Empowering Domain Recursion in Symmetric Weighted First-Order Model Counting

21st January 2022

1 Basic Definitions

Things I might need to explain.

- notation: Im
- atom, literal, constant, predicate, variable
- inequality constraint
- Vars, $\mathrm{Vars}(c) = \mathrm{Vars}(P) \cup \mathrm{Vars}(N) \cup \mathrm{Vars}(C)$
- Doms on both formulas and clauses. $\mathrm{Doms}(c) = \operatorname{Im} \delta_c$, and $\mathrm{Doms}(\phi) = \bigcup_{c \in \phi} \mathrm{Doms}(c)$.
- substitution
- size of a domain, how each domain is partitioned into two during compilation.
- maybe: notation for partial function, notation for powerset, domain size
- notation for projection (or avoid it?)
- WMC
- constraint removal operation
- two parts: compilation and inference.
- introduce and use arrows for bijections, injections, set inclusions, etc.
- $\emptyset$ for an empty partial map.

Let $\mathscr{V}$ be the set of circuit nodes.

TODO

- later: maybe mathcal instead of mathscr
- maybe π is global enough to have a more unique name
- maybe $c[\beta, \gamma]$ is better than $c * (\beta, \gamma)$

$$\begin{tikzcd}
a \arrow[d] & & b \arrow[dl] \arrow[dr] & \\
\boxed{a'} & b^{\top} & & b^{\perp}
\end{tikzcd}$$

Figure 1: The forest of domains from Example 1. The directed edges correspond to the pairs of domains in π but in the opposite direction, e.g., $\pi(a') = a$. The domain enclosed in a rectangle is the only domain created by constraint removal, i.e., $\mathscr{C} = \{\, a' \,\}$.

Definition 1. A *domain* is a set with elements not used anywhere else.[1] Let $\mathscr{D}$ be the set of all domains and $\mathscr{C} \subset \mathscr{D}$ be the subset of domains introduced as a consequence of constraint removal. Note that both sets (can) expand during the compilation phase.

Let $\pi \colon \mathscr{D} \nrightarrow \mathscr{D}$ be a partial endomorphism on $\mathscr{D}$ that denotes the *parent* relation, i.e., if $\pi(d) = e$ for some $d, e \in \mathscr{D}$, then we call e the parent (domain) of d, and e a child of d. Intuitively, π arranges all domains into a forest—thus, we often use graph theoretical terminology to describe properties of and relationships between domains.

Definition 2. A *clause* is a triple $c = (P, N, C, \delta_c)$, where P and N are sets of atoms interpreted as positive and negative literals respectively, C is a set of inequality constraints, and $\delta_c \colon \mathrm{Vars}(c) \to \mathscr{D}$ is a function that maps all variables in c to their domains. Two clauses c and $d = (P', N', C', \delta_d)$ are *isomorphic* (written $c \cong d$) if there is a bijection $\beta \colon \mathrm{Vars}(c) \to \mathrm{Vars}(d)$ such that $c\beta = d\beta$. TODO: we will always use this subscript notation for the δ's. Equality of clauses is defined in the usual way (i.e., all variables, domains, etc. must match).

A *formula* is a set of clauses.

We use hash codes to efficiently check whether a recursive relationship between two formulas is plausible. (It is plausible if the formulas are equal up to variables and domains.) The hash code of a clause $c = (P, N, C, \delta_c)$ combines the hash codes of the sets of constants and predicates in c, the numbers of positive and negative literals (i.e., $|P|$ and $|N|$), the number of inequality constraints $|C|$, and the number of variables $|\mathrm{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted $\#\phi$.

2 Identifying Possibilities for Recursion

Definition 3 (Notation)**.** For any clause $c = (P, N, C, \delta_c)$, bijection $\beta \colon \mathrm{Vars}(c) \to V$ (for some set of variables V) and function $\gamma \colon \mathrm{Doms}(c) \to \mathscr{D}$, let $c * (\beta, \gamma) = d$ be the clause with all occurrences of any variable $v \in \mathrm{Vars}(c)$ in P, N, and C replaced with $\beta(v)$ (so $\mathrm{Vars}(d) = V$) and $\delta_d \colon V \to \mathscr{D}$ defined as $\delta_d \coloneqq \gamma \circ \delta_c \circ \beta^{-1}$. In other words, δ_d is the unique function that makes the following diagram commute:

$$\begin{tikzcd}
\mathrm{Vars}(c) \arrow[r, tail, two heads, "\beta"] \arrow[d, "\delta_c"'] & V = \mathrm{Vars}(d) \arrow[d, dashed, "\exists!\delta_d"] \\
\mathrm{Doms}(c) \arrow[r, "\gamma"'] & \mathscr{D}.
\end{tikzcd}$$

The function `traceAncestors` returns `null` if domain $d \in \mathscr{D}$ is not an ancestor of domain $e \in \mathscr{D}$. Otherwise, it returns `true` if the size of e is guaranteed to be strictly smaller than the size of d (i.e., there is domain created by the constraint removal rule on the path from d to e) and `false` if their sizes will be equal at some point during inference.

Notation: For partial functions $\alpha, \beta \colon A \nrightarrow B$ such that $\alpha|_{\mathrm{dom}(\alpha)\cap\mathrm{dom}(\beta)} = \beta|_{\mathrm{dom}(\alpha)\cap\mathrm{dom}(\beta)}$, we write $\alpha \cup \beta$ for the unique partial function such that $\alpha \cup \beta|_{\mathrm{dom}(\alpha)} = \alpha$, and $\alpha \cup \beta|_{\mathrm{dom}(\beta)} = \beta$.

[1]In the context of functions, the domain of a function f retains its usual meaning and is denoted $\mathrm{dom}(f)$.

TODO

- explain why $\rho \cup \gamma$ is possible
- explain what the second return statement is about and why a third one is not necessary
- mention which one is the main function, what each function takes and returns
- explain the yield keyword
- in the example below: write down both formula using the Forclift format

The algorithm could be improved in two ways:

- by constructing a domain map first and then using it to reduce the number of viable variable bijections.
- by similarly using the domain map ρ.

However, it is not clear that this would result in an overall performance improvement, since the number of variables in instances of interest never exceeds three and the identity bijection is typically the right one.

Diagramatically, `constructDomainMap` attempts to find $\gamma\colon \mathrm{Doms}(c) \to \mathrm{Doms}(d)$ such that the following diagram commutes (and returns `null` if such a function does not exist):

$$\begin{tikzcd}
\mathrm{Vars}(c) \arrow[r, tail, two heads, "\beta"] \arrow[d, "\delta_c"'] & \mathrm{Vars}(d) \arrow[d, "\delta_d"] \\
\mathrm{Doms}(c) \arrow[r, dashed, "\gamma"] \arrow[d, hook] & \mathrm{Doms}(d) \arrow[d, hook] \\
\mathscr{D} \arrow[r, "\rho"'] & \mathscr{D}.
\end{tikzcd}$$

Example 1. Let ϕ be the formula

$$\forall X \in a.\forall Y \in b.\forall Z \in b.Y \neq Z \implies \neg p(X,Y) \lor \neg p(X,Z) \tag{1}$$
$$\forall X \in a.\forall Y \in b.\forall Z \in a.X \neq Z \implies \neg p(X,Y) \lor \neg p(Z,Y). \tag{2}$$

and ψ be the formula

$$\forall X \in a'.\forall Y \in b^{\perp}.\forall Z \in b^{\perp}.Z \neq Y \implies \neg p(X,Y) \lor \neg p(X,Z) \tag{3}$$
$$\forall X \in a'.\forall Y \in b^{\perp}.\forall Z \in a'.X \neq Z \implies \neg p(X,Y) \lor \neg p(Z,Y) \tag{4}$$

The relevant domains and the definition of π is in Fig. 1. Since $\#\phi = \#\psi$, and the formulas are non-empty, the algorithm proceeds with the for-loops on Lines 6 to 8. Suppose c in the algorithm refers to Eq. (1), and d to Eq. (3). Since both clauses have three variables, in the worst case, function `generateMaps` would have $3! = 6$ bijections to check. Suppose the identity bijection is picked first. Then `constructDomainMap` is called with the following parameters:

- $V = \{ X, Y, Z \}$,
- $\delta_c = \{ X \mapsto a, Y \mapsto b, Z \mapsto b \}$,
- $\delta_d = \{ X \mapsto a', Y \mapsto b^{\perp}, Z \mapsto b^{\perp} \}$,
- $\beta = \{ X \mapsto X, Y \mapsto Y, Z \mapsto Z \}$,
- $\rho = \emptyset$.

Since $\delta_i(Y) = \delta_i(Z)$ for $i \in \{ c, d \}$, `constructDomainMap` returns $\gamma = \{ a \mapsto a', b \mapsto b^\perp \}$. Thus, `generateMaps` yields its first pair of maps (β, γ) to Line 8. Furthermore, the pair satisfies the $c * (\beta, \gamma) = d$ condition.

Since $\pi(a') = a$, and $a' \in \mathscr{C}$, `traceAncestors(`a`, `a'`)` returns `true`, which sets $\mathsf{foundConstraintRemoval}'$ to `true` as well. When $e = b$, however, `traceAncestors(`b, $b^\perp$`)` returns `false` since $b^\perp$ is a descendant of b but not created by the constraint removal compilation rule. On Line 18, a recursive call to `identifyRecursion(`ϕ'`, `ψ'`, `γ`, true)` is made, where ϕ' and ψ' are new formulas with one clause each: Eq. (2) and Eq. (4), respectively.

Again we have two non-empty formulas with equal hash codes, so `generateMaps` is called with c set to Eq. (2), d set to Eq. (4), and $\rho = \{ a \mapsto a', b \mapsto b^\perp \}$. Suppose Line 22 picks the identity bijection first again. Then `constructDomainMap` is called with the following parameters:

- $V = \{ X, Y, Z \}$,
- $\delta_c = \{ X \mapsto a, Y \mapsto b, Z \mapsto a \}$,
- $\delta_d = \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto a' \}$,
- $\beta = \{ X \mapsto X, Y \mapsto Y, Z \mapsto Z \}$,
- $\rho = \{ a \mapsto a', b \mapsto b^\perp \}$.

Since β and ρ 'commute' (TODO: as in the diagram above), and there are no new domains in Doms(c) and Doms(d), γ exists and is equal to ρ. Again, the returned pair (β, γ) satisfies the condition $c * (\beta, \gamma) = d$. This γ passes the `traceAncestors` checks exactly the same way as the one before, and Line 18 calls `identifyRecursion(`$\emptyset$`, `$\emptyset$`, `ρ`, true)`, which immediately returns $\rho = \{ a \mapsto a', b \mapsto b^\perp \}$ as the final answer. This means that one can indeed use a circuit for WMC(ϕ) to compute WMC(ψ) by replacing every mention of a with a' and every mention of b with $b^\perp$.

2.1 Evaluation

This ρ is saved as a parameter of the new circuit node. During inference, there is a domain size map $\sigma\colon \mathscr{D} \to \mathbb{N}_0$, which gets updated to

$$\sigma'(x) = \begin{cases} \sigma(\rho(x)) & \text{if } x \in \operatorname{dom}(\rho) \\ \sigma(x) & \text{otherwise} \end{cases}$$

for all $x \in \mathscr{D}$.

3 New Operations

TODO: compilation rules, node types, their evaluation during inference.

3.1 Improved Domain Recursion

The original version of domain recursion is here [1].

3.2 Constraint Removal

4 Other Topics

- domains, smoothing, and avoiding infinite cycles
- new rules that don't create nodes (e.g., duplicate removal, unconditional contradiction detection, etc.)

5 Circuit Evaluation

TODO: new node types, their algebraic/graphical representation, what info they hold, and how they're created.

TODO: describe evaluation of: and, counting, constraint removal, ref, unit, contradiction, improved domain recursion. Most of this will be from [2].

References

[1] Van den Broeck, G. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain* (2011), J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, Eds., pp. 1386–1394.

[2] Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., and De Raedt, L. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011* (2011), T. Walsh, Ed., IJCAI/AAAI, pp. 2178–2185.

Algorithm 1: A recursive function for checking whether one can reuse the circuit for computing WMC(ϕ) to compute WMC(ψ). Both ϕ and ψ are formulas, and $\rho\colon \mathrm{Doms}(\phi) \nrightarrow \mathrm{Doms}(\psi)$ is a partial map.

```
Function identifyRecursion($\phi$, $\psi$, $\rho = \emptyset$, foundConstraintRemoval = false):
    if $|\phi| \neq |\psi|$ or $\#\phi \neq \#\psi$ then return null;
    if $\phi = \emptyset$ then
        if foundConstraintRemoval then return $\rho$;
        return null;
    foreach clause $c \in \phi$ do
        foreach clause $d \in \psi$ such that $\#d = \#c$ do
            foreach $(\beta, \gamma) \in$ generateMaps($c$, $d$, $\rho$) such that $c * (\beta, \gamma) = d$ do
                foundConstraintRemoval$'$ $\leftarrow$ foundConstraintRemoval;
                suitableBijection $\leftarrow$ true;
                foreach $e \in \mathrm{Doms}(c)$ do
                    foundConstraintRemoval$''$ $\leftarrow$ traceAncestors($e$, $\gamma(e)$);
                    if foundConstraintRemoval$''$ = null then
                        suitableBijection $\leftarrow$ false;
                        break;
                    if foundConstraintRemoval$''$ then foundConstraintRemoval$'$ $\leftarrow$ true;
                if suitableBijection then
                    $\rho'' \leftarrow$ identifyRecursion($\phi \setminus \{ c \}$, $\psi \setminus \{ d \}$, $\rho \cup \gamma$, foundConstraintRemoval$'$);
                    if $\rho'' \neq$ null then return $\rho''$;
        return null;
Function generateMaps($c$, $d$, $\rho$):
    foreach bijection $\beta\colon \mathrm{Vars}(c) \to \mathrm{Vars}(d)$ do
        $\gamma \leftarrow$ constructDomainMap($\mathrm{Vars}(c)$, $\delta_c$, $\delta_d$, $\beta$, $\rho$);
        if $\gamma \neq$ null then yield $(\beta, \gamma)$;
Function constructDomainMap($V$, $\delta_c$, $\delta_d$, $\beta$, $\rho$):
    $\gamma \leftarrow \emptyset$;
    foreach $v \in V$ do
        if $\delta_c(v) \in \mathrm{dom}(\rho)$ and $\rho(\delta_c(v)) \neq \delta_d(\beta(v))$ then return null;
        if $\delta_c(v) \notin \mathrm{dom}(\gamma)$ then $\gamma \leftarrow \gamma \cup \{ \delta_c(v) \mapsto \delta_d(\beta(v)) \}$;
        else if $\gamma(\delta_c(v)) \neq \delta_d(\beta(v))$ then return null;
    return $\gamma$;
Function traceAncestors($d$, $e$):
    foundConstraintRemoval $\leftarrow$ false;
    while $e \neq d$ and $e \in \mathrm{dom}(\pi)$ do
        if $e \in \mathscr{C}$ then foundConstraintRemoval $\leftarrow$ true;
        $e \leftarrow \pi(e)$;
    if $e = d$ then return foundConstraintRemoval;
    return null;
```