Benchmarking Inference Algorithms for Probabilistic Relational Models

Tristan Potten and Tanya Braun[0000−0003−0282−4284]

Institute for Information Systems, University of Lübeck, Lübeck
{potten, braun}@ifis.uni-luebeck.de

Abstract. In the absence of benchmark datasets for inference algorithms in probabilistic relational models, we propose an extendable benchmarking suite named ComPI that contains modules for automatic model generation, model translation, and inference benchmarking. The functionality of ComPI is demonstrated in a case study investigating both average runtimes and accuracy for multiple openly available algorithm implementations. Relatively frequent execution failures along with issues regarding, e.g., numerical representations of probabilities, show the need for more robust and efficient implementations for real-world applications.

Keywords: StaRAI · Lifted inference · Probabilistic inference.

1 Introduction

At the heart of many machine learning algorithms lie large probabilistic models that use random variables (randvars) to describe behaviour or structure hidden in data. After a surge in effective machine learning algorithms, efficient algorithms for inference come into focus to make use of the models learned or to optimise machine learning algorithms further [5]. This need has lead to advances in probabilistic relational modelling for artificial intelligence (also called statistical relational AI, StaRAI for short). Probabilistic relational models combine the fields of reasoning under uncertainty and modelling incorporating relations and objects in the vein of first-order logic. Handling sets of indistinguishable objects using representatives enables tractable inference [8] w.r.t. the number of objects.

Very few datasets exist as a common baseline for comparing different approaches beyond models of limited size (e.g., epidemic [11], workshops [7], smokers [18]). Therefore, we present an extendable benchmarking suite named ComPI (Compare Probabilistic Inference) that allows for benchmarking implementations of inference algorithms. The suite consists of (1) an automatic generator for models and queries, (2) a translation tool for providing the generated models in the format needed for the implementations under test, and (3) a measurement module that batch-executes implementations for the generated models and collects information on runtimes and inference results.

In the following, we begin with a formal definition of the problem that the benchmarked inference algorithms solve. Afterwards, we present the functions of the modules of ComPI. Lastly, we present a case study carried out with ComPI.

2 Inference in Probabilistic Relational Models

The algorithms considered in `ComPI` solve query answering problems on a model that defines a full joint probability distribution. In this section, we formally define a model and the query answering problem in such models. Additionally, we give intuitions about how query answering algorithms solve these problems.

2.1 Parameterised Models

Parameterised models consist of parametric factors (parfactors). A parfactor describes a function, mapping argument values to real values (potentials). Parameterised randvars (PRVs) constitute arguments of parfactors. A PRV is a randvar parameterised with logical variables (logvars) to compactly represent sets of randvars [9]. Definitions are based on [13].

Definition 1. *Let* $\mathbf{R}$ *be a set of randvar names,* $\mathbf{L}$ *a set of logvar names,* Φ *a set of factor names, and* $\mathbf{D}$ *a set of constants (universe). All sets are finite. Each logvar* L *has a domain* $\mathcal{D}(L) \subseteq \mathbf{D}$*. A* constraint *is a tuple* $(\mathcal{X}, C_{\mathbf{X}})$ *of a sequence of logvars* $\mathcal{X} = (X_1, \ldots, X_n)$ *and a set* $C_{\mathcal{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$*. The symbol* $\top$ *for* C *marks that no restrictions apply, i.e.,* $C_{\mathcal{X}} = \times_{i=1}^{n} \mathcal{D}(X_i)$*. A* PRV $R(L_1, \ldots, L_n), n \geq 0$ *is a syntactical construct of a randvar* $R \in \mathbf{R}$ *possibly combined with logvars* $L_1, \ldots, L_n \in \mathbf{L}$*. If* $n = 0$*, the PRV is parameterless and constitutes a propositional randvar. The term* $\mathcal{R}(A)$ *denotes the possible values (range) of a PRV* A*. An* event $A = a$ *denotes the occurrence of PRV* A *with range value* $a \in \mathcal{R}(A)$*. We denote a* parfactor g *by* $\phi(\mathcal{A})_{|C}$ *with* $\mathcal{A} = (A_1, \ldots, A_n)$ *a sequence of PRVs,* $\phi : \times_{i=1}^{n} \mathcal{R}(A_i) \mapsto \mathbb{R}^+$ *a function with name* $\phi \in \Phi$*, and* C *a constraint on the logvars of* $\mathcal{A}$*. A set of parfactors forms a* model $G := \{g_i\}_{i=1}^{n}$*.*

The term $gr(P)$ denotes the set of all instances of P w.r.t. given constraints. An instance is a grounding, substituting the logvars in P with constants from given constraints. The *semantics* of a model G is given by grounding and building a full joint distribution P_G. Query answering refers to computing probability distributions, which boils down to computing marginals on P_G. Lifted algorithms seek to avoid grounding and building P_G. A formal definition follows.

Definition 2. *With* Z *as normalising constant, a model* G *represents the full joint distribution* $P_G = \frac{1}{Z} \prod_{f \in gr(G)} f$*. The term* $P(Q|\mathbf{E})$ *denotes a* query *in* G *with* Q *a grounded PRV and* $\mathbf{E}$ *a set of events. The* query answering problem *refers to solving a query w.r.t.* P_G*.*

2.2 Exact Lifted Inference Algorithms

The very first lifted algorithm is lifted variable elimination (LVE) [9], which has been further refined [11,7,13,2]. LVE takes a model G and answers a query $P(Q|\mathbf{E})$ by absorbing evidence $\mathbf{E}$ and eliminating all remaining PRVs except Q from G (cf. [13] for further details). With a new query, LVE restarts with the original model. Therefore, further research concentrates on efficiently solving

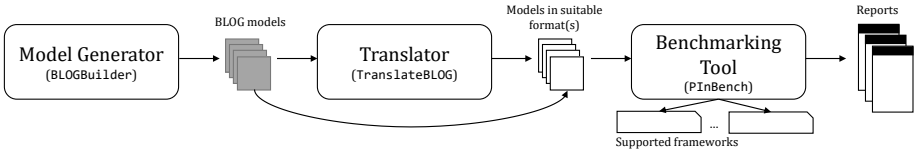

Fig. 1. Modules and workflow of `ComPI`.

multiple query answering problems on a given model, often by building a helper structure based on a model, yielding algorithms such as (i) the lifted junction tree algorithm (LJT) [1], (ii) first-order knowledge compilation (FOKC) [18,17], or (iii) probabilistic theorem proving (PTP) [3]. LJT solves the above defined query answering problem while FOKC and PTP actually solve a weighted first order model counting (WFOMC) problem to answer a query $P(Q|\mathbf{E})$.

For all algorithms mentioned, implementations are freely available (see Section 3.3). LVE and LJT work with models as defined above. The other algorithms usually use a different modelling formalism, namely Markov logic networks (MLNs) [10]. One can transform parameterised models to MLNs and vice versa [16]. Next, we present `ComPI`.

3 The Benchmarking Suite `ComPI`

`ComPI` allows for collecting runtimes and inference results from implementations given automatically generated models. Figure 1 shows a schematic description of `ComPI` consisting of three main parts, (i) a *model generator* `BLOGBuilder` that allows the automatic creation of multiple models (in the BLOG grammar [6]) following different generation strategies, (ii) a *translator* `TranslateBLOG` for translation from the BLOG format to correct input format for individual frameworks (if necessary), and (iii) a *benchmarking tool* `PInBench` for collection of runtimes and inference results, summarized in multiple reports.

The modules can be accessed at https://github.com/tristndev/ComPI. Below, we briefly highlight each module.

3.1 Model Generation

The goal is to generate models as given in Definition 1, generating logvar/randvar names and domain sizes, combining logvars and randvars as well as forming parfactors with random potentials. The input to `BLOGBuilder` is a model creation specification, which selects a model creation and augmentation strategy, which describe how models are created possibly based on a previous model.

Running `BLOGBuilder` creates a number of models as well as a number of reports and logs that describe specific characteristics of the created models. The output format of the models is BLOG. Additionally, `BLOGBuilder` generates

reports describing the model creation process and highlighting possible deviations from the model creation specification. The module can be extended with additional model creation and augmentation strategies.

3.2 Model Translation

The task is to translate models from the baseline BLOG format into equivalent models of those different formats required for the frameworks in `PInBench`. Accordingly, `TranslateBLOG` takes a number of model files in the BLOG format as input. It subsequently translates the parsed models into different formats of model specifications. As of now, the generation of the following output types is possible: (i) Markov logic networks (MLNs) [10], (ii) dynamic MLNs [4], and (iii) dynamic BLOG files. `TranslateBLOG` is easily extendable as new output formats can be added by specifying and implementing corresponding translation rules and syntactic grammar to create valid outputs.

3.3 Benchmarking

The final module in `ComPI` is `PInBench` (Probabilistic Inference Benchmarking), which serves to batch process the previously created model files, run inferences, and collect data on these runs. `PInBench` can be interpreted as a control unit that coordinates the running of an external implementation which it continuously and sequentially supplies with the available model files. The following implementations in conjunction with the corresponding input formats are supported:

- GC-FOVE[1] with propositional variable elimination and LVE,
- WFOMC[2] with FOKC,
- Alchemy[3] with PTP and sampling-based alternatives, and
- the junction tree algorithm[4] in both its propositional and lifted form.

We also developed a dynamic version of `PInBench`, named `DPInBench`, for dynamic models, i.e., models with a sequence of state, which could refer to passage of time. Currently, the implementations of UUMLN, short for University of Ulm Markov Logic Networks[5] and the Lifted Dynamic Junction Tree (LDJT)[6]) as well as their input formats are supported. The data collected by `PInBench` is stored in multiple reports, e.g., giving an overview on the inference success per file and query (queries on big, complex models might fail) or summarizing the run times and resulting inference probabilities.

Implementations not yet supported can easily be added by wrapping each in an executable file and specifying calls needed for execution. Additionally, parsing logics for generated outputs need to be implemented to extract information.

[1] dtai.cs.kuleuven.be/software/gcfove (accessed 16 Apr. 2020)
[2] dtai.cs.kuleuven.be/software/wfomc (accessed 16 Apr. 2020)
[3] alchemy.cs.washington.edu/ (accessed 16 Apr. 2020)
[4] ifis.uni-luebeck.de/index.php?id=518#c1216 (accessed 16 Apr. 2020)
[5] uni-ulm.de/en/in/ki/inst/alumni/thomas-geier/ (accessed 16 Apr. 2020)
[6] ifis.uni-luebeck.de/index.php?id=483 (accessed 16 Apr. 2020)

4 Case Study

To demonstrate the process of benchmarking different implementations using `ComPI`, we present an exemplary case study. Within the case study, the process consists of model generation with `BLOGBuilder`, model translation with `TranslateBLOG`, and benchmarking of inference runs with `PInBench`. All implementations currently supported by `ComPI` are included here. We do not consider sampling-based algorithms implemented in Alchemy as performance highly depends on the parameter setting for sampling, which requires an analysis of its own and is therefore not part of this case study.

4.1 Model Generation

Model generation starts with creating base models given a set of specified parameters (number of logvars/randvars/parfactors, domain sizes). Subsequently, these base models are augmented according to one of the following strategies:

- **Strategy A** - *Parallel Factor Augmentation*: The previous model is cloned and each parfactor extended by one additionally created PRV with randomly chosen existing logvars.
- **Strategy B** - *Increment By Model*: The base model is duplicated (renaming names) and appended to the current. A random randvar from the duplicate is connected with a random randvar of the current model via a new parfactor.

The strategies are set up to increase complexity based on LVE. Strategy A increases the so-called tree width (see, e.g., [12] for details). Strategy B increases the model size, while keeping the tree width close to constant.

It is non-trivial to generate series of models that increase in complexity without failing executions. Overall, we intend the generation of "balanced" models with not too strongly connected factors to allow all methods to demonstrate their individual strengths and weaknesses while maintaining manageable runtimes. Creating multiple base models with random influences (e.g., regarding relations between model objects) leads to multiple candidate model series, which allows for selecting one series that leads to runs with the least amount of errors. We tested multiple parameter settings for both strategies to generate candidate series. The specific numbers for parameters are random but small to generate models of limited size, ensuring that the tested programs successfully finish running them. We vary the domain sizes while keeping the number of logvars small for the models to be liftable from a theoretical point of view.[7] More precisely, the settings given by the Cartesian product of the following parameters have been evaluated for generating base models for each strategy:

- **Strategy A**: `domain_size` $\in \{10, 100, 1000\}$, `#logvar` $\in \{2\}$, `#randvar` $\in \{3, 4, 5\}$, `#factor` = `#randvar` $- 1$, `max_randvar_args` $= 2$. Augmentation in 16 steps.

[7] Models with a maximum of two logvars per parfactor are guaranteed to have inference runs without any groundings during its calculations [15,14].

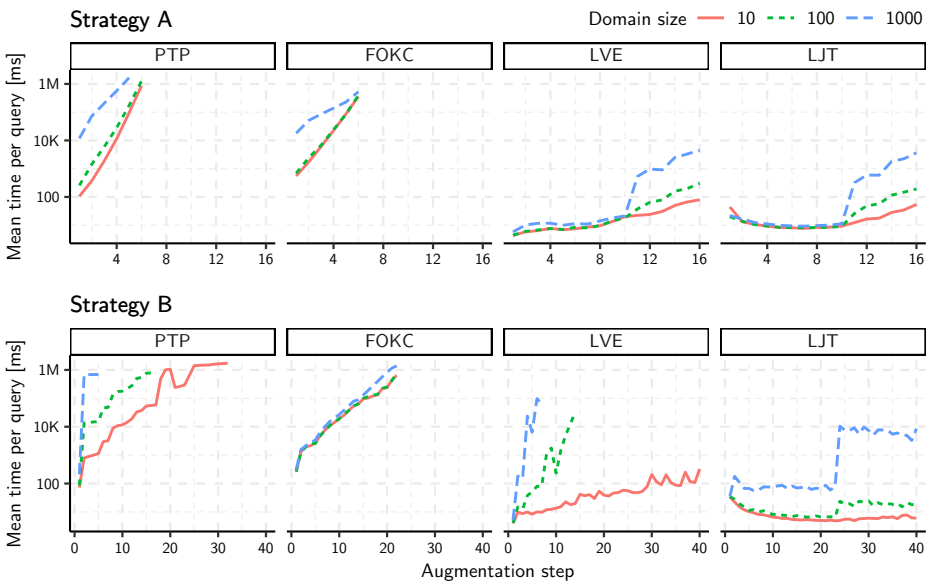

Fig. 2. Mean inference times per query (with logarithmic y-axis).

– **Strategy B**: `domain_size` $\in \{10, 100, 1000\}$, `#logvar` $\in \{1\}$, `#randvar` $\in \{2\}$, `#factor` $\in \{1\}$. Further restrictions: `max_randvar_occurrences` $= 4$, `max_randvar_args` $= 2$, `max_factor_args` $= 2$. Augmentation in 40 steps.

In each model file, one query is created per randvar. Model generation was executed in three independent runs to obtain multiple candidate models due to the included factors of randomness. The randomness may also lead to model series of potentially varying complexity. Preliminary test runs have led to the selection of the model series investigated below.

4.2 Evaluation Results

We analyse the given frameworks regarding two aspects: Firstly, runtimes of inference and secondly, inference accuracy.

Runtimes Figure 2 shows runtimes, displaying the relation between augmentation steps, domain sizes, and mean query answering times for PTP, FOKC, LVE and LJT. The two propositional algorithms supported are not shown in the plots as both presented very steep increases along with failures on early augmentation steps for bigger domain sizes (as expected without lifting).

Regarding *Strategy A*, the mean time per query increases for bigger domain sizes for all frameworks. PTP and FOKC have a similar early increase and fail on the models after augmentation steps 6 and 7, respectively. They are the only implementations working with MLNs as input. As the original models have

random potentials for each argument value combination in each parfactor, the translated MLNs do not have any local symmetries, which these algorithms would be able to exploit. Generating models with many local symmetries, e.g., only two different potentials per whole parfactor, may lead to better results for FOKC and PTP, with the remaining algorithms taking longer. Another possible explanation may (of course) also be that there are bugs in implementations or employed heuristics may be improvable. One could actually use `ComPI` to test an implementation with random inputs for bugs or better heuristics. The lifted algorithms LJT and LVE have steeper increases for high augmentation steps and the biggest domain size. The reason lies in so-called count-conversions that have a higher complexity than normal elimination operations and become necessary starting with augmentation step 11.

A technical note on count conversions: A count-conversion is a more involved concept of lifted inference where a logical variable is counted to remove it from the list of logical variables [13]. A count-conversion involves reformulating the parfactor, which enlarges it. A PRV is replaced by a so called counted PRV (CRV), which has histograms as range values. Consider a Boolean PRV $R(X)$ with three possible X values, then a CRV $\#_X[R(X)]$ has the range values $[0, 3]$, $[1, 2]$, $[2, 1]$ and $[3, 0]$, which denote that given a histogram $[n_1, n_2]$, n_1 ground instances of $R(X)$ have the value *true* and n_2 instances the value $false$. Given the range size r of the PRV and the domain size n of the logical variable that gets counted, the new CRV has $\binom{n+r-1}{n-1}$ range values, which lies in $O(r^n)$ [12]. In the case study, we use boolean PRVs, so $r = 2$. If n is small, the blow up by a count conversion is easily manageable. However, with large n, the blow up leads to a noticeable increase in runtimes, which is the reason why the increase in runtimes with step 11, at which point count conversion become necessary, is more noticeable with larger domain sizes.

Given *Strategy B*, the collected times display less extreme increases over the augmentation steps compared to Strategy A. PTP and FOKC exhibit similar behaviour compared to Strategy A. Again, changing domain sizes has little effect on FOKC. LVE only manages to finish models with small domain sizes as the models become too large overall. LJT is able to finish the models even for larger augmentation steps. The reason is that runtimes of LJT mainly depend on the tree width, i.e., the models of Strategy B have roughly the same complexity during query answering for LJT. The jump between augmentation step 23 and 24 again can be explained by more count conversions occurring.

Inference Accuracy Investigating the actual inference results presents an alternative approach to analysing collected data. Since the tested implementations all perform exact inference, we expect the implementations to obtain the same results on equivalent files and queries. Looking at accuracy allows for identifying bugs. If adding implementations of approximate inference algorithms to `ComPI`, one could compare accuracy of approximate inference against exact results. Figure 3 shows a comparison of probabilities queried for one (smaller) model generated for this case study. In this case, the implementations under in-

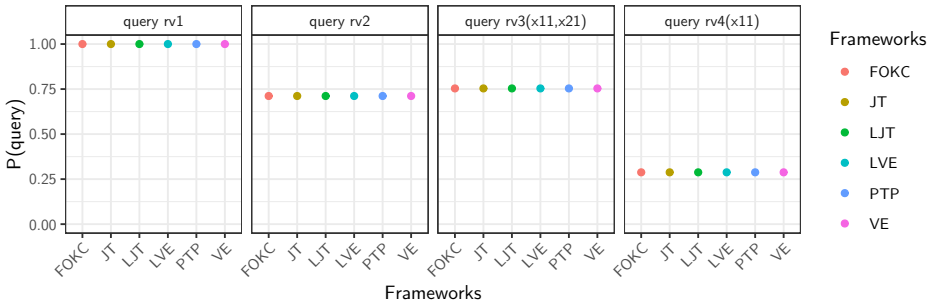

Fig. 3. Comparison of inference results between different frameworks.

vestigation calculated identical results for all shown queries. However, examining models created at later augmentation steps reveals that increased model complexities lead to errors in the outputted probabilities or even to no interpretable values at all. The latter case is caused by flaws in the numerical representations: `NaN` values occur when potentials get so small that they become 0 if represented as a float. When normalising a distribution, the normalising constant is a sum over 0, leading to dividing 0 by 0, which results in a `NaN`.

5 Conclusion

We present `ComPI`, a benchmarking suite for comparing various probabilistic inference implementations. `ComPI` consists of extendable tools for automatic model generation, model translation, and inference benchmarking. A case study demonstrates the simplicity of comparative analyses carried out with `ComPI`. Similar analyses are needed in the evaluation of future novel algorithms in the field of probabilistic inference. The extensibility of `ComPI` has the potential to reduce the efforts for these evaluations.

For the tested implementations, it becomes evident that there is still a need to provide bulletproof implementations. Occurring issues range from high memory consumption to the lack of robustness regarding heuristics used by the algorithms, exact numerical representations of probabilities, or reliable handling of errors. Future work will need to address these points.

References

1. Braun, T., Möller, R.: Preventing Groundings and Handling Evidence in the Lifted Junction Tree Algorithm. In: Proceedings of KI 2017: Advances in AI. pp. 85–98. Springer (2017)
2. Braun, T., Möller, R.: Parameterised Queries and Lifted Query Answering. In: IJCAI-18 Proceedings of the 27th International Joint Conference on AI. pp. 4980–4986. IJCAI Organization (2018)
3. Gogate, V., Domingos, P.: Probabilistic Theorem Proving. In: UAI-11 Proceedings of the 27th Conference on Uncertainty in AI. pp. 256–265. AUAI Press (2011)

4. Kersting, K., Ahmadi, B., Natarajan, S.: Counting Belief Propagation. In: UAI-09 Proceedings of the 25th Conference on Uncertainty in AI. pp. 277–284. AUAI Press (2009)
5. LeCun, Y.: Learning World Models: the Next Step Towards AI. Invited Talk at IJCAI-ECAI 2018 (2018), https://www.youtube.com/watch?v=U2mhZ9E8Fk8, accessed November 19, 2018
6. Milch, B., Marthi, B., Russell, S., Sontag, D., Long, D.L., Kolobov, A.: BLOG: Probabilistic Models with Unknown Objects. In: IJCAI-05 Proceedings of the 19rd International Joint Conference on AI. pp. 1352–1359. IJCAI Organization (2005)
7. Milch, B., Zettelmoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted Probabilistic Inference with Counting Formulas. In: AAAI-08 Proceedings of the 23rd AAAI Conference on AI. pp. 1062–1068. AAAI Press (2008)
8. Niepert, M., Van den Broeck, G.: Tractability through Exchangeability: A New Perspective on Efficient Probabilistic Inference. In: AAAI-14 Proceedings of the 28th AAAI Conference on AI. pp. 2467–2475. AAAI Press (2014)
9. Poole, D.: First-order Probabilistic Inference. In: IJCAI-03 Proceedings of the 18th International Joint Conference on AI. pp. 985–991. IJCAI Organization (2003)
10. Richardson, M., Domingos, P.: Markov Logic Networks. Machine Learning **62**(1–2), 107–136 (2006)
11. de Salvo Braz, R., Amir, E., Roth, D.: Lifted First-order Probabilistic Inference. In: IJCAI-05 Proceedings of the 19th International Joint Conference on AI. pp. 1319–1325. IJCAI Organization (2005)
12. Taghipour, N., Davis, J., Blockeel, H.: First-order Decomposition Trees. In: NIPS-13 Advances in Neural Information Processing Systems 26, pp. 1052–1060. Curran Associates, Inc. (2013)
13. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. Journal of Artificial Intelligence Research **47**(1), 393–439 (2013)
14. Taghipour, N., Fierens, D., Van den Broeck, G., Davis, J., Blockeel, H.: Completeness Results for Lifted Variable Elimination. In: AISTATS-13 Proceedings of the 16th International Conference on AI and Statistics. pp. 572–580. AAAI Press (2013)
15. Van den Broeck, G.: On the Completeness of First-order Knowledge Compilation for Lifted Probabilistic Inference. In: NIPS-11 Advances in Neural Information Processing Systems 24. pp. 1386–1394. Curran Associates, Inc. (2011)
16. Van den Broeck, G.: Lifted Inference and Learning in Statistical Relational Models. Ph.D. thesis, KU Leuven (2013)
17. Van den Broeck, G., Davis, J.: Conditioning in First-Order Knowledge Compilation and Lifted Probabilistic Inference. In: AAAI-12 Proceedings of the 26th AAAI Conference on AI. pp. 1961–1967. AAAI Press (2012)
18. Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., De Raedt, L.: Lifted Probabilistic Inference by First-order Knowledge Compilation. In: IJCAI-11 Proceedings of the 22nd International Joint Conference on AI. pp. 2178–2185. IJCAI Organization (2011)