with a queue for each vertex accessed in cocO(n).

Incorporating into our approach a selection of the discussed algorithms and memory structure, we are able to meet the algorithmic requirements (*i.e.* SR_7 from Table 2.1 at page 18) for the cocO(*n*) algorithm. The pre-processing algorithm that we suggest:

- is an adoption of Dijkstra's algorithm, where the "best-first" approach is replaced by an "evaluate all" strategy;
- applies Johnson's algorithm, which is run without the adjustment of the negative cycles;
- avoids the memory/time cost having a heap storing/extracting intermediate data, as the non-matrix algorithms of Dijkstra and Johnson do.

Before outlining the details of the cocO(*n*) algorithm, and its formal proof, we are interested in evaluating the cost of memory access for our ontology benchmark set. The pre-processing of our cocO(*n*) algorithm implies storing ontological properties in memory. In the next section we provide a methodology of testing beneficial access-patterns. The goal is to assess the usefulness of our suggested approach, *i.e.* with regard to the importance of the structure storing the pre-processed results.

Chapter 4

Test Methodology for Impact of Data Representation

In this chapter we provide methodology for analyzing the cost of storage schemes, including the random-access-pattern performed for ONTO-PERL, before applying the methodology to different data-sets in the next chapter. The methodology covered in this chapter was also presented in our previous work[24].

4.1 Why a Benchmark Analysis is of Importance

Until now we have assumed that the design of a data structure has a relative high impact on an algorithms performance. These assumptions are based on

1. informal observations of running-time (when reasoning on big ontologies),
2. analysis of our ontology-benchmark and
3. a review of important research in the field of memory handling.

Accepting correctness of these assumptions, the implicit conclusion (that a well-implemented data structure speeds up the software's performance) might still be wrong:

1. The long running time may be due to hidden complexity unrelated to memory handling.
2. The relative path-length may overall be short. One of cocO(n)'s goals is reducing the time for finding longest paths: if the speed-improvement increases with longer

path-lengths, short paths would decrease the performance impact. We expect a sequence of connectivity-queries to be called by the user. As the order in which terms are queried (for their paths) are unknown, the users access pattern can not be incorporated into the design, thereby the benefit of the pre-processing (*i.e.* improved memory handling) may also be unknown.

3. To our knowledge, no research exists regarding analysis of memory handling[9, 40, 26, 64, 21]. It is therefore unknown how this research (discussed at page 23) relates to access patterns in biomedical ontologies.

Evaluating the arguments (in the above list), we observe the requirement of a proper analysis of the impact of memory access-patterns on the handling of biomedical ontologies. A methodology is therefore needed to determine the type of memory-access patterns which cover our requirement (*i.e.* SR_7 from Table 2.1 at page 18). Our goal is to construct an efficient algorithm extracting important ontology facts for fast look-up (*i.e.* retrieval). A short summary of what we know:

1. a complete ontology may reside in memory, *i.e.* without passing the memory threshold;

2. the number of terms grows approximate linearly, *i.e.* both with regard to time-consumption and relative path lengths (given the ontologies in our benchmark-set);

3. we expect the sequences of visited relations in the ontology to result in a non-predictable access pattern, *i.e.* at random (which is illustrated in Table B.1 at page 121).

As a first approach to estimate the cost of random access, we may use data from our ontology-benchmark (as seen in section 2.1 at page 7). Assuming that relations from the ontology are stored at random locations in the memory, the probability of finding a relation for pro-reasoned-v24.0..obo is

$$100 * \frac{1}{|V|}\% = \quad \%\text{(chance)} \tag{4.1}$$

$$100 * \frac{1}{27869}\% = 0.0036\% \text{ chance}, \tag{4.2}$$

which is the likelihood of the computer to randomly find a term in memory (*i.e.* given an ontology consisting of 27.869 terms). This formula is only an estimation of the likelihood. Estimating the benefit of storage schemes, we need concrete measurements. The measurements target optimization strategies for improved usage of low-latency cache.

4.2 The Targets for our Memory Access Benchmark

When the number of cache misses in a function is considerable compared to its number of accesses, it is worth investigating. Statistics tying cache usage and number of misses to particular parts of a memory access pattern are therefore of interest. The "Cachegrind" program (`http://valgrind.org/docs/manual/cg-manual.html`) provides such functionality, integrated as an extension to the binary analysis tool "Valgrind" (`http://valgrind.org/`).

During our memory benchmark, we explore efficient representations for ontology reasoning. The benchmark set consist of a subset of commonly used data structures for ontology iteration. The data structures are analyzed with regard to time and memory footprint. Tests enabling parametrization are therefore of importance. In order to capture the influence of data access patterns, and the consequence of using relative memory addresses (*i.e.* pointers).

For the purpose of comparison, we perform a set of operations for each alternative:

1. allocate memory;
2. add data;
3. extract data;
4. reallocate memory.

Testing the benefit of structure reuse, we explore adding- and extracting of data without reallocating memory, *i.e.* running the second and third operation multiple times.

The preferred access patterns we intend to identify, will not translate directly into computational time costs performing the operations of interest. Comparison of naive implementations against our cocO(n) library may give indications of the actual cost-difference in reasoning. The latter approach implies an ambiguity. A full-fledged study is therefore recommended, as stated in the list of future work on page 111.

The micro-benchmarks we describe, are written in C++. The source code is made accessible through flow diagrams and summaries (at our repository `https://code.google.com/p/ontowiz/`), using the Doxygen tool (`http://www.stack.nl/~dimitri/doxygen/`).

4.3 Configuration of the Benchmark

The systems of interest consist of two hardware cache layers in each of the CPUs two cores. The first layer is divided into disjoint parts handling instructions and data, while the

latter is unified. If not explicitly stated, we choose to abstract both cache levels as unified.

The purpose of pre-fetching is to reduce the delay factor for memory loading. The instructions executed (fetched) by the CPU depend on the state (conditions) of the software at a particular execution point. Examples of such conditions are "if", "else" or "switch". If the computer could have by-passed the waiting time, the cost of memory fetches is reduced. This is made possible using instruction-level parallelism (ILP) on modern cores. Similar to the data access, there also exists support for memory-level parallelism (MLP)[26].

The operations we test make it hard for the processor to execute instructions out-of-order. Simplifying the analysis, the data will not exceed the physical memory on the test platforms, *i.e.* the additional complexity of disk access is not explored. Restricting our scope, we choose not to discuss the more complex case of thread parallelism. For future work we would recommend investigate this, *i.e.* adding the latter as part of an extended parameter space.

The CPU behavior is isolated in our experimental setup. The tests we provide have a low computational intensity, compared to the complexity of memory access. In practical terms, we perform a basic sum of all the vertices regarded as a member of the result, *e.g.* the sum of vertex keys classified as ancestors for a given vertex. This simple computational task of arithmetic/floating point operations is executed in order to assert correctness, and prevent other factors from obscuring the result.

Complexities in underlying hardware- and specifics of environment the system runs in, make the details of the tests system-specific. An example of the latter is cache optimization techniques, which are first of interest when the data set exceeds the size of the cache. Investigating a Dell Latitude E6510 laptop machine, we observe the smallest Data cache L1 and the unified last-level (L2) cache, respectively to be 32KB and 3072KB. Benchmarking the Dell laptop will therefore require data sizes greater than 32KB. Studying the effects on different systems, additional measurements are made on a the NTNU kongull server and the special-purpose biogw-db system:

Platform	L1 cache	L2 cache	Architecture	OS	Memory	Processor	Processor count
Dell Latitude E6510	32KB	3072KB	amd64	Ubuntu 12.04	3.79GB	i5 2.67GHz, FSB 2500 MHz	4
Kongull	64KB	512KB	amd64	Red Hat Enterprise Linux Server release 5.7 (Tikanga)	24GB	AMD Opteron 2431 2.4GHz, FSB 2200 MHz	6
Biogw-db	32KB	3072KB	amd64	Red Hat Enterprise Linux Server release 5.7 (Tikanga)	125GB	Intel Xeon X7460 2.66GHz, FSB 1066 MHz	24

Table 4.1: The micro benchmark platforms, identified at the leftmost column, are those we conduct our study at. Evaluating the properties of the platforms, we observe the size of L1 cache in column two and size of L2 cache in column three. Column four describe the architecture of the platforms, while column five the operating systems (OS) of them. The memory/RAM of a platform is given in column six, a memory which is distributed among the processors, with properties of the processors listed in column seven and the processor count in column eight.

Studying the above systems, the core architectures are similar. In general terms, those pieces of main memory most recently accessed are found in the cache. When a processor requires instructions or data, a request is made to the cache. If not found, its fetched from memory and counted as a miss, *i.e.* not a cache hit. Software with high cache hit rates decrease the running time. The latter parameters do not provide the complete picture, as the amount of accessed data may influence the result, *i.e.*

$$requestCost(x) = |x| * \chi * \psi_{cache} + |x| * (1 - \chi) * \psi_{memory}$$

$$= |x| * (\chi * \psi_{cache} + (1 - \chi) * \psi_{memory}) \tag{4.3}$$

where $|x|$ describes the amount of data to retrieve, χ the likelihood of a cache hit and ψ the fetch-time of the data to retrieve. In this model we assume cache optimal data structures to be of increasing importance as the relation between memory access costs decreases, *i.e.*

$$\frac{\psi_{cache}}{\psi_{memory}} \to 0 \tag{4.4}$$

Combining Cachegrind with a high resolution time measurement and using the number of clock ticks from the start of an operation- to its end, it is easy to measure the relative time of an operation. The return value from the program "clock_t times(struct tms *buf);" provide such data. An approach using system ticks has several disadvantages:

1. dependency on systems makes it improper for cross-system comparison;
2. the low resolution of the time-measurements hides variability due to multiple runs on the same code;
3. it does not distinguish between the time spent on I/O-waits versus the CPU time.

The latter issue (3) is solved by filling the cache with random numbers before each measurement. The standardized hardware-to-second conversion value "_SC_CLK_TCK" is used for converting from the internal time representation- to seconds. Including the user- and system time in our measurement, using data from the updated parameter, the problems of system-dependency and I/O waiting are rectified.

Observing the Cachegrind output, the measurements are the sum for each code block. This makes it impossible to distinguish which of the implementations caused the cache misses, *e.g.* when performing memory allocations.

The alternatives are therefore run separately, merging them later into a result table. Using a Perl script, the collection process is automated. Including the graph library in our script, a subset of the latter table is visually presented. The Perl-script calls the micro-benchmark code with a specific operation- and range. A Perl-function wraps the set of calls. Prior to the micro benchmark, the source is compiled in a mode discarding internal tests used only for logical validation. System specific setup of the compilation process is made using the CMake tool[52].

Chapter 5

Practical Benchmark Results

The structural ontology evaluation in section 2.1 at page 7 proposes to store the complete set of implicit relations from an ontology. From the structural ontology evaluation, and recommendations presented for a pre-processing algorithm in section 3.3 at page 24, we suggest to store all the precomputed queries in ordered arrays by their access patterns. Building these arrays has a pre-processing cost. The benefit of our approach versus the alternative is explored in this section. We measure the optimization-effect for the operations investigated. An example of this is the time finding all the 60M ancestors related to a vertex in an ontology. To enhance the benefit of a data structure, we are interested in locating the bottlenecks. In this study the components are looked at, first in isolation, and then in the context of ontology reasoning, *i.e.* subsets of the result data are represented in the graphs, while the complete set is found at our repository (`https://code.google.com/p/ontowiz/source/browse/datasets`). Most of the results in this chapter are based on our previous work[24].

5.1 The Isolated Component Representations

Exploring the data structures, we investigate the behavior with regard to time, memory consumption and cache utilization. Comparing different approaches, strengths becomes visible. An example is the *linked list*, which avoid the cache read misses upfront by allocating memory on a demand basis. The alternatives we are interested in testing, is:

Ordered List: All the addresses are adjacent. A negative effect is the upfront allocating of memory. Illustrating the latter, we explore memory reuse and the option of gradually

memory allocation. Verifying the independence of incremental and reverse access order, both are analyzed.

Random List: corresponds to the ontology input expected for our structure, *i.e.* an adjacency list. The randomness is due to the unordered criteria when relations are stored, *e.g.* when iterating through the relations in an OBO or OWL file. The random list accesses its elements either directly, or with an additional pointer based memory reference.

Stacks and Queues: important for graph searches. Uses absolute memory addresses (pointers) as index items. Testing the optimal case, we have not introduced any noise in the cache while building it. This gives a high likelihood that the addresses will be adjacent, with the negative side effect that the higher complexity of the structure gives a higher memory footprint. The stacks and queues provided by C++ Standard Library (STL) are of interest, and are therefore used as benchmark.

Linked Lists: an experimental alternative to the STL stacks and queues. The lists are of length corresponding to the cache size, and connected by head and tail to the others. The purpose is to test the benefit of hiding the cost of memory allocations, as the memory-allocations are performed on a need basis, when compared to an upfront allocations of memory as seen for the *ordered lists*. We expect the extra complexity overhead of the program to be marginal when compared with the number of elements in the lists.

The alternatives we test are a subset of the relatively large space of data structures available. Our goal is to find the best implementations for ontology handling. Bare-bone data structures simplify the mapping between source code and measurable results. Additionally, corresponding structures form the STL are tested in order to estimate the performance impact of using generic implementations.

When iterating through vertices, we intend to extract specific values, *e.g.* a vertex name-space. These data may either be stored together, or in separate lists. Including an optional padding for each element, we test the effect of vertex compression. It is worth noticing that the extra bytes an item is padded with, are not accessed by the benchmark code if otherwise not explicitly stated.

The importance of factoring out highly accessed items in separate lists is therefore analyzed. Implementation details are provided in the benchmark-documentation at `http://folk.ntnu.no/olekrie/ontowiz_cocoon_documentation/dir_2df16cb89d4b3d202e1759e8becf8da0.html`.

5.2 Data Representations in the Context of Ontology Reasoning

Investigating the representations, a fixed number of accesses are performed for each. A problem with analyzing the current results is the complexity of the data structures involved. For each of the sub-cases, we therefore decompose the micro-benchmark results.

The relevance of pre-processing is due to the size and format of the ontologies used as input. In our investigation, we focus on minimizing the memory access overhead, i.e.

$$minimum\left(\frac{\alpha}{\beta_i}\right) \tag{5.1}$$

where α corresponds to the time cost of the arithmetic operation, and β describes the time cost of the operations using memory accesses. The arithmetic operation in this context is the ideal situation, *i.e.* it is not possible on computers to find a set of numbers in memory faster than the time summing the set of numbers. Figure 5.1 illustrates the relative cost of memory accesses:

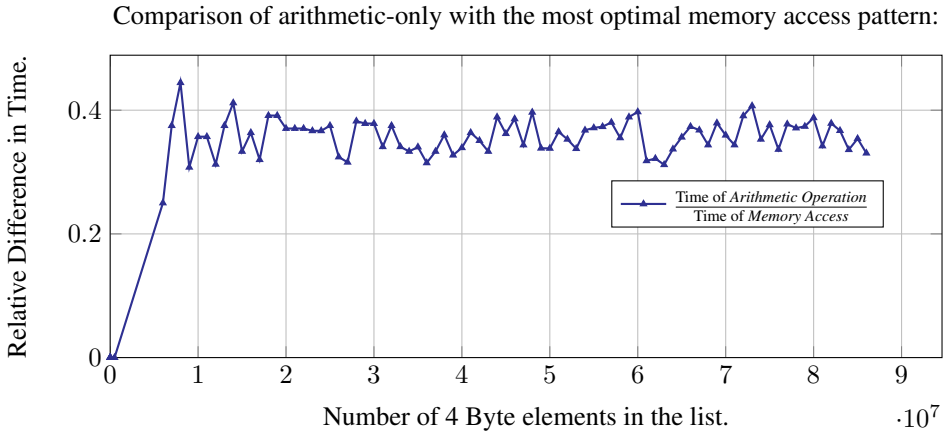

Figure 5.1: The result comparing arithmetic operations with or without additional memory accesses. Only the relative measurement is given, using the ordered list as basis for comparison.

The curve in Figure 5.1 corresponds to the cost of memory accesses, where we explore the relative difference in time when iterating through a fixed space of integers. Comparing the optimal alternative, we use a statically ordered list as comparison. From the figure we observe a performance difference of approximately 2x when memory is accessed, in

comparison with a statically ordered list. We regard this overhead as low, also in the context of discussions provided by *e.g.* Tumeo *et al.*[66] and Ferdman *et al*[26].

The statically ordered list-access-pattern corresponds to our resulting data-structure, *i.e.* of our pre-processing. The benefit of pre-processing is illustrated comparing with alternatives. Figure 5.2 shows the running time for a subsets of our measurements:

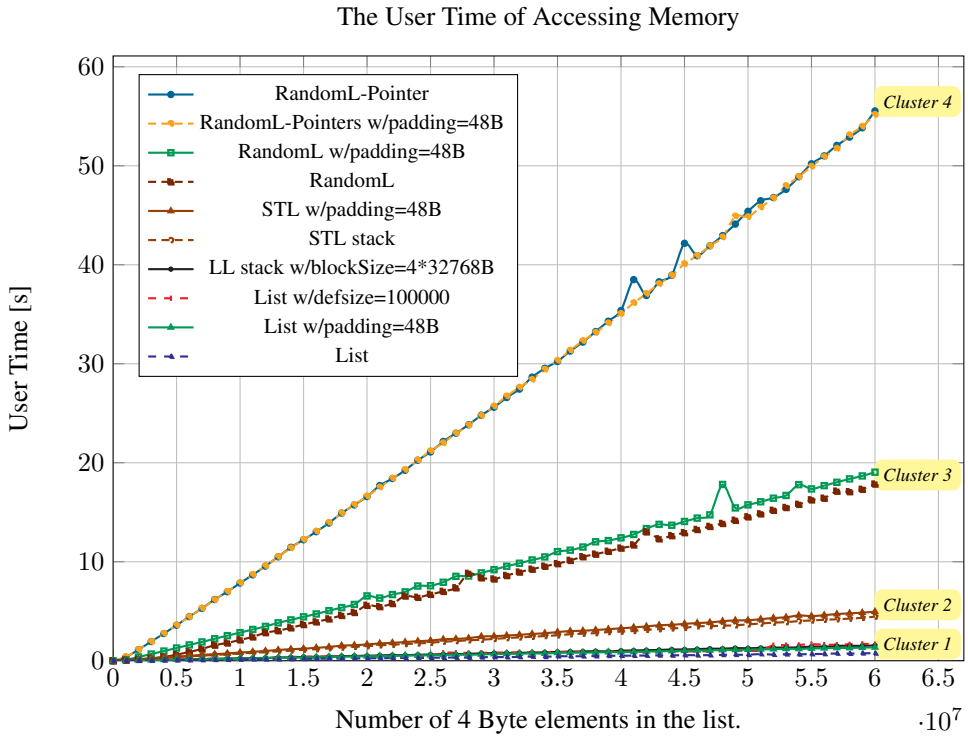

Figure 5.2: Time measurements of different memory access patterns on the biogw system described in Table 4.1 at page 31, having hardware similar to our Dell Laptop. We have accessed 4 Byte elements in range [1,000,000:60,000,000] iteratively increasing the size by 1M for the access patterns that we had a look at. The highlighted yellow text (at the rightmost part of the figure) is a classification of the clusters which the measurements form.

The curved lines correspond to different data access patterns. From the bottom legend in Figure 5.2, we observe the best-performing access patterns. The lines in Figure 5.2 are clustered in sets, where each cluster is highlighted in yellow at the rightmost part of the figure. In the ontology simulation (*i.e.* as seen in Figure 5.2) only a subset of the result is presented. The ontology access cost is illustrated through mapping the user experienced

time along the vertical axis with list length along the horizontal line, *i.e.* for iteration through a graph of similar size.

We observe how the time difference increases with the ontology size:

1. In the ordered list we find the best performing memory access pattern, *i.e.* the most beneficial access pattern. The ordered list is found in the cluster closest to the bottom of the figure. Measurements for the ordered list are included in two flavors: static and dynamic list size. The static list size has optimal length, while the dynamic list iteratively increases it by a factor of two. From the underlying benchmark-data we observe a periodic variety in its performance, which is in correspondence with our expectation. The upper time limit for the latter correspond to the stack-version of the Linked List (LL). The LL consists of static memory blocks knotted together. Experimentation with different block sizes gives a variety in performance. LL performs best when the static block fits into the smallest cache. Details are tabulated at (`https://code.google.com/p/ontowiz/source/browse/#hg%2Fmeasurements_memory`).

2. The second cluster, seen from the bottom, describes the STL stack performance. In our measurements we have experimented with integers of size 4 Bytes, and an optional padding of 48 Bytes: a padding of 48 Bytes is of interest as it correspond to our knowledge from Figure 2.1 at page 8, *i.e.* padding a vertex with two triplets of 12 Bytes for both the vertex child relations and parent relations of a vertex. We observe a negligible difference in user experienced time, *e.g.* a 10 percent difference for 60M vertices. Factoring out the padding (*e.g.* a vertex meta data) in separate structures therefore gives a low optimization benefit. The stack-alternatives differ by 2.4x for 60M elements. Simplifying our experiments, we assume that the LL and STL queues correspond in complexity and operation pattern to the stacks. We therefore infer that the 2.4x improvement also holds for the queues.

3. In the third cluster, accessing data at random is measured. The additional cost of padding data to the vertices corresponds to the STL stacks. In contrast with this and LL, the degree of relative memory accesses (pointers) is lower for this representation. On the other hand, the randomness makes the pre-fetching hard for the system, degrading the performance.

Stressing the importance of the latter observations in the above list, we measure a structure that arbitrary accesses all its memory at random, *i.e.* a representation using both random and pointer based access, which correspond to *Cluster 4* in Figure 5.2. The consequence is a 66x performance-degradation, compared to the optimal case. Similar to the

STL stacks and random lists, the cluster consists of measurements testing the implication of additional padding. In contrast with this, padding extra memory to it has no extra cost. We attribute this to locality of the padding, as it is stored along the same memory reference as the integer we retrieve, *i.e.* the cost of padding is hidden in the unpredictability of access.

The above analysis depends on the result found when measuring the user time for each of the representations. The use of this generalizes the underlying data. Shifting our focus towards the system time in Figure 5.3, we observe that the number of instruction fetches increases when pointer-based representations are used:

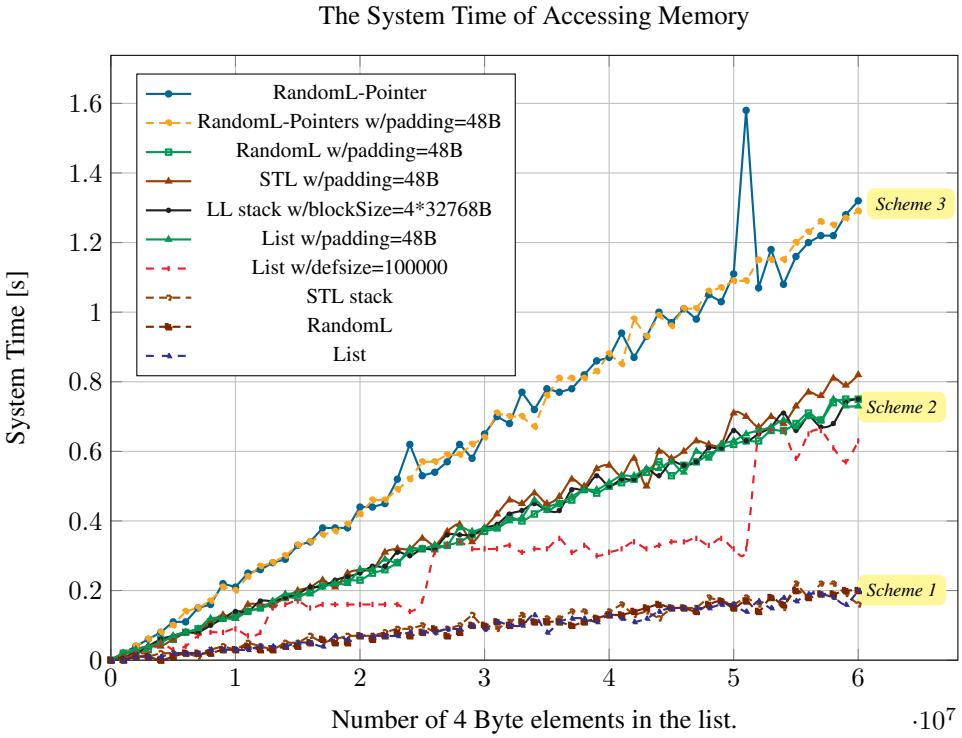

Figure 5.3: System time versus number of 4B elements. The individual graphs are described in the text box in the top of the figure. The plot shows three clusters (Schemes) related to specific operations (see text).

In Figure 5.3 we study the system time, which provide an indication of an access patterns system-dependency and I/O waiting. The best-performing access-patterns have a slight shift, compared to the previous user time. From the above figure we can classify

three schemes:

1. Low intensive pointer based operations, *e.g.* a single pointer, as for "Random list" and "List". Both use fixed-size memory. If we instead dynamically allocate it, a fluctuation between this and the next cluster is observed.

2. Medium intensive pointer based operations: each access requires at least one unpredictable pointer-memory access, *e.g.* as illustrated using "STL Stack" and "LL stack". Interestingly we observe that the number of cache misses increase with the object size. The latter is illustrated when the random structure is provided with a padding. We attribute this to the ratio of cache misses, *i.e.* that the provided padding decreases the MLP-effect.

3. High intensive pointer based operations. The access patterns covered by this scheme, have two unpredictable memory look-ups for each value to retrieve. In contrast with the latter scheme, padding has no effect. This is explained by the lack of locality for the random accesses, *i.e.* with regard to the systems MLP ability.

Understanding the significance of the system time for the overall software performance, we compare it in Figure 5.4 with the user time *i.e.*

$$\frac{Time(system_i)}{Time(user_i)} \tag{5.2}$$

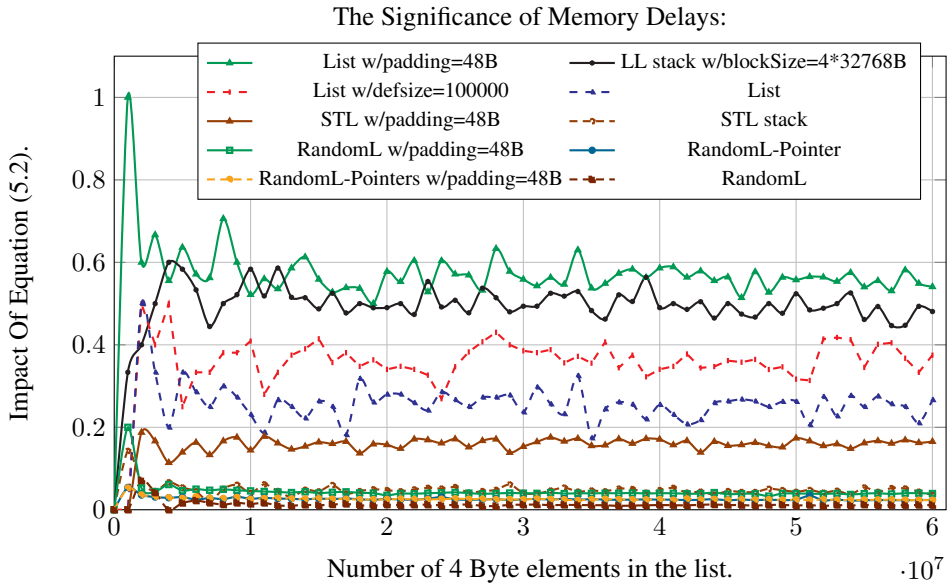

Figure 5.4: The significance of memory delays. In the graph we observe that all measurements are below one, *i.e.* the CPU stalls due to lag in memory transfer. We therefore denote the operations as memory intensive. Of special interest is how the higher complexity of the LL is reflected by a score similar to the ordered list.

In the Figure 5.4 the system-time is mapped against the user time along the vertical axis. The curve located on the top reflects the relation for the LL stack. On the opposite end the random access patterns are found. The individual graphs are described in the text box in the top of the figure.

All the measurements have a values lower than one. The importance of low system-dependency and i/O waiting is therefore out-weighted by the lag in memory retrieval.

The LL-stack is represented by the first curve, counted from the figures top. LL manages relative well to balance the CPU operations with the system load. But as we observe, this does not tell the whole story, *i.e.* the high utilization of system calls reflects the complexity in the source code of LL.

At optimum, the system time would correspond to the user time. The curves fluctuating in the middle of the graph, represent variations of the ordered list. The third curve from the figures top represents the fixed-size memory allocation allocation, *i.e.* the best alternative looses approximate 4x of the CPUs maximum performance.

Reflecting our analysis, we compare in Figure 5.5 the above results with the optimal

case, *i.e.* as before, Figure 5.5 normalizes the results to the optimal case

$$\frac{Time(user_i)}{Time(ordered\_list)} \tag{5.3}$$

where the numerator correspond to each of the identity labels in the graphs above:

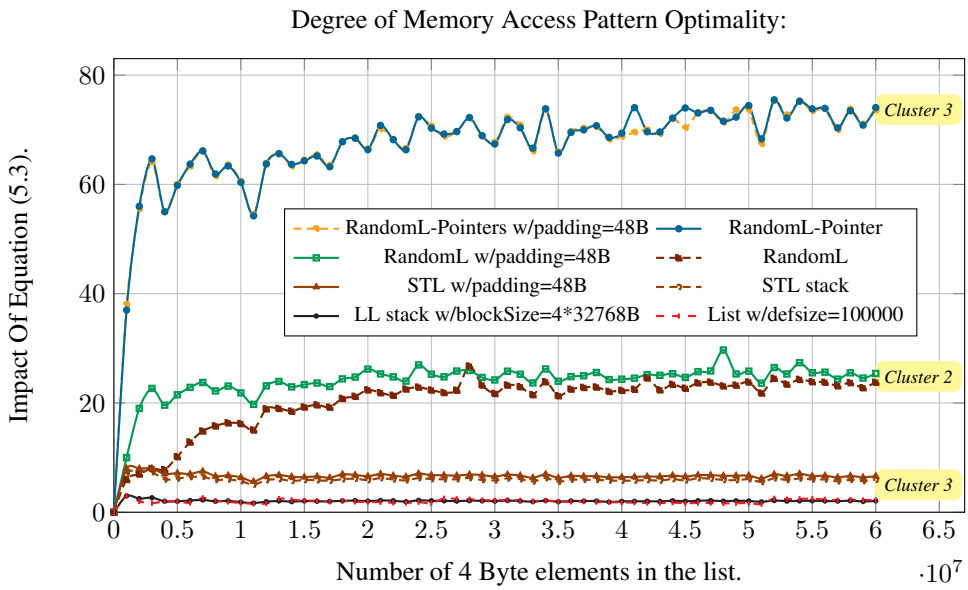

Figure 5.5: The cost of the different memory accesses is illustrated comparing each of the memory access patterns with the optimal case, in which we apply Equation (5.3) on the data sets in Figure 5.2. The individual graphs are described in the text box in the top of the figure.

Returning to the measurement of user time, we here visualize the difference between the alternatives, given the best performing result. The order and clustering in the graph corresponds to our earlier discussion, *e.g.* that the linked list and dynamically sized list are among the best-performers, while the random accesses perform worst. The clear distinction between memory offset and pointer based random access is of interest, which our comparison underlines. Confirming earlier observations, the type of representation is more important than the amount of memory padding added. This importance of access patterns suggests that our focus on data structure is worth investigation further, *i.e.* that benchmarking of the complete set of operations is of interest. These results correspond to those for memory reads, shown in Figure 5.6.

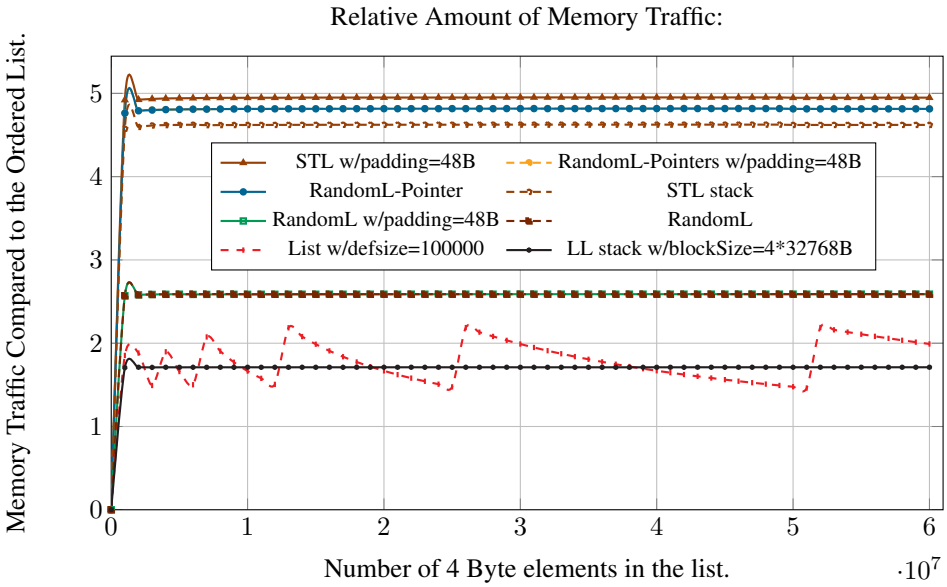

Figure 5.6: The relative memory traffic, compared to the linear lists access pattern. The individual graphs are described in the text box in the center of the figure.

In the graph we observe the relative memory traffic. The lowest curve, represented as a solid black line of asterisks, defines the usage for the linked list, Fluctuating along the latter, is the dynamically sized list, *i.e.* the dashed red curve. Above the fluctuating curve, we observe the random-accessed list using memory offsets. After a leap in the memory consumption, the usage for the random pointer accessed list, the STL stack and STL stack with offset are included.

The periodic fluctuations of the dynamically allocated list is expected, due to doubling of its memory usage when the list size passes a certain threshold. For the other access patterns, we observe a constant relation to their level of memory usage, *i.e.* a vertex accounts for a fixed number of memory reads.

In representations using unpredictable access patterns, we have in Figure 5.6 visualized the insignificance of adding a 48B padding. As we observe, a padding of 48B does not change the number of reads. We assume this is due to the pre-fetching effort giving an overhead, camouflaging our introduced padding.

In contrast the predictable access pattern followed by the STL stack gives an increase in memory usage similar to the time measurements: When a padding of 48B is added, we get an increase of 10 percent in both running time and memory consumption.

The results in Figure 5.7 indicates a compression-technique for random accesses:

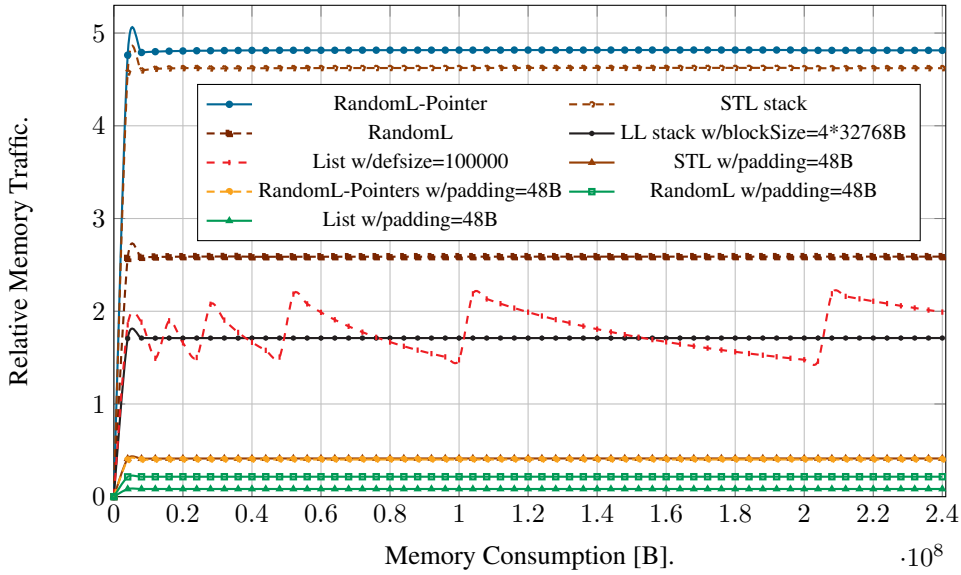

Figure 5.7: The compressing opportunity using short segments of sequential accesses is here illustrated. The individual graphs are described in the text box in the center of the figure. For each of the data structure elements we access the padding, which is the memory overhead we provide for each access.

In Figure 5.7 we have plotted actual memory consumption along the x-axis. The results correspond to difference as is earlier seen. The relation between memory and time consumption is of interests, as it indicates a dependency between cache hits and non-contiguous memory accesses.

Verifying our assumption of a large number of data misses for the random accesses, we compare the number of cache data misses with the optimal case, *i.e.*

$$\frac{\rho_i}{\rho_\mu} \tag{5.4}$$

where ρ corresponds to the number of data misses, μ the ordered representation and i to the access patterns in our benchmarking. Comparing the data sets using Equation (5.4), the result is illustrated in Figure 5.8 below:

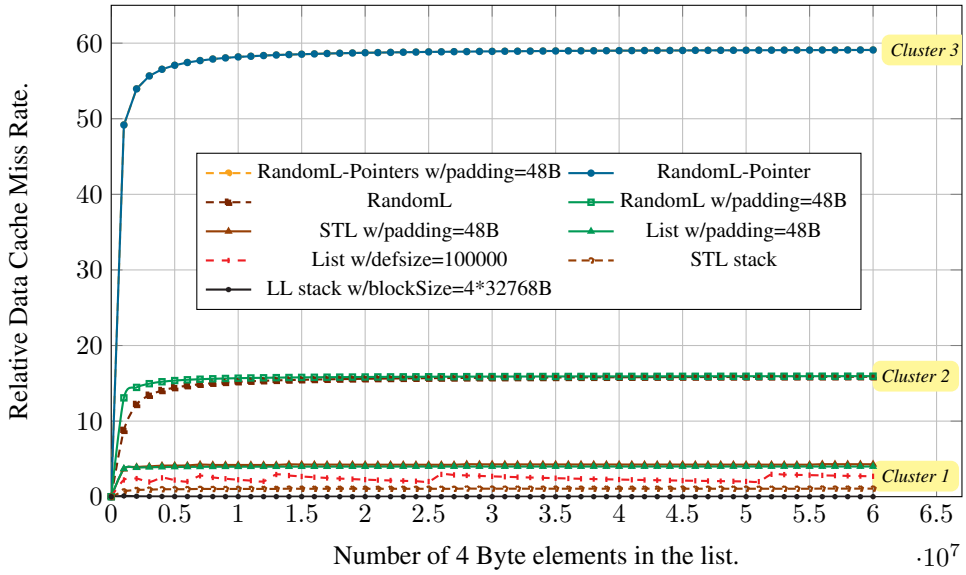

Figure 5.8: Relative data cache read misses, compared to the linear access pattern. The individual graphs are described in the text box in the center of the figure. The plot shows three classes of relationships with high internal similarity: *cluster 1*, *cluster 2* and *cluster 3*.

In the above Figure 5.8 we observe a familiar performance of access patterns, *i.e.* the pointer based random access patterns as the worst, and LL-stack and dynamically list as the best-performing. The value along the vertical axis corresponds to the total relative count of misses in both L1 and L2 cache are compared the the ordered list. The curve pattern we observe indicates that the amount of data found in cache is largely constant.

As earlier, we observe clusters in the graph. The clusters correspond to our earlier observations. We observe a high cache miss rate for the random structures, *i.e.* that data structures with a predictable access pattern behave better compared to the random alternative. This indicates a performance reduction when pre-fetching is used. The result correlates with observation by others, such as Tumeo[66]. On the other hand pre-fetching optimizes highly predictable structures. Our LL-stack implementation illustrates this. We observe that the LL stack has approximately 100 percent hits on its memory accesses. For the latter we infer a relation between the number of reads and cache misses. The optimality of LL visualizes the cost of the the best approach, *i.e.* the upfront allocation of a large buffer.

Comparing the dynamically sized list, we observe a 2.7x increase in cache misses

when 60M elements are accessed. The reason for this relative high rate, is the need for copy operation we perform during list extension, *i.e.* the hardware accesses all elements when the new list is filled with the old values. The periodicity of the curve is continuous.

In the graph, most curves are horizontally aligned, *i.e.* the miss rate stabilizes, and becomes constant. This correspond to both to the relative user and system time, and the amount of memory read performed.

The importance of data cache misses indicates a low benefit regarding structure reuse, *i.e.* to use allocated data containers multiple times, in order to avoid the upfront allocation cost. Figure 5.9 we shows this approach:

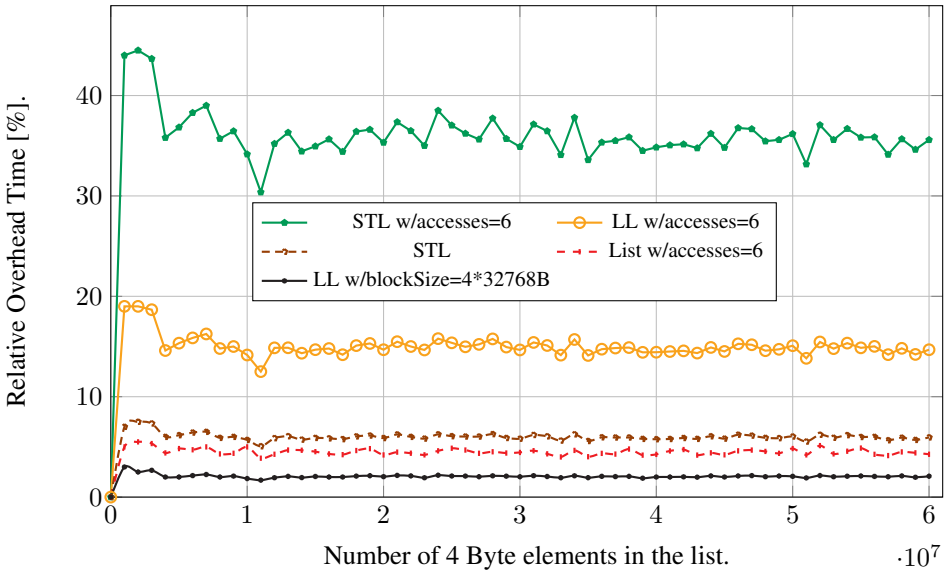

Figure 5.9: Relationship between Overhead time and 4 Byte elements. The individual graphs are described in the text box in the center of the figure. Studying the best-performing structures in our benchmark, we test the benefit of memory re-usage, and compare the results to the linear access pattern.

In Figure 5.9 we compared a subset of the representations. Testing the benefit of structure-re-usage, the same allocated memory was used multiple times for a given access pattern:

1. The STL stack accessed size times are the top curve.
2. Close to the middle of the figure, the LL-stack with six accesses.
3. The STL stack accessed a single time is the densely dashed brown line.

4. Second from the bottom the ordered list accessed six times.

5. At the graphs bottom the STL stack's performance is observed.

In the result data we observe an difference of 4.3x when six times as many items are accessed, *i.e.* a relative improvement for multiple accessed using the latter alternative.

The result corresponds to our expectations, *i.e.* that the upfront cost of memory allocation is small, compared with the cost of cache misses. The latter implies that the cost of memory allocations does not affect the result. Studying the background result material, we observe this holds for all the representations we test. The pre-fetcher is therefore not able to utilize its internal software statistics to optimize the accesses. Further, the difference in ability reusing structures fit our expectations. This due to the number of pointer references required in the access procedure degrades the efficiency of reuse.

5.3 Analysis of the Result Material

We observe that the program speed is clearly affected by cache utilization. Validity of the findings is ensured by a measurement difference of less than 10%, as seen in the measurements (`https://code.google.com/p/ontowiz/source/browse/datasets`). From the results in our analysis of data access patterns, we conclude that complexity in access patterns is an important parameter for the performance of software tools. In this context, the deviation between user and system time highlights the difference between CPU speed and memory fetching.

The two structures we propose, *i.e.* the LL and the ordered lists, perform best in our measurements, both with regard to actual time and cache utilization: the ordered list using statically allocated memory, performs best.

Using different metrics, we have examined tendencies in the results, *e.g.* between the worst-performers of user time and data cache misses. We attribute this to a dependency on memory access patterns. To validate this assumption, random access patterns are investigated for comparison. In the result graphs we find a considerable increase in the load factor for each memory request, *e.g.* as highlighted when normalizing to the optimal access pattern.

The extra cost of accessing vertices with meta data included depends on how it is stored, *i.e.* the number of memory requests which is required need to locate it (*i.e.* the data in question). In our measurements, we placed the padded information adjacent to the data of primary interest. This approach gave a difference of 10 percent when the data set increased by a factor of 13x. We therefore regard memory adjacency to be of higher importance than compression techniques.

Representation	Access Patterns		Time Cost		Impact Of			
	Explicit Storage	Random Access	User	System	Padding	Def. Size	$(1-\chi)$	CPU stalls
STL Stacks	yes	no	low	med.	high	low	low	med.
LL Stacks	yes	no	low	med.	high	low	low	low
Random Lists	yes	yes	med.	low	med.	low	med.	high
Random Lists	no	yes	high	high	low	low	high	high
Ordered List	yes	no	low	low	high	nil	low	med.
Reversed List	yes	no	low	low	high	nil	low	med.

Table 5.1: Parameters of two micro-benchmarks. The terms *low*, *med.* and *high* are used for classification. The terms reflect the relative importance, given the complete set of measurement data. Column one are the identities we have used to explain the representation of memory access patterns, which is described by its access patterns: column two describes the case where an additional pointer based memory reference is required to identify a value, while column three if the accessed memory addresses are adjacent. The user time is given in column four and system time in column five. In column six to column nine we include the impact of specific properties: column six compare the implication of padding (*i.e.* compression), column seven the consequences of a allocated memory space at start of executing the software, column eight the likelihood of a cache miss and column nine the chance of CPU stalls.

In Table 5.1 we observe the efficiency of the access patterns. A comparison of the time cost columns provides an overview of important observations. Our results illustrate the benefit of graph pre-processing in the context of ontology reasoning. The cache-efficient data structure utilizes the performance of commodity hardware, as documented in the results. Approximating a value for the difference, Equation (4.4) is used:

$$\frac{\psi_{cache}}{\psi_{memory}} \approx \frac{0.84}{38.04} = 0.022 = 2.2\% \tag{5.5}$$

In Equation (5.5) the ordered list is used as an estimate of ψ_{cache} and the pointer-based random list for ψ_{memory}, *i.e.* in correspondence with the importance of explicit storage and access-pattern. The numbers correspond to values measured on systems of interest (see Table 4.1). As the measured systems differ in their cache sizes, we assume that the cache size is of minor importance compared to the access pattern.

With the presented approach, we identify a reduction in the complexity of memory accesses. The observation is of importance for tools were the number of memory accesses is (considerably) larger than the number of arithmetic operations, such as for ontology

reasoning. We therefore regard non-random accesses in ordered list with explicit storage as a principle component in ontology reasoning tools.

The difference we have identified (*i.e.* between the random and ordered accesses) indicates a preference for it (due to the approximate performance-gain of 50x for a random-access-application which changes its access-pattern to the ordered-access-path). Given this knowledge, we suggest application of this storage-scheme in memory-intensive software, such as the cocO(n) software described in the next section.

Chapter 6

cocO(*n*); A High-Performing Ontology API

cocO(*n*) in the core of our approach for high-speed ontology reasoning. The library (*i.e.* cocO(*n*)) applies knowledge gathered in the master thesis' previous sections. What we intend to cover in this chapter is complex. Simplifying the complexity of our algorithms, without loosing its important details, cocO(*n*) is an algorithm/software/library where:

1. The algorithms that contribute to the high performance are knotted into the cache-efficient storage scheme by accessing memory through ordered list accesses, *i.e.* using the best access pattern as described at page 46. Examples of procedures that utilize the memory-scheme in cocO(*n*) are those calculating the longest path or intersection/union of vertices.

2. The cache-efficient storage schemes are built during the pre-processing, where each vertex in the ontology is provided with an explicit coverage, *i.e.* a representation of its (the vertex) implicit relations. This representation holds both the set of non rule based paths connecting a vertex to any of its ancestors, and the set of ancestors/descendants, *i.e.* as we stated at SR_7 in Table 2.1 at page 18.

3. The cocO(*n*) library is accessed through its Application Process Interface (API). The API of cocO(*n*) provides access to complex ontology properties. Examples of such are intersection and union of a terms ancestors or descendants, transitive closures, longest paths to a root vertex, *etc*. Onto*Wiz* interacts with cocO(*n*) using this API.

Reaching this goal, this chapter is organized starting with a gentle introduction to how cocO(*n*) treats ontologies, then moving into algorithmic descriptions, before giving conclusive evidence as to why cocO(*n*) qualifies as a *high-speed ontology reasoning tool*. In short this chapter approaches the internal milestones in the below order:

1. a brief overview of the ontology-input for cocO(*n*)'s algorithm: while onto*Wiz* works on the complete ontology, cocO(*n*) only looks upon subsets of it.

2. steps of the pre-processing, which is the process translating an ontology into data-structures for short look-up-time.

3. support rule-based extensions and contractions, such as transitive closures and transitive reductions: presents algorithmic extensions of the pre-processing which does not increase the time of the pre-processing of an ontology.

6.1 The ontology given as input

An ontology in its simplest form is a set of unrelated vertices (*i.e.* without relatedness). As an example, we look at a small subset of an on ontology, *i.e.* as seen in Figure 6.1(a).

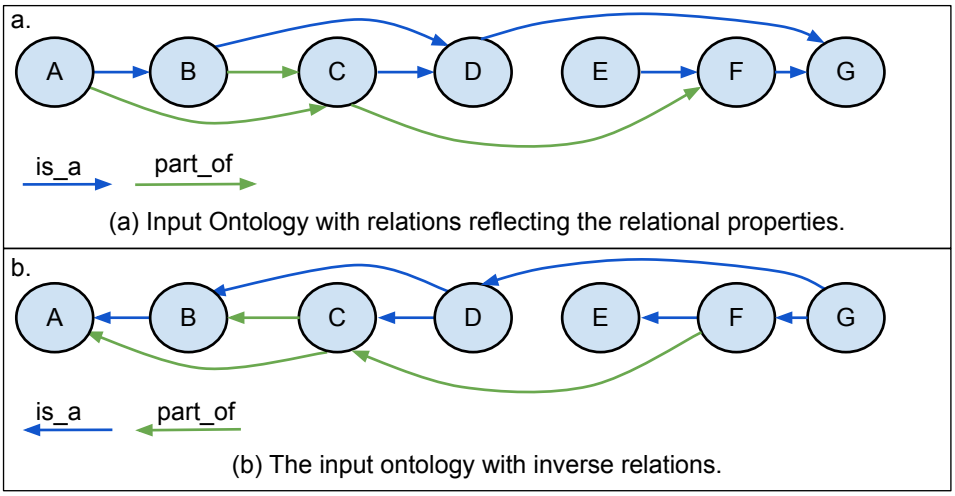

Figure 6.1: Schematic ontology depiction. We observe a set of vertices (circles) and arcs (arrows): cartoon show both ontology's two relation type, *i.e.* the `is_a` and `part_of` relation type. The cartoons refer to the same ontology. In sub-figure (a) the ontology with the correct arc-direction is included. In sub-figure (b) the mirror of the input ontology is provided.

The ontologies in Figure 6.1 illustrate what cocO(*n*) takes as input: we observe that inverting the arc-direction of the first ontology yields the second ontology (*i.e.* Figure 6.1(b)). For the two ontologies, which both are without cycles, *i.e.* directed a-cyclic graph (DAG), we assign the properties of reflexivity, anti-symmetry and transitivity to each of the relations, which correspond to the definition of a partially ordered set (poset)[69]. When translating an ontology into a poset we loose important attributes of the ontology, such as the underlying meaning of inferred relations. The application of a poset allows us to construct super-sets of ancestors and descendents, which we use in operations such as building rule-based cover, sub-ontologies, comparison of sub-ontologies, *etc*. To simplify our discussion we assume that an ontology is a poset if we not explicitly states the properties of the ontology's relation types (*e.g.* anti-symmetric without the transitive property).

The ontology which cocO(*n*) takes as input is a limited set of a biomedical ontology. We observe this by the attributes which are present in Figure 6.1, exemplified by the Example-Ontology 6.1:

Example-Ontology 6.1: Extract from taxonomic-rank.obo.

```
[Term]
id: TAXRANK:0000001
name: phylum
synonym: "division" EXACT []
xref: http://rs.tdwg.org/ontology/voc/TaxonRank#Phylum
xref: NCBITaxon:phylum
is_a: TAXRANK:0000000 ! taxonomic_rank

[Term]
id: TAXRANK:0000000
name: taxonomic_rank
def: "A level of depth of a taxon in a taxonomic hierarchy."
    [TAXRANK:curator]
```

Comparing Example-Ontology 6.1 with Figure 6.1 we observe that cocO(*n*) performs reasoning on a limited set of the properties of an ontology. From our knowledge of the limited ontology which cocO(*n*) takes as input, we are ready investigating the properties of our high-speed ontology handling software.

6.2 The Pre-Processing: Algorithms and Proofs of Correctness

The high performance of cocO(*n*) is achieved by the pre-processing. The pre-processing is about making all vertices aware of its connections. The connections are those other vertices a vertex is linked to, *i.e.* given one or more relations, such as

$$A \xrightarrow{is\_a} B \xrightarrow{part\_of} C \xrightarrow{is\_a} D \tag{6.1}$$

which implies that vertex *A* is connected to both vertex *B*, vertex *C* and vertex *D*. This is understood as an explicit representation of the implicit knowledge in Equation (6.1). Without translation-rules, the relations in Equation (6.1) do not provide any deeper meaning, *i.e.* of the implicit relations in the underlying data. For this purpose rules of inference are needed, *e.g.* those covering transitivity and anti-symmetry: details of relation types were given in section 2.2 at page 11. Providing an example of implicit relations, we apply the rules of transitivity and anti-symmetry to the explicit relations in Equation (6.2):

$$A \xrightarrow{is\_a} B \qquad A \xrightarrow{part\_of} C \qquad A \xrightarrow{part\_of} D \tag{6.2a}$$

$$B \xrightarrow{part\_of} C \qquad B \xrightarrow{part\_of} D \qquad C \xrightarrow{is\_a} D \tag{6.2b}$$

Equation (6.2) transforms the implicit representation of the relations from Equation (6.1) into its explicit representation. We call the transformation an *expansion* of the ontology, as the number of ontology-relations increases. When inference-rules are applied to the set of related vertices, the result is a deeper understanding of each vertex (*e.g. term* or *instance*) in the ontology. Our goal is building the explicit representation (of the ontology) during the pre-processing. We expect the pre-processing to result in explicit representation of the implicit knowledge. This explicit representation of implicit knowledge is important for high-speed reasoning.

High-speed reasoning depends on easy access to information, which in this context is about giving each vertex an awareness of its implicit relations. From our ontology and memory access benchmarks we have seen preferences for ordered representations of an ontology's internal structure. The high-speed access to query results is due to our pre-processing: while traversing the ontology *awareness*, the implicit knowledge of relatedness is propagated along the arcs. Achieving awareness throughout the ontology therefore requires all vertices to systematic be visited. An example of such a systematical approach is seen in the ontology pre-processing in Figure 6.2:

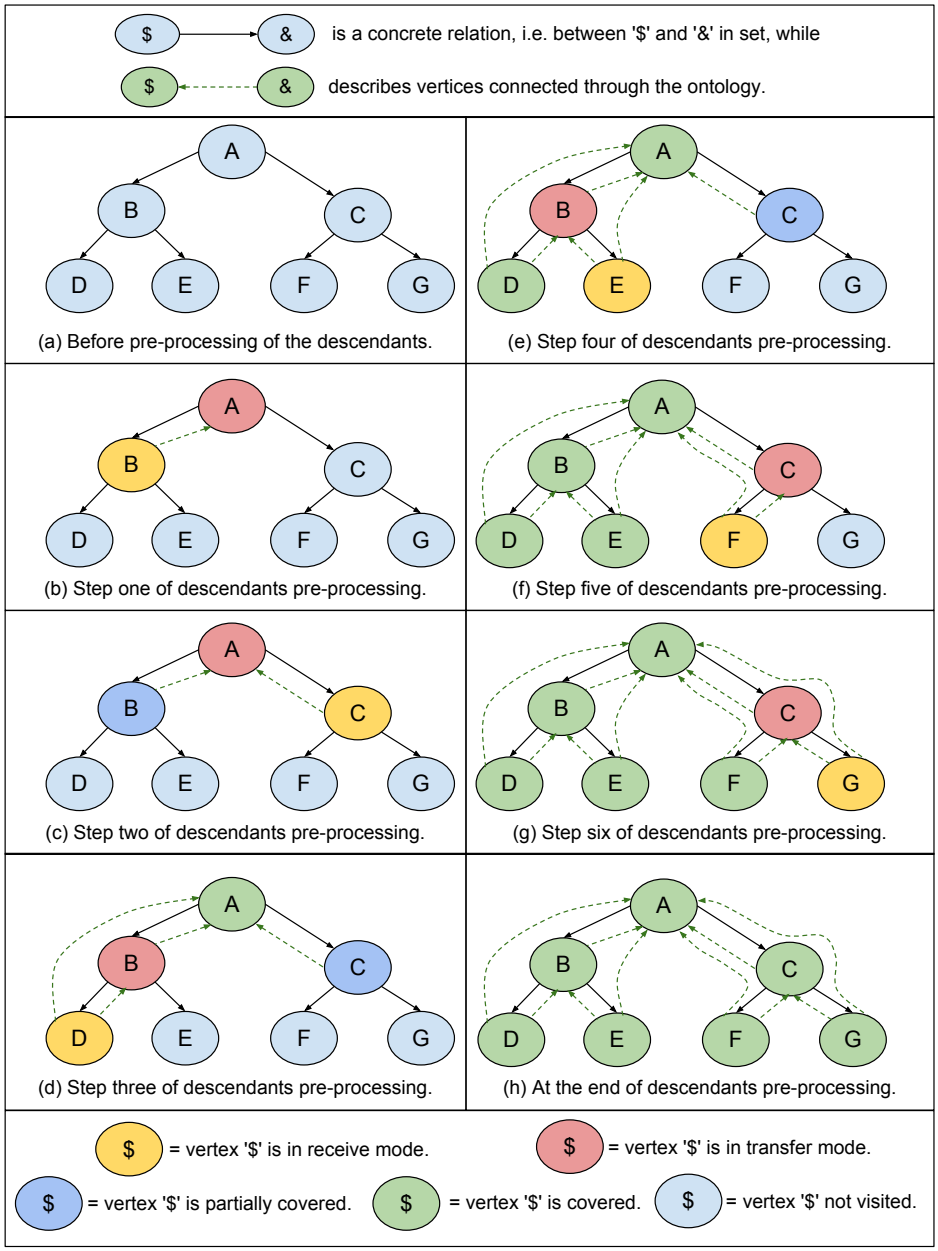

Figure 6.2: Schematic ontology pre-processing. We observe the steps of the cocO(n) algorithm. The input is an ontology, shown in sub-figure (a). At the end of of the pre-processing, all vertices have awareness of their relations, *i.e.* as illustrated in sub-figure (h). In each of the sub-figures $[(b)....(g)]$ a vertex updates a child with its own awareness. An example of such an awareness-transfer is seen in sub-figure (d), were vertex *C* becomes aware of its connection to vertex *A*. In sub-figure (c) vertex *B* is colored dark blue. With this we mean that its awareness is not completed: after it has received all the data from *C* it is completely covered, *i.e.* as illustrated in sub-figure (d).

The result of the pre-processing algorithm is seen in Figure 6.2. The figure conveys an high-level understanding of the cocO(*n*) algorithm. We observe the sequence by which the vertices were visited. The sequence corresponds to the BFS algorithm. The BFS algorithm is explained in chapter C at page 123, while a brief summary of our usage of the BFS is provided in section 3.3 at page 24. The dashed arrows in Figure 6.2 propagate the knowledge of the implicit relations:

- the continuous black arrows are the ontology's explicit relations;
- the dashed green arrows are understood as arrows propagating relational awareness;
- the colors are the awareness-state of each vertex.

In Figure 6.2 we observe that the dashed arrows never penetrate a vertex. We understand the arrows as those propagating information. Therefore, if the vertices found along the arrows length had penetrated the arrow, path-awareness would have been the result. By not penetrating the vertices along the arrows length, a vertex knows nothing about intermediary paths. Path-awareness is interesting as it enables finding the shortest path between a set of vertices. Achieving this, we slightly extend our procedure by adding an extra set of arrows with the task of remembering the vertices along the paths. This extension requires cocO(*n*) to store three types of knowledge:

1. The set of *unique vertices* explicitly or implicitly related to a vertex, henceforth, its *unique set*.
2. Subsets of unique sets, *restricted* by relation type or sub-name-space.
3. Sets of *concrete paths*, storing not only the related vertices, but also how they related (*i.e.* their intermediate vertices).

Highlighting the concept, we investigate the pre-processing in the mirror ontology of Figure 6.2, as seen in Figure 6.3:

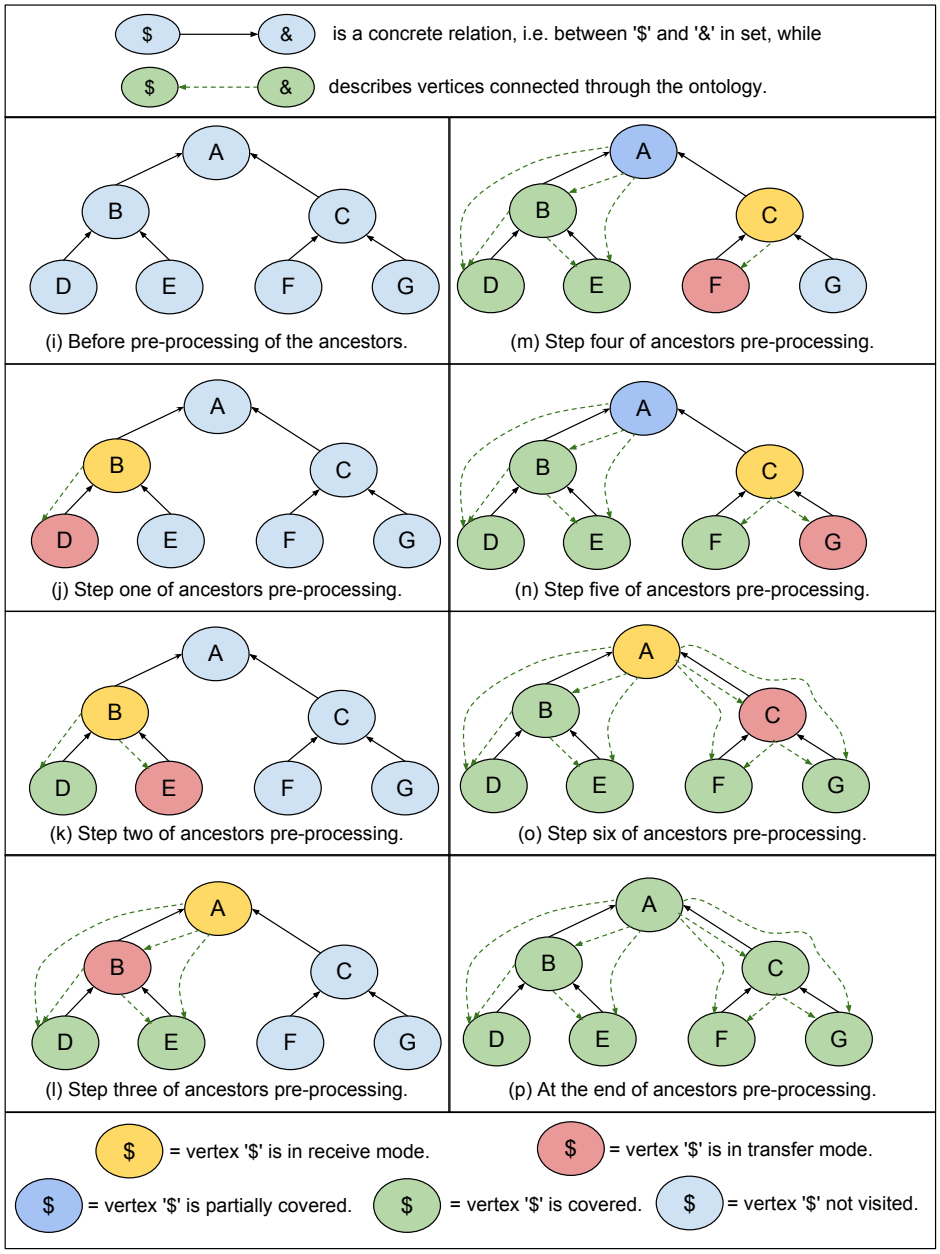

Figure 6.3: Schematic ontology pre-processing. Overview of the pre-processing steps when starting from the leafs of an ontology, which is the mirror of the ontology described in Figure 6.2.

From Figure 6.3 we observe the result of the pre-processing algorithm of cocO(n) when

the algorithm starts from the leaf vertices, which is the mirror of the ontology described in Figure 6.2. In the applied algorithm a vertex parent knows everything about its descendents, but is limited about its parents. When combining the knowledge from Figure 6.3 with 6.2 we observe that this, *i.e.* the limited knowledge of a vertex descendents and ancestors, is true until the two pre-processed input-ontologies are combined. An issue of confusion in this context, is that the ancestors pre-processing describes the set of descendants, and the descendants pre-processing describes the ancestors. If this during the reading becomes unclear, remember the opposite direction of the dashed versus the continuous arrows seen in Figure 6.3: translating Figure 6.3 into a tangible algorithm, the result is Algorithm 6.1:

Algorithm 6.1: A tangible algorithm for making each vertex aware of its relations.

```
/* Globally accessible result containers. */
global inputParents ←new Set();
global inputChildren ←new Set();
global parentsReceived ←new Set(); // the number of received
    parents.
/**
   brief: The main-method for the pre-processing.
   param: <cocoOn> the object containing the set of inputs.
**/
class cocoOnPreProcessing(cocoOn) {
 /* Set the algorithmic input. */
 inputParents ←cocoOn.getInputParents();
 inputChildren ←cocoOn.getInputChildren();
 /* Declare internal variables. */
 global childCount ←new set(); // number of times a child is added.
 global queue ←new Queue; // should be empty at end of for-each.
 global rootSet ←new Set();
 /* Builds the set of roots. */
 for each root ∈ {inputChildren} {
  if(inputParents[root] = ∅ ) {
   //! A root is a vertex without outgoing relations:
   rootSet ←{rootSet ∪ root}; // updates the set.
  }
 }
 //! Start with vertices already covered, i.e. the roots:
 for each root ∈  rootSet {
  //! The root adds data; colors it red.
  for each relation ∈ {inputChildren[root]} then {
   //! The child is aware of a parent; colors it with yellow;
   global child ←relation.tail; // color the child in yellow.
   queue ←{relation ∪ queue}; // updates the set.
   childCount[child] += 1; // The number of children.
  } // The root is now safe to extend; color it green.
 }
 //! Sets the ancestor coverage for each vertex:
 setAncestorCoverageForEachVertex(childCount, queue);
```

Algorithm 6.1 both initiates the result-containers which are used to answer queries in cocO(n), and iterate through the roots of the ontology, *i.e.* the vertices without ingoing arcs. Each of the roots is added to the queue (which is accessed in a first-in-first-out access pattern). When the roots are visited, cocO(n) builds the ancestor coverage for each vertex,

as explained in Algorithm 6.2:

Algorithm 6.2: Sets the ancestor coverage for each vertex.

```
/**
   brief: Sets the coverage for each vertex.
   param: <childCount> counts the number of times a vertex is added.
   param: <queue> stores the next relations to be processed.
**/
void setAncestorCoverageForEachVertex(childCount, queue) {
 //! Iterate through the internal vertices using a modified BFS:
 while(queue ≠ ∅) {
  //! Gets relation at front of queue:
  global transferRelation ←queue.front;
  global head ←transferRelation.head;
  global type ←transferRelation.type; // i.e. the relation type.
  queue ←queue - transferRelation; // removes relation from queue.
  if(childCount[head] ≠  inputParents[head].size) then {
   //! The parent is not covered:
   add the transferRelation to the queue; // color it blue.
  } else { // we color the tail with red;
   parentsReceived[tail]++; // received one more parent relation.
   global tail ←transferRelation.tail; // the relation type
   //! Update the data-structures:
   updateAnswersToReasoningTasks(head, tail, type);
   //! First add the children when the tail has a parent-coverage:
   if(parentsReceived[tail] = inputParents[tail].size) then {
    //! Iterate through tails immediate descendants:
    for each relation ∈ {inputChildren[tail]} then {
     //! Update the child with the tails own awareness:
     queue ←{relation ∪ queue};
     childCount[child] += 1; // The number of children added.
    } // The tail-vertex is now safe to extend; color it green.
   }
  }
 }
}
```

Algorithm 6.2 covers all of the ontology's relations: each vertex stores the set of implicit relations. We denote this process as a "transfer-receive" operation, as the implicit relations are received from the parents of a vertex through a data transfer operation. The use of a queue makes Algorithm 6.2 similar to the BFS algorithm, where the BFS algorithm describe the order in which the vertices are processed in the transfer-receive operation.

The set of implicit relations are stored in function `updateAnswersToReasoningTasks(head,tail,type)`, which is called from line 21 in Algorithm 6.2. The data-structure in which cocO(*n*) stores the implicit relations is a set of lists with relations ordered to support memory efficient access of the result. When a vertex receives a set of implicit relations from a parent, cocO(*n*) uses three different operations to concatenate the received set with existing sets of implicit relations:

- "$\cup_s$" is a union-operation applied with sorting. An example is $\upsilon = \alpha \cup_s \beta$, where υ holds holds a sorted set of both α and β.
- "$\cup_a$" is a binary operator appends the rightmost item to the end of the leftmost set. An example is $\upsilon = \alpha \cup_a \beta$, where $\upsilon = \{\alpha\beta\}$.
- "$= \cup$" is a syntactic sugar for concatenation of sets. An example is $\upsilon = \cup\, \alpha$ which corresponds to $\upsilon = \upsilon \cup \alpha$.

In the structure of our correctness arguments, which the above notation serves, we generally follow the approach of stating properties of ontology subsets that the pre-processing should compute. We relate the properties of ontology subsets to expressions which are derived from specific points in stated algorithms, such as Algorithm 6.3 for for updating the data-structures holding the implicit relations:

Algorithm 6.3: Logic for storing important results of the pre-processing

```
/* Globally accessible result containers. */
global uniqueSet ←new Set(); // the set of ancestors for each
    vertex.
global restrictedSet ←new Set(); // the restricted set of ancestors
    for each vertex.
global pathSet ←new Set(); // the vertices simple paths to the
    roots.

/**
   brief: Logic for storing important results of the pre-processing.
   param: <head> The head of a given relation, i.e. the parent
       vertex.
   param: <tail> The tail of a given relation, i.e. the current
       vertex.
   param: <type> The relation type linking the head with the tail.
**/
void updateAnswersToReasoningTasks(head, tail, type) {
  //! Get the the parents complete coverage, and sort (s) the set:
  uniqueSet[tail] ←uniqueSet[tail] ∪s head ;
  //! Concatenate the tails restricted set with the parents:
  restrictedSet[tail][type] ← ∪s head ∪s restrictedSet[head][type];
  global space ←head.subNameSpace; // i.e. for the restricted space.
  restrictedSet[tail][space] ← ∪s head ∪s
      restrictedSet[head][space];
  for each path ∈ {pathSet[head]} then { // update the path set:
    global extended ←path ∪a (head, type); // extends the path.
    //! Adds the new path to the set:
    pathSet[tail] ←pathSet[tail] ∪ extended;
  }
}
```

Algorithm 6.3 updates three data sets. The algorithm is performed on both ontologies, *i.e.* the two ontologies respectively representing the ancestors and descendants, which we explained in Figure 6.1 at page 50. The three data sets which are updated by Algorithm 6.3 are:

1. the set of *unique vertices* (*uniqueSet*), which for each vertex stores a sorted representation of all the ancestors in the ontology.

2. the set of *restricted vertices* (*restrictedSet*), which is similar to the set of unique vertices, with the exception that the set of ancestors are limited to a given type, *i.e.*

a specific relation type or sub-name-space which must hold for all the ancestors in the set.

3. the set of the *concrete paths* (*pathSet*), which stores a vertex' concrete path(s) to the root(s). The concrete paths are only built for the descendant's ontology-representation, *i.e.* iterating backwards from the leafs of the ancestors ontology-representation.

The above listed data sets are the core of our reasoning-approach. They therefore need evaluation of their formal correctness in order to know that they formally cover associated tasks, such as the operations of union, intersection and shortest/longest path. In our notation

$$v_i \subset_\beta v_k,\ \{v_i, v_k\} \in V \tag{6.3}$$

denotes a vertex v_i which is the descendant of vertex v_k by the relation type β.

6.2.1 Building the Set of Unique Vertices

The set of unique vertices are used in operations such as intersection and union. The set of unique vertices consist of all the ancestors for a given vertex, and is formally defined[65] as

$$Ancestors(v_i) = \left\{ v_k \in V \;\middle|\; v_i \subseteq_\beta v_k \;\&\&\; v_k \neq v_i \right\} \tag{6.4}$$

where v_i and v_k are vertices in the Ontology V, and β an arbitrary relation type connecting vertex v_i to vertex v_k. When building the sets of unique vertices (*i.e.* of the given ontology's ancestors), Algorithm 6.1 is run twice, in order to handle the two representations of the ontology. We therefore have two disjoint sets of unique vertices, *i.e.* for the ancestors and descendants, denoting a vertex' set of ancestors as the vertex' complete cover of its parents:

Lemma 6.2.1 (A parent covers the ancestors)**.** *For each visited vertex, a complete coverage for the data sets exists, i.e. from the parents, through its parents intermediate ancestors, and to the roots.*

We will later prove the correctness of Lemma 6.2.1, but first a summary of the algorithm's properties.

From Lemma 6.2.1 and the properties of Algorithm 6.1, we observe that each vertex is aware of its relations. The awareness (*i.e.* the relational knowledge) is stored in sets.

The sets are built iteratively. Structural features of the ontology, which we discussed in section 2.3 at page 16, suggests the use of precomputed sorted lists. Reducing the time cost of memory accesses, the access pattern of our sets will therefore replicate the ordered list accesses, *i.e.* as seen at page 46. The benefit of applying the sorted property to the unique set, constitutes the efficiency when searching for a vertex or comparing sets. A set *M* of unique vertices is *sorted* with respect to vertex identifiers *k*

$$\forall_{k\in[2,|M|-1]} v_{k-1} < v_k < v_{k+1} \tag{6.5}$$

We require that all unique sets are sorted. The sorting is performed for each parent-child relation by line 14 in Algorithm 6.3 (*i.e.* list concatenated with the $\cup_s$ operator). The child relations of a vertex are added at the lines [17....31] in Algorithm 6.2. The building of the unique vertices is therefore an algorithm for stepwise add/merge. This makes our algorithm similar, but not equal, to Dijkstra's algorithm, which was discussed in section 3.3 at page 24. Formalizing our approach, we combine the knowledge from Lemma 6.2.1 with Algorithm 6.1, Algorithm 6.2 and Algorithm 6.3:

$$\Phi\left[v_i\right] = \emptyset \; v_i \in Roots \quad \text{initializes root/start vertices.} \tag{6.6a}$$

$$\Phi\left[v_i\right] = \bigcup_s \underbrace{\left\{ \forall_{v_k\in\pi(v_i)} \left\{ \underbrace{v_k \cup_s \Phi\left[v_k\right]}_{\Delta_2} \right\} \right\}}_{\Delta_1} \tag{6.6b}$$

where

- Φ defines the unique set (*i.e. uniqueSet* in the algorithm,
- v_i a vertex in the ontology (graph),
- $\pi(v_i)$ the set of parents for v_i,
- $\cup_s$ and is a union-operation applied with sorting,
- $\pi(v_i)$ the set of parents for v_i,
- Δ_1 correspond to the lines [25 .... 29] found in Algorithm 6.2, and
- Δ_2 correspond to the line of code 14 found in Algorithm 6.3.

Comparing Equation (6.6) with Algorithm 6.2 and Algorithm 6.3, which is their formalized representation, we note that both the algorithms and the equation uses the $\forall$ (*i.e.* for-all) approach; difference is in their order. The algorithms implements Lemma 6.2.1: while Equation (6.6) implicitly assumes that a vertex has its parents covered, *i.e.* Algorithm 6.3 explicitly performs the task. We therefore assume that the vertices in the ontology have a complete coverage of its ancestors denoted by Φ in Equation (6.6). From the assumption we hypothesize that a vertex has a complete coverage of its ancestors, which we prove below:

Proof of vertex coverage. We will now show that Algorithm 6.2 and Algorithm 6.3 visit every vertex in the ontology, and construct complete paths to their roots, proving Lemma 6.2.1:

Reference-id	Statement	Reference
OLC_1	Add its children to the queue after it has received all of its parents vertices.	Algorithm 6.2, line 23.
OLC_2	Stops modifying its internal data-structures (representing the set of simple paths and set of unique vertices) when the vertex child-relations are added to the queue: a vertex first starts the usage of a parents data-set when the parent is covered.	Algorithm 6.2, lines [17....31].
OLC_3	Propagates the parents knowledge (*i.e.* the set of data-structures) into all of its children: a vertex' parents therefore have a complete coverage for their own parents.	Algorithm 6.3.

Table 6.1: The set of observations used in the process of deducing the correctness of Lemma 6.2.1. We observe that the reference-id (in the leftmost column) consists of a prefix (*i.e.* OLC) and a number: construction of the proof will refer to the reference-id.

Table 6.1 lists three implications of the algorithms in question. The implications support the proofs of the generalized BFS algorithm[46, 55], *i.e.* that all of the vertices in the ontology are visited. A simplification of the latter is seen in Equation (6.7)

$$\forall \left(v_i \in \tilde{\pi} \left(v_k \right) \right) \tag{6.7}$$

where v_i and v_k are vertices in the ontology (graph), and $\tilde{\pi}(v_i)$ represents the set of children for v_k. The observation in Equation (6.7) implies that a vertex has received data from all of its parents, *i.e.*

$$\forall \left(v_k \in \pi \left(v_i \right) \right) \tag{6.8}$$

where v_i and v_k are vertices in the ontology (graph), and $\pi(v_k)$ represents the set of parents for v_i. The latter (*i.e.* that a vertex has received data from all of its parents) is in accordance with OLC_1.

To prove the correctness of a vertex coverage, we observe that each of the vertex' parents has a set of unique vertices representing its ancestors, where the received set of unique vertices is copied into the vertex' own data-structure. Correctness of the operation require that (a) the parent does not modify its own data-structure and (b) that copy-operation is correct. Requirement (a) is covered by OLC_2 while requirement (b) is covered by the properties of the union-operator. If the parent has coverage for all its ancestors, the coverage of an arbitrary vertex is correct. From the additional knowledge stated in OLC_3 the latter is verified. Lemma 6.2.1 is therefore proven. ■

The set of distinct vertices support operations such as intersection and union. The intersection of vertices' unique sets is understood as

$$\psi(v_i, v_k) = \{ y \supseteq_\beta v_k \;\&\&\; y \supseteq_\beta v_i \} \tag{6.9}$$

where v_i and v_k are the vertices which get intersected, β an arbitrary relation type connecting vertex v_i to vertex v_k, and ψ the intersected set of v_i and v_k. The intersection may use either the complete set of unique vertices, or the restricted set (*i.e.* a subset of the unique vertices). The sets may use either the ancestors or the descendants' version of the ontology. For brevity we omit including the algorithm for intersection. From Lemma 6.2.1 we conclude that a vertex has complete coverage of its ancestors. The operation of intersection is performed as

$$\psi(v_i, v_k) = \Phi \left[v_i \right] \cap_s \Phi \left[v_k \right] \tag{6.10}$$

where $\cap_s$ is the set-operator resulting in a sorted subset of distinct elements from the operations.

Corollary 6.2.1 (Correctness of the intersection algorithm)**.** *The property's of the intersection is defined in Equation* (6.9)*. From the definition in Equation* (6.9)*, we observe that finding the intersection* $\psi(v_i, v_k)$ *require examining all ancestors of vertices* v_i*,* v_k*.*

Lemma 6.2.1states that these are completely covered by the unique sets $\Phi\left[v_i\right]$, $\Phi\left[v_k\right]$, *so it follows that their intersection is computed in Equation* (6.10).

Similar to the operation of intersection is the task of calculating the union between two sub-ontologies of vertex v_i and v_k:

$$\tilde{\psi}(v_i, v_k) = \left\{ y \supseteq_\beta v_k \;\middle|\middle|\; y \supseteq_\beta v_i \right\} \tag{6.11}$$

where v_i and v_k are the vertices which receive the union, β an arbitrary relation type connecting vertex v_i to vertex v_k, and $\tilde{\psi}$ is the union of v_i and v_k. Applying the definition in Equation (6.11) to the pre-processed data-structure in cocO(*n*), the algorithm is described as

$$\tilde{\psi}(v_i, v_k) = \Phi\left[v_i\right] \cup_s \Phi\left[v_k\right] \tag{6.12}$$

where $\cup_s$ is the set-operator resulting in a sorted super-set of distinct elements from the operations.

Corollary 6.2.2 (Correctness of the algorithm for union)**.** *The difference between the algorithms of union and intersection is found in the set-operator, i.e.* $\cap_s$ *versus* $\cup_s$ *as seen in Equation* (6.11) *and Equation* (6.12)*. From Corollary 6.2.1 we have proven the correctness of the intersection algorithm. Therefore the algorithm for the union-operation is also correct.*

6.2.2 Building a Restricted Set of Unique Vertices

The set of unique vertices consists of all linked vertices. In some contexts it is of interest restricting the set. Examples of such a restriction are for subsets were all the descendants have the sub-name-space 'P', or the stretch of vertices only connected by the "part of" relation type. A restricted set of unique ancestors (henceforth *the restricted set*) is understood as

$$Ancestors(v_i) = \begin{cases} \{v_k, v_l\} \in V & \text{where} \\ v_i \subseteq_\beta \{v_k, v_l\} & \text{and} \\ v_l \neq v_i & \text{and} \\ v_l \neq v_k & \text{and} \\ v_k \xrightarrow{\beta} v_l & \text{holds.} \end{cases} \tag{6.13}$$

where v_i, v_k and v_l are vertices in the Ontology V and β a sub-name-space/relation type connecting two vertices. When initializing Equation (6.13) with $v_k = v_i$, the implication is an explicit linkage between vertices of the same type, *i.e.* as required by the $v_k \xrightarrow{\beta} v_l$ property. The latter condition separate the restricted set from a unique set. The difference is seen when comparing Equation (6.13) with Equation (6.4), where the latter does not apply conditions for testing the type of relatedness. From the executing algorithm we observe the similarities of the approaches (*i.e.* equations):

- the sets (of the unique and restricted set) are constructed in Algorithm 6.3.
- the unique set uses a one-dimensional list, *i.e.* as specified in Equation (6.6) and at line 14 in Algorithm 6.3.
- the restricted set uses a two-dimensional list for storing the relations, *i.e.* as seen at lines 16 and 18 in Algorithm 6.3 .

Formalization of the algorithm for restricted set therefore implies a modification to Equation (6.6):

$$\{\text{relation type},\ \text{sub-name-space}\} \in \tau \tag{6.14a}$$

$$\Phi\left[v_i\right]_\tau = \emptyset\ v_i \in Roots \quad \text{initializes root/start vertices.} \tag{6.14b}$$

$$\Phi\left[v_i\right]_\tau = \bigcup_s \left\{ \underbrace{\forall_{v_k \in \pi(v_i,\beta)} \left\{ \underbrace{v_k \cup_s \Phi\left[v_k\right]_{\tau,\beta}}_{\Delta_2} \right\}}_{\Delta_1} \right\} \tag{6.14c}$$

where

- τ is a function representing the restricted set.
- Φ is modified to the unique set (*i.e. uniqueSet* in Algorithm 6.3),
- $\pi(v_i, \beta)$ is extended with the properties of relation types/sub-name-space,
- Δ_1 correspond to the lines [25 .... 29] in Algorithm 6.2, and
- Δ_2 correspond to the line 16 (in the code) for the relation type and line 18 for the sub-name-space, both found in Algorithm 6.3.

The similarity between the set of ancestors with/without restrictions, indicate correctness of the restricted set:

Lemma 6.2.2 (A vertex has a complete coverage for its restricted set)**.** *For the restricted set, a vertex has a complete coverage for all its ancestors. The cover is defined by Equation* (6.13)*, and holds for every vertex in the ontology.*

Correctness of the assumption in Lemma 6.2.2 is indicated by the similarity between Equation (6.6) and Equation (6.14)), and verified by the following proof:

Proof for the correctness of restricted sets. Proving the correctness of the restricted set, we observe similarities and differences with the unique set of vertices, *i.e.* as seen in Table 6.2:

Reference-id	Statement	Observation
OLR_1	Definitions of the restricted and unique set differ only in their requirement of explicit linkage, *i.e.* the $v_k \xrightarrow{\beta} v_l$ requirement.	Equations (6.13) and (6.4).
OLR_2	The unique and restricted sets differ only in their extra relation type/sub-name-space identifiers β and τ.	Equations (6.6) and (6.14).
OLR_3	The relation type/sub-name-space identifiers are defined for both the unique and restricted sets: if a vertex has a complete cover of its ancestors, then the properties of them must be correct, implying that the identifiers are correctly set.	Lemma 6.2.1, line 12 in Algorithm 6.2 and line 17 in Algorithm 6.3.
OLR_4	The algorithm for updating restricted sets only differs from (the algorithm of) updating the unique set with regard to the extra dimensions of their set: if the unique set is correctly updated, then the restricted set is correctly updated.	Lines 14, 16 and 18 in Algorithm 6.3.

Table 6.2: The set of observations used in the process of deducing the correctness of Lemma 6.2.2. We observe that the reference-id (in the leftmost column) consists of a prefix (*i.e.* OLR) and a number: supporting the operation of deduction, the proof will refer to the reference-id.

From OLR_1 and OLR_2 in Table 6.2 we induce that correctness of the identifier (*e.g.* relation type/sub-name-space) gives correctness of Lemma 6.2.2, *i.e.* as the set of operations which are equal between the restricted and unique set is proven for Lemma 6.2.1.

Our task is therefore to prove the correctness of both the identifiers and algorithmic modifications to the code for updating the restricted set.

If the identifiers are set correctly, the restricted set is correct. Assuming that the vertex does not hold a cover, it must not hold a cover for its ancestors, and therefore have been updated incorrectly. This would contradict Lemma 6.2.1 and our statement in OLR_3, which imply that the identifiers are correct.

The second difference between the restricted and unique set is due to procedures for updating the sets them self. From OLR_4 we observe that correctness in updating the unique set implies correctness in updating the restricted set, which implies correctness for the algorithm of updating the restricted set. ■

6.2.3 Building of the Concrete Paths

The set of (all) concrete paths provide answers to algorithms such as shortest or longest paths between arbitrary vertices. The set of all concrete paths is constructed in order to answer problems related to a term's linkage, *i.e.* the implicit relations of an ontology. A short summary of general properties regarding path-construction:

1. each vertex has zero or more paths to a root.

2. each path involves at least one vertex.

3. a vertex position in the path corresponds to its distance from the root.

The paths (*i.e.* the implicit relations) are the possible linkages between a vertex v_i and a root r_n. The linkage is constructed without taking into account the rules covering the relation types. Given the latter simplification, the existence of a path between the two vertices v_i and v_k is understood as

$$\{v_i \rightsquigarrow v_k\} \subseteq_\beta \{v_i \rightsquigarrow v_k \rightsquigarrow r_n\} \tag{6.15}$$

where β denotes an arbitrary relation type connecting two vertices, $\rightsquigarrow$ implies an implicit relation and the path $v_i \rightsquigarrow v_k$ is covered by the set of paths from a vertex v_i to a root r_n. From the observation we get Lemma 6.2.3:

Lemma 6.2.3. *If a vertex v_i has a coverage of all concrete paths to a root, then v_i has coverage of all concrete paths to any of its vertices that is member of the set covered by Lemma 6.2.1 (i.e. its ancestor-set).*

A proof of Lemma 6.2.3 require a definition of the path-set in an ontology. Generalizing the observation of Equation (6.15), we observe that the set of root-paths contain the set of paths linking all vertices

$$\Omega\left[v_i\right] = \emptyset \text{ for } v_i \in Roots \quad \text{initializes the root/start vertices.} \tag{6.16a}$$

$$\Omega\left[v_i\right] = \bigcup_{ae} \left\{ \underbrace{\Omega\left[v_k\right] \cup_{ae} \left(v_k, \beta\right)}_{\Delta_1} \right\} \quad \text{for each} \quad \underbrace{v_{k,\beta} \in \pi\left(v_i\right)}_{\Delta_2} \tag{6.16b}$$

where

- Ω hold the set of all paths to the ancestors,
- $\bigcup_{ae}$ appends the set of recursive paths (generated by $\Omega[v_k]$) to the end of each of the paths stored at a vertex (*e.g.* for $\Omega[v_i]$),
- v_i, v_k are members of the ontology,
- β is a relation type connecting v_i with v_k,
- $\pi_{v_i,\beta}$ is the set of concrete parent-relations for vertex v_i,
- Δ_1 performs the task the task of concatenating the ancestor-paths and
- Δ_2 represents the immediate ancestors which are explored.

The formalization of all paths in Equation (6.16) define the set of all concrete paths from a vertex to its set of ancestors:

Proof of a Lemma 6.2.3. From Δ_2 in Equation (6.16) we observe that a all of a vertex v_i's ancestors are visited. The equation construct path-sets, *i.e.* those labeled with Δ_1. When a root r_i is a member of the ancestors of v_i, we have

$$v_i \subset_\beta v_k \subset_\beta r_i \tag{6.17}$$

where v_k is an arbitrary ancestor of v_i and β an arbitrary relation type connecting two vertices. Given the pre-condition in Equation (6.17) the resulting sets of paths $\Omega[v_i]$ are given by

$$\underset{\omega_n \in \Omega[v_i]}{\exists} \left(\underset{\omega_p \in \Omega[v_k]}{\exists} \omega_n(0) \quad = \quad r_i \quad \&\& \quad \omega_p \subset_\beta \omega_n \right) \tag{6.18}$$

which implies that any vertex along the path from v_i to r_i is covered by a concrete path.

■

Constructing a path set $\omega_n \in \Omega[v_k]$ connecting a vertex to a root r_i we append a vertex' own identity at the end of parent vertex v_i path list $\omega_p \in \Omega[v_i]$, as outlined in line-block [19....23] in Algorithm 6.3. A formalization of the procedure in which we iteratively construct vertex' path-set is given in Equation (6.19):

$$\Omega\left[v_i\right] = \emptyset \; v_i \in Roots \quad \text{initializes root/start vertices.} \tag{6.19a}$$

$$\Omega[v_i] = \bigcup \left\{ \underbrace{\forall_{(v_k,\beta)\in\pi(v_i)} \left(\underbrace{\forall_{\omega\in\Omega[v_k]} \left\{ \underbrace{\omega \cup_a (v_k, \beta)}_{\Delta_3} \right\}}_{\Delta_2} \right)}_{\Delta_1} \right\} \tag{6.19b}$$

where

- $\pi(v_i)$ is the set of parents for v_i,
- β is a relation type connecting v_i with v_k,
- Ω holds the set of paths to the roots,
- ω is a simple path (*i.e.* set of vertices without furcation) to the the root,
- $\cup_a$ appends the rightmost item to the end of the (path) set,
- Δ_1 correspond to the lines [25 .... 29] in Algorithm 6.2,
- Δ_2 correspond to the lines [19 .... 23] in Algorithm 6.3, and
- Δ_3 correspond to the line 20 and 22 (in the code), both found in Algorithm 6.3.

Evaluating the correctness of the path-construction, Equation (6.19) is compared with our similar algorithms for pre-processing: the algorithms which build the unique and restricted set iterate through the set of parent-relations, appending their parents coverage to their own. Corollary 6.2.3 formalizes the latter observation:

Corollary 6.2.3 (Path-coverage for each vertex)**.** *The all-path implementation visits all of a vertex' ancestors. We observe that the for-each loop in Equation* (6.19) *corresponds to Equation* (6.14)*. From the proof of Lemma 6.2.2 we conclude that all ancestor-relations*

are visited. Therefore the set of paths for a vertex v_i contains a coverage for all of its ancestors.

Corollary 6.2.3 prove the correctness of a vertex' coverage. The corollary assume correctness of internal attributes of a path. Example of such an attribute is the distance-measurement, which is important when calculating the shortest and longest paths between arbitrary vertices. Lemma 6.2.4 formalizes the assumption of correctness with regard to a path's property:

Lemma 6.2.4. *A concrete path ω_i holds a string of connected vertices without branches, i.e.*

$$\{v_i \rightsquigarrow v_j \rightsquigarrow v_k\} \subseteq \omega_n \subseteq \Omega[v_i] \tag{6.20}$$

where the positions in ω_n (i.e. the path), which is a member of the path-set $\Omega[v_i]$ for vertex v_i, correspond to the distance from the root.

An implication of Lemma 6.2.4 is that a root is stored as the paths first index,

$$\omega_n(0) = r_i \tag{6.21}$$

where r_i is an arbitrary root stored on the arbitrary path ω_n. The task is to prove the correctness of a concrete path, *i.e.* to verify that a path is a true member of an ontology G.

Proof of Lemma 6.2.4. The paths are built from a successive number of set-concatenations. We observe the use of the $\cup_a$ operator in the definition at Equation (6.16) and implementation in Equation (6.19). The $\cup_a$ operator pushes a relation at the end of a list. The first vertex which the BFS visits is the root. The implication of the $\cup_a$ operator is

$$id\left(\omega_n(0)\right) = r_i \quad \text{which is the first vertex along a path,} \tag{6.22a}$$

$$\left(id(\omega_{n+1}) \xrightarrow{\beta(\omega_n)} id(\omega_n)\right) \in E \quad \text{for} \quad n \in [1 \ldots (|\omega| - 1)] \tag{6.22b}$$

where $id(\omega_n)$ describes the vertex-id at index n in the path. The vertex-pairs in Equation (6.22) represent a concrete relation. Construction of the path-set correspond to the lines 20 and 22 in Algorithm 6.3. An illustration of the path-set construction is provided in Equation (6.23):

$$\left\{ \underbrace{\pi\left(v_i\right) \ni v_k}_{d(v_k,\pi(v_i))=3} \right\} \subset_\beta \left\{ \underbrace{\pi\left(v_k\right) \ni v_l}_{d(v_l,\pi(v_k))=2} \right\} \subset_\beta \left\{ \underbrace{\pi\left(v_l\right) \ni r_i}_{d(r_i,\pi(v_l))=1} \right\} \tag{6.23}$$

where $d\left(v_i, v_k\right)$ represents the length (*i.e.* number of relations) connecting two arbitrary vertices along the path $v_i \rightsquigarrow r_i$, and β an arbitrary relation type linking two vertices. From Equation (6.23) we observe that the distances from a vertex v_i to any of its roots r_n is given by

$$d\left(v_i, r_n\right) \text{ equals } \underbrace{\begin{cases} \left(d\left(v_j, r_n\right) + 1\right) \\ \forall_{\omega \in \Omega[v_k]} \left(|\omega| + 1\right) \end{cases}}_{\Delta_1} \quad \text{where} \quad \underbrace{v_k \in \pi(v_i, \beta)}_{\Delta_2} \tag{6.24}$$

where $|\omega|$ describes the length of the ω set and $d\left(v_j, r_n\right)$ the distance from vertex v_i to the root r_n. Proof of the latter is by contradiction. Let the ontology G be a DAG, and assume that

$$d(v_i) < d(v_k) \tag{6.25}$$

represent the ancestor distance between v_i and v_k, where

$$v_k \in \Omega[v_i] \tag{6.26}$$

which implies that there exists a path (either implicit or explicit) connecting v_i to v_k. From Lemma 6.2.4 and assumption of the DAG, this is a contradiction, *i.e.* part Δ_1 of Equation (6.24) is proved. Correctness of the Δ_2 part regards the property of a vertex coverage. Correctness of the path-coverage is given by Lemma 6.2.5. From Equation (6.22) we observe that all vertices are connected in the same order as they are stored in the path-set, *i.e.* as a pair of neighboring vertices in the path correspond to a concrete relation in set E of arcs in the ontology. The root is the first inserted vertex. As the paths are concrete and the order corresponds to concrete relations, the implication is that an index in the path corresponding to the vertex distance from the root. An illustration of this is given in Equation (6.23). Lemma 6.2.4 is therefore proven. ■

The proof of Lemma 6.2.4 describe properties of a concrete path. Operations such as finding the shortest path between arbitrary vertices require a complete coverage for *all* the possible paths, *i.e.* as formalized in Lemma 6.2.5:

Lemma 6.2.5. *The set of of all concrete paths $\Omega[v_i]$ for an arbitrary vertex v_i cover all the possible paths connecting a vertex to any of its ancestors.*

From the proof of Lemma 6.2.4 we know that a path is a true member of an ontology G. Our task is to prove that all paths are member of the path-set Ω.

Proof of Lemma 6.2.5. Corollary 6.2.3 states that all vertices are covered during the path-construction. Therefore if each of the paths for vertex v_k are correctly extended at its child v_i, the set of paths correspond to those found in G, *i.e.* as stated in the proof of Lemma 6.2.5. The task is to prove correctness for the inner for-each loop (*i.e.* Δ_2) in Equation (6.19). The for-each loop corresponds to the lines [19 .... 23] in Algorithm 6.3. From these lines we know that, given correctness of the path extension, a set is correctly copied. As the path extension were proven for Lemma 6.2.4, we have therefore proven the correctness of Lemma 6.2.5 . ■

The proof of Lemma 6.2.5 states that a vertex v_i holds a complete coverage of its implicit ancestor-paths. The implication is that the distances to any of v_i's ancestors are pre-computed. Formalizing the latter, we get Corollary 6.2.4:

Corollary 6.2.4. *The set of distances from a vertex v_i to any of its ancestors v_k is*

$$d\left(v_i, v_k\right) = \left\{ \forall_{\omega \in \Omega[v_i]} \Big| \underbrace{\omega\left(|\omega|\right) \ldots \left(id\left(\omega(k) = v_k\right)\right)}_{\Delta_1} \Big| \right\} \tag{6.27}$$

where $d\left(v_i, v_k\right)$ describe the length of the sub-set, and Δ_1 the number of vertices in the subset. From the proof of Lemma 6.2.4 we know that a path's distance corresponds to the number of vertices along it. The latter is identical to Δ_1 in Equation (6.27)*, which prove this corollary.*

Corollary 6.2.4 define the correctness of the ordered access to any of a vertex v_i's concrete relations. From this we know the correctness of the algorithm for finding the paths, and their length, *i.e.* between arbitrary vertices. Proving the correctness of extracting the shortest and longest paths is therefore straightforward. The operations are defined as

$$\omega\left[v_i\right] = \bigcup_{ae} \left\{ \underbrace{\psi\left(\omega\left[v_k\right]\right) \cup_{ae} \left(v_k, \beta\right)}_{\Delta_1} \right\} \quad \text{for each} \quad \underbrace{v_{k,\beta} \in \pi\left(v_i\right)}_{\Delta_2} \tag{6.28}$$

where ψ is an arithmetic operator either selecting the shortest or longest path(s), *i.e.* of those found in the ancestors path-set $\omega[v_k]$. The path-length between the vertices is given by Equation (6.27) in Corollary 6.2.4. Applying the *min* operator to ψ in the equation, we have a function in Equation (6.29) describing the shortest or longest path:

$$d\left(v_i, v_k\right) = \psi \left\{ \forall_{\omega \in \Omega[v_i]} \Big| \underbrace{\omega\left(|\omega|\right) \ldots \left(id\left(\omega(n) = v_k\right)\right)}_{\Delta_1} \Big| \right\} \ , \ 0 < n < |\omega| \tag{6.29}$$

The correctness of the shortest and longest path operation is given by Corollary 6.2.5:

Corollary 6.2.5. *Corollary 6.2.4 states the correctness of finding the set of distances between two arbitrary vertices v_i and v_k. Equation* (6.29) *selects either the shortest or longest path using the basic arithmetic operator named ψ. As we assume the correctness of the basic arithmetic operators, the extension provided by the shortest and longest path algorithm is correct.*

6.2.4 Summary of Operations During The Pre-Processing

The procedures for calculating the set of unique sets, restricted sets and path sets are similar:

1. from Lemma 6.2.2 we know that all the relations are visited in the ontology;
2. from Lemma 6.2.1 we know that all the vertices have a complete coverage of their data-structures;
3. from Table 5.1 (at page 47) we know that the approach of storing data in lists implies a highly efficient utilization of memory (*i.e.* reduced running-time of the software).

The operations were given separate descriptions due to their differences:

The number of sets: while a single set represents a vertex' related vertices (*i.e.* in the context of unique and restricted sets), on average more than one concrete path will link a vertex to the roots. (For details about the latter, see description of our benchmark-ontologies in section 2.1 at page 7.)

The order in which the vertices are stored: the index (location) of a vertex in a concrete path correspond to its distance from the root.

Type of set-concatenation: The modified union $\cup_a$ operator appends a vertex to the end of the list, this in contrast with the sorting-operator $\cup_s$ in the building of the unique and restricted sets.

In this section, we explored the non-rule based pre-processing algorithms, and verified correctness of the algorithms in the pre-processing. The pre-processing algorithms assumed that an arc connecting two vertices was both anti-symmetric, reflexive and transitive, *i.e.* did not cover specific rules for each relation type. The queries which is understood as the most time consuming operations in biomedical reasoning concerns the application of inference rules, such as the operation of building of all-some closures, which we described in section 2.2.5 at page 14.

6.3 Efficient Support of Rule-Based Querying

We have recognized opportunities for increasing the performance of rule-based ontology queries, an algorithmic extension which is presented in this section. The extended pre-processing is motivated both by our initial performance analyze of ONTO-PERL, as seen in Table B.1 at page 121, and a description of the cocO(n) algorithm, as given in the previous section. When designing the rule-based extension, we restrict our scope to the cases of ontology contraction and expansion: an ontology is

- contracted (*i.e.* reduced) when it holds the minimal path-set and
- expanded (*i.e.* closed) when the new ontology holds all possible paths.

If the rules of ontology contraction and expansion cover all of an ontology's relation types, we might think that there will only be one path connecting two arbitrary vertices V and S. In some cases the latter assumption is wrong. The special case occurs when none of V's children is the subset of the other. To highlight the cases regarding the rule-based reduction, we include them in Figure 6.4:

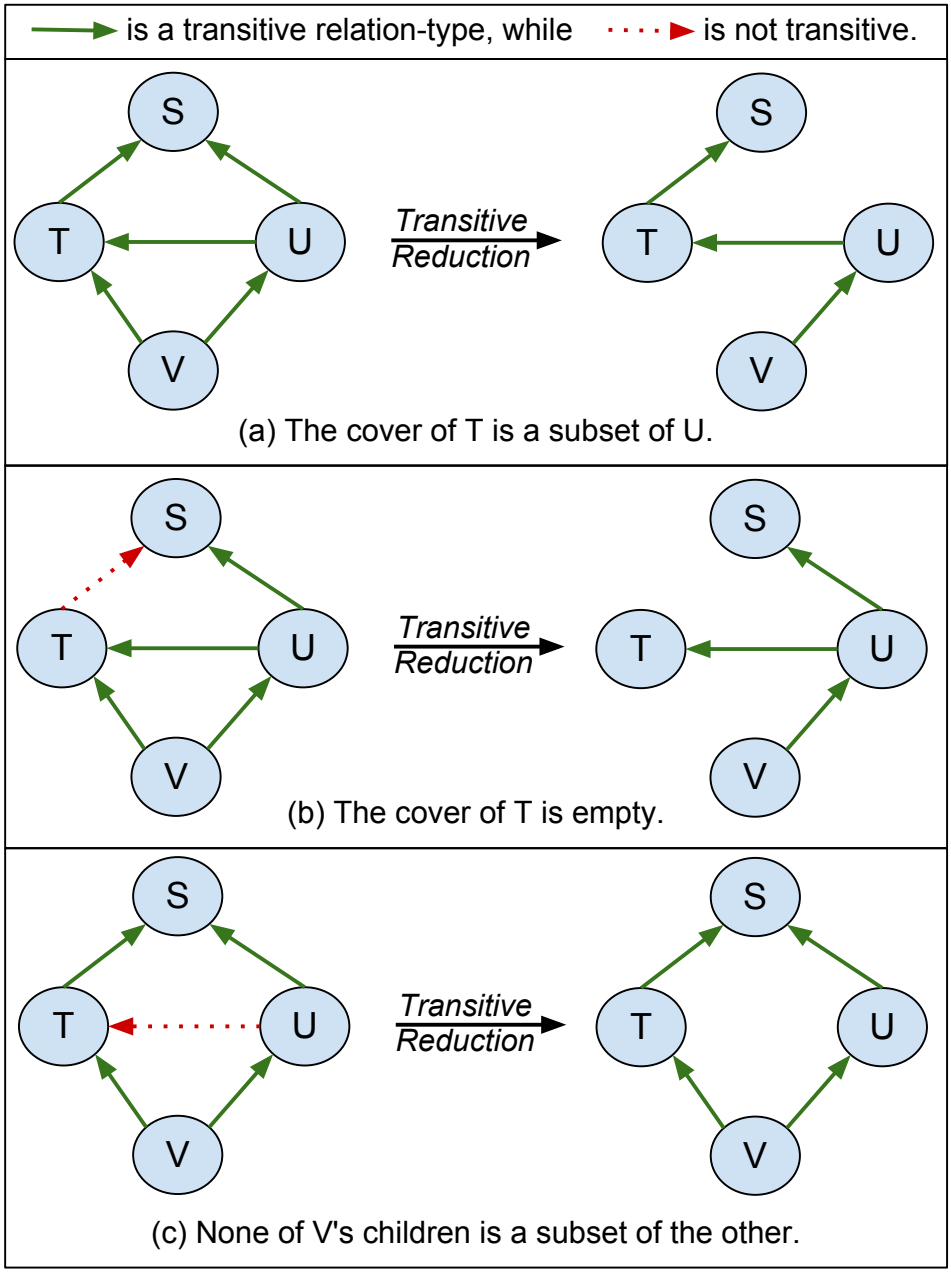

Figure 6.4: Rule-based ontology reduction. We observe the rule-based ontology reduction we observe the rule-based expansion for three sub-ontologies. To generalize our representation, the relation types are not explicitly stated: the green arrow symbolizes the relation types covered by the rule, while the dotted red arrow are those not to be expanded.

Figure 6.4 illustrates an ontology's special cases with regard to the rules of reduction. Making our illustration less abstract, we have applied the rules of transitive reduction, *i.e.* from the left to the right sub-figures. The three situations provided illustrate cases handled by our ruled-based algorithmic extension:

Sub-figure 6.4a: if one of the vertex-child *T* is the rule-based sub-set of the others (named *U*), then the descendants of *T* are completely covered by the rules for *U*.

Sub-figure 6.4b: if the rule-based cover of *V*'s child *T* is empty, then *V*'s rule-based cover is not extended.

Sub-figure 6.4c: if neither the descendants of *U* nor *T* provide a complete subset of the other, the contracted ontology is not updated: this implies that there exists more than one path connecting the vertices V and S.

The red arrows in Figure 6.4 are not part of a vertex' rule-based cover. For some ontologies this considerably reduce the number of paths connecting two vertices. The ontologies in our benchmark-set do not provide such challenges. (For details about the path-growth in our benchmark-ontologies, see Equation (2.3) at page 10.) If our benchmark ontologies were expanded before applying the pre-processing, the opposite would be true (*i.e.* an intractable amount of paths connecting two vertices). Ontology reduction provides an option of considerably reducing the number of paths connecting two arbitrary vertices. The reduction-factor depends upon the fraction of relation types covered by the applied rule (*e.g.* using the rule for transitivity instead of rules for the all-some property).

Illustrating the above cases, Figure 6.5 presents the query-result of ontology contraction and expansion:

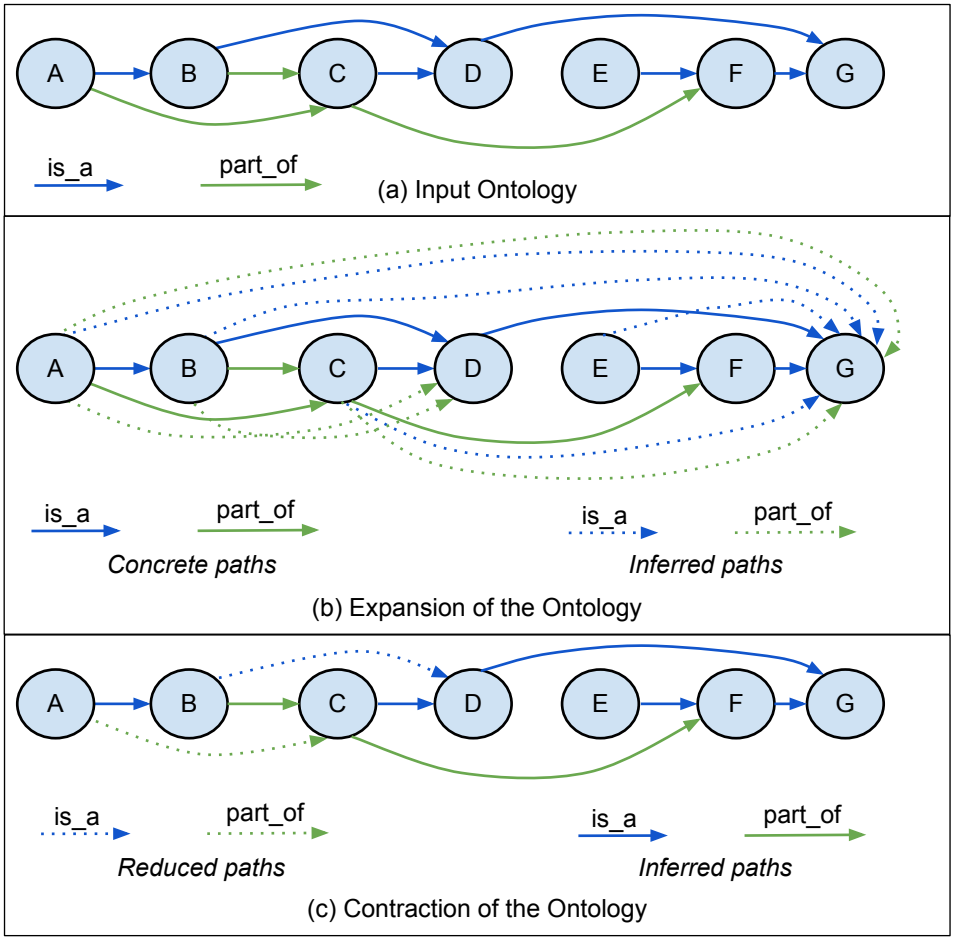

Figure 6.5: Ontology contraction and expansion. We observe here the contraction and expansion on the input ontology. To simplify our representation, we have chosen to use only two relation types: `part_of` and `is_a`. The input ontology is illustrated in sub-figure (a) with the concrete arrows connecting the vertices. Inferences are made in sub-figure (b) and (c). An inference is illustrated using a dotted arrow: for each vertex the resulting set of paths consists of both the inferred and concrete arrows: the expanded ontology is illustrated in sub-figure (b), while the contracted ontology is included in (c).

Ontology expansion and contraction are visualized in Figure 6.5. A short re-statement of the basics:

- the task of contraction is to remove concrete relations, while
- expansion (in most cases) increases the set of concrete relations.

From the figure we observe that

$$r \in R \qquad \text{is the set of query-rules, and} \tag{6.30a}$$

$$H[v_i]_r = \begin{cases} \{v_k, v_l\} \in V \quad \text{where} \\ v_i \subseteq_{r(\beta)} \{v_k, v_l\} \quad \text{and} \\ v_l \neq v_i \quad \text{and} \\ v_l \neq v_k \quad \text{and} \\ v_i \xrightarrow{\beta \in R} v_k \quad \text{and} \\ v_k \xrightarrow{\zeta \in R} v_l \quad \text{and} \\ v_k \xrightarrow{r(\beta,\zeta)} v_l \end{cases} \qquad \text{holds.} \tag{6.30b}$$

where

- v_i, v_k and v_l are vertices in the Ontology V,
- β and ζ are relation types covered by a rule r found in the rule-set R, and
- $H[v_i]_r$ is the set of reachable relations, given rule r, for vertex v_i.

Similar to Equation (6.13) at page 65 we initialize Equation 6.30 with $v_k = v_i$. In short $H[v_i]$ represents the set of inferred paths which are reachable from a vertex v_i.

The set of inferred paths require a translation of relation types, as seen for $v_k \xrightarrow{r(\beta,\zeta)} v_l$ in Equation 6.30, where the rule r translates the relation type β into a new relation type. Looking at the inferred paths in Figure 6.5, we observe that a single relation type results from concatenation of multiple relation types. Until now we have ignored the relation types in our algorithmic description. Without explicit translation rules it is difficult to correctly concatenate the relation types. Why awareness of the relations types is of importance now becomes obvious: in Figure 6.5 the relation types `is_a` and `part_of` were concatenated. An example of a correctly applied translation of Definition 2.2.4 (at page 13) is seen in Equation (6.31):

$$\begin{aligned} A \xrightarrow{is\_a} B \xrightarrow{part\_of} C \quad & \text{a set of relations, where} \\ A \xrightarrow{part\_of} C \quad & \text{is the ruled-based translation.} \end{aligned} \tag{6.31}$$

We have here used the rules for property chains. When only the relations affected by the property of chains are covered by the contraction and expansion rules, they are respectively denoted as the operations of *chain reduction* and *chain closure*. In Algorithm 6.4 the chain-rules for the relation types `part_of` and `is_a` are translated into an algorithm:

Algorithm 6.4: Ontology-rules for the `part_of` and `is_a` relation type.

```
/* Extends the algorithmic input with the set of rules. */
global inputRules ←cocoOn.getInputRules();
/**
   brief: Get the inferred relation type using the input-rule.
   param: <types> are set of connecting relation types.
   return: the relation type inferred by the input-rule.
**/
relationType getRelationTypeByRule(types) {
 if(types ∈ inputRules) {
   if(types ∈ {'is_a'}) then {return 'is_a';}
   if(types ∈ {'part_of'}) then {return 'part_of';}
   else {
     if(types ∈ {'is_a' ∪ 'part_of'}) then {return 'part_of';}
   }
 } else {
   return ∅; // the relation types were not covered by the rules.
 }
}
```

The concatenation rules are defined by the user. The provided example of chain-based rules (for the relation types `part_of` and `is_a`) is a simplification. Real-world expansion and contraction rules are more complicated. At page 84 we provide a detailed algorithm for the rules regarding chain-based expansion and closure: before the example is presented, the context of the rules and their application should be clear:

$$r \in R \qquad \text{the query-rules,} \tag{6.32a}$$

$$v_k \in V \qquad \text{the ontology,} \tag{6.32b}$$

$$H(v_k)_r \in H(v_k) \qquad \text{reachable vertices,} \tag{6.32c}$$

$$\Upsilon\left(v_k\right)_r = \pi(v_k) - \left\{\forall_{r\in R}\forall_{(v_i,\beta)\in\pi(v_k)} r\left(H\left(v_i\right)_r, \beta\right)\right\} \qquad \text{the } \textit{contracted} \text{ set,} \tag{6.32d}$$

$$\Xi\left(v_k\right)_r = \pi(v_k) + \left\{\forall_{r\in R}\forall_{(v_i,\beta)\in\pi(v_i)} r\left(H\left(v_k\right)_r, \beta\right)\right\} \qquad \text{the } \textit{extended} \text{ set.} \tag{6.32e}$$

Equation (6.32) define the contracted and expanded sets where

- Υ covers the contracted set,
- Ξ the extended set,
- $H(v_k)$ the set of reachable vertices for a vertex v_k, which was defined in Equation 6.30.

- $\pi(v_k)$ the set of parents for v_k, and
- r the rule which is applied.

Equation (6.32) applies the translation rules for relation types, which was defined in section 2.2 at page 11. The rule-based ontology may either be built separately, *i.e.* before the generation of all paths is applied, or used directly as part of the path-building operation. The provided option when running the cocO(*n*) algorithm therefore becomes:

1. either be part of the pre-processing (default option),
2. the only part of the pre-processing or
3. built from the ontology or a sub-part (sub-ontology) of it.

An example of the provided user-convenience is seen in above point (3), where cocO(*n*) support ontology reduction using multiple reduction rules.

From the the acquired understanding of the query-rules, and the different sub-ontology cases they are to be applied on, we are able to formulate an algorithm: translating the acquired knowledge into an high-level algorithm, the result is seen in Algorithm 6.5:

Algorithm 6.5: An algorithmic extension performing a rule-based pre-processing.

```
    /* Extend the algorithmic output containers. */
2   global relatedRules ←new Set; // reachable parent-vertices.
    global contractedParents ←new Set; // immediate parent-vertices.
    global expandedParents ←new Set; // immediate parent-vertices.

    /**
7      brief: Builds the set of rule-based relations.
       param: <current> is the vertex in question.
       param: <parent> is a vertex who added current to the queue.
       param: <type> The relation type linking the head with the tail.
       remarks: is the new function we have extended the algorithm with.
12  **/
    void insertRuleBasedRelation(current, parent, type) then {
     if(parent = ∅) then { // it is a root.
      relatedRules[current] ←{}; // initiates it.
     } else {
17    if(expandedParents[current] = ∅) then {
       //! Add relations both following- and not following the rule:
       {contractedParents[current]} ←inputParents[current];
       {expandedParents[current]} ←inputParents[current];
      }
22    for each rule ∈ {inputRules} then {
       if(current.relationType ∈ rule) then {
        //! Remove reachable relations:
        contractedParents[current][rule] ←
            contractedParents[current][rule] -
            rule(relatedRules[parent][rule],type);
        //! Append reachable relations:
27      expandedParents[current][rule] ←
            expandedParents[current][rule] +
            {rule(relatedRules[parent][rule],type)};
        //! Update the reachable set:
        relatedRules[current][rule] ←relatedRules[current][rule]
        ∪s  {rule(relatedRules[parent][rule], type)} ∪s  rule(parent,
            type);
       } // else the relation is not part of the accepted set.
32    }
     }
    }
```

From Algorithm 6.5 we observe the application of query rules, such as for line 25, which corresponds to $v_k \xrightarrow{r(\beta,\zeta)} v_l$ in Equation 6.30. To verify correctness of the reachable

set (*i.e. relatedRules* in Algorithm 6.5), we formalize the algorithm:

$$r \in R \quad \text{is the set of query-rules in } R, \tag{6.33a}$$

$$H\left[v_i\right]_r = \emptyset \; v_i \in Roots \quad \text{initializes root/start vertices.} \tag{6.33b}$$

$$H\left[v_i\right]_r = \bigcup_s \left\{ \underbrace{\forall_{(v_k,\beta)\in\pi(\beta)} \left\{ \underbrace{(v_k, r(\beta)) \cup_s \left\{ r\left(H\left[v_k\right]_r, \beta\right)\right\}}_{\Delta_2} \right\}}_{\Delta_1} \right\} \tag{6.33c}$$

where

- $H[v_i]_r$ holds the reachable set,
- Δ_1 corresponds to the lines [25 .... 29] in Algorithm 6.2, and
- Δ_2 corresponds to the line 30 in Algorithm 6.5.

If the reachable set has a complete coverage of its ancestors, given a rule *r*, we assert it to be correct:

Corollary 6.3.1. *A reachable set for a vertex has a complete coverage of its ancestors when given a query-rule.*
Comparing Equation (6.33) *with Equation* (6.14)*, the difference is found in the* Δ_2 *part of the equations, which concern translation of relation types: if the rules are correctly applied, and the restricted set is correctly updated, then the reachable set is correct. From the proof of Lemma 6.2.2 we know the correctness of a restricted set's coverage. Application of a query-rule is given by its definition, which we outlined in section 2.2 at page 11. We therefore assert the correctness of a reachable set's coverage for a vertex, given a query-rule "*r*".*

From the correctness of the reachable set in Corollary 6.3.1 we are ready proving the correctness of the *extended* and *contracted* sets:

Corollary 6.3.2 (Correctness of the extended and contracted sets)**.** *Given the similarities of the extended and contracted set, we first prove the correctness of the extended set before inducing the correctness of the contracted set.*

Comparing our algorithm to the definition of the expanded set, we observe that

- *line 20 in Algorithm 6.5 correspond to* $\pi(v_k)$ *in Equation* (6.32e)*, i.e. the expanded set is correctly initiated.*

- *line 27 in Algorithm 6.5 correspond to* $r\left(H\left(v_k\right)_r, \beta\right)$ *in Equation* (6.32e)*, i.e. that the expanded set is correctly updated if our reachable set is correct.*

From Corollary 6.3.1 we have correctness of the reachable set, therefore the expanded set is correctly updated, i.e. given the above two points.

Comparing the expanded set to the contracted set, the difference is seen in the "$+$*" vs "*$-$*" operator, i.e. the contracted set is correctly updated.*

6.3.1 Example: A Rule-set for Biomedical Ontologies

Until know we have vaguely described the rules of relation type translation. In short, the rules have been described as a change from one relation type to another. To highlight the operations of the rule-based algorithm, we use the rule set covering life science ontologies. To limit our scope, we specify the rules for the set

$$\{A, B, C, D, E\} \in V \quad \text{and } V \quad \text{is the set of vertices in our scope,} \tag{6.34a}$$

$$\{\alpha, \beta, \zeta\} \quad \in R \quad \text{where } R \quad \text{is the set of relation types.} \tag{6.34b}$$

$$\{\alpha, \beta\} \quad \in R_T \quad \text{where } R_T \text{ is the transitive sub-set.} \tag{6.34c}$$

$$\alpha \prec \beta \quad \text{implying that } \beta \text{ is a sub-chain of } \alpha\text{:} \tag{6.34d}$$

Equation (6.34a) forms the basis for our example of rule-based querying. The equation covers the sets of the relation types and vertices: α may in this context be substituted by `is_a` and β for `part_of`. The rule-based translation is important as it provides knowledge of the

- extended set which relations are member of,
- contracted set which relations are member of, and
- relation types connecting the inferred relations in the extended set.

The rules have the goal of identifying (a) the members of the set and (b) the translation of relation types. Our example uses the definitions outlined from pages 11...16:

- The connectivity of relations (*i.e.* transitivity) is defined in Definition 2.2.6.
- Definition 2.2.5 (the all-some property at page 14) covers the changes in relation types.
- Definition 2.2.7 explicitly states how super-sets are a generalization of subsets.

To simplify our case, our example is limited to the two relation types α and β, *i.e.*

$$A \xrightarrow{\alpha} B \xrightarrow{\beta} C \quad \text{which is our example,} \tag{6.35}$$

$$A \xrightarrow{\beta} C \quad \text{is the result of the mapping from Equation (6.34d).} \tag{6.36}$$

Equation (6.35) illustrates the case looking at only two of the immediate relations. It might be of interest considering more than two immediate relations. The latter is therefore included as part of our future-work.

We have here covered the translations using generalized representation of the relation types. The set notation does not explicitly specify the priority of specific relation types (*e.g.* `is_a` or `part_of`). The concrete mapping require knowledge of:

1. the relation type along the first relation, *e.g.* α in Equation (6.35),
2. the relation type along the second relation, *e.g.* β in Equation (6.35),
3. the inferred relation type, *e.g.* β in Equation (6.36),
4. the related-rule-set the mapping is a member of, *e.g. 1* (as there exists only one possible set of inferences), and
5. if the rule covers chains (*i.e.* $\prec$) or sub-classes (*i.e.* $\subset$), *e.g.* $\prec$ in Equation (6.34d).

The above examples (of relation type knowledge) is translated into a four-tuple, *i.e.*

$$(\beta, \beta, 1, \prec) \tag{6.37}$$

We observe that the new four-tuple consists of the last four points in the above list. Equation (6.34a) (at page 84) a presented a set of rules. Translating the rule-set into triplets, the result is seen in Table 6.3:

Relation-Head	**Mapping of Relation-Tail**
α	$(\beta, \beta, 1, \prec)$, $(\alpha, \beta, 1, \prec)$, $(\alpha, \alpha, \$, \prec)$
β	$(\beta, \beta, 1, \prec)$, $(\beta, \beta, \$, \prec)$, $(\alpha, \beta, 1, \prec)$
ζ	$\emptyset$

Table 6.3: Translating rule-sets into triples. Using four-tuplets to translate set-notation into explicit syntax. The last row represents inferences for relation type ζ; the set of inferences is defined using $\emptyset$, which implies that it has no inferences. For the other relation types α and β, inferences are defined. Each inference is a member of an inference-set. The membership is defined by the four-tuplets rightmost symbol. CocO(*n*) already stores knowledge of closed set of relation types: for details, see section 6.2.2 at page 65. The closed set is identified by the *$* symbol. An example of such is seen for the four-tuplet $(\beta, \beta, \$, \prec)$.

Table 6.3 illustrates the explicit translation of set-based knowledge. The inferences are ordered. The order is seen in column two of the table: from the most generalized relation type at left, into the most specific case at the leftmost side. For two relations connected by relation type α, two translations exists. Equation (6.38a) provides an example of such a case:

$$A \xrightarrow{\alpha} B \qquad \text{gives the two alternative translations} \tag{6.38a}$$

$$A \xrightarrow{\alpha} B \qquad \text{which is the first alternative} \tag{6.38b}$$

$$A \xrightarrow{\beta} B \qquad \text{which is the other solution.} \tag{6.38c}$$

In the process of contraction we choose alternative in Equation (6.38c), while in the process of expansion both alternatives are used. The latter is due to the all-some property in Definition 2.2.5. We observe that the translations are used both in operations of contraction and extension: for contraction a single relation type is to be chosen.

6.4 CocO(*n*)'s Implementation and Future Work

In this chapter we have presented an approach for making ontology reasoning fast. The chapter has applied the knowledge of properties regarding ontology benchmarks, algorithms with the task of reducing redundant operations, and the knowledge of preferred memory access patterns:

1. implemented in C++ using the principles of efficient memory access patterns;

2. applied in the pre-processing algorithms discussed at page 24;
3. consists of approx. 17.000 lines of code distributed on 43 distinct classes;
4. accessible through its (library) API, which is located at `https://code.google.com/p/ontowiz/source/checkout` and described at `http://folk.ntnu.no/olekrie/ontowiz_cocoon_documentation/classcocoOn.html`;
5. extensively documented at `http://folk.ntnu.no/olekrie/ontowiz_cocoon_documentation/annotated.html`;

Through the pre-processing, CocO(*n*) has followed the design criteria outlined in the previous chapters, by taking the approach of storing data sequentially in ordered lists. Implementations of all the discussed algorithms are found at `https://code.google.com/p/ontowiz/source/checkout`, with the sole exception of the API support for rule-based querying, which still poses some restrictions on the expressive power of rule-based definitions at the time of writing. Since implementing the full range of desirable options for ontology reasoning would be a significant additional undertaking, we regard extended API support for rule-based querying as part of our future work.

The performance impact of our approach has not been evaluated. Such an approach requires cocO(*n*) to be glued to an ontology-parser. In the next chapter we describe this part, *i.e.* ontology reasoning through the onto*Wiz* API, where onto*Wiz* enables performance testing.

Chapter 7

ontoWiz; Implementation and Performance Analysis

OntoWiz is an engineering tool for biomedical ontologies that uses cocO(n) for high-speed ontology reasoning. As the performance measurements of cocO(n) depend on the interconnection between the ontology-parser and ontology-interaction (both written in Perl), in section 7.1 we describe the interplay, in more detail.

It is obvious that in order to be useful our ontoWiz implementation should produce correct results correspond to the operations that are described in the software-documentation. The C++ source code is documented at `http://folk.ntnu.no/olekrie/ontowiz_cocoon_documentation/`, while the documentation for the Perl code is found in each source-code file at `https://code.google.com/p/ontowiz/source/browse/#hg%2Fml_ontology%2FOntoWiz`.

The approach for validating correctness, which we discuss in section 7.2, concerns both ontoWiz and cocO(n), we therefore omitted a separate discussion of correctness of the cocO(n) implementation in the previous chapter. Our assessment of assuring correctness in ontoWiz, show that the implementation is trustworthy. Therefore, measuring the the performance impact (*i.e.* processing time) of the ontoWiz approach becomes relevant and highly interesting. From the measurements in section 7.3 we observe that the processing-time of ontology-reasoning stems from three aspects:

1. the number of memory accesses,

2. the type of memory accesses, and

3. the size of the ontology.

By analyzing these aspects we are able to assess their contribution to the processing time-With a summary in section 7.4 of both the implementation and the performance analysis, we suggest extensions to our implementation/performance analysis.

7.1 Implementation

Onto*Wiz* is written in both C++ and Perl, using SWIG's auto-generated Perl and C++ code as a glue (*i.e.* for interconnection between the two programming languages). The main use of onto*Wiz* is to provide an interface for cocO(*n*). The layered division of their relationship is shown in Figure 7.1:

From Figure 7.1 we observe how cocO(*n*) is an integral part of onto*Wiz*. The onto*Wiz* interface is specially designed for tasks of interest in biomedical ontology engineering.

Users interact with onto*Wiz*through an API. The interface is similar to ONTO-PERL.

The storage scheme (for biomedical ontologies as described in Figure 7.1) requires a design which is flexible w.r.t. the ontology-formats (*i.e.* given requirements SR_1 and SR_2 in Table 2.1). As an ontology consists mostly of relations and attributes (*i.e.* compositions of strings), it is therefore possible to develop two parallel storage schemes:

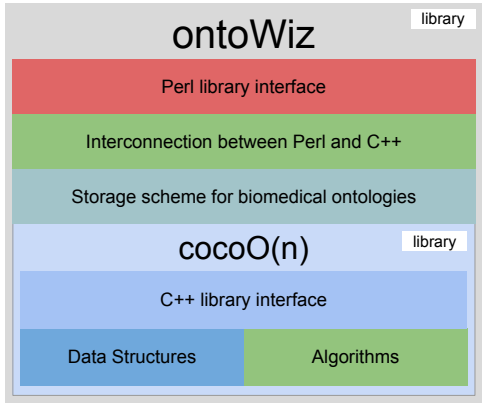

Figure 7.1: Layered work-division of onto*Wiz* and cocO(*n*); dependencies are to be read from top to down of the figure, *i.e. Perl library interface* is dependent on fields below it, such as the cocO(*n*) library.

- relations, such as for the (Term, Term, relation-type) triple, and
- sets of strings, such as (database, accession number, description, modifier).

ONTO-PERL stores the data in structures corresponding to the grammar of each ontology (*i.e.* linked lists of indirect memory references), resulting in a high number of cache misses for each operation (*i.e.* as mentioned in Table B.1 at page 121). In contrast to ONTO-PERL, onto*Wiz* separates the data storage and access routines:

- attributes for Terms, Instances, Relation-types, and general properties of the ontology are stored/accessed using the same underlying class/functionality, an approach which differs from ONTO-PERL's strategy;
- the list of relations (which is the input for cocO(n)) is stored in one continuous memory block, *i.e.* designed for reducing the number of cache misses;
- the user-based interaction (*e.g.* retrieval of a *term's* attributes) is independent of the data storage scheme, *i.e.* changes in interaction-pattern will not affect the data-storage routines.

From these facts we observe that the ontology storage scheme used by onto*Wiz* is capable of handling any type as long as they are of OWL, OBO and RDFS, that may be divided into sets of relations and sets of attributes. Our approach depends upon a generalized grammar of the biomedical ontologies.

Each of the OBO, OWL and RDFS ontology formats has a grammar specified in the Extended Backus Naur Format (EBNF) language (`https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form`), such as for the OBO Format at `http://oboformat.googlecode.com/svn/trunk/doc/obo-syntax.html`. Building a generalized grammar is the core of our approach (*i.e.* for ontology storage without loss of expressivity), as formalized in EBNF Grammar 7.1:

EBNF Grammar 7.1: Grammar for generic storage of ontology attributes.

```
<attribute_set>       ::={<attribute>
                          (<internal_set>)? <attribute_delimiter>}*
                          <attribute> <attribute_end>
<attribute>           ::={<string>}
<internal_set>        ::={<string>} <end_of_internal_set>
<attribute_delimiter> ::=\4
<internal_delimiter>  ::=\5
<end_of_internal_set> ::=\6
<attribute_end>       ::=\0
```

EBNF Grammar 7.1 defines the generic scheme for storage of ontology attributes. Our grammar uses the W3C EBNF flavor (`http://www.w3.org/TR/REC-xml/#sec-notation`). From the grammar we observe that an attribute is either

1. a composition of other attributes,

2. a string (*i.e.* a set of chars), or

3. a digit starting with the \ symbol (in our grammar), which correspond to a non-printable symbols in the American Standard Code for Information Interchange (ASCII).

The implication from EBNF Grammar 7.1 is that our internal representation is a generalized representation of the ontology models. We therefore avoid the problem of having the data storage structure explicitly bound to the ontology-grammars in question: the application of four distinct delimiters (*i.e. attribute_delimiter*, *internal_delimit-or*, *end_of_internal_set*, *attribute_end* in EBNF Grammar 7.1) makes it possible for cocO(*n*) to describe all types of attribute-compositions in the ontologies. If we know that the grammar, and the handling of the grammar (*i.e.* as concrete code in the C++ language) is correct, it opens opportunities to extend it to new components (*i.e.* a shorter implementation time when extending the software to new ontology-formats). For further understanding of the attributes storage scheme, it is of interest looking at the ontology's dependency scheme, which is illustrated in Figure 7.2:

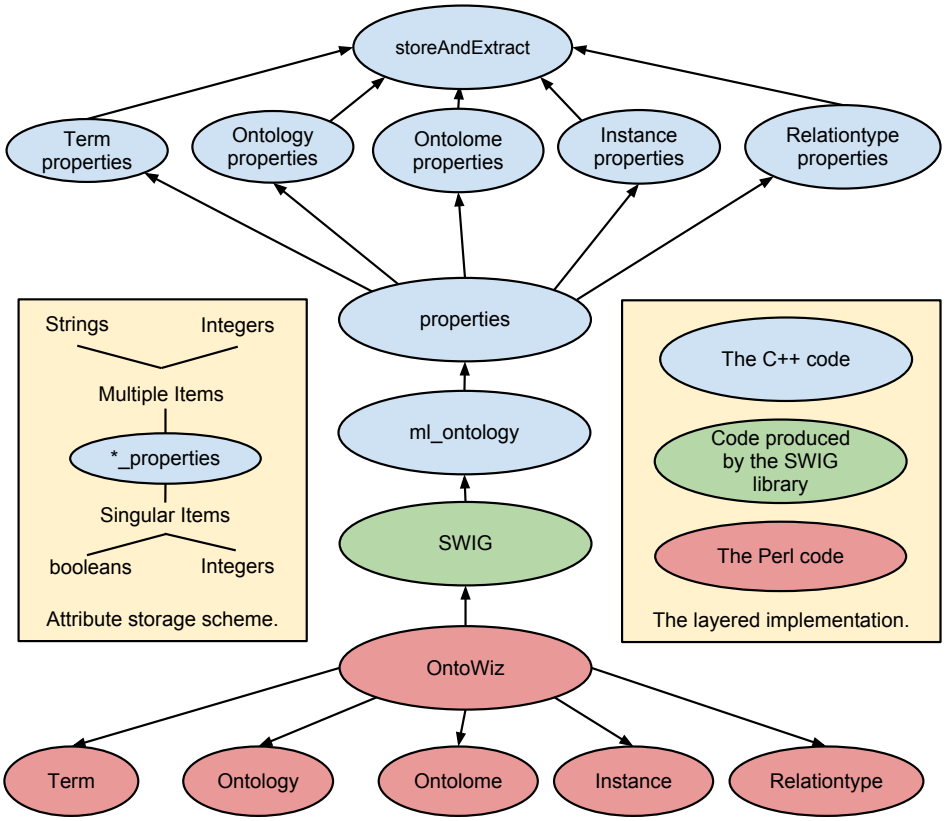

Figure 7.2: Conceptual organization of ontoWiz' dependency scheme for handling of biomedical ontologies; in the below text we explain the figure.

Figure 7.2 illustrates the dependencies of ontology storage: we observe a tree with two boxes at each side of it. The tree describes a simplified architectural design of ontoWiz[1]. In short:

- the tree describe access and storage of an ontology; the top blue vertex (named *storeAndExtract*) stores and extract ontology properties applying the memory access pattern in Table 5.1 at page 47; the red vertices at the tree's bottom represent the Perl interface modules in ontoWiz.

- the left square represent the four data structures which is used to densely store the ontology properties when using EBNF Grammar 7.1.

[1]Both documentation and the (complete) dependency graph of *ml_ontology* is located at `http://folk.ntnu.no/olekrie/ontowiz_cocoon_documentation/ml__ontology_8h.html`.

- the color scheme in the right square provides an organization of the programming-languages and glue (*i.e.* as described in the layered division in Figure 7.1).

From the dependencies we observe that it is easy to extend with new objects such as property-sets (*i.e.* by appending a new vertex to the second and last row of the dependency tree in Figure 7.2). We therefore argue that it is a flexible storage scheme. The approach has sought to balance a generic approach with error-checking and correctness of usage, and integrates the benefits of cache efficiency into the resulting interface.

The SWIG provides two alternative approaches:

- transfer of complex data-objects (such as 3-dimensional lists) *without* adding a state-dependency in the C++ ontology object, or
- use of state-dependency, which implies (in this context) that multiple calls are made to insert and retrieve a complex object, such as a variable-sized 3-dimensional list.

Onto*Wiz* uses the state-dependency-approach. The approach of applying state-dependencies avoids the need for a high-level language-specific setting, as language-specific settings require detailed domain knowledge of the given languages to support: for Perl see `http://www.fnal.gov/docs/products/perl/pod.new/5.00503/pod/perlguts.html`.

The downside of our approach is a higher complexity of the C++ code. The support of state-dependency requires a framework for identification of the order and meaning of an inserted object (*i.e.* implementation of EBNF Grammar 7.1). An example of this is functionality for explicit marking of the start of a table-row. A challenge is to detect erroneous usage, such as when values in a new row are inserted without first marking the explicit start of the new row. The problem is minimized by separating the operations in their own class, *i.e.* using scopes to clarify the operations. We therefore need an approach to evaluate correctness, both with regard to internal logic and correctness of usage of onto*Wiz*.

7.2 Correctness of Executing Code in onto*Wiz* and coc$O(n)$

Trustworthiness, extendability and performance-measurements of onto*Wiz* and coc$O(n)$ all rely on correctness of the executing/implemented code. Correctness of the packages syntax (*i.e.* that the language is expressed in accordance with the languages rules) is evaluated through the Perl-interpreter/C++ compiler. Correctness is understood as a match between the documented behavior of each component (*i.e.* class/function) versus

the actual behavior (*i.e.* the output returned from the objects/functions input parameters). Documentation of the behavior of each component is found in the C++/Perl source files (located in the repository `https://code.google.com/p/ontowiz/`). Improving the readability of this, we provide a structured version (of the documentation for improved readability) at `http://folk.ntnu.no/olekrie/ontowiz_cocoon_documentation/annotated.html`.

The task of evaluating this correctness is organized in three levels:

The functionality level: Asserts correctness of each function. Example of evaluations are those of validating correctness of list extensions, copying of data structures and performing sub-set operations. Such tests are implicitly performed at all stages during debug-mode (*i.e.* when compiled with the "./install_debug.bash" script), while a limited set of tests are active during run-time mode (*i.e.* when the packages are compiled with the "./install.bash" script).

The class level: Included in the static routine "[class_name]::assert_class(...)". Uses known input to validate the output. Tests are bundled together in the executable found in each of the source-code sub-directories.

The API level: Compares the results of ontoWiz with ONTO-PERL using the benchmark-ontologies; validates the result of ontology-querying, both with regard to attribute-storage and reasoning, *i.e.* correctness is evaluated for both ontoWiz and cocO(*n*). *Class level* tests are found in the ontoWiz source directory (`https://code.google.com/p/ontowiz/source/browse/#hg%2Fml_ontology%2FOntoWiz`). An example of a class-level test is AssertTermModule.pm, which asserts the correctness of retrieving ontology-properties. The *API Level* tests are performed for each of the files in our benchmark-ontologies.

The above three levels of inspection apply knowledge from the object-oriented approach (`http://www.mysciencedictionary.com/definition-of-object-oriented-paradigm/`) which is the paradigm applied by both ontoWiz and cocO(*n*). The object-oriented approach implies that functionality/expressivity is increased for each included component (*e.g.* for a class). When a component is tested at the above three levels, we assert that it is safe to assume correctness of the component's behavior.

The described tests of correctness are formal. From code inspection, we roughly estimate that 50% of the code is designed for the task of validation. We know from experience that such tests does not give a 100% coverage of the logic's involved in calculation of queries. To catch errors not framed by the formal tests, we also perform an *informal* inspection of the code, by evaluating:

1. the match between the components documentation and their implementation,
2. the coverage of tests versus the expected behavior given our ontology benchmarks, and
3. the ability to catch and explain errors due to wrong usage, such as providing for instance a cyclic ontology as input.

The informal inspection is supported by applying ontology-visualization, which is enabled through the Visual Tool Kit (VTK) library[60, 61]. VTK is integrated in cocO(*n*), and enhances the identification of cyclic ontologies, multiple vertex linkage, vertex leafs, *etc*. When errors are in the code are discovered through informal inspections, which we described above, the formal tests are updated[2]. Supporting test-maintenance, simplicity is balanced against complexity, giving the best specificity (*i.e.* as it is difficult to verify and update complex test-code). From our measurements we know that application of *functionality level* tests during run-time reduces the performance by a factor greater than 3x. Most of the tests are therefore de-activated at run-time mode, *i.e.* when the "./install.bash" script is used for building libraries of onto*Wiz* and cocO(*n*).

7.3 Performance Measurements

Measuring the performance, we compare the performance of onto*Wiz* against ONTO-PERL. One of the parameters we evaluate is the number of computed relations/terms. From our performance measurements we observe that both tools calculate similar answers to the same queries; interestingly, differences in the calculated answers are due to false negatives in ONTO-PERL. The correctness of the onto*Wiz* tool provides us with confidence to compare the performance of onto*Wiz* versus ONTO-PERL. We do this comparison by measuring the performance on queries which we used to evaluate the structural attributes of our ontology benchmark (as discussed in section 2.1 at page 7). The queries we evaluate are those listed at `https://code.google.com/p/ontowiz/wiki/PerformanceBenchmark`. The results may be reproduced using the benchmark script at `https://code.google.com/p/ontowiz/source/browse/bm_core/benchmark.pl`.

Surprisingly, a difference of more than **six orders** of magnitude is seen when comparing the performance difference between onto*Wiz* and ONTO-PERL. The difference is evaluated with a methodology in sub-section 7.3.1. When applying this methodology in

[2]Error-tracking is supported by the "Issues" list in our code repository (`https://code.google.com/p/ontowiz/issues/list`).

sub-section 7.3.2, we observe that this difference is due to the extremely high number of redundant operations and large frequency of cache misses in ONTO-PERL. Sub-chapter 7.3.3 ends with a summary of the performance difference.

7.3.1 Methodology of the Performance Measurements

The purpose of the performance measurements is to evaluate the impact of the onto*Wiz* approach, and empirically identify reasons for the observed difference. In our measurement configuration we allow a number of distinct programs (*i.e.* background processes) to execute simultaneously, from which we expect a variance in our measurements. Empirical correctness of the measurements is therefore validated applying multiple measurements, and for each measurement we discuss:

1. the averaged measurement-value for each ontology (*e.g.* the time finding the set of descendants for the roots),

2. the variance of the measurement-value for each ontology, and

3. the outliers (*i.e.* the lowest and biggest values) in the measurement-value for each ontology.

The number of multiple measurement samples differs both with regard to the type of measurement (*e.g.* 1 measurement for querying the restricted set, while 10 measurements for the task of finding all the roots) and the number of ontologies which is measured. As some of the measurements takes days to complete, the most time-consuming ontologies are evaluated with fewer overlapping sample-measurements than the smaller ontologies.

Correctness of each calculation (*i.e.* of the queries which we measure) is validated by our correctness approach, which was described at page 94. For the ontologies with a file size less than 2 MB we compared the results generated by onto*Wiz* and ONTO-PERL. From the comparison we observed that onto*Wiz* found more paths than ONTO-PERL. Investigating the difference, we discovered that ONTO-PERL was not able to infer a subset of the paths. To test if this was the case for ontologies of a file size greater than 2 MB, we compared the path lengths generated by ONTO-PERL and onto*Wiz*: the paths length detected by onto*Wiz* were always greater than the path lengths detected by ONTO-PERL, which is in correspondence with the false negatives of ONTO-PERL discussed in Table B.1 at page 121.

In chapter 5 at page 33 we presented measurement results collected by applying a memory measurement tool. The memory-measurement is not suited for measuring Perl

code. Memory-measurements are therefore done indirectly by applying the gathered knowledge of how cache-misses correlates with both the user-time and number of cache misses (as stated in Table 5.1 at page 47). In short, we investigate the memory access pattern of the executing code and compare it with the number of memory accesses and the executing time of the operation. Measuring the number of memory accesses, we count the number of times a query in ONTO-PERL accesses the ontology. A query function accesses the ontology through two distinct procedures:

1. access to the ontology object, *e.g.* set of relations, and

2. access to temporary storage containers, with tasks such as sorting the set of accessed ontology objects.

We understand the number of redundant operations as the number of unnecessary memory accesses, both with regard to the ontology object and accesses in the query-functions temporary memory structure. The number of redundant operations in ONTO-PERL, and their implication, is evaluated by including a counter to each of ONTO-PERL's operations. The counter measures the number of times the ontology is accessed (point (1) in the above list), but not the number of internal accesses to temporary allocated memory by the functions performing the querying (point (2) in the above list). Comparing the number of ontology accesses through distinct queries, we get a relation between the number of operations and the time performing them, which makes it possible to evaluate the impact of temporary storage. We do not expect the operation-counter to reduce the performance of ONTO-PERL, as the 8 Byte counter we apply will not occupy a significant part of the computers cache, nor increase the processing time (*i.e.* due to its simple arithmetic nature).

Presenting the measurements, we use continuous trend-lines, which is different from the alternative of using step-lines. Formally, as the ontologies are discontinuous, it would be more correct using a step line. The problem when applying a step-line to the graph is the clutter of the curves, which make it more difficult to visually tracking the trends of the measurements.

The computer which we measure on is the Dell-laptop, described in Table 4.1 at page 31, which we use to evaluate all of the queries for ontoWiz and ONTO-PERL. When computing the queries with ONTO-PERL, standard servers, such as the biogw-db, are used for the CCO project. The short running-time of ontoWiz makes it tractable running the software on a laptop, *i.e.* as the computation takes a negligible amount of time. We expect the measurements on our laptop to correspond to other hardware, such as those listed in Table 4.1, due to our discussion of the memory access benchmark in section 5.3 at page 46.

7.3.2 Analysis of the Performance Measurements

Measuring the performance-impact of onto*Wiz*, we evaluate 7 parameters to the 21 queries on the 62 benchmark ontologies for both onto*Wiz* and ONTO-PERL[3]. The analysis we present evaluates the result of all the parameters/queries/ontologies. To simplify our discussion, we highlight the important findings, *i.e.* only a subset of our measurements is presented. The under-performance of ONTO-PERL, when compared to onto*Wiz*, is explained by the findings from our analysis of the performance-measurements, a measurement which is illustrated in Figure 7.3:

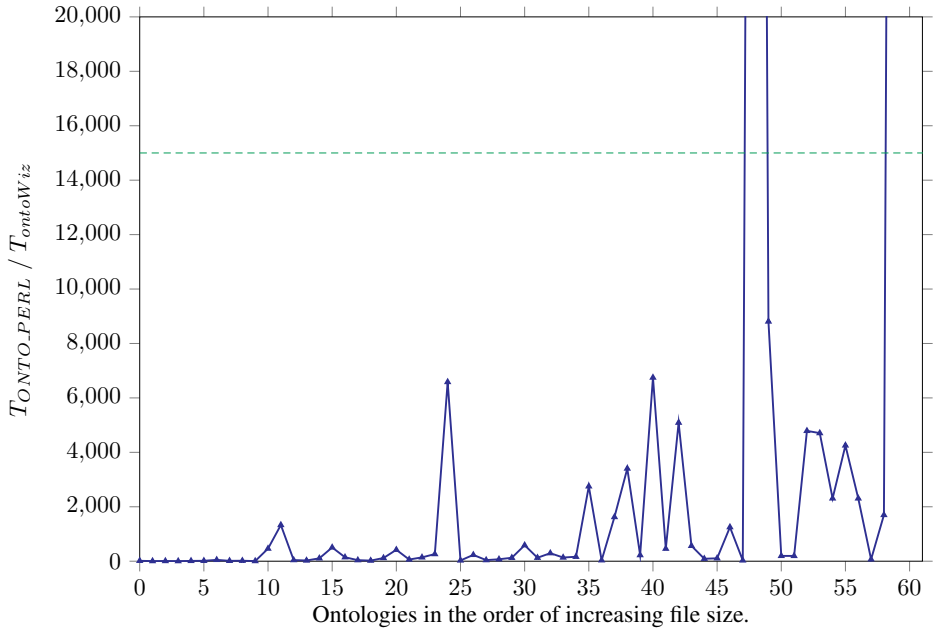

Figure 7.3: Comparison of onto*Wiz* and ONTO-PERL. The performance of the tools was compared when executing 21 queries on 62 benchmark ontologies. For each of the benchmarked ontologies we gave onto*Wiz* and ONTO-PERL an upper time threshold of 172,800 seconds for completing the queries, which is illustrated by the ---- line.

Figure 7.3 shows that the relative time consumption of ONTO-PERL to onto*Wiz* reaches the order of thousands for the larger benchmark ontologies. The limit of 172,800 seconds corresponds to two days of computing, and the ontologies at indices 48, 59, 60 and 61 surpassed this threshold when running ONTO-PERL. Analysis of the underlying mea-

[3]For a list of parameters, queries and ontologies of the performance benchmark see `https://code.google.com/p/ontowiz/wiki/PerformanceBenchmark`.

surements for each of the 21 queries indicates that the performance difference exceeds six orders of magnitude: increasing the threshold to 345,600 seconds we queried the ontology at index 60 (the GO ontology) for the descendants roots, but was aborted as the processing time passed our updated threshold. Given the result,

$$\frac{\text{Time(ONTO-PERL)} > 345,600s}{\text{Time(ontoWiz) = 0.04s}} >= 8.64 * 10^6 \tag{7.1}$$

is a lower bound of the performance difference.

From the underlying data we observe that the measurement error of each of our processing time measurements is comparatively small, *i.e.* when compared to the performance-difference between ontoWiz and ONTO-PERL. We have not managed to identify reasons for the error, though we suspect a combination of:

1. system scheduling due to other programs which simultaneously uses the hardware, and
2. granularity of the time-measurements.

The differences seemed to correlate with the number of parameters (in our benchmark script) extracted for each ontology in our benchmark: the processing time increased for unrelated measurements when the set of extracted parameters increased. We suspected that the problem was linked to the re-use of data in cache from earlier computations, as we changed the order and type of accessed data previous to a query. To investigate the effect, we tried polluting the cache, *i.e.* to fill the cache with unrelated data. The cache-polluting measurements did not produce any significant difference when compared to the maximum, minimum and variance of the non-cache-pollution measurements.

The missing impact of polluting the cache is an indication that ONTO-PERL does not utilize the cache when getting data, *i.e.* that ONTO-PERL does not manage to reuse pre-loaded memory in cache when querying the ontology. Studying the underlying measurement-data, we observe that the huge performance impact of our approach, is due to both the type of memory access and the algorithmic overhead:

1. ontoWiz uses the pre-processed ontology for generating answers, while ONTO-PERL iterates through an ontology using random access pattern;
2. a high number of redundant operations is seen for most of the queries performed by ONTO-PERL;
3. the cost of redundant operations is amplified by the time cost of the cache misses for each memory request.

When the number of redundant operations is low, the number of operations performed by ONTO-PERL maps roughly to the processing time, which can be seen in Figure 7.4:

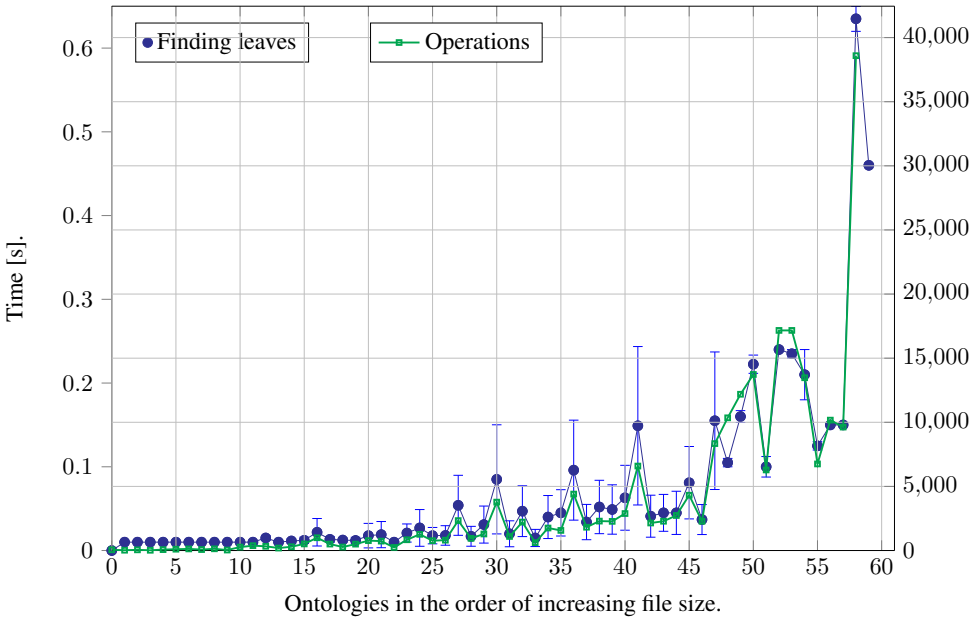

Figure 7.4: Impact of memory accesses for the query of finding all of the leafs ancestors. We evaluate the processing time of ONTO-PERL along the left y-axis and compare it with the total number of operations accessing the ontology (operations) along the right y-axis.

Comparing the number of memory accesses to the processing time, we observe from Figure 7.4 a correspondence between the time-measurements and the change in the number of ontology memory access operations performed by ONTO-PERL. Given the large variance in each measurement, we investigate the underlying data with respect the each measurement sample. From the inspection we observe that the measurements follow the same periodicity, from which we assess that there is a true periodic correlation between the number of memory accesses and the execution time.

We understand a high-performance ontology reasoning tool as a tool which achieves a performance close to the possible reasoning speed offered by commodity computers, as discussed in chapter 1 at page 1. It is therefore of interest evaluating the performance of ONTO-PERL against the memory speed of the laptop (where a laptop is understood as a commodity computer). The timer we are using has a resolution of 0.01 seconds (*i.e.* as seen by our underlying measurement data). The query time for the operation of finding the leaf ancestors using onto*Wiz* takes less than 0.01 seconds, *i.e.* zero seconds. As the

memory access speed of our hardware is greater than the memory access speed achieved by onto*Wiz*/cocO(*n*), it is therefore hard to estimate a concrete number (other than infinite) for the performance-difference between the memory speed of our hardware, the performance of onto*Wiz*, and ONTO-PERL.

The infinite difference, which we observe when comparing the query time of onto*Wiz* against ONTO-PERL in Figure 7.4, is due to memory accesses: even when there is a low level of redundant operations, there exists an overhead with regard to time. From the underlying data we observe that the number of leafs which is found maps to the number of performed operations. The small difference implies that the number of redundant operations is low for finding the set of ancestors for each of the leafs, when seen in comparison to the other queries which ONTO-PERL supports. Inspecting the effect for algorithms that have a higher number of redundant operations and memory accesses, we analyze a query designed for the task of finding all the descendants of the roots, as seen in Figure 7.5:

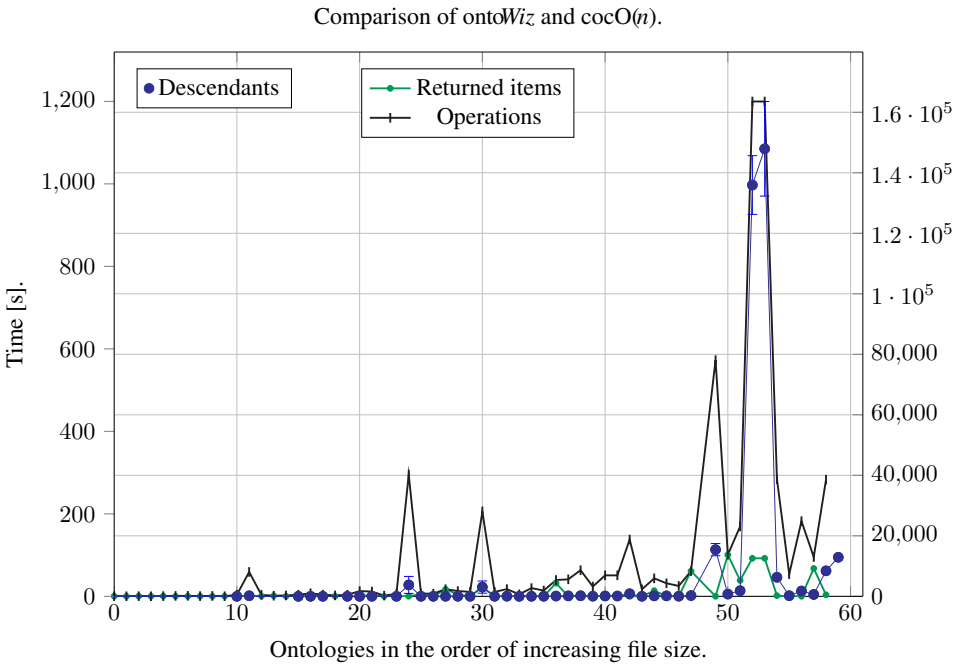

Figure 7.5: Comparison of ontoWiz and ONTO-PERL. The impact of an increased number of memory accesses is analyzed by querying the roots for their descendants. The processing time of ONTO-PERL is described by the left y-axis while the right y-axis describes the number of operations accessing the ontology (operations) and number of found descendants.

From Figure 7.5 we observe a mapping between the number of operations and processing time. In contrast, the number of returned items, measured along the left y-axis, only partially follows the other curves, instead following a pattern which correspond to the internal structure of the ontologies. In the measurements we observe insignificant differences, *i.e.* a difference on average less than 1%, and a maximum difference at benchmark-ontology at index 48 with a measurement difference of 3.5% between the samples.

Evaluating the impact of an algorithms redundant operations, we observe that a high number of redundant operations in ONTO-PERL translates into a significant increase in processing time, as illustrated in Figure 7.5. The correlation we identify between the processing time and number of operations an algorithm executes, provides an example of consequences when not applying well-established algorithms for ontology reasoning. Of special interest is the non-linear increase we observe between growth in the execution time and the number of memory accesses. The observation highlights the cost of iterating

through ontologies with a big internal relation-structure: the growth pattern in Figure 7.5 corresponds to the number of root-descendants, which we discussed in Figure 2.2 at page 9.

The benchmark ontologies with the biggest increase in measured processing time are those above the Level 2 cache limit, *i.e.* the ontologies at index 52, 52, 59, 60 and 61 in our measurements, where the last three ontologies were omitted due to our time threshold. The observation of a correlation between processing time and cache usage seems logic; when the chance of finding a given item in cache decreases, the processing time increases. The observation relates to our measurements of memory access patterns, from which we know that the running-time is bound to both the number of memory-accesses and the order of the memory accesses.

Comparing the query time in Figure 7.5 with the time of applying the rule based extension of the transitive closure (described in section 6.3 at page 75), we observe a similar pattern of performance:

1. the peaks in the measurements of performance are due to the number of ontology memory accesses;

2. the difference in performance between onto*Wiz* and ONTO-PERL is of more than six orders of magnitude in difference, which corresponds to the difference for the query which we calculated when generating Figure 7.5;

3. the performance of onto*Wiz* is relatively close to the memory speed offered by our test platform, which we observe from the underlying performance benchmarks (when compared to ONTO-PERL). The difference between the performance of onto*Wiz* corresponds to the number of executed queries, which is as expected given the discussion in section 4.1 at page 27.

The pattern of performance which we observe in the above list, correspond to the already discussed correlation between the number of operations an algorithm executes and processing time for the queries we evaluate. Investigating our assertion we evaluate the dependency between the number of relations in an ontology, the number of operations performed by an algorithm and the execution time, which is presented in Figure 7.6:

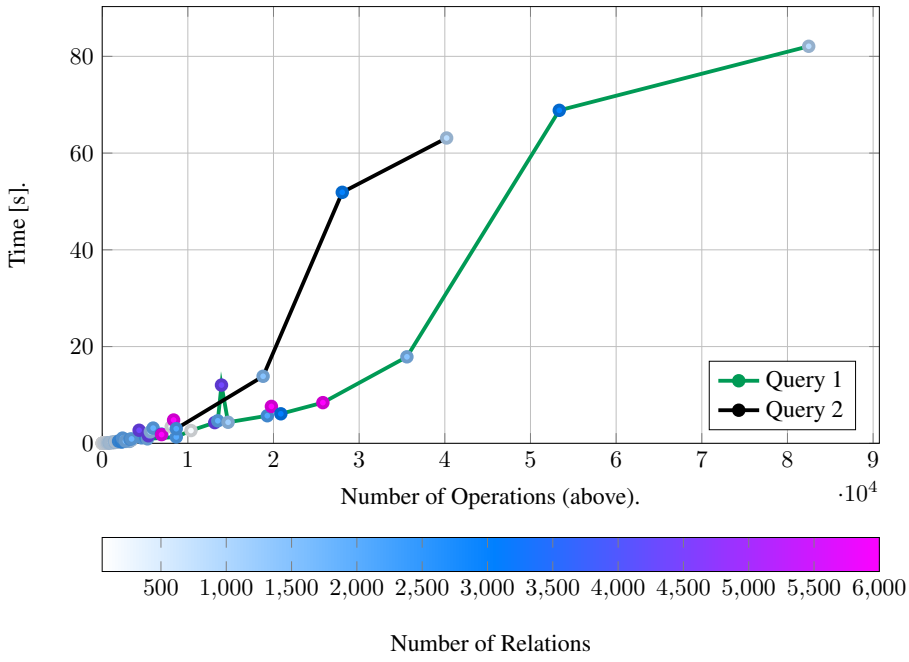

Figure 7.6: Comparison of queries showing correlation between memory accesses and processing time for ONTO-PERL. Two queries are compared: *Query 1* requests all the roots on an ontology and *Query 2* the descendents of the roots.

Figure 7.6 describe the impact of our queries using three parameters:

1. the number of explicit ontology relations using a color at each measurement-point;
2. the number of operations performed by an algorithm along the x-axis;
3. the execution time along the y-axis.

Given the color-scale (below the x-axis in the figure), measurement-points in a pink color are those with the biggest memory-consumption (when looking at the explicit relations of each ontology). To increase the level of details in our plot, we only present the 40 smallest ontologies with respect to file size. Of special interest is the observation that both queries we illustrate follow the same growth-pattern; a measurement point with a pink color has higher processing time than a measurement point with a light blue color and equal number of operations.

The execution time of a query depends on its number of operations and the size of an ontology, as seen in Figure 7.6. Given our underlying data we observe that the impact of

an algorithms operations and the size of the accessed ontology is influenced by a third parameter, *i.e.* the number intermediate memory accesses in a function. Inspecting the source code of ONTO-PERL we observe a relative high degree of insert and extract operations from a queue for *Query 2*, which correspond to the growth difference of the two queries in Figure 7.6.

Evaluating the importance of the memory space consumed by the relations of an ontology, we analyze the impact on processing time by reducing the area of Figure 7.6, which is seen in Figure 7.7:

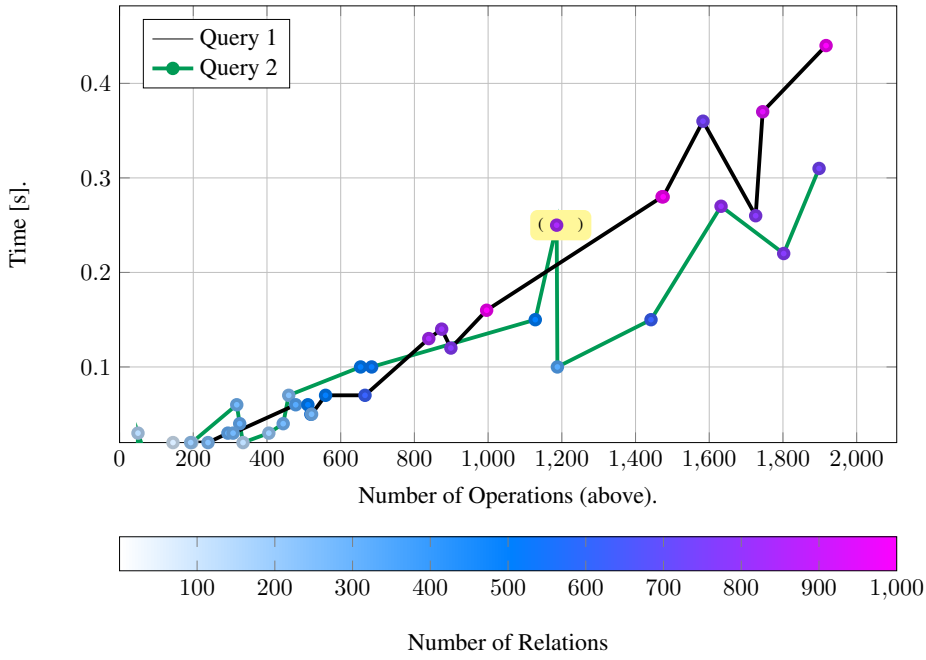

Figure 7.7: Detail from Figure 7.6.

To show the sensitivity of memory load, Figure 7.7 enlarges the details of the lowest measurements from Figure 7.6. The measurement point highlighted in yellow shows a correlation between the number of operations, accessed relations, and time spent on a query. When accessing small ontologies, it is more important to reduce the memory requirement of an ontology than the number of times it is accessed. The importance of compressing the set of relations for small ontologies, which we observe, correspond to how the cache works: when the ontology does not fit into cache, parts of it is swapped out of cache, a swapping which increases the cost of ONTO-PERL for accessing memory[40, 26, 64, 21].

7.3.3 Summary of the Performance Measurements

The motivation of our performance analysis was to understand why ontology reasoning tools under-perform. We have identified the components of the under-performance, *i.e.* the memory access patterns, the number of memory accesses and the size of the accessed memory space (*i.e.* size of the ontology).

A considerably waste of processing time was discovered when comparing the memory speed of the laptops hardware with the time for ONTO-PERL running a server and calculating our queries for the benchmark ontologies. We observed that ontoWiz had an processing time close to the memory speed of the hardware. We believe that other software's should apply our pre-processing approach in their own algorithms. From our memory benchmark at chapter 5 at page 33 we observed a large benefit from ordered list accesses, a conclusion which is supported in this section by the measurements of ontoWiz and ONTO-PERL.

Building of the ordered lists depends on the pre-processing of the ontology. The time of the pre-processing is considerably less than the time of parsing an ontology, *i.e.* the time to read an ontology from a file and into a data-structure. In section 2.3 at page 16 we stated that the parsing time is not regarded as a performance issue, from which we conclude that the time of the pre-processing can be left out from our comparison.

The set of rules covering our benchmark ontologies was described in section 2.2 at page 11. Rules are important for tasks such as hypothesis generation in the field of biomedical ontology engineering, such as querying the GO ontology as part of drug discovery[47]. In short, application of ontology contraction and expansion rules increases the range of possible queries. Given our measurements for transitive closures, and the rule based discussion in section 6.3 at page 75, we assess that the time cost of rule based queries for ontoWiz will be considerably less than the parsing time of ontologies. At page 1 we provided an example of ontology knowledge gathering, asking "How many generations have lapsed (in average) since the first Norwegian entered Norway?". A similar query is "Find all pairs of people where the first person is a (direct or indirect) ancestor of the second", which Hitzler *et al.*[33] describes as an intractable query with regard to processing time of reasoning tools. Given the rule based nature of the query, we expect ontoWiz to answer the query in a time similar to the time measurement for transitive closure, *i.e.* to be performed in a negligible amount of time.

Each executed query introduces a chance for a cache miss: when the number of independent queries increases, the processing time of ontoWiz decreases. Compared to the processing time of ONTO-PERL, the decrease in running time for ontoWiz is minimal, though compared to the optimal case of memory handling we observe a small overhead.

From our knowledge of the queries and the implementations of onto*Wiz* we know that the small overhead in memory handling for onto*Wiz* is due to the numbers of executed queries: if we reduce the number of queries, the memory handling overhead of onto*Wiz* is lowered. We are therefore interested in reducing the number of executed queries for a given task. An example from ONTO-PERL is the operation of intersecting ontologies, which ONTO-PERL does not support; using ONTO-PERL an ontology engineer must first query the ancestors/descendants before building an own algorithm intersecting the results, which increases the number of cache misses. To solve the issue we therefore suggest to increase the number of possible queries of our onto*Wiz* API by identifying dependencies among queries executed by the user.

The performance measurements which we have presented were done on a Dell laptop. We know from the research of Rivoire *et al.*[58] that a laptop uses considerably less energy for the same computations compared to a standard server. The high energy efficiency of laptops, when compared to the use of standard servers, implies that onto*Wiz* provides an impact of reduced energy consumption greater than six orders of magnitude for each of the computed queries. We therefore expect that ontoWiz can even contribute to reducing the carbon footprint of biomedical ontology reasoning.

7.4 Summary of onto*Wiz* and Future Work

We have in this chapter presented onto*Wiz*, which applies the cocO(n) library. From the presented facts we conclude that the cocO(n) software has great potential for high-speed reasoning. The preference of integrating cocO(n) as part of an ontology handling software is seen both with regard to the performance impact of ontology reasoning and the low integration-cost into existing software.

Onto*Wiz* provides support for more than 350 different functions in its API, meaning most of the functions in ONTO-PERL. A subset of the functions covered by ONTO-PERLis, however, not covered, and for some of these, onto*Wiz* may have a potential for speed-improvement. Examples of such functions are comparison of ontologies with regard to the ontologies attribute-set, or integration of a rule based pre-processing interface, as discussed at section 6.3.1 at page 84. We regard both examples as possible parts of our future work.

Our analysis of onto*Wiz* and cocO(n) indicate improvements in:

The running-time: Our benchmarks clearly states that onto*Wiz* is a high-performing tool for ontology reasoning, *i.e.* giving a speed-improvement of six orders of magnitude when compared to ONTO-PERL.

The implementation/extension time: The properties are stored using an internal grammar in our data storage scheme. Combined with the utilization of cocO(*n*), we assert that ontoWiz is easy to extend, both with regard to new functionality and connection to other program interfaces (*e.g.* to apply different front-ends than the ONTO-PERL API).

The correctness of code: Comparing our approach to the likelihood of errors in our use, by performing an informal evaluation of the tests coverage, we assert that both ontoWiz and cocO(*n*) are more correct than ONTO-PERL: while ONTO-PERL only performs a limited number of *Class level* tests, both ontoWiz and cocO(*n*) combine a three-layered formal evaluation of correctness with informal inspections (*i.e.* to detect errors).

Given this evaluation of ontoWiz, we conclude that ontoWiz covers the requirements (which was stated in Table 2.1 at page 18). Analyzing the implications of our approach, we will in the next chapter present the conclusions of our work.

Chapter 8

Conclusions and Future Work

In this master thesis we have presented a tool for high-speed ontology reasoning, ontoWiz/cocO(n). The ontoWiz/cocO(n) library is made freely available through our wiki (`https://code.google.com/p/ontowiz/`). The tremendous speed impact of six orders of magnitude which we have achieved is due to our algorithm and memory structures design, which is specially suited for biomedical ontologies. Its suitability and correctness are validated using queries on real-world ontologies. The library is designed to work both on commodity computers (*e.g.* a laptop) and advanced supercomputers, and is suitable for integration in existing software-pipelines avoiding the cost of software adjustment. Its ease and performance obviates expensive hardware and user threshold associated with using special-purpose systems such as supercomputers.

The core of our high performance approach is the pre-processing algorithm of cocO(n), which we presented in chapter 6 at page 49. The cocO(n) algorithm translates an ontology into disjoint sets of memory chunks, where each memory chunk correspond to a subset of a user-based query. Using the most beneficial memory access pattern, (discussed in section 5.3 at page 46), we have reduced the impact of memory latency.

The programming effort which we have presented in this master thesis, depends on our micro benchmarks. From the structural analysis of our benchmark-ontologies we formulated properties of the pre-processing algorithm. Combined with our micro benchmarks of memory access patterns, and our brief study of related work, we formulated the cocO(n) algorithm. Connecting the ontoWiz interface to cocO(n), we micro-benchmarked the performance of the cocO(n) algorithm. The performance analysis of ontoWiz/cocO(n) pointed to conclusions which was similar to our initial micro-benchmarks, *i.e.* the cost of redundant operations and type of memory access.

In our work, both with regard to micro benchmarks and implementation, we have focused on effective use of commonly available hardware. Combined with the approach of rule based query support, onto*Wiz*/cocO(*n*) provides high-speed support of tasks which due to their time-complexity used to be regarded as intractable by other software, *i.e.* as described in section 7.3.3 at page 107.

In our discussion, we have spanned the fields of life-science, ontology reasoning and hardware oriented programming techniques, such as pre-fetching, data-compression and locality.

The high speed of our approach makes it interesting to extend our work to new cases. In our work we have identified a set of properties that specify high-performance ontology reasoning tools. For the future we suggest to refine our performance micro benchmark. We have seen that the ability to profit from earlier measurements partly depends on the internal structure of the ontology. Querying in biomedical ontologies are performed for tasks such as hypothesis generation[12, 19], pattern identification[36] and reasoning in transcriptional gene expressions, *e.g.* in GreXKB[67]. For future, we want to increase the complexities of the supported queries and the number of ontologies to investigate the promise of onto*Wiz*/cocO(*n*).

Appendix A

Bibliography

[1] Abo, A. V., *et al*. (2007a). Compilers; Principles, Techniques and Tools. Addison Wesley, second edition.

[2] Alam, S. R., *et al*. (2006a). Characterizing Applications on the Cray MTA-2 Multithreading Architecture. In Proc. of CUG Conf.

[3] Andersson, A. (1993). Balanced search trees made simple. In F. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, Algorithms and Data Structures, volume 709 of Lecture Notes in Computer Science, pages 60–71. Springer Berlin Heidelberg.

[4] Antezana, E., *et al*. (2008a). ONTO-PERL: an api for supporting the development and analysis of bio-ontologies. Bioinformatics, **24**(6), 885–887.

[5] Antezana, E., *et al*. (2008b). Structuring the life science resourceome for semantic systems biology: lessons from the biogateway project. In Proceedings of the Workshop on Semantic Web Applications and Tools for Life Sciences (SWAT4LS): November 28, 2008; Edinburgh, United Kingdom.

[6] Antezana, E., *et al*. (2009 b). Biogateway: A Semantic Systems Biology Tool for the Life Sciences. BMC Bioinformatics, **10**, S11.

[7] Aranguren, M. E., *et al*. (2007b). Understanding and using the meaning of statements in a bio-ontology: recasting the gene ontology in owl. BMC bioinformatics, **8**(1), 57.

[8] Asanovic, K., *et al*. (2006b). The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[9] Bader, D., *et al*. (2005a). On the architectural requirements for efficient execution of graph algorithms. In Parallel Processing, 2005. ICPP 2005. International Conference on, pages 547 – 556.

[10] Bayer, R. (1972). Symmetric binary b-trees: Data structure and maintenance algorithms. Acta Informatica, **1**, 290–306. 10.1007/BF00289509.

[11] Bayer, R. *et al*. (1972). Organization and maintenance of large ordered indexes. Acta Informatica, **1**, 173–189.

[12] Bell, L., *et al*. (2011a). Integrated bio-entity network: a system for biological knowledge discovery. PloS one, **6**(6), e21474.

[13] Bellman, R. (1958). On a routing problem. Quarterly of Applied Mathematics, **16**, 87–90.

[14] Berry, J., *et al*. (2007c). Software and algorithms for graph queries on multithreaded architectures. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–14.

[15] Blonde, W. (2012). Metarel, an ontology facilitating advanced querying of biomedical knowledge. Ph.D. thesis, Department of Mathematical Modelling, Statistics and Bioinformatics, Ghent University, Ghent, Belgium.

[16] Blondé, W., *et al*. (2011b). Reasoning with bio-ontologies: using relational closure rules to enable practical querying. Oxford Bioinformatics, **27**, 1562–1568.

[17] Boeker, M., *et al*. (2011c). Unintended consequences of existential quantifications in biomedical ontologies. BMC bioinformatics, **12**(1), 456.

[18] Briggs, P. *et al*. (1993). An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems (LOPLAS).

[19] Callahan, A., *et al*. (2011d). Hyque: evaluating hypotheses using semantic web technologies. Journal of biomedical semantics, **2**(Suppl 2), S3.

[20] Chilimbi, T., *et al*. (2000). Making pointer-based data structures cache conscious. Computer, **33**(12), 67 – 74.

[21] Cormen, T. H., *et al*. (2001). Introduction to Algorithms. The MIT Press, second edition.

[22] Dijkstra, E. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, **1**(1).

[23] Dittrich, J.-P., *et al*. (2002). Progressive merge join: a generic and non-blocking sort-based join algorithm.

[24] Ekseth, O. K. (2012). Cache efficient ontology querying. Technical report, NTNU. A feasability study of improving ontology reasoning through increased utilization of a computers hardware.

[25] Ekseth, O. K., *et al*. (2010a). Turboortho – a high performance alternative to orthomcl. In European Conference on Computational Biology.

[26] Ferdman, M., *et al*. (2012a). Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 37–48, New York, NY, USA. ACM.

[27] Floyd, R. (1962). Algorithm 97: Shortest path. Commun. ACM, **5**(6), 345–345.

[28] Fredman, M. *et al*. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM, **34**(3), 596–615.

[29] Goodman, E. L., *et al*. (2011e). High-performance computing applied to semantic databases. In G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. Leenheer, and J. Pan, editors, The Semanic Web: Research and Applications, volume 6644 of Lecture Notes in Computer Science, pages 31–45. Springer Berlin Heidelberg.

[30] Guibas, L. J. *et al*. (1978). A dichromatic framework for balanced trees. In Proceedings of the 19th Annual Symposium on Foundations of Computer Science, pages 8–21, Washington, DC, USA. IEEE Computer Society.

[31] Haarslev, V. *et al*. (2001). High performance reasoning with very large knowledge bases: A practical case study. In International Joint Conference on Artificial Intelligence, volume 17, pages 161–168. LAWRENCE ERLBAUM ASSOCIATES LTD.

[32] Harispe, S., *et al*. (2013). From theoretical framework to generic semantic measures library. Forwarded by Dr. Mironov.

[33] Hitzler, P., *et al*. (2010b). Foundations of Semantic Web Technologies. CRC Press.

[34] Hoare, C. A. (1962). Quicksort. The Computer Journal, **5**(1), 10–16.

[35] Hoehndorf, R., *et al*. (2010c). Relations as patterns: bridging the gap between obo and owl. BMC bioinformatics, **11**(1), 441.

[36] Hollunder, J., *et al*. (2007d). Dass: efficient discovery and p-value calculation of substructures in unordered data. Bioinformatics, **23**(1), 77–83.

[37] Hopcroft, J. *et al*. (1971). Efficient algorithms for graph manipulation. Technical report, Stanford University, Stanford, CA, USA.

[38] Horrocks, I. *et al*. (1999). Optimizing description logic subsumption. Journal of Logic and Computation, **9**(3), 267–293.

[39] Hustadt, U., *et al*. (2005b). Data complexity of reasoning in very expressive description logics. In Proceedings of the 19th international joint conference on Artificial intelligence, IJCAI'05, pages 466–471, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[40] Jamriska, O., *et al*. (2012b). Cache-efficient graph cuts on structured grids. In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference, pages 3673–3680.

[41] Jensen, L. J. *et al*. (2010). Ontologies in quantitative biology: A basis for comparison, integration, and discovery. PLoS Biol, **8**(5), e1000374.

[42] Johnson, D. B. (1977). Efficient algorithms for shortest paths in sparse networks. J. ACM, **24**(1), 1–13.

[43] Jones, N. C. *et al*. (2004). An introduction to bioinformatics algorithms. The MIT Press, first edition.

[44] Kiupel, M., *et al*. (2003). West Nile virus infection in eastern fox squirrels (sciurus niger). Veterinary Pathology, **40**(6), 703–7.

[45] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society, **7**(1), 48–50.

[46] Lee, C. Y. (1961). An algorithm for path connections and its applications. Electronic Computers, IRE Transactions on, **EC-10**(3), 346 –365.

[47] Loging, W., *et al*. (2007e). High-throughput electronic biology: mining information for drug discovery. Nature reviews Drug discovery, **6**(3), 220–230.

[48] Lu, S.-L., *et al*. (2012c). Scaling the x201C memory wall x201D designer track. In Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on, pages 271–272.

[49] M. Fredman, J. K. *et al*. (1984). Storing a sparse table with 0(1) worst case access time. J. ACM, **31**(3), 538–544.

[50] M. Thuresson, L. S. *et al*. (2008). Memory-link compression schemes: A value locality perspective. IEEE Trans. Comput., **57**(7), 916–927.

[51] Madduri, K., *et al*. (2007f). An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In Workshop on Algorithm Engineering and Experiments (ALENEX).

[52] Martin, K., *et al*. (2005c). Mastering CMake: A cross-platform build system. Kitware Incorporated, fifth edition.

[53] Martínez-Prieto, M. A., *et al*. (2012d). Exchange and consumption of huge rdf data. In E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, editors, The Semantic Web: Research and Applications, volume 7295 of Lecture Notes in Computer Science, pages 437–452. Springer Berlin / Heidelberg.

[54] Meyer, U. *et al*. (2003). delta-stepping: a parallelizable shortest path algorithm. Journal of Algorithms, **49**(1), 114 – 152. 1998 European Symposium on Algorithms.

[55] Moore, E. (1957). The shortest path through a maze. Proceedings of an international sym- posium on the theory of switching, Part 1I,.

[56] Pesquita, C. *et al*. (2012). Predicting the extension of biomedical ontologies. PLoS Comput Biol, **8**(9), e1002630.

[57] Prim, R. C. (1957). Shortest connection networks and some generalizations. Bell system technical journal, **36**(6), 1389–1401.

[58] Rivoire, S., *et al*. (2007g). Models and metrics to enable energy-efficiency optimizations. Computer, **40**(12), 39–48.

[59] Rusell, S. *et al*. (2003). Artificial Intelligence, A Modern Approach. Prentice Hall, Person Education International, second edition.

[60] Schroeder, W., *et al*. (2006c). An Object-Oriented Approach To 3D Graphics. Kitware, forth edition.

[61] Schroeder, W. J. (2010). The VTK user's guide. Kitware, 11th edition.

[62] Smith, B., *et al*. (2005d). Relations in biomedical ontologies. Genome biology, **6**(5), R46.

[63] Smith, B., *et al*. (2007h). The obo foundry: coordinated evolution of ontologies to support biomedical data integration. Nature biotechnology, **25**(11), 1251–1255.

[64] Stallings, W. (2009). Operating systems; Internals and Design Principles. Pearson Education, sixth edition.

[65] Tarkoma, S. (2007). Chained forests for fast subsumption matching. In Proceedings of the 2007 inaugural international conference on Distributed event-based systems, pages 97–102. ACM.

[66] Tumeo, A., *et al*. (2012e). Designing next-generation massively multithreaded architectures for irregular applications. Computer, **45**(8), 53–61.

[67] Venkatesan, A., *et al*. (2012f). Towards an integrated knowledge system for capturing gene expression events. In Proceedings of the 3rd International Conference on Biomedical Ontology, KR-MED Series, Graz, Austria (R. Cornet and R. Stevens, eds.), volume 897, pages 85–90.

[68] Vuillemin, J. (1978). A data structure for manipulating priority queues. Commun. ACM, **21**(4), 309–315.

[69] W. Kainz, M. J. E. *et al*. (1993). Modeling spatial relations and operations with partially ordered sets. INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SYSTEMS, **7**, 215–229.

[70] Williams, J. W. J. (1964). Algorithm 232: heapsort. Communications of the ACM, **7**(6), 347–348.

Appendix B

An Analysis of ONTO-PERL's Implementation

In this appendix we briefly analyze ONTO-PERL with respect its implementation. From our knowledge of related tools in biomedical ontology engineering, the problems faced by ONTO-PERL are similar to those faced by other software (which was discussed in chapter 3 at page 21). It is therefore of interest gaining a proper understanding of the issues, which implies an investigation into the behavior of our use-case (*i.e.* ONTO-PERL). The analysis should cover the underlying source code, and actual performance with regard to running time and correctness of results. The source code of ONTO-PERL is inspected with regard to the type of memory storage scheme, the degree of indirect memory references (which is the ability to predict future memory accesses), and the number of times the ontology must be iterated in order to get the answers. The analyze makes it possible to evaluate the impact of

1. random accesses, which indicates the likelihood of cache misses;
2. redundant operations, *i.e.* the waste of work by ONTO-PERL;
3. the degree of false negatives produced by ONTO-PERL, which is due to logical errors in the Perl code.

The analysis is provided in Table B.1: for brevity we only include the result of our evaluation.

	Algorithm/Data-Flow			*Impact Of*		
Operation	**Storage Scheme**	**Indirect Reference**	**Ontology Iterations**	**Random Access**	**Redundant Operations**	**False Negatives**
Get attributes for a Term.	Hash	High	None	Med.	no	no
Get the set of relations for a given Term.	Hash	High	None	Med.	no	no
Get ancestor Terms	Hash	High	Med.	Med.	Med.	no
Get descendant Terms	Hash	High	Med.	Med.	Med.	no
Get restricted set of ancestor Terms, given either a relation-type of sub-name-space.	Hash	High	Med.	Med.	Med.	no
Get restricted set of descendant Terms, given either a relation-type of sub-name-space.	Hash	High	Med.	Med.	Med.	no
Get the paths connecting two Terms.	Hash	High	Med.	Med.	Med.	yes
Get the paths connecting *n* Terms.	Hash	High	Med.	High	Med.	yes
Get a sub-ontology from a set of root Terms.	Hash	High	High	High	High	no
Get union of two ontologies.	Hash	High	High.	High	High	no
Get intersection of two ontologies.	Hash	High	High	High	High	yes
Get an ontologies transitive reduction.	Hash	High	High	High	High	yes
Get an ontology's transitive closure.	Hash	High	High	High	High	yes

(Continued on next page.)

Table B.1 – continued from previous page.

Table B.1: The table describe the subset of ONTO-PERL's functionality which is regarded as a performance issue. The analyze focus on the type of storage schemes, and the number of ontology iterations (in the graph) which is required in order to answer a question (*i.e.* given user-based parameters to an operation).

Table B.1 describes the subset of operations classified as performance-bottlenecks. The classification combines inspection of the underlying source code, ONTO-PERL's documentation and discussion with the developers of ONTO-PERL. From the table we observe that:

1. ONTO-PERL uses hashes where a key (*e.g.* a relation) is stored as a string. Perl uses hashes as indirect memory references, which (with regard to running time) correspond to random access of memory. The implication of this random access is a high likelihood for accessing memory which is not pre-loaded into cache, *i.e.* of cache misses.

2. ONTO-PERL stores each attribute as a set of linked hashes. As a consequence, each operation require several indirect references (*i.e.* random memory accesses), as seen for the operation of returning the set of immediate children.

3. when running queries (*i.e.* tasks of reasoning), ONTO-PERL iterates through the ontology several times. Applying established algorithms it is possible to reduce the number of operations. The naive algorithms of ONTO-PERL contribute to an increased running time, among others due to the high number of redundant operations.

4. The simplification in the data-structure and complexity in the algorithms make ONTO-PERL rather error-prone: true relations are sometimes missed, giving rise to false negatives. We found no evidence that false positives are generated. These errors were unknown to the developers of ONTO-PERL. Results generated by ONTO-PERL should therefore be used with some care when verifying the result of onto*Wiz*.

The complexities of the approach that ONTO-PERL has followed make it hard to (a) extend its functionality, (b) verify correctness and (c) improve its performance. In short, ONTO-PERL does not provide a separation between the front-end ontology (object) model,

and the actual reasoning. An implication of the non-existing separation is the increased likelihood of under-performing memory accesses, as the ontology is stored in an order which is different from the order it is accessed.

Appendix C

Brief Survey of Graph Representations and their Application

The brief survey in this chapter was also presented in our previous work[24].

An ontology in a general form can be represented by a graph, offering a multitude of different implementations. As ontology reasoning is a subset of graph operations, understanding important contributions is of importance. Surveying other representations, we aim at extracting principles to be used in our suggested representation. With this purpose we briefly introduce the area of graph representations.

A graph G=(V,E) is defined as a set of vertices V with a set of edges E, where E(V) describes the set of edges incident on vertex $v \in V$. In this report all edges are assumed to be directed, and are referred to as arcs. Traversing it in a particular manner, algorithms to answer specific queries are equivalent to particular searches in ontological graphs.

Numerous ways exist to represent graphs and their components[21]. A *tree* in computer science (CS) is a set of linked vertices (nodes) simulating a hierarchical structure. In comparison with a tree, a forest does not limit the set of root vertices to one. A root-vertex in this context is a data-structure-object without inbound arcs to itself. Requiring a tree to comply with a property, a heap is formed. An example of such a property is the constraint of a vertex' children having a value equal- or less to itself, *i.e.* the max-heap property. Limiting the set of operations to insert- and extraction, the stack-type is developed. A stack is a data representation where the first element to be extracted corresponds to the last

inserted, therefore referred to as a LIFO (Last In First Out) structure. A corresponding data storage representation is the queue, where the last element to be inserted is the last to be extracted, *i.e.* a FIFO (First In, First Out) policy.

Representing the above terms in memory, the basic types of either a linked list or of an array is normally used. An array in this context constitutes a contiguous space in memory, accessed using offsets from the first allocated bytes. In contrast, linked lists (sometimes called adjacency list (`http://www.boost.org/doc/libs/1_51_0/libs/graph/doc/graph_theory_review.html`)) do not hold this linear order, as they are determined by the pointer in each object. A pointer in this context may here either be the offset in an array, or a memory address. Iterating through a linked list set therefore involves higher memory complexity compared to the alternative.

Accessing sorted sets, several algorithms have been developed. A sorted set satisfies the condition $array[i-1] <= array[i] <= array[i+1]$, where 'i' represents the memory offset. One such method is the "Insertion sort", regarded as efficient for sorting small sets, though for larger sets its $O(|V|^2)$ worst case running time makes it less tractable, where the set length $n = |V|$. Instead of storing the set in an array, "Heap-sort"[70] utilizes the max-heap-property, combining interchanging- and popping of vertices, with an $O\left(|V| * log(|V|)\right)$ worst-case running time. Splitting the set into smaller parts before sorting, "Merge-sort"[21] combines the subsets into a resulting order. A weakness of the latter approach, though it manages the sorting in $O(|V| * log(|V|))$, is the fact that it does not operate in-place with regard to memory.

In contrast "Quick-sort"[34] handles the data in-place- and like Insertion-sort provides a tight code, having an average-case running time of $O(|V| * log(|V|))$. Like merge-sort the latter is based on divide-and-conquer paradigm, conquering the subdivided arrays by sorting using recursive calls to itself. In the above sorting algorithms, information about the input sequence is gained comparing each element, *i.e.* they are classified as part of the "comparison sort" methodology. In contrast "Counting sort" determines the number of elements by counting the properties less than each of the values in the set. The performance of the latter is (k+n), where 'k' is the number of possible values, *i.e.* highest number in the set with zero as its lowest value. Including "counting sort" as a subroutine in the related "Radix sort", the range of the first is extended. Radix sort operates by first sorting the values at the rightmost radix (`http://en.wikipedia.org/wiki/Radix`) (*i.e.* least significant digit), and continues until all radices are in order. For the special case when values are in a uniform distribution, bucket sort manages a time complexity of $O(|V|)$ as average case. Extending the problem to sort data larger than the physical memory, Dittrich *et al.*[23] describes the "Progressive Merge-Join" method. In

the Progressive Merge-Join method an initial parallel phase is combined with a serial stage: For the parallel phase initial sorting is performed, *e.g.* using Quick-sort or merge-sort on subsets, before merging the partially ordered sets at the serial stage.

Algorithms later to be introduced, depends on efficient methods for search. The "Breadth First Search" (BFS)[46, 55] is an algorithm limited to the operations of exploring a vertex, and visit neighboring vertices. For this purpose the BFS uses a queue to store/retrieve the next vertex to process. A similar approach is followed by the "Depth First Search" (DFS)[37], with the only exception that the queue is replaced by a stack. Combining both, we get the iterative-depending algorithm[59].

Sorting a set in topological order, gives a horizontally aligned diagram with regard to the order of the dependencies. A common implementation is using the DFS for the task. The resulting diagram is when the vertices are found closest to the root at the leftmost side- and those without outgoing arcs to the rightmost side.

Building data representations for specific algorithms, functionality is integrated with the properties, such as maintaining a sorted order in a "binary search tree". In a binary search tree, the leftmost sub-tree has values less than- or equal to the rightmost sub-tree. The problem in the latter approach is the possibility of unbalanced trees when insertion- and deletions are performed. Solving this issue, "Red-black tree"[10, 30, 3] constrains the path from root to leaf, making the red-black trees approximately balanced. Similar to this approach, "B-trees"[11] were developed. Better optimized for I/O operations, they exist in several flavors, such as the the "B(+)" tree. The "B(+)" tree stores all the satellite information in the leaves, maximizing the branching factor. For the task of extracting the minimum- or maximum of a set, heap representations are designed. "Binomial heaps"[68] is a collection of binomial trees, *i.e.* an ordered tree defined recursively. A more complex structure is the "Fibonacci heap"[28], desirable when the number of times minimum are extracted- or an element is deleted is relatively few compared to other operations. In contrast with binomial heaps, "Fibonacci heaps" are rooted, but unordered. Due to the high programming complexity, the latter is rather to be regarded as of theoretical interest- than of practical importance[21]. When a set consists of subsets to be searched for, *e.g.* words in a dictionary, hashing is used.

A special case of hashing, *i.e.* of mapping subsets to numbers, is the concept of perfect hashing[49]. An interesting implementation is the cmph library (`http://cmph.sourceforge.net/`). The efficiency of such implementations, depends on the subset being static. A static set of subsets in this context implies that it is not changed, *e.g.* no word is added after hash-table construction.

Studying the graph topology for extracting sub/super sets, several problems exist. One

such is the task of finding a subset of edges including every vertex, *i.e.* the minimum spanning tree. Kruskal[45] generates the solution using a set of trees, *i.e.* a forest. At the running time of $O(|E| * log(|V|))$ it greedily selects the smallest forest to extend. In contrast Prim[57] maintains a single tree, starting at a given root vertex, using a depth-first-search selecting the optimal edge. Similar to Kruskal, Prim uses a greedy approach for the edge selection, thereby finding the edge giving the least increase in the total path length. A greedy approach is here understood as taking the best choice based on knowledge at a given instance.

Modifying the problem, we now require knowledge about the shortest paths linking a given vertex to the other vertices in a graph. A famous work is the dynamic programming[43] algorithm described by Dijkstra[22], where a greedy approach[43] is used solving the problem. Studying the greedy step, we observe it is similar to the one found in Prim, but with the exception that Dijkstra uses the total weight for evaluation, instead of Prims use of individual arc weights. In contrast, the algorithm described by Bellman and Ford (BF)[13] evaluates all of the edges as an alternative to the above greedy approach, answering the wider problem of finding all paths given a single source. As the size of graphs passes the size of physical memory, alternative algorithms are required. One such implementation is proposed by Madduri *et al.*[51] using the Δ-stepping parallel algorithm of Meyer[54].

Extending the problems size of memory-fittable graphs to cover all sources, the work of Floyd and Warshall[27] proposes an implementation using matrix-operations, with a time complexity of $\theta(|V|^3)$ comparisons in a graph. The problem with the latter algorithm is the expectation of a dense graph of vertices for optimal performance, resulting in an overhead compared to our expectation of sparse graph as input. Suggesting an alternative for dense graphs "Johnson's algorithm"[42] combines the step of negative-weight-adjustment using BF with the shortest-path-operation using Dijkstra for each vertex in the graph.

Summing up the essential qualities of the above brief survey, we observe the usage of important techniques, *e.g.* in-place memory handling and reuse of earlier calculations. An interesting aspect is the back-bone integration of data model in some of the alternatives. Regarding ontology reasoning, several representations describe approaches of vertex search- and sorting, while the connectivity problem is outlined in different flavors. The principles described in this section are therefore implemented in our suggested cocO(n) representation, and outlined in the next sections.

Appendix D

List of External Tools

Our implementation depends upon several tools. Supporting re-production of our results, we provide the extensive list of tools used during development, *i.e.* for

1. onto*Wiz*,
2. cocO(*n*) and
3. the memory-access-benchmark.

The set of tools, their task, accessibility and version number is provided in Table D.1:

Tag	Description
Tool-name	Cachegrind
Task	Collects memory access patterns of a program. Its collected data forms the basis for our memory-access-benchmark.
Homepage	`http://valgrind.org/docs/manual/cg-manual.html`
Version	(Part of the valgrind-installation-package; see Valgrinds version number.)
Tool-name	CMake
Task	Controls the software compilation process. Serves as the users interface during installation. Builds the releasable binaries (found at our home-page).
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://cmake.org/`
Version	2.8.7

(Continued on next page.)

Table D.1 – continued from previous page.

Tag	Description
Tool-name	Cmph
Task	A library for generating of fast key-value (hash) look-up. Requires the keys to be known before keys are mapped to the keys (*i.e.* strings). onto*Wiz* uses cmph to map string identifiers (*e.g.* term-ids) into indices.
Homepage	`http://cmph.sourceforge.net/`
Version	2.0
Tool-name	Doxygen
Task	Supports development and inspection of the onto*Wiz*, coc$O(n)$and benchmark-libraries. Were used building the documentation.
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://www.stack.nl/~dimitri/doxygen/`.
Version	1.7.6.1
Tool-name	g++
Task	Translate the C++ source code into machine-readable syntax: The C++ code was compiled with g++.
Homepage	`http://gcc.gnu.org/projects/cxx0x.html`
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Version	4.6.3
Tool-name	Make
Task	Organizes and improves the speed of software compilation.
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`https://www.gnu.org/software/make/`
Version	3.81
Tool-name	Mercurial
Task	A repository tool. Supports consistency of source code and the content of this document (*i.e.* the report).
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://mercurial.selenic.com/`
Version	2.0.2
Tool-name	Perl

(Continued on next page.)

Table D.1 – continued from previous page.

Tag	Description
Task	Interprets the Perl source code into machine-code.
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://www.perl.org/`
Version	5.14.12
Tool-name	SWIG
Task	Glues programming languages together, *e.g.* Perl and C++. SWIG is used in our project connecting the Perl layer into the C++ layer.
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://www.swig.org/`
Version	2.0.4
Tool-name	Time
Task	Measures system and user time.
Availability	As package from the Ubuntu repository.
Version	1.7-23.1
Tool-name	Valgrind
Task	Dynamically analyzes memory traffic on softwares. Used in our project to remove memory leakages, bug tracking and bug fixing.
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://www.valgrind.org/`
Version	1:3.7.0-0ubuntu3.1
Tool-name	VTK
Task	Visualizes scientific data. Used in this context to visualize ontologies of interest.
Availability	Either as package from the Ubuntu repository, or as source from the homepage.
Homepage	`http://www.vtk.org/VTK`
Version	5.10.1

Table D.1: The list of external tools, describing their accessibility. Included to support re-producability of our work.

Appendix E

Glossary

API Application Programming Interface; makes it possible for programs to communicate.

Arc An *arc* is a directed edge connecting two vertices.

BF The algorithm described by Bellman and Ford (BF) evaluates all of the edges as an alternative to the greedy approach, answering the problem of finding all paths given a single source.

BFS The *Breadth First Search* (BFS) is an algorithm limited to the operations of exploring a vertex, visiting neighboring vertices.

C++ A programming language, which supports low-level implementations. The benefit is higher speed of memory-sensitive operations at the cost of a longer implementation time (*i.e.* the time writing a software). Both cocO(n) and parts of onto*Wiz* are written in C++.

Cache *Cache* (or *CPU cache*) is the high-speed memory found in a computer. The cache is a physical component which is part of the CPU (core) architecture. When a processor requires instructions or data, a request is made to the cache.

Class For ontologies in the OWL format, a class corresponds to a specific type of a vertex, which in Description Logic (DL) is classified as a *concept*.

CPU The *Central Processing Unit* (CPU) is a physical component found in computers. The CPU is often understood as the computers command center. All applications

depends upon a CPU. A CPU may consist of several processors, which are connected on the same chip through data-paths. A CPU has several sub-units. One of them of them is the cache.

Depth The *depth* in one of our benchmark ontologies is the longest path connecting an arbitrary leaf vertex to an arbitrary root vertex.

Description Logic (DL) *DL* is a group of languages in which knowledge is formally represented. DL is used for the task of formally evaluating ontology properties.

DFS The *Depth First Search* (DFS) is similar to the BFS. The difference is that the queue (see *Queue*) is replaced by a stack (see *Stack*).

FIFO A *First In, First-Out* (FIFO) policy is a data storage representation where the last element to be inserted is the last to be extracted

Forest A *forest* in computer science (CS) is a set of linked vertices. In contrast to a tree (see *Tree*), a forest does not limit the set of root vertices to one.

Graph A *graph* is defined as a set of vertices V with a set of edges E, where E(V) describes the set of edges connected to a vertex $v \in V$.

Heap A *heap* is a tree shaped memory storage scheme in which the order of the elements in the tree must comply with a given set of properties, called the *heap properties*. An example of a property is the *min-heap* property, where the minimum value of the data-set is found at the root, and the value of each vertex must be less than or equal to its children.

ILP The *Instruction-Level Parallelism* (ILP) increases memory speed when instructions (*i.e.* set of commands which a program consists of) are transported from the computers memory (see *Memory*) into the CPU. Similar to MLP.

Instance Given the context of ontologies in the OWL/OBO format, an instance corresponds to a specific type of a vertex, which in Description Logic (DL) is classified as an *individual*.

L1 cache The *L1 cache* is a part of a computers cache (see *cache*), where the L1 cache is both the fastest and smallest cache: on our test platforms the L1 cache size is either 32KB or 64KB (for details see Table 4.1 at page 31).

L2 cache The *L2 cache* is a part of a computers cache (see *cache*). Compared to the L1 cache (see *L1 cache*) the L2 cache is both slower and bigger, with a size on our test platforms of either 512KB or 3072KB.

Leaf A *leaf* (often denoted as *root-vertex*) is a data-structure-object without outbound arcs.

LIFO A *Last In First Out* in a data storage is a representation where the first element to be extracted corresponds to the last inserted. This is often denoted as a *stack*.

Memory The *memory* of a computer is often denoted as *Random Access Memory* (*RAM*), and is a set of physical chips which is connected to the CPU through data-paths. The memory size on our test platforms varies from 2.79GB on our Dell laptop to 125GB on the Biogw-db server (for details see Table 4.1 at page 31).

MLP The *Memory-Level Parallelism* (MLP) reduces delay when data (*e.g.* parts of an ontology) are transported from memory (see *Memory*) into the CPU. Similar to ILP.

Naive algorithm An algorithm which does not reuse knowledge (*i.e.* collected facts) from earlier operations.

Node Corresponds to a *vertex*.

OBOF *Open Biological and Biomedical Ontology Format* (*OBOF*); one of the formats which ontologies are written in.

OWL *Ontology Web Language* (*OWL*); like OBO, it is one of the formats that ontologies can be represented in.

Path A *path* is a set of relations to traverse in order to get from one vertex to another in the ontology.

Perl *Perl* is a programming language. Provides an abstraction from the memory layout of modern computers, which implies ease of writing code at the cost of reduced speed when writing software dependent of fast memory access. Parts of the onto*Wiz* library are written in Perl.

Poset A *Partially ordered set* (*poset*) is an ontology where all of the relations have the properties of *reflexivity*, *anti-symmetry* and *transitivity*.

Queue A *queue* is a memory storage scheme applying the FIFO (see *FIFO*) order.

RAM See *Memory*.

RDFS *Resource Description Framework Scheme* (*RDFS*); like OWL and OBO it is one of the formats that ontologies can be represented in.

Relation A connection between vertices.

Relation type A *relation type* labels a relation with a set of properties. An example is the `is_a` relation type, which labels the relation with the reflexivity, anti-symmetry and transitivity properties.

Root A *root* (often denoted as *root-vertex*) is a data-structure-object without inbound arcs.

Stack A *stack* is a memory storage scheme applying the LIFO (see *LIFO*) order.

Term For ontologies in the OBO format, a *term* corresponds to a specific type of a vertex, which in Description Logic (DL) is classified as a *concept*.

Transitive The *transitive* property concerns relations. Illustrating the transitive property, we provide an example: if vertex A relates to vertex B through a transitive relation type α and vertex B relates to vertex C through a transitive relation type α, then given transitivity vertex A relates to C through relation type α.

Tree A *tree* in computer science (CS) is a set of linked vertices (nodes) representing a hierarchical structure.

Triplet An ordered 3-tuple to represent a relation, *i.e.* (vertex1, relation type, vertex2).

Vertex A *vertex* is an individual of a *graph*.

Vertex cover A *vertex cover*, in the context of ontology pre-processing, is understood as the set of ancestors which a vertex relates to.

Vertices Plural for *vertex*. When two vertices are connected through a relation type, they form a triplet of a relation.

VTK *Visual Tool Kit* (VTK); an open source visualization library covering a wide area of fields (`http://vtk.org/`), used in this context to visualize ontologies of interest.

Width The *width* in one of our benchmark ontologies is the number of leaf vertices in the ontology.

Index