Towards a Concurrent Approximate Description Logic Reasoner

Raj Kamal Yadav[1], Gunjan Singh[1], Raghava Mutharaju[1], Sumit Bhatia[2]

[1] Knowledgeable Computing and Reasoning Lab, IIIT-Delhi, India
{raj16076,gunjans,raghava.mutharaju}@iiitd.ac.in
[2] IBM Research AI, Delhi, India
sumitbhatia@in.ibm.com

1 Introduction

Ontologies are used in a variety of applications in domains such as healthcare, geoscience, IoT, and e-commerce. Many commonly used ontologies are manually constructed, and may not be large in size but could be very expressive. With the advances in automated knowledge base construction and ontology learning [2], it is becoming increasingly common to build very large ontologies. OWL 2 knowledge representation language, a W3C standard, provides several profiles such as OWL 2 Full, OWL 2 DL, OWL 2 EL, OWL 2 QL, and OWL 2 RL that vary in terms of their expressivity, and hence, the complexity of reasoning. While there are several existing reasoners [1] for different OWL profiles, including OWL 2 DL, they generally do not scale well for large ontologies and even medium-sized ontologies in the OWL 2 DL profile.

Approximate reasoning [5] offers an attractive alternative in such cases by sacrificing either soundness or completeness in favor of reasoning runtime. Approximate reasoning is useful in applications where i) response time is crucial, or ii) the reasoning is performed in a resource-constrained environment, and iii) *good enough* answers from the reasoner are sufficient. TrOWL [4], is a well known approximate description logic reasoner that uses syntactic language weakening for approximating $SROIQ$ TBox axioms to $\mathcal{EL}^{++}$ axioms. It also makes use of data structures to maintain complement and cardinality information. While offering significant improvements in reasoning runtime, TrOWL does not scale well for large ontologies (see Section 2). An alternate way of reducing the reasoning runtime is to better utilize the computing resources (multi-core, multi-processor architectures). ELK [3] is one such concurrent rule-based reasoner for the description logic $\mathcal{EL}_{\perp}^{+}$. Axioms are assigned to lock-free data structures called *contexts*, and inferences are computed independently and in parallel offering significant speedup when compared to the state-of-the-art.

In this paper, we describe our ongoing efforts for developing a concurrent, approximate description logic reasoner. We propose to make use of the approximation rules of TrOWL and the concurrent strategy of ELK to create an efficient and concurrent approximate reasoner.

2 Efficiency and Scalability of Existing Reasoners

In this section, We compare the runtime performance of TrOWL (an approximate reasoner), ELK (a concurrent reasoner), and three other commonly used reasoners (Pellet, Hermit, and JFact). We use five different ontologies of varying sizes – GALEN (37,696 axioms), GO (107,909 axioms), FMA (126,548 axioms), Anatomy (268,513 axioms),

	Anatomy			**FMA**			**Galen**			**GO**			**SNOMED CT**		
Copies	1	5	10	1	5	10	1	5	10	1	5	10	1	5	10
ELK (1)	28.23	383.95	1203.28	4.86	37.65	87.37	3.30	11.71	25.63	5.66	32.44	130.30	38.89	254.02	505.97
ELK (5)	10.84	115.01	799.94	3.37	28.23	60.16	1.89	5.77	12.42	3.74	16.97	122.54	20.48	161.63	317.19
ELK (10)	7.44	80.92	621.48	3.2	25.92	67.12	1.75	4.88	10.18	3.68	20.23	121.11	14.89	139.06	261.75
TrOWL-EL	4.34	35.31	79.96	7.99	62.08	212.41	63.91	jh	jh	75.5	520.93	1168.5	104.04	1212.15	to
TrOWL-DL	4.47	36.08	85.61	7.14	62.15	205.43	79.26	jh	jh	72.95	462.1	1170.59	105.35	1208.82	to
Pellet	so	so	so	580.62	3490.06	to	gc	gc	gc	to	gc	gc	369.49	gc	gc
Hermit	to	to	to	21.18	330.2	1178.44	gc	gc	gc	742.74	to	to	1810.86	to	to
JFact	so	so	so	410.36	to	to	gc	gc	gc	to	to	to	to	to	to

Table 1. Runtime (in seconds) taken by different reasoners for different ontologies. In the table, so represents Stack Overflow Error due to Heap Queue, to represents Timeout Exception, gc represents Garbage Collection Overflow and jh respresents Java Heap Space OverFlow errors.

and Snomed (569,701 axioms). In order to further test the reasoners' scalability, we created interlinked copies of ontologies where independent copies of the ontologies are connected with each other. For example, this could lead to axioms such as $A_1 \sqsubseteq B_2$, where A_1 and B_2 are copies of classes A and B in the original ontology. We report classification times achieved by different reasoners with 1, 5, and 10 interlinked copies (Table 1). The experiments were performed on an Intel(R) Xeon(R) 2.30GHz $x86\_64$ server with 96GB RAM, 40 CPUs, 20 cores per CPU, and 2 threads per core. The maximum Java Heap Space allocated was 24GB and the timeout was set to 3600 seconds. The source code for replicating our experiments and creating interlinked ontologies is at `https://github.com/kracr/dl-approx-reasoner`. We observe from Table 1 that except for 1 copy of FMA ontology, JFact is not able to complete the reasoning tasks with in the specified time (1 hour) using available computing resources. Pellet and Hermit perform slightly better with successful completion of the tasks for FMA ontology, but they also do not scale to other ontologies and their larger interlinked variants. Both the variants of TrOWL successfully complete the tasks, except for larger variants of Galen (5 and 10 copies) and Snomed (10 copies). ELK on the other hand is able to successfully complete the tasks for all the ontologies, and their larger interlinked variants. Also note that except for Anatomy ontology, ELK outperforms TrOWL due to its concurrent architecture. For Anatomy, however, TrOWL outperforms ELK (even with 10 parallel processes). We speculate that this could be attributed to the nature of axioms present in the ontology and a large number of missing results due to TrOWL being a sound, but incomplete reasoner. We also note that the runtimes reduce with increasing the number of parallel workers employed by ELK (shown in parentheses). It was observed during experiments that large gains in performance could be achieved by increasing the number of parallel processes, and the gains saturate at about 10 parallel workers (Fig. 2). These experimental observations provide the *motivating evidence* for our efforts towards *a concurrent approximate description logic reasoner* that can scale to large, expressive ontologies and perform reasoning tasks efficiently in resource constrained environments.

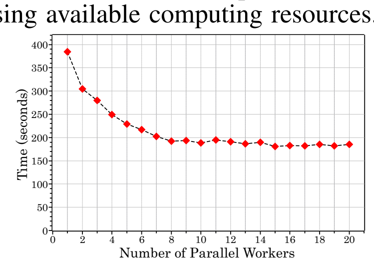

Fig. 1. ELK Performance for 8 interlinked copies of Snomed using different no. of parallel workers

3 Concurrent Approximate DL Reasoner

We extend the completion rules of ELK reasoner [3] with the complement and cardinality rules of TrOWL [4], which are 8 in number. These rules are sound but not complete. We translate these 8 rules into a form that is suitable for parallel processing using ELK style lock-free data-structures. We used ELK notations and translated the 8 complement and cardinality rules into 10 rules.

Algorithm 1: C.process(expression): Subsumption

C.process(*Sub*(D)):
  if *C.subs.add*(D) **then**
    if *bottom.negOccurs>0* **then**
      for each D_2,*G* **with** *C.subs*(D_2)**and** *conCompl*(D_2, G) **and** *conEq*(D, G) **do**
        C.enqueue($add(newSub(bottom))$) // $R_\neg$
    for each *G,H* **with** *conCompl*(D, G) **and** *conCompl*(C, H) **do**
      G.enqueue($newSub(H)$) // $R_\neg^+$
    if $D =$ *bottom* **and** *C* **instanceOf** *IdxConjunction* **and** *conCompl*($C.secondConj, G$) **then**
      C.firstConj.enqueue($newSub(G)$) // $R_\neg^-$
    if *D* **instanceOf** *IdxCardinality* **then**
      D.filler.enqueue(
      $newBackCardLink($
      $D.card, D.role, C)$
      C.enqueue(
      $newForwCardLink($
      $D.card, D.role, D.filler)$ // R_c^-
    for each *R,S,E,F* **with** *hier*(R, S) **and** *C.backCardLink*(i, R, E) **and** *D.backCardLink*(j, S, F) **do**
      E.enqueue($newSub(F)$) // $R_\sqsubseteq^c$

Algorithm 2: C.process(expression): Existential Links

C.process(*BackLink*(R, E)):
  if *C.backLink.add*(R, E) **then**
    for each D, R_2, S_1, S_2, S **with** *C.forwCardLink*(i, R_2, D) **and** *roleComp*(S_1, S_2, S) **and** *hier*(R, S_1) **and** *hier*(R_2, S_2) **do**
      D.enqueue($newBackLink(S, E)$)
      E.enqueue($newForwLink(S, D)$) // R_{o2}^c
C.process(*ForwLink*(R, E)):
  if *C.forwLink.add*(R, E) **then**
    for each E, R_1, S_1, S_2, S **with** *C.backCardLink*(j, R_1, E) **and** *roleComp*(S_1, S_2, S) **and** *hier*(R_1, S_1) **and** *hier*(R, S_2) **do**
      D.enqueue($newBackLink(S, E)$)
      E.enqueue($newForwLink(S, D)$) // R_{o2}^c

Algorithm 3: C.process(expression): Cardinality Links

C.process(*BackCardLink*(i, R, E)):
  if *C.backCardLink.add*(i, R, E) **then**
    if *C.subs.contains*($bottom$) **then**
      E.enqueue($newSub(bottom)$) // $R_c^\perp$
    for each *D,F,S* **with** *C.subs*(D) **and** *negCard*(i, S, D) **and** *hier*(R, S) **do**
      E.enqueue($newSub(F)$) // R_c^+
    for each D, R_2, S_1, S_2, S **with** *C.forwLink*(R_2, D) **and** *roleComp*(S_1, S_2, S) **and** *hier*(R, S_1) **and** *hier*(R_2, S_2) **do**
      D.enqueue($newBackLink(S, E)$)
      E.enqueue($newForwLink(S, D)$) // R_{o1}^c
    for eachD, R_2, S_1, S_2, S **with** *C.forwCardLink*(j, R_2, D) **and** *roleComp*(S_1, S_2, S) **and** *hier*(R, S_1) **and** *hier*(R_2, S_2) **do**
      D.enqueue($newBackLink(S, E)$)
      E.enqueue($newForwLink(S, D)$) // R_{o3}^c
    C.enqueue($Init$);
C.process(*ForwCardLink*(i, R, D)):
  if *C.forwCardLink.add*(i, R, D) **then**
    for each E, R_1, S_1, S_2, S **with** *C.backLink*(R_1, E) **and** *roleComp*(S_1, S_2, S) **and** *hier*(R_1, S_1) **and** *hier*(R, S_2) **do**
      D.enqueue($newBackLink(S, E)$)
      E.enqueue($newForwLink(S, D)$) // R_{o1}^c
    for each E, R_1, S_1, S_2, S **with** *C.backCardLink*(j, R_1, E) **and** *roleComp*(S_1, S_2, S) **and** *hier*(R_1, S_1) **and***hier*(R, S_2) **do**
      D.enqueue($newBackLink(S, E)$)
      E.enqueue($newForwLink(S, D)$) // R_{o3}^c

From among these 8 rules, one of the rules (R13 from [4]) is split into two rules (R_c^- and R_c^+) and the rule involving $\perp$ ($R_c^\perp$) has been added. Since the rules are from TrOWL but are recast into ELK style inference rules, the tractability and soundness proofs from TrOWL can be carried over as well (but we will not show them here due to lack of space). These 10 rules are given below.

$$\mathrm{R}_\neg\ \frac{C \sqsubseteq D \quad C \sqsubseteq E}{C \sqsubseteq \perp} : \begin{array}{c} D \equiv \neg E \\ \perp \text{ occurs negatively in } \mathcal{O} \end{array} \qquad R_\neg^+\ \frac{C \sqsubseteq D}{\neg D \sqsubseteq \neg C} \qquad R_\neg^-\ \frac{C \sqcap D \sqsubseteq \perp}{C \sqsubseteq \neg D}$$

$$R_c^\sqsubseteq\ \frac{A \sqsubseteq B \qquad X \xrightarrow{\text{i,R}} A \qquad Y \xrightarrow{\text{j,S}} B}{X \sqsubseteq Y} : R \sqsubseteq_{\mathcal{O}}^{\star} S \qquad R_c^\perp\ \frac{E \xrightarrow{\text{i,R}} C \quad C \sqsubseteq \perp}{E \sqsubseteq \perp} \qquad R_c^-\ \frac{E \sqsubseteq iR.C}{E \xrightarrow{\text{i,R}} C}$$

$$R_c^{+}\ \frac{E \xrightarrow{i,R} C \quad C \sqsubseteq D}{E \sqsubseteq iS.D} : \begin{array}{c} R \sqsubseteq_{\mathcal{O}}^{\star} S \\ iS.D \text{ occurs negatively in } \mathcal{O} \end{array} \qquad R_o^{c_1}\ \frac{E \xrightarrow{\text{i,R}} C \quad C \xrightarrow{R_2} D}{E \xrightarrow{\text{S}} D} : \begin{array}{c} R \sqsubseteq_{\mathcal{O}}^{\star} S_1 \\ R_2 \sqsubseteq_{\mathcal{O}}^{\star} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array}$$

$$R_o^{c_2}\ \frac{E \xrightarrow{\text{R}} C \quad C \xrightarrow{i,R_2} D}{E \xrightarrow{\text{S}} D} : \begin{array}{c} R \sqsubseteq_{\mathcal{O}}^{\star} S_1 \\ R_2 \sqsubseteq_{\mathcal{O}}^{\star} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array} \qquad R_o^{c_3}\ \frac{E \xrightarrow{\text{i,R}} C \quad C \xrightarrow{j,R_2} D}{E \xrightarrow{\text{S}} D} : \begin{array}{c} R \sqsubseteq_{\mathcal{O}}^{\star} S_1 \\ R_2 \sqsubseteq_{\mathcal{O}}^{\star} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array}$$

For efficient look-up of side conditions of inference rules, different look-up tables are constructed. For example, *negConjs* holds conjuncts of the form $C \sqcap D$. For the complement and cardinality rules, we added two additional look-up tables, *conCompl* and *negCard*. conCompl consists of pairs $\langle A, \neg A \rangle$ for each concept A in the ontology. negCard holds information about qualified cardinality expressions in the form of tuple $\langle i, S, D, iS.D \rangle$. For concurrent execution, *contexts* are assigned to each class expression on the basis of the inference rules. Every context c has a separate queue *c.Todo* and a set *c.Closure* which helps in achieving concurrency without using locks. c.Todo holds expressions that are yet to be processed and are initialized with the axioms from the input ontology. c.Closure holds all the processed expressions to which inference rules have already been applied. The expressions in c.Todo are represented with the corresponding number of parameters (Sub(D), BackLink(R, E), ForwLink(R, C), BackCardLink(R, E), ForwCardLink(R, C)), whereas the expressions in Closure are represented using tables for the respective types of expression (D $\in$ C.subs, $\langle R, E \rangle \in$ C.backLinks, $\langle R, C \rangle \in$ E, forwLinks $\langle i, R, E \rangle \in$ C.backCardLinks, $\langle i, R, C \rangle \in$ E.forwCardLinks). Note that two links (Back and Forw) are being used for both Existential and Cardinality because let's say we have expression $E \xrightarrow{\text{R}} C$; this expression would be added to both contexts E and C and similarly for cardinality $E \xrightarrow{\text{i,R}} C$ (unlike expression of the form $C \sqsubseteq D$ which will be added to context C only). Adding an expression to contexts activates it and each item is taken by some worker (thread) for execution.

The pseudocode for the 10 rules of complement and cardinality are given in Algorithms 1, 2, and 3. When a worker takes an expression from c.Todo, c.process(expression) is called and depending on the type of that expression, one method from Algorithm 1, 2, or 3 is executed. On calling c.process(expression), expression is added to its corresponding c.closure and inferences are performed between elements of c.closure by applying inference rules given in the method body. Also, other than the expressions from c.closure, data structures such as concIncs, roleComps, conCompl, negCard, hier etc are used for efficient look-up of side conditions. Note that here we have given only additional rules in each method body but inferences would be drawn using these new rules and the original rules provided in ELK. We are currently implementing the proposed algorithms and modifications and plan to make the implementation available for the community to use and build upon.

References

1. Antoniou, G., et al.: A Survey of Large-Scale Reasoning on the Web of Data. The Knowledge Engineering Review **33**. https://doi.org/10.1017/S0269888918000255
2. Asim, M.N., Wasim, M., Khan, M.U.G., Mahmood, W., Abbasi, H.M.: A survey of ontology learning techniques and applications. Database **2018** (2018)
3. Kazakov, Y., Krötzsch, M., Simančík, F.: The Incredible ELK: From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ ontologies. Journal of Automated Reasoning **53**(1), 1–61 (2014)
4. Ren, Y., Pan, J.Z., Zhao, Y.: Soundness Preserving Approximation for TBox Reasoning. In: AAAI. pp. 351–356 (2010)
5. Rudolph, S., Tserendorj, T., Hitzler, P.: What Is Approximate Reasoning? In: Web Reasoning and Rule Systems. pp. 150–164. Lecture Notes in Computer Science (2008)