# Knowledge Compilation for Lifted Probabilistic Inference: Compiling to a Low-Level Language

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

## Abstract

Algorithms based on first-order knowledge compilation are currently the state-of-the-art for lifted inference. These algorithms typically compile a probabilistic relational model into an intermediate data structure and use it to answer many inference queries. In this paper, we propose compiling a probabilistic relational model directly into a low-level target (e.g., C or C++) program instead of an intermediate data structure and taking advantage of advances in program compilation. Our experiments represent orders of magnitude speedup compared to existing approaches.

Probabilistic relational models (PRMs) (Getoor and Taskar 2007) are forms of graphical models where there are probabilistic dependencies among relations of individuals. The problem of lifted inference for PRMs was first explicitly proposed by Poole (2003) who formulated the problem as first-order variable elimination (FOVE). Current representations for FOVE are not closed under all inference operations. Search-based algorithms (Gogate and Domingos 2011; Poole, Bacchus, and Kisynski 2011) were proposed as alternatives for FOVE algorithms and gained more popularity.

Van den Broeck (2013) follows a knowledge compilation approach to lifted inference by evaluating a search-based lifted inference algorithm symbolically (instead of numerically) and extracting a data structure on which many inference queries can be efficiently answered. Lifted inference by knowledge compilation offers huge speedups because the compilation is done only once and then many queries can be answered efficiently by reusing the compiled model. In order to answer new queries in the other algorithms, however, even though they can reuse a cache, they must reason with the original model and repeat many of the operations.

In this paper, we propose compiling relational models into low-level (e.g., C or C++) programs by symbolically evaluating a search-based lifted inference algorithm and extracting a program instead of a data structure, and taking advantage of advances in program compilation. Our work is inspired by the work of Huang and Darwiche (2007) who turn exhaustive search algorithms into knowledge compilers by recording the trace of the search. Their traces can be seen as straight-line programs generated by searching the entire space. In contrast, our programs have loops over the values of the variables and result in more compact representations. Our generated programs have similar semantics to the FO-NNFs of Van den Broeck (2013). The inference engine for FO-NNFs can be viewed as an *interpreter* that executes a FO-NNF node-by-node, but we can *compile* our programs and perform inference faster.

We focus on inference for Markov logic networks (MLNs) (Richardson and Domingos 2006), but our ideas can be used for other relational representations. We compile an MLN into a low-level program by symbolic evaluation of lifted recursive conditioning algorithm (Gogate and Domingos 2011; Poole, Bacchus, and Kisynski 2011) because of its simplicity to describe and implement. Our idea, however, can be used for any search-based lifted inference algorithm.

## Notation and Background

A **population** is a set of **individuals** and corresponds to a domain in logic. A logical variable (LV) is written in lower case and is typed with a population; we let $\Delta_x$ represent the population associated with $x$ and $|\Delta_x|$ represent the size of the population. A lower case letter in bold represents a tuple of LVs. Constants, denoting individuals, are written starting with an upper-case letter.

A **parametrized random variable (PRV)** is of the form $R(t_1, \dots, t_k)$ where $R$ is a k-ary predicate symbol and each $t_i$ is an LV or a constant. A **grounding** of a PRV can be achieved by replacing each of the LVs with one of the individuals in their domains. A **literal** is an assignment of a PRV to *True* or *False*. We represent $R(\dots) = True$ by $r(\dots)$ and $R(\dots) = False$ by $\neg r(\dots)$. A **world** is an assignment of truth values to each grounding of each PRV. A **formula** is made up of literals connected with conjunctions and/or disjunctions.

A **weighted formula (WF)** is a triple $\langle L, F, w\rangle$, where $L$ is a set of LVs with $|L| = \prod_{x \in L} |\Delta_x|$, $F$ is a formula whose LVs are a subset of LVs in $L$, and $w$ is a real-valued weight. $\langle \{x, y, z\}, g(x, y) \wedge \neg h(y), 1.2\rangle$ is an example of a WF. For a given WF $\langle L, F, w\rangle$ and a world $\omega$, we let $\eta(L, F, \omega)$ represent the number of assignments of individuals to the LVs in $L$ for which $F$ holds in $\omega$.

A **Markov logic network (MLN)** is a set of WFs and induces the following probability distribution:

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

where $\omega$ is a world, the product is over all WFs in the MLN, and $Z = \sum_{\omega'} (\prod_{\langle L,F,w \rangle} (\exp(\eta(L,WF,\omega') * w))$ is the partition (normalization) function. It is common to assume formulae in WFs of MLNs are in conjunctive or disjunctive form. We assume they are in conjunctive form.

We assume input MLNs are shattered (de Salvo Braz, Amir, and Roth 2005) based on observations. An MLN can be conditioned on random variables by updating the WFs based on the observed values for the variables. It can be also conditioned on some counts (the number of times a PRV with one LV is *True* or *False*) as in the following example:

**Example 1.** Suppose for an MLN containing PRV $R(x)$ we observe that $R(x) = True$ for exactly 2 out of 5 individuals. We create two new LVs $x_1$ and $x_2$ representing the subsets of $x$ having $R$ *True* and *False* respectively, with $|\Delta_{x_1}| = 2$ and $|\Delta_{x_2}| = 3$. We update PRVs based on the new LVs and replace $r(x_1)$ with *True* and $r(x_2)$ with *False*.

## Lifted Recursive Conditioning

Algorithm 1 gives a high-level description of a search-based lifted inference algorithm obtained by combining the ideas in (Poole, Bacchus, and Kisynski 2011) and (Gogate and Domingos 2011). Following (Poole, Bacchus, and Kisynski 2011), we call the algorithm *lifted recursive conditioning (LRC)*. The cache is initialized with $\langle \{\}, 1 \rangle$.

```
Algorithm 1 LRC(MLN $M$)
Input: A shattered MLN $M$.
Output: $Z(M)$.
  if $\langle M, Val \rangle \in Cache$ then
    return $Val$
  if $\exists$WF = $\langle L,F,w \rangle \in M$ s.t. $F \equiv True$ then
    return $exp(|L| * w) * LRC(M \setminus WF)$
  if $\exists$WF = $\langle L,F,w \rangle \in M$ s.t. $F \equiv False$ then
    return $2^{nterv(M,WF)} * LRC(M \setminus \text{WF})$
  if $|$CC = connected_components(M)$| > 1$ then
    return $\prod_{cc \in CC} LRC(cc)$
  if $\exists \mathbf{x}$ s.t. $decomposer(M, \mathbf{x})$ then
    return $LRC(decompose(M, \mathbf{x}))^{\#GCC(M,\mathbf{x})}$
  Select $P(\mathbf{x})$ from the branching order
  if $P$ has no LVs then
    $sum = \sum_{v \in \{True, False\}} LRC(M \mid P = v)$
  if $P$ has one LV $x$ then
    $sum = \sum_{i=0}^{|\Delta_x|} \binom{|\Delta_x|}{i} * LRC(M \mid P = True$ exactly $i$ times$)$
  $Cache = Cache \cup \langle M, sum \rangle$
  return $sum$
```

For an input MLN $M$, LRC first checks a few possibilities that can potentially save computations. If $Z(M)$ has been computed before, *LRC* returns the value from the cache. If the formula of a WF is equivalent to *True*, *LRC* evaluates the WF ($exp(|L| * w)$) and removes it from the set of WFs. If the formula of a WF is equivalent to *False*, *LRC* removes the WF. However, if there are random variables in this WF that do not appear in any other WFs, *LRC* multiplies the result by $2^{nterv(M,WF)}$ where $nterv(M,WF)$ calculates the **n**umber of **t**otally **e**liminated **r**andom **v**ariables from the ground MLN after removing the WF, to account for the possible number of value assignments to these variables. If the input MLN $M$ can be divided into more than one connected components, $Z(M)$ is the product of the $Z$ of these components. If the network consists of one connected component but the grounding is disconnected[1], the connected components are the same in the grounding up to renaming the constants from one or more LVs $\mathbf{x}$. $\mathbf{x}$ is called the decomposer of the network. In this case, *LRC* replaces LVs in $\mathbf{x}$ by an assignment of individuals (aka decomposing the network on $\mathbf{x}$), calculates $Z$ of the new model, and raises it to the power of $\#GCC(M, \mathbf{x})$: number of connected components in the grounding of $M$ with $\mathbf{x}$ as the decomposer.

If none of the above cases hold, *LRC* proceeds by a case analysis on one of the PRVs. If the PRV $P$ selected for case analysis has no LVs, $Z(M) = Z(M \mid P = True) + Z(M \mid P = False)$. These two $Z$s can be computed recursively as MLNs are closed under conditioning (6th if). If $P$ has one LV $x$, the case analysis should sum over $2^{|\Delta_x|}$ cases: one for each assignment of values to the $|\Delta_x|$ random variables. However, the individuals are exchangeable, i.e. we only care about the number of times $P(x)$ is *True*, not about the individuals that make it *True*. Thus, we only sum over $|\Delta_x| + 1$ cases with the $i$th case being the case where for exactly $i$ out of $|\Delta_x|$ individuals $P(x)$ is *True*. We also multiply the $i$th case by $\binom{|\Delta_x|}{i}$ to take into account the number of different assignments to the individuals in $\Delta_x$ for which $P(x)$ is exactly $i$ times *True*. The sum can be computed with $|\Delta_x| + 1$ recursive calls (7th if). In this paper, we assume the input MLN is *recursively unary*:

**Definition 1.** An order for the PRVs of an MLN is a *recursively unary order* if for performing case analysis on the PRVs with that order while recognizing disconnectedness and decomposability, no population needs to be grounded.

**Definition 2.** An MLN is *recursively unary* if there exists at least one recursively unary order for its PRVs.

Other MLNs can be partially grounded and turned into a recursively unary MLN offline. There exist heuristics that guarantee generating recursively unary orders for recursively unary networks. Using such heuristics, we do not need to consider in Algorithm 1 the case where the PRV selected for case analysis has two or more LVs.

## Compiling to a Target Program

Algorithm 1 finds the $Z$ of the input MLN. Inspired by the knowledge compilation approach of Van den Broeck (2013) and its advantages, we develop Algorithm 2 which evaluates Algorithm 1 symbolically and extracts a low-level program instead of finding $Z$. We chose C++ as the low-level program because of its efficiency, availability, and available compiling/optimizing packages.

[1] See (Poole, Bacchus, and Kisynski 2011) for detailed analysis.

In Algorithm 2, $VNG()$ (variable name generator) and $ING()$ (iterator name generator) return unique names to be used for C++ variables and loop iterators respectively. Each time we call $LRC2CPP(M, vname)$, it returns a C++ code that stores $Z(M)$ in a variable named *vname*.

**Example 2.** Consider compiling the MLN $M_1$ with a WF $\langle\{x,s\}, f(x) \wedge g(x,s) \wedge h(s), 1.2\rangle$ with $|\Delta_x| = 5$ and $|\Delta_s| = 8$ to a C++ program by following Algorithm 2. Initially, we call $LRC2CPP(M_1, "v1")$. Suppose the algorithm chooses $F(x)$ for a case analysis. Then it generates:

```
    $v1 = 0;$
    $for(int\ i = 0;\ i <= 5;\ i{+}{+})\{$
        $Code\ for\ LRC2CPP(M_2, "v2")$
        $v1 \mathrel{+}= C(5,i) * v2;$
    $\}$
```

where $M_2 \equiv M_1 \mid F(x) = True$ exactly $i$ times. In $M_2$, $s$ is a decomposer and $\#GCC(M_2, s) = |\Delta_s| = 8$. Therefore, $LRC2CPP(M_2, "v2")$ generates:

```
    $Code\ for\ LRC2CPP(M_3, "v3")$
    $v2 = pow(v3, 8);$
```

where $M_3 \equiv decompose(M_2, s)$. In $M_3$, $s$ is replaced by an individual, say, $S$. $LRC2CPP(M_3, "v3")$ may proceed by a case analysis on $H(S)$ and generate:

```
    $Code\ for\ LRC2CPP(M_4, "v4")$
    $Code\ for\ LRC2CPP(M_5, "v5")$
    $v3 = v4 + v5;$
```

where $M_4 \equiv M_3 \mid H(S) = True$ and $M_5 \equiv M_3 \mid H(S) = False$. We can follow the algorithm further and generate the program.

**Caching as Needed** In the compilation stage, we keep record of the cache entries that are used in future and remove from the C++ program all other cache inserts.

**Pruning** Since our target program is C++, we can take advantage of the available packages developed for pruning/optimizing C++ programs. In particular, we use the $-O3$ flag when compiling our programs which optimizes the code at compile time. Using $-O3$ slightly increases the compile time, but substantially reduces the run time when the program and the population sizes are large. We show the effect of pruning our programs in the experiments.

## MinNestedLoops Heuristic

The maximum number of nested loops (MNNL) in the C++ program is a good indicator of the time complexity and a suitable criteria to minimize using a suitable elimination ordering heuristic. To do so, we start with the order taken from the MinTableSize (MTS) heuristic (Kazemi and Poole 2014), count the MNNL in the C++ program generated when using this order, and then perform $k$ steps of stochastic local search on the order to minimize the MNNL. We call this heuristic *MinNestedLoops (MNL)*. Note that the search is performed only once in the compilation phase. The value of $k$ can be set according to how large the network is, how large the population sizes are, how much time we want to spend on the compilation, etc.

**Proposition 1.** *MinNestedLoops heuristic generates recursively unary orders for recursively unary networks.*

**Algorithm 2** LRC2CPP(MLN $M$, String vname)

```
Input: A shattered MLN $M$ and a variable name.
Output: A C++ program storing $Z(M)$ in $vname$.
  if $M \in Cache$ then
    return "$\{vname\} = cache.at(\{M.id\});$"
  if $\exists$WF = $\langle L, F, w\rangle \in M$ s.t. $F \equiv True$ then
    nname = VNG()
    return LRC2CPP(M\WF, nname) + "$\{vname\} = \{nname\} * exp(w * |L|);$"
  if $\exists$WF = $\langle L, F, w\rangle \in M$ s.t. $F \equiv False$ then
    nname = VNG()
    return LRC2CPP(M\WF, nname) + "$\{vname\} = \{nname\} * pow(2, \{nterv(M, WF)\});$"
  if |CC = connected_components(M)| > 1 then
    retVal = ""
    for each cc $\in$ CC do
      nname[cc] = VNG()
      retVal += LRC2CPP(cc, nname[cc])
    return $retVal$ + "$\{vname\} = \{nname.join("*")\};$"
  if $\exists \mathbf{x}$ s.t. $decomposer(M, \mathbf{x})$ then
    nname = VNG()
    return $LRC(decompose(M, \mathbf{x}), nname)$ + "$\{vname\} = pow(\{nname\}, \{\#GCC(M, \mathbf{x})\});$"
  Select $P(\mathbf{x})$ from the branching order
  if $P$ has no LVs then
    nname1 = VNG(), nname2 = VNG()
    retVal = $LRC2CPP(M \mid P = True, nname1)$
    retVal += $LRC2CPP(M \mid P = False, nname2)$
    retVal += "$\{vname\} = \{nname1\} + \{nname2\};$"
  if $P$ has one LV $x$ then
    retVal = "$\{vname\} = 0;$"
    i = ING(), nname = VNG()
    retVal += "for(int $\{i\}$=0; $\{i\}$ <= $\{|\Delta_x|\}$; $\{i\}$++){"
    retVal += $LRC2CPP(M$ | P=True exactly $i$ times, $nname)$
    retVal += "$\{vname\}$ += $C(\{|\Delta_x|\}, \{i\})$ * $\{nname\}$;"
    retVal += "}"
  retVal += "cache.insert($\{M.id\}, \{vname\}$);"
  $Cache = Cache \cup M$
  return retVal
```

## Experiments and Results

We implemented Algorithm 2 in Ruby and did our experiments using Ruby 2.1.5 on a 2.8GH core with 4GB RAM under MacOSX. We used the g++ compiler with $-O3$ flag to compile and optimize the C++ programs.

We compared our results with *weighted first-order model counting (WFOMC)*[2] and *probabilistic theorem proving (PTP)*[3], the state-of-the-art lifted inference algorithms. We allowed at most 1000 seconds for each algorithm. All queries were ran multiple times and the average was reported. When using MNL as our heuristic, we did 25 iterations of local search.

Fig. 1 represents the overall running times of LRC2CPP

[2]Available at: https://dtai.cs.kuleuven.be/software/wfomc

[3]Available in Alchemy-2 (Kok et al. 2005)

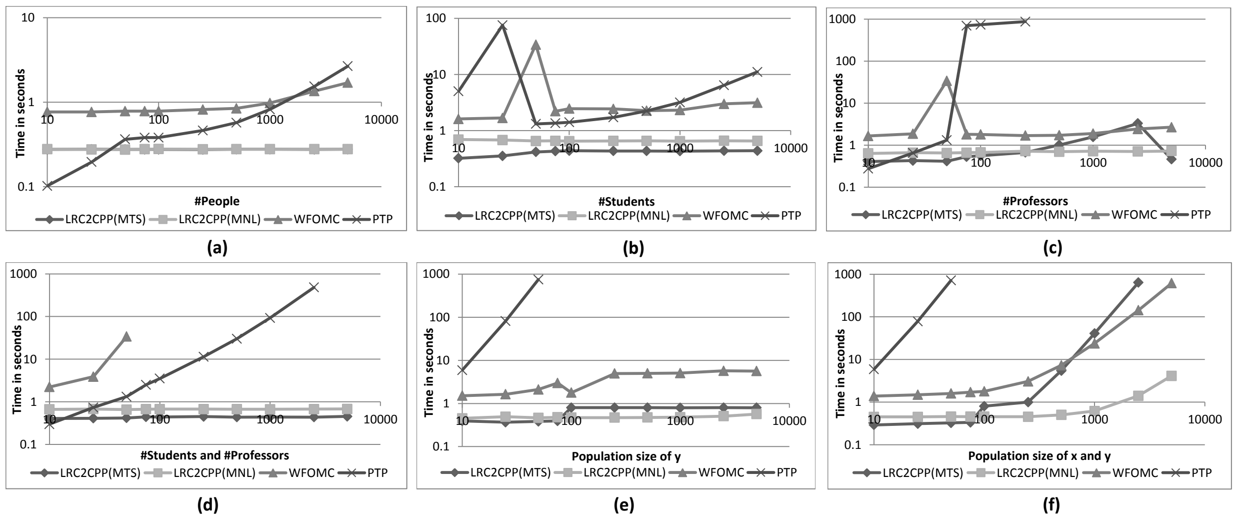

Figure 1: Run-times for LRC2CPP using MTS and MNL heuristics, WFOMC, and PTP on several benchmarks.

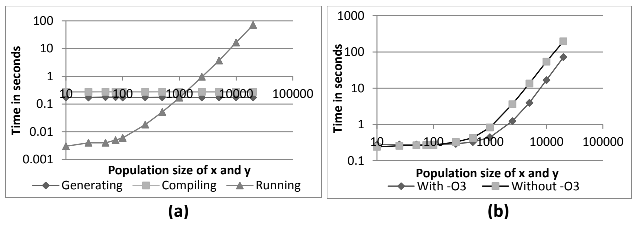

Figure 2: (a) Time spent for each step of Algorithm 2. (b) Compile + run time with and without $-O3$.

with MTS and with MNL as well as the running times of WFOMC and PTP for: **a)** competing workshops network with 50 workshops and varying #people, **b)** link prediction network taken from the second experiment in (Gogate and Domingos 2011) using conjunctive formulae and querying $FutureProf(S)$ with 50 professors and varying #students, **c)** similar to **(b)** but with 50 students and varying #professors, **d)** similar to **(b)** and **(c)** but with varying #students and #professors at the same time, **e)** a network similar to the first experiment in (Gogate and Domingos 2011) with weighted formulae: $\{a(x) \wedge b(y)\}, \{a(x) \wedge c(x)\}, \{b(y) \wedge d(y)\}, \{c(x) \wedge d(y)\}, \{e \wedge d(y)\}$ querying $E$ with $|\Delta_x| = 100$ and varying $|\Delta_y|$, and **f)** similar to **(e)** but changing both $|\Delta_x|$ and $|\Delta_y|$ at the same time. The obtained results show that LRC2CPP (using either of the two heuristics) outperforms WFOMC and PTP by an order of magnitude on different networks.

Answering queries in our settings consists of: 1) generating the program, 2) compiling it, and 3) running it. Fig. 2(a) represents the amount of time spent for each step of the algorithm for the network in Fig. 1(f) (using MNL). Fig. 2(b) compares the time spent on compiling and running for this network with and without using $-O3$. The increase in the compile time caused by $-O3$ becomes negligible as the size of the population grows.

## Conclusion

We presented an algorithm to compile a relational network into a C++ program. Compiling to C++ obviates the need to work with intermediate data structures, thus inference only requires summations and additions, and enables taking advantage of advances in program compilation. Obtained results represent orders of magnitude speed-up over the state-of-the-art lifted inference techniques on several benchmarks.

## References

de Salvo Braz, R.; Amir, E.; and Roth, D. 2005. *Lifted first-order probabilistic inference. In L. Getoor and B. Taskar (Eds).* MIT Press.

Getoor, L., and Taskar, B. 2007. *Introduction to Statistical Relational Learning*. MIT Press, Cambridge, MA.

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

Huang, J., and Darwiche, A. 2007. The language of search. *J. of Artificial Intelligence Research (JAIR)* 29:191–219.

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

Kok, S.; Singla, P.; Richardson, M.; and Domingos, P. 2005. The alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

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 *Proc. of IJCAI*, 985–991.

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

Van den Broeck, G. 2013. *Lifted Inference and Learning in Statistical Relational Models*. Ph.D. Dissertation, KU Leuven.