Why is Compiling Lifted Inference into a Low-Level Language so Effective?*

Seyed Mehran Kazemi and **David Poole**
The University of British Columbia
Vancouver, BC, V6T 1Z4
{smkazemi, poole}@cs.ubc.ca

Abstract

First-order knowledge compilation techniques have proven efficient for lifted inference. They compile a relational probability model into a target circuit on which many inference queries can be answered efficiently. Early methods used data structures as their target circuit. In our KR-2016 paper, we showed that compiling to a low-level program instead of a data structure offers orders of magnitude speedup, resulting in the state-of-the-art lifted inference technique. In this paper, we conduct experiments to address two questions regarding our KR-2016 results: 1- does the speedup come from more efficient compilation or more efficient reasoning with the target circuit?, and 2- why are low-level programs more efficient target circuits than data structures?

Probabilistic relational models (Getoor, 2007; De Raedt et al., 2016) (PRMs), or template-based models (Koller and Friedman, 2009), are extensions of Markov and belief networks (Pearl, 1988) that allow modelling of the dependencies among relations of individuals, and use a form of exchangeability: individuals about which there exists the same information are treated identically. The promise of lifted probabilistic inference (Poole, 2003; Kersting, 2012) is to carry out probabilistic inference for a PRM without needing to reason about each individual separately (grounding out the representation) by instead exploiting exchangeability to count undistinguished individuals.

The problem of lifted probabilistic inference was first explicitly proposed by Poole (2003), who formulated the problem in terms of parametrized random variables, introduced the use of splitting to complement unification, the parametric factor (parfactor) representation of intermediate results, and an algorithm for summing out parametrized random variables and multiplying parfactors in a lifted manner. This work was advanced by the introduction of counting formulae, the development of counting elimination algorithms, and lifting the aggregator functions (de Salvo Braz, Amir, and Roth, 2005; Milch et al., 2008; Kisynski and Poole, 2009; Choi, de Salvo Braz, and Bui, 2011; Taghipour, Davis, and Blockeel, 2014). The main problem with these proposals is that they are based on variable elimination. Variable elimination (Zhang and Poole, 1994) (VE) is a dynamic programming approach which requires a representation of the intermediate results, and the current representations for such results are not closed under all operations used for inference.

An alternative to VE is to use search-based methods based on conditioning such as recursive conditioning (Darwiche, 2001), AND-OR search (Dechter and Mateescu, 2007) and other related works (e.g., Bacchus, Dalmao, and Pitassi (2009)). While for lifted inference these methods require propositionalization in the same cases VE does, the advantage of these methods is that conditioning simplifies the representations rather than complicating them, and these methods exploit context specific independence (Boutilier et al., 1996) and determinism. The use of lifted search-based inference was proposed by Jha et al. (2010), Gogate and Domingos (2011) and Poole, Bacchus, and Kisynski (2011). These methods take as input a probabilistic relational model, a query, and some observations, and output the probability of the query given the observations.

Van den Broeck et al. (2011) and Kazemi and Poole (2016) follow a knowledge compilation approach to lifted inference by evaluating a search-based lifted inference algorithm symbolically (instead of numerically) and extracting a target circuit on which many inference queries can be efficiently answered. While the target circuit used by Van den Broeck et al. (2011) is a data structure, Kazemi and Poole (2016) showed that using a low-level program as a target circuit is more efficient and results in orders of magnitude speedup. In a simultaneous work, Wu et al. (2016) showed that compiling to a low-level program is effective for BLOG (Milch et al., 2005) and offers substantial speedup.

Two issues remained unanswered in Kazemi and Poole (2016)'s results: 1- they compared end-to-end (compiling to a target circuit and reasoning with the circuit) run-times of their work with Van den Broeck et al. (2011)'s weighted first-order model counting, leaving the question of where exactly the speedup comes from, and 2- the actual reason behind the speedup gained by compiling to a program instead of a data structure remained untested. In this paper, we answer to these two issues. We conduct our experiments on Markov logic networks (MLNs) (Richardson and Domingos, 2006) and argue that our results also hold for other representations (e.g., (Jaeger, 1997; De Raedt, Kimmig, and Toivonen, 2007; Suciu et al., 2011; Kazemi et al., 2014)).

*In IJCAI-16 Statistical Relational AI Workshop.

Background and Notations

A **population** is a set of **individuals** (a.k.a. things, entities or objects). The **population size** is a nonnegative integer indicating the cardinality of the population. A **logical variable** is written in lower case and is typed with a population. For a logical variable x, we let Δ_x and $|\Delta_x|$ represent the population associated with x and its cardinality respectively. A lower case letter in bold represents a tuple of logical variables. Constants, denoting individuals, are written starting with an upper-case letter. A **term** is a logical variable or a constant.

A **parametrized random variable (PRV)** consists of a k-ary predicate R and k terms t_i and is represented as $R(t_1,\ldots,t_k)$. If every t_i is a constant, the PRV corresponds to a random variable. When k = 0, we omit the parentheses. A **grounding** of a PRV can be achieved by replacing each of the logical variables with one of the individuals in their domains. A **literal** is an assignment of a value to a PRV. We represent $R(\ldots) = True$ and $R(\ldots) = False$ by $r(\ldots)$ and $\neg r(\ldots)$ respectively. A **world** is a truth assignment to all groundings of all PRVs. A **formula** is made up of literals connected with logical connectives (conjunctions and disjunctions).

Following Kazemi and Poole (2016), we represent a **weighted formula (WF)** as a triple $\langle L,F,w\rangle$, where L is a set of logical variables, F is a formula whose logical variables are a subset of L, and w is a real-valued weight. For a given WF $\langle L,F,w\rangle$ and a world ω, we let $\eta(L,F,\omega)$ represent the number of assignments of individuals to the logical variables in L for which F holds in ω.

Markov Logic Networks

A **Markov Logic Network (MLN)** consists of a set ψ of WFs. It induces the following probability distribution:

$$Prob(\omega) = \frac{1}{Z} \prod_{\langle L,F,w\rangle \in \psi} \exp(\eta(L,F,\omega) * w) \qquad (1)$$

where ω is a world and

$$Z = \sum_{\omega'} \Big(\prod_{\langle L,F,w\rangle \in \psi} (\exp(\eta(L,F,\omega') * w)) \qquad (2)$$

is the partition (normalization) function. In this paper, we focus on calculating the partition function as many inference queries on MLNs reduce to calculating the partition function. Following Kazemi and Poole (2016), we assume the formulae in WFs of MLNs are in conjunctive form.

Example 1. Consider an MLN over three PRVs $R(x,m)$, $S(x,m)$ and $T(x)$, where $\Delta_x = \{X_1,X_2,X_3,X_4,X_5\}$ and $\Delta_m = \{M_1,M_2\}$, with the following WFs:

$\{\langle\{x,m\}, r(x,m) \wedge s(x,m), 1.2\rangle, \langle\{x,m\}, s(x,m) \wedge t(x), 0.2\rangle\}$

and a world ω in which $R(X_1,M_1)$, $S(X_1,M_1)$, $S(X_1,M_2)$ and $T(X_1)$ are $True$ and the other ground PRVs are $False$. Then $\eta(\{x,m\}, r(x,m) \wedge s(x,m), \omega) = 1$ as there is only one assignment of individuals to x and m ($x = X_1$ and $m = M_1$) for which $r(x,m) \wedge s(x,m)$ holds in ω and $\eta(\{x,m\}, s(x,m) \wedge t(x), \omega) = 2$. Therefore:

$$Pr(\omega) = \tfrac{1}{Z}(\exp(1 * 1.2) * \exp(2 * 0.2))$$

An MLN can be conditioned on a PRV having no logical variables by replacing the PRV in all formulae of WFs with its observed value. Observations on individuals can be handled by a process called shattering.

Example 2. Suppose for the MLN in Example 1 we observe that $T(X_1)$ and $T(X_2)$ are $True$. Since we have more information about X_1 and X_2 compared to other individuals in Δ_x, the individuals in Δ_x are no longer exchangeable. In order to handle this, we create two new logical variables x_1 and x_2 with $\Delta_{x_1} = \{X_1,X_2\}$ representing the individuals for which we have observed T is $True$, and $\Delta_{x_2} = \{X_3,X_4,X_5\}$ representing the individuals for which we have not observed T. Then we create new WFs with our new logical variables as follows:

$$\{\langle\{x_1,m\}, r(x_1,m) \wedge s(x_1,m), 1.2\rangle,$$
$$\langle\{x_2,m\}, r(x_2,m) \wedge s(x_2,m), 1.2\rangle,$$
$$\langle\{x_1,m\}, s(x_1,m) \wedge t(x_1), 0.2\rangle,$$
$$\langle\{x_2,m\}, s(x_2,m) \wedge t(x_2), 0.2\rangle\}$$

Then we replace $t(x_1)$ with $True$. For every logical variable x in the above MLN, the individuals in Δ_x are now exchangeable. This process is called shattering. We assume our input MLNs have been shattered based on the observations and refer interested readers to (de Salvo Braz, Amir, and Roth, 2005) for the details.

An MLN can be also conditioned on some counts: the number of times a PRV with one logical variable is $True$ or $False$. For a PRV $T(x)$, we let $Obs(T(x),i)$ represent a count observation on $T(x)$ indicating T is $True$ for exactly i out of $|\Delta_x|$ individuals.

Example 3. Suppose for the MLN in Example 1 we observe $Obs(T(x),2)$. We create two new logical variables x_1 and x_2 representing the subset of x having T $True$ and $False$ respectively, with $|\Delta_{x_1}| = 2$ and $|\Delta_{x_2}| = 3$.[1] Then we create new WFs as in Example 2, and replace $t(x_1)$ with $True$ and $t(x_2)$ with $False$.

Search-based Lifted Inference Rules

There are several rules that are used in search-based lifted inference algorithms. In this section, we describe some of these rules using examples.

Lifted Decomposition

Example 4. Consider the MLN in Example 1. On the relational level, all PRVs are connected to each other and we only have one connected component. On the grounding, however, for every individual $X_i \in \Delta_x$, we have the following WFs mentioning X_i:

$$\{\langle\{\}, r(X_i,M_1) \wedge s(X_i,M_1), 1.2\rangle,$$
$$\langle\{\}, s(X_i,M_1) \wedge t(X_i), 0.2\rangle,$$
$$\langle\{\}, r(X_i,M_2) \wedge s(X_i,M_2), 1.2\rangle,$$
$$\langle\{\}, s(X_i,M_2) \wedge t(X_i), 0.2\rangle\}$$

Notice that the WFs mentioning X_i in the grounding are totally disconnected from the other WFs. Therefore, we have

[1] The domains can be assigned randomly due to the exchangeability of the individuals.

$|\Delta_x|$ connected components that are equivalent up to renaming of the X_i individuals. In this case, x is called a *decomposer* of the network. Given the exchangeability of the individuals, the Z of all these connected components are the same. Therefore, we compute the Z for only one of these connected components, e.g., for an MLN with the following WFs, and raise it to the power of $|\Delta_x|$.

$$\{\langle\{m\}, r(X_1,m) \wedge s(X_1,m), 1.2\rangle, \\ \langle\{m\}, s(X_1,m) \wedge t(X_1), 0.2\rangle\}$$

In the above MLN, x has been replaced by one of its individuals. We refer to this as decomposing the MLN on logical variable x. While in this example only one logical variable is the decomposer, note that in general a set of logical variables can be the decomposer of an MLN. We point interested readers to (Poole, Bacchus, and Kisynski, 2011) for a detailed analysis of when a set of logical variables $\mathbf{x}$ is a decomposer of a network.

Lifted Case Analysis

Example 5. Consider the resulting MLN in Example 4 after being decomposed on x. We can find the partition function for this MLN by a case analysis on the values of a PRV. Suppose we do a case analysis on $S(X_1,m)$. Given that $S(X_1,m)$ represents $|\Delta_m|$ random variables in the grounding, one may think $2^{|\Delta_m|}$ cases must be considered: one for each assignment of values to the random variables. However, the individuals are exchangeable, i.e. we only care about the number of times $S(X_1,m)$ is $True$, not about the individuals that make it $True$. Thus, we only consider $|\Delta_m|+1$ cases with the ith case being the case where for i out of $|\Delta_m|$ individuals $S(X_1,m)$ is $True$. We also multiply the ith case to $\binom{|\Delta_m|}{i}$ to take into account the number of different assignments to the individuals in Δ_m for which $S(X_1,m)$ is exactly i times $True$. The case analysis for this PRV will then be:
$Z(M) = \sum_{i=0}^{|\Delta_m|} \binom{|\Delta_m|}{i} Z(M|Obs(S(X_1,m),i))$
where $M|Obs(S(X_1,m),i)$ has the following WFs:

$$\{\langle\{m_1\}, True \wedge r(X_1,m_1), 1.2\rangle, \\ \langle\{m_2\}, False \wedge r(X_1,m_2), 1.2\rangle, \\ \langle\{m_1\}, True \wedge t(X_1), 0.2\rangle, \\ \langle\{m_2\}, False \wedge t(X_1), 0.2\rangle\}$$

Removing False Formulae

Example 6. Consider the resulting MLN in Example 5 after the case analysis on $S(X_1,m)$. The formulae of the second and the fourth WFs are equivalent to $False$ and can be removed from the MLN. However, removing these WFs causes the random variables in $R(X_1,m_2)$ to be totally eliminated from the MLN. To address the effect of these variables, we calculate the Z of the MLN having only the first and third WFs and multiply it by $2^{|\Delta_{m_2}|}$, i.e. the number of possible assignments to the $|\Delta_{m_2}|$ random variables in $R(X_1,m_2)$.

Decomposition

Example 7. Consider the resulting MLN in Example 6 after removing the WFs whose formulae are equivalent to $False$. The resulting MLN has two WFs each mentioning different PRVs, i.e. the two WFs are disconnected. In this case, we can find the Z of the first and second formulae (more generally: first and second connected components) separately and return the product.

Case Analysis

Example 8. Consider the second connected component of the MLN in Example 7. The partition function of this MLN can be found by a case analysis on $T(X_1)$ as: $Z(M) = Z(M \mid T(X_1) = True) + Z(M \mid T(X_1) = False)$.

Evaluating True Formulae

Example 9. Consider the MLN in Example 8 conditioned on $T(X_1) = True$. This MLN has one WF:

$$\{\langle\{m_2\}, True, 1.2\rangle\}$$

Since the formula of the WF is equivalent to True, we can evaluate this WF as $\exp(1.2 * |\Delta_{m_2}|)$.

Caching

The above rules each generate new MLNs, find their partition functions, combine and return the results. As we apply the above rules, we keep the partition functions of the generated MLNs in a cache so we can potentially use them in future when the partition function of the same MLN is required.

Lifted Inference by Compiling into a Low-Level Program

We explain Kazemi and Poole (2016)'s LRC2CPP algorithm for compiling an MLN into a C++ program using an example. LRC2CPP is a recursive algorithm which takes as input an MLN M and a variable name *vname*, and outputs a C++ code which computes $Z(M)$ and stores it in a variable called *vname*.

Example 10. Consider compiling the MLN $\mathbb{MLN}_1$ in Example 1 to a C++ program by following LRC2CPP. Initially, LRC2CPP calls $LRC2CPP(\mathbb{MLN}_1, \texttt{"v1"})$.

As explained in Example 4, x is a decomposer of $\mathbb{MLN}_1$. Let $\mathbb{MLN}_2$ denote $decompose(\mathbb{MLN}_1, x)$ (i.e. the resulting MLN in Example 4 after decomposition). LRC2CPP generates the following C++ program:

$Code\ for\ LRC2CPP(\mathbb{MLN}_2, \texttt{"v2"})$
$v1 = pow(v2, 5);$

where 5 represents $|\Delta_x|$. For $LRC2CPP(\mathbb{MLN}_2, \texttt{"v2"})$, suppose we choose to do a case analysis on $S(X_1,m)$ as in Example 5. Assuming $\mathbb{MLN}_3$ represents $\mathbb{MLN}_2$ conditioned on $Obs(S(X_1,m),i)$, LRC2CPP generates a *for loop* as follows:

$v2 = 0;$
$for(int\ i = 0;\ i <= 2;\ i{+}{+})\{$
  $Code\ for\ LRC2CPP(\mathbb{MLN}_3, \texttt{"v3"})$
  $v2\ \texttt{+=}\ Choose(2,i) * v3;$
$\}$

where 2 represents $|\Delta_m|$, and $Choose(2,i)$ computes $\binom{2}{i}$. $\mathbb{MLN}_3$ has the WFs in the resulting MLN of Example 5.

The formulae of the second and fourth WFs are $False$ and will be removed from $\mathbb{MLN}_3$. However, as explained in Example 6, after removing these two WFs, $R(X_1,m_2)$ will be totally eliminated. Assuming $\mathbb{MLN}_4$ represents $\mathbb{MLN}_3$ after

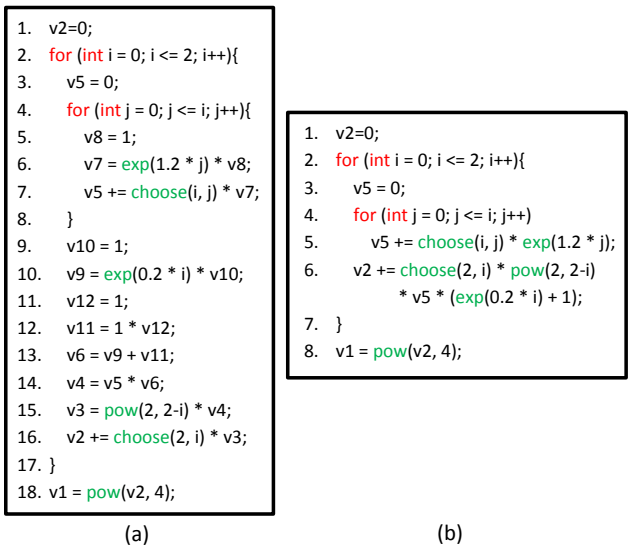

Figure 1: (a) The C++ program for the MLN in Example 1. The partition function (Z) is stored in $v1$. (b) The C++ program in part (a) after pruning.

removing its second and fourth WFs, LRC2CPP generates:

$Code\ for\ LRC2CPP(\mathbb{MLN}_4, "v4")$

$v3 = pow(2, 2-i) * v4;$

where $2-i$ refers to the number of ground variables in $R(X_1, m_2)$ (i.e. $|\Delta_{m_2}|$).

$\mathbb{MLN}_4$ is disconnected as explained in Example 7. Let $\mathbb{MLN}_{41}$ and $\mathbb{MLN}_{42}$ represent the first and second connected components. LRC2CPP generates:

$Code\ for\ LRC2CPP(\mathbb{MLN}_{41}, "v5")$

$Code\ for\ LRC2CPP(\mathbb{MLN}_{42}, "v6")$

$v4 = v5 * v6;$

The first component requires a case analysis of a PRV with one logical variable which generates another for loop, and the second component requires a case analysis of a PRV with no logical variables. We can continue following LRC2CPP for these components and get the C++ program in Figure 1(a).

Optimizing C++ Programs

Since the program obtained from LRC2CPP is generated automatically (not by a developer), a post-pruning step might seem required to reduce the size of the program. For instance, one can remove lines 11 and 12 of the program in Figure 1(a) and replace line 13 with "$v6 = v9 + 1;$". The same can be done for lines 5 and 9. One may also notice that some variables are set to some values and are then being used only once. For example in the program of Figure 1(a), $v7$ and $v9$ are two such variables. The program can be pruned by removing these lines and replacing them with their values whenever they are being used. One can obtain the program in Figure 1(b) by pruning the program in Figure 1(a). Pruning can potentially save time and memory at run-time, but the pruning itself may be time-consuming.

Kazemi and Poole (2016) use available optimization packages for C++ programs which optimize the code at compile time. In particular, they use the $-O3$ flag at compile time to optimize their generated programs before running them.

Experiments and Results

Kazemi and Poole (2016) compared their end-to-end running times to those of WFOMC (Van den Broeck et al., 2011) and *probabilistic theorem proving (PTP)* (Gogate and Domingos, 2011) on six benchmarks. By varying the population sizes of the logical variables for these benchmarks, they showed that LRC2CPP beats these two approaches for most population sizes, especially when the population sizes are large. WFOMC was the closest rival of LRC2CPP. A question which remained unanswered in Kazemi and Poole (2016)'s experiments was to whether LRC2CPP outperforms WFOMC because the compilation to a target circuit is faster in LRC2CPP, or because reasoning with the target circuit generated by LRC2CPP is more efficient than that of WFOMC.

In order to address the above question, we measured the time spent by LRC2CPP and WFOMC on each of the reasoning steps for three networks: 1- the network used in Figure 1(f) of (Kazemi and Poole, 2016), 2- a network with only one WF $A(x) \wedge B(x) \wedge C(x,m) \wedge D(m) \wedge E(m) \wedge F$, and 3- another network with only with WF $A(x) \wedge B(x) \wedge C(x) \wedge D(x,m) \wedge E(m) \wedge F(m) \wedge G(m) \wedge H$. For LRC2CPP, we used the MinNestedLoops heuristic (Kazemi and Poole, 2016) to select the (lifted) case analysis order of PRVs. MinNestedLoops starts with the order obtained from MinTableSize (Kazemi and Poole, 2014) and tries to improve it in terms of the *maximum number of nested loops* it produces in the C++ program using stochastic local search. All experiments were conducted on a 2.8GH core with 4GB RAM under MacOSX. Unless stated otherwise, the C++ programs of LRC2CPP were compiled using g++ compiler.

For the three networks, it takes LRC2CPP 0.173, 0.029, and 0.138 seconds and it takes WFOMC 0.768, 0.373, and 0.512 seconds respectively to generate their target circuits. Figure 2(a), (b) represent the time spent by LRC2CPP and WFOMC for reasoning with their circuits[2] for the first and second networks when the population of the logical variables varies at the same time (WFOMC could not solve the third circuit for population sizes ≥ 500, so we did not include the run-time diagram for the third network).

Obtained results represent that the compilation part takes almost the same amount of time in both LRC2CPP and WFOMC. For small population sizes, reasoning with WFOMC's circuit is more efficient because LRC2CPP's circuit needs a program compilation step. However, as the population size grows, the program compilation time becomes negligible and reasoning with the programs generated by LRC2CPP becomes much faster than reasoning with the data structures generated by WFOMC. As an example, it can be seen from the diagrams that reasoning with LRC2CPP's

[2]Reasoning with LRC2CPP programs are considered as the time spent on compiling the C++ codes plus the run time.

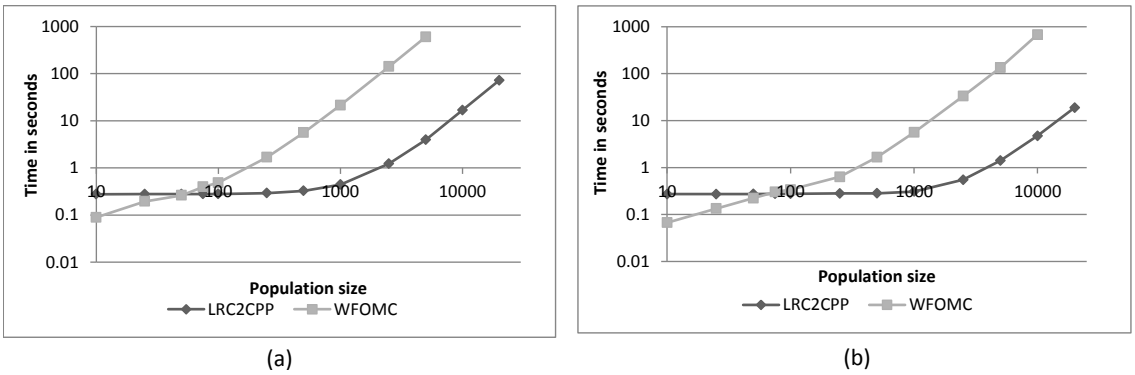

Figure 2: The amount of time spent for reasoning with the target circuit in LRC2CPP and WFOMC on two benchmarks and for different population sizes.

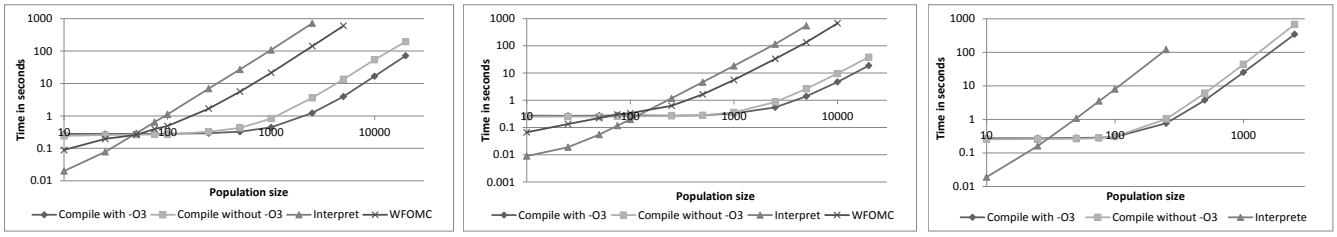

Figure 3: The amount of time spent for reasoning with the programs generated by LRC2CPP when the program is interpreted, compiled, or compiled and optimized as well as the amount of time spent for reasoning with WFOMC's target circuit on three benchmarks and for different population sizes (WFOMC failed to produce an answer in less than 1000s for the third network when the population size was 500 or higher, so we did not include it in the diagram).

program offers about 163x speedup compared to WFOMC's data structure for the first network when the population sizes are 5000. It is also interesting to note that the slope of the diagrams are the same, meaning reasoning with both circuits has the same time complexity.

Our first experiment indicates that the speedup in LRC2CPP is mostly due to the reasoning step. The next question to be answered is why reasoning with LRC2CPP's programs is more efficient than reasoning with WFOMC's data structures. Kazemi and Poole (2016) hypothesized that the speedup is due to the fact that LRC2CPP's programs can be compiled and optimized, while reasoning with WFOMC's data structures requires an interpreter: a virtual machine that executes the data structure node-by-node. Validating this hypothesis by comparing the runtimes of LRC2CPP and WFOMC softwares is not sensible as there might be several implementation or other differences (e.g., case analysis order) between the two softwares.

In order to test Kazemi and Poole (2016)'s hypothesis in an implementation-independent way, we used LRC2CPP to generate programs for the three networks in our previous experiment. For the reasoning step, we ran the programs in three different ways: 1- compiling and optimizing the programs using $-O3$ flag, 2- compiling without optimizing the programs, and 3- running the programs using Ch 7.5.3 which is an interpreter for C++ programs[3]. Obtained results can be viewed in Figure 3. We also included the run time of WFOMC in the first two diagrams (as explained before, WFOMC failed on the third network for population sizes of 500 or more). It can be viewed that interpreting the C++ programs produces similar run times as working with WFOMC's data structures. The diagram for WFOMC is slightly below the diagram for interpreting the C++ program. One reason can be the non-optimality of the interpreter used for interpreting the C++ programs. It is interesting to note that by interpreting the C++ programs for small population sizes, and compiling and optimizing them for larger population sizes, in our benchmarks LRC2CPP's programs are always more efficient than the WFOMC's data structures.

In order to compare the speedup caused by compilation (instead of interpreting) with the speedup caused by optimization, we measured the percentage of speedup caused by each of them for our three benchmarks (we only considered the cases where both of them contributed to the speedup). We found that on average, 99.7% of the speedup is caused by compilation, and only 0.3% of it is caused by optimization. For the largest population where interpreting produced

[3]Note that C++ interpreters are mostly used for teaching purposes and may not be highly optimized.

an answer in less than 1000*s*, we found that compilation offers an average of *175x* speedup compared to interpretation. Furthermore, for the largest population where compilation produced an answer, we found that optimization offers an average of *2.3x* speedup compared to running the code without optimizing it.

Conclusion

Compiling relational models into low-level programs for lifted probabilistic inference is a promising approach and offers huge speedups compared to the other approaches. In this paper, we conducted two experiments to explore the reasons behind the efficiency of this approach. In our first experiment, we compared compiling to low-level programs vs. WFOMC (which compiles into data structures) regarding the amount of time spent on different steps of the reasoning process. Our results indicated that the compilation step takes almost the same time in both approaches and almost all the speedup comes from reasoning with a low-level program instead of a data structure. In our second experiment, we explored why reasoning with a low-level program is more efficient than reasoning with a data structure. We designed an implementation-independent experiment using which we tested and validated Kazemi and Poole (2016)'s hypothesis stating that low-level programs can be *compiled* and *optimized*, while reasoning with a data structure requires a virtual machine to *interpret* the computations, and compilers are known to be faster than interpreters.

References

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2009. Solving #SAT and Bayesian inference with backtracking search. *Journal of Artificial Intelligence Research* 391–442.

Boutilier, C.; Friedman, N.; Goldszmidt, M.; and Koller, D. 1996. Context-specific independence in Bayesian networks. In *Proceedings of UAI*, 115–123.

Choi, J.; de Salvo Braz, R.; and Bui, H. H. 2011. Efficient methods for lifted inference with aggregate factors. In *AAAI*.

Darwiche, A. 2001. Recursive conditioning. *Artificial Intelligence* 126(1-2):5–41.

De Raedt, L.; Kersting, K.; Natarajan, S.; and Poole, D. 2016. Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 10(2):1–189.

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7.

de Salvo Braz, R.; Amir, E.; and Roth, D. 2005. Lifted first-order probabilistic inference. In *IJCAI*, 1319–1325.

Dechter, R., and Mateescu, R. 2007. And/or search spaces for graphical models. *Artificial intelligence* 171(2):73–106.

Getoor, L. 2007. *Introduction to statistical relational learning*. MIT press.

Gogate, V., and Domingos, P. 2011. Probabilistic theorem proving. In *UAI*, 256–265.

Jaeger, M. 1997. Relational Bayesian networks. In *UAI*. Morgan Kaufmann Publishers Inc.

Jha, A.; Gogate, V.; Meliou, A.; and Suciu, D. 2010. Lifted inference seen from the other side: The tractable features. In *NIPS*, 973–981.

Kazemi, S. M., and Poole, D. 2014. Elimination ordering in first-order probabilistic inference. In *AAAI*.

Kazemi, S. M., and Poole, D. 2016. Knowledge compilation for lifted probabilistic inference: Compiling to a low-level language. In *KR*.

Kazemi, S. M.; Buchman, D.; Kersting, K.; Natarajan, S.; and Poole, D. 2014. Relational logistic regression. In *KR*.

Kersting, K. 2012. Lifted probabilistic inference. In *ECAI*, 33–38.

Kisynski, J., and Poole, D. 2009. Constraint processing in lifted probabilistic inference. In *UAI*, 293–302.

Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA.

Milch, B.; Marthi, B.; Russell, S.; Sontag, D.; Ong, D. L.; and Kolobov, A. 2005. BLOG: Probabilistic models with unknown objects. In *Proc. IJCAI*, 1352–1359.

Milch, B.; Zettlemoyer, L. S.; Kersting, K.; Haimes, M.; and Kaelbling, L. P. 2008. Lifted probabilistic inference with counting formulae. In *AAAI*, 1062–1068.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaumann.

Poole, D.; Bacchus, F.; and Kisynski, J. 2011. Towards completely lifted search-based probabilistic inference. *arXiv:1107.4035 [cs.AI]*.

Poole, D. 2003. First-order probabilistic inference. In *IJCAI*, 985–991.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62:107–136.

Suciu, D.; Olteanu, D.; Ré, C.; and Koch, C. 2011. Probabilistic databases. *Synthesis Lectures on Data Management* 3(2):1–180.

Taghipour, N.; Davis, J.; and Blockeel, H. 2014. Generalized counting for lifted variable elimination. *Inductive Logic Programming. Springer Berlin Heidelberg* 107–122.

Van den Broeck, G.; Taghipour, N.; Meert, W.; Davis, J.; and De Raedt, L. 2011. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI*, 2178–2185.

Wu, Y.; Li, L.; Russell, S.; and Bodik, R. 2016. Swift: Compiled inference for probabilistic programs. In *Proc. IJCAI*.

Zhang, N. L., and Poole, D. 1994. A simple approach to Bayesian network computations. In *Proceedings of the 10th Canadian Conference on AI*, 171–178.