Answer Set Programming for Egg Extraction and More

Ziyi Yang
yangziyi@u.nus.edu
National University of Singapore
Singapore, Singapore

Ilya Sergey
ilya@nus.edu.sg
National University of Singapore
Singapore, Singapore

Abstract

Three years ago, Philip Zucker posted an attempt to use answer set programming (ASP) for term extraction from e-graphs [10]. Although the task is NP-hard and ASP offers a natural modelling of e-graph terms, the initial attempt did not yield convincing results.

From the aspect of practical ASP users, we first pinpoint the way to make ASP work and work well on the task of e-graph extraction. The initial results show the naïve ASP encoding is comparable on efficiency to the well-optimised ILP-based exact DAG extraction in the extraction-gym, and find several extra optimal extraction on the complex instances. This leads us to a further agenda: with the "better together of egg+Datalog", is there a better "better together" by having ASP as a more powerful Datalog? We discuss the potential benefit from each other.

1 DAG Extraction in E-Graphs

Term extraction in e-graphs is a central optimisation step for equality saturation systems. The exact extraction problem is known to be NP-hard [6], which makes solver choice and encoding strategy directly relevant for practical use. Several recent works have explored the solution from theory to practice [3, 4, 7, 8].

Prior work highlights that tree-cost intuition does not transfer directly to exact DAG extraction: sharing in DAGs creates global interactions that are not captured by purely local, compositional tree costs. Importantly, Philip's ASP formulation [10] already targets DAG-cost extraction by using the stable-model semantics of ASP to enforce acyclicity. That encoding is compact and expressive, but baseline experiments in the original post indicated practical scalability limitations on larger instances. This extended abstract revisits the same task with a solver-oriented ASP workflow.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 The Attempts and the Problems

2.1 How ASP Models E-Graph Extraction

The key attraction of Philip's encoding is that it captures DAG extraction with a tiny set of relations and founded rules, without explicitly encoding loops. The e-graph is represented as facts in logic programming over classes and enodes, as shown in Figure 1: each enode is identified inside an eclass (operator and local cost), and child links point from an enode to child eclasses. In practice this can be read as three tables: `root(E)`, `enode(E,I,Op,C)`, and `child(E,I,Ec)`.

The essential rule pattern is a support-style choice: an enode may be selected only if its child eclasses are selected; selecting an enode induces selection of its eclass; and every root eclass must be selected. Together with cost minimization, this yields a founded selected subgraph. The important point is that cycles are avoided through stable-model foundedness (no self-supporting selection), rather than by adding an explicit reachability or transitive-closure encoding.

Figure 2 (left) shows the core rules for the initial bottom-up encoding from [10]. Line 2 in the left column enforces support before selecting an enode, line 5 lifts selected enodes to selected eclasses, and line 14 makes root selection mandatory. Line 8 adds a one-enode-per-class restriction, while line 11 sets the optimisation objective.

The problem. It is unfortunate that the initial encoding "appears to be slow on the large problems" (as the original post states). For example, the `lambda function repeat` instance takes forever to solve in the ASP encoding, while even the naïve ILP encoding can solve it in less than a second.

2.2 A Top-Down Alternative?

Although not discussed explicitly, the natural question is whether the top-down alternative for the extraction is valid. Figure 2 (right) sketches a top-down encoding: root classes are marked as needed (line 2), and for each needed class, exactly one enode is selected (line 5). Selecting an enode makes its children needed too (line 8), which creates a demand-driven support condition. The same cost minimization applies to the selected nodes (line 11).

The problem. Unlike the bottom-up encoding, the search from the roots fails to skip the cycles in the e-graph, which means incorrectness of the DAG extraction. The question is whether we can fix this easily with ASP constraints.

A natural attempt is to add a reachability constraint: if an enode is selected, we collect all its reachable nodes from the

Logical facts of an e-graph

```
% root eclass
root(1).

% enode(Eclass, Id, Op, Cost)
enode(1,0,"add",1).
enode(1,1,"add",1).
enode(2,0,"x",1).
enode(3,0,"y",1).

% child(Eclass, Id, ChildEclass)
child(1,0,2). child(1,0,3).
child(1,1,3). child(1,1,2).
```

E-graph view (right)

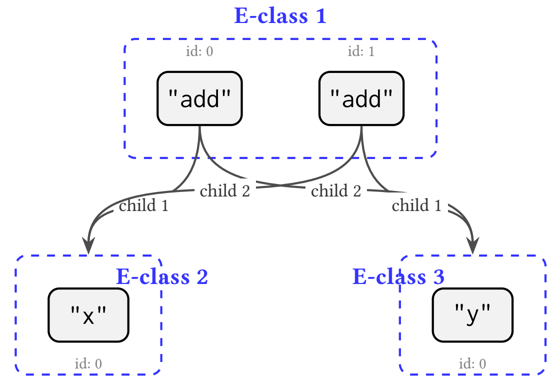

Figure 1. A commutativity example ($x + y \equiv y + x$) shown as ASP facts and as an e-graph from [10].

Philip's bottom-up encoding

```
1  % choose an enode only if all child classes are selected
2  {sel(E,I)} :- enode(E,I,_,_), selclass(Ec): child(E,I,Ec).
3
4  % selecting an enode selects its class
5  selclass(E) :- sel(E,_).
6
7  % at most one enode per class
8  :- enode(E,_,_,_), #count { I : sel(E,I) } > 1.
9
10 % optimize extraction cost
11 #minimize { C,E,I : sel(E,I), enode(E,I,_,C) }.
12
13 % roots must be selected
14 :- root(E), not selclass(E).
```

A top-down alternative?

```
1  % Top-down: root classes need extraction
2  need(E) :- root(E).
3
4  % For each needed class, choose exactly one node
5  { selnode(I) : enode(E,I,_,_) } = 1 :- need(E).
6
7  % Selecting a node makes its children's classes needed too
8  need(E) :- selnode(I), echild(I,E).
9
10 % The actual cost function
11 #minimize { C,E,I : selnode(I), enode(E,I,_,C) }.
```

Figure 2. ASP encoding of egg extraction (left: bottom-up from [10], right: top-down ignoring DAG acyclicity).

selected enodes, and require that none of the selected nodes are reachable from themselves. A direct encoding is like

```
% Selected dependency graph
sel_edge(E,Ec) :- selnode(I), enode(E,I,_), echild(I,Ec).

% Transitive closure of selected dependency graph
sel_reach(E,Ec) :- sel_edge(E,Ec).
sel_reach(E,Ez) :- sel_reach(E,Ey), sel_edge(Ey,Ez).

% Cycle prohibition: extracted structure must be acyclic
:- sel_reach(E,E).
```

However, this encoding is not effective since the real e-graphs can have large strongly connected components (SCCs) with many nodes, making the $O(n^3)$ transitive closure encoding too heavy for the solver.

3 The Pragmatic Solutions

We say it is unfortunate (about the poor scalability), because Philip was almost there: if he had tried using parallel solving [2], even with two threads, ASP version should work. This is actually unfortunate due to the barrier of Clingo [5], the most popular ASP solver. What is helping is actually the *UNSAT-core* (*a.k.a* usc) based optimisation in Clingo [1], which is a powerful technique for solving optimisation problems (and enabled by the second thread with parallel solving). With single-thread solving on usc, the ASP encoding three years ago is competitive with the latest ILP-based exact DAG extraction, for which we will show the results later.

The solution for the top-down encoding is more involved in principle: to detect and eliminate cycles is not complex, but the declarative logic programming is limited in expressing such procedures. The good thing is, Clingo provides custom propagators [5]. So like how SMT solvers use theory solvers, the propagator can be used to implement imperative procedures to support the solving, which is answer set programming modulo acyclicity in this case. The funny thing is, such propagator exists in Clingo (but without documentation). So it only takes one more line to achieve the SCC elimination:

```
#edge (E,Ec): selnode(I), enode(E,I,_), echild(I,Ec).
```

It is exactly the same edge definition as the one above, but the reachability is handled outside the ASP solver.

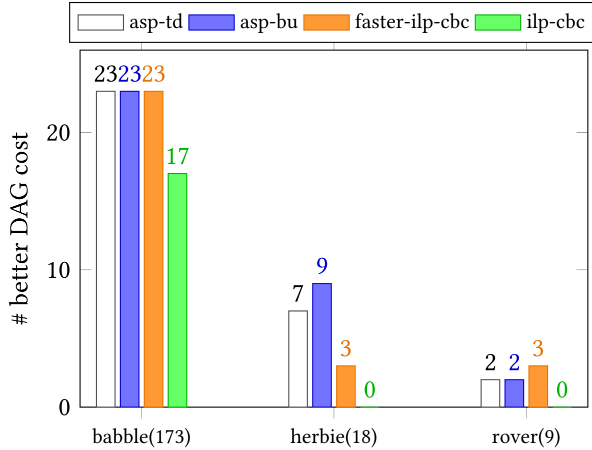

Figure 3. Per-suite counts of instances with strictly better DAG cost than `faster-greedy-dag`. Suite sizes are shown on the x-axis.

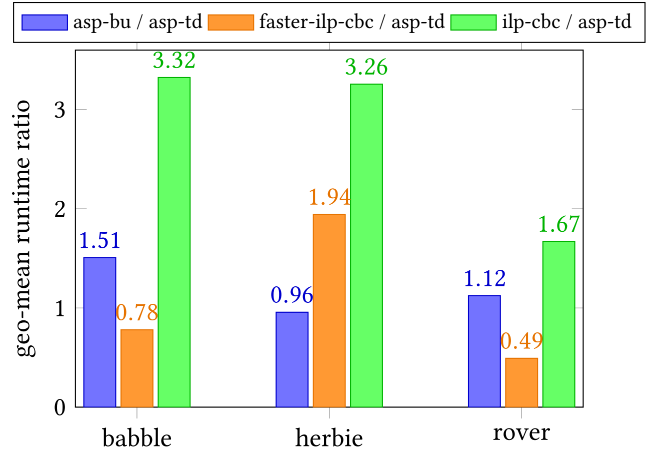

Figure 4. Geometric-mean runtime ratio by suite from pair comparisons to the top-down `asp-td` encoding. Values below 1 indicate faster than `asp-td`.

4 Benchmarking

We evaluate against extraction-gym[1] baselines on three suites (babble, herbie, rover), which are a large collection of e-graphs from different domains. We use *faster-greedy-dag* as the baseline for solution quality, and compare the top-down and bottom-up ASP encodings against the naïve ILP encoding and an optimised ILP variant (with several optimisations outside the ILP solving). All exact methods are given a default 10-second timeout in the extraction-gym.

Figure 3 summarizes solution quality as the number of instances where each method improves DAG cost over the greedy. Runtime trade-offs are shown in Figure 4 as geometric-mean ratios against the `asp-td` (lower is better). The top-down series is omitted from the plot, while the bottom-up ASP and the ILP variants are shown for comparison.

Overall, the propagator-based top-down encoding (asp-td) gives a good balance in our evaluation: it preserves solution quality while staying competitive in runtime. By contrast, the naïve ILP baseline is clearly not competitive.

The optimised ILP variant can be very fast, especially on rover. Our interpretation is that this comes largely from non-solver engineering choices (including specialized optimisations such as zero-cost-oriented simplifications) that can significantly help certain classes of instances. With Clingo's API, similar engineering hooks are also feasible for ASP.

Compared with asp-bu, asp-td is generally faster overall, but the quality differences are concentrated in three outlier instances, two of which are additional optimal extractions found by asp-bu. These outliers are root-heavy (around 400 and 1400 roots), where bottom-up search can be advantageous because it does not start from roots. At the same time, the runtime benefit of asp-bu on the suites where it helps remains modest (for example, geo-mean ratio is only about 0.96 on herbie), so asp-bu is still slower overall. This indicates that search direction is strongly case-sensitive.

There are several limitations to the current evaluation. First, the weak constraints for optimisation in Clingo only cover integer costs, which means that the current ASP encodings are not directly applicable to non-integer cost e-graphs. Second, we did not investigate more solver configurations and features: the only extra one we tried is a 4-thread parallel solving for Clingo, which only achieves better suboptimal results but not better optimal results in our experiments. A portfolio-based approach may be a good way to leverage different encodings, optimisation techniques, and solver features for different instance classes.

5 More than Extraction?

The immediate takeaway is practical: the two-phase egg extraction–first egg then extraction–is effective and efficient with the right ASP encoding and solver configuration. Looking forward, this opens a broader agenda around solver-enhanced equality saturation: if egg and Datalog are "better together," then ASP may provide an even richer integration point by combining declarative modelling with stronger global optimisation and constraint handling. In the following, we discuss our ongoing work on this agenda.

5.1 What does ASP do?

It searches–in a generate-and-test style–searching for

$$\exists M.\ M \in G \wedge T(M)$$

where G is the search space defined by the ASP encoding, and T is the test condition defined by the constraints (more layers of quantification in the case of optimisation problems).

[1] https://github.com/egraphs-good/extraction-gym

5.2 What does EqSat do?

It compactly represents a congruence-closed search space of equivalent terms. An e-graph maintains a set of e-classes, each partitioning a collection of e-nodes (operator applications) that are known to be equivalent under the current rewrite state. The central congruence invariant is: *equivalent children induce equivalent parents, so every term extractable from an e-class is equivalent to every other term extractable from the same e-class.* Rewrite rules extend the e-graph by merging e-classes, growing the space of represented terms while preserving this invariant throughout.

5.3 The Combined Formulation of Extraction

This makes the two frameworks naturally complementary. In egglog [9], the combination provides a compact, equivalence-closed search space

$$G_{egraph} = lfp(Datalog_E)(facts(t_0))$$

computed as the Datalog least fixpoint of the rewrite rules, while ASP navigates that space with full combinatorial search and optimisation:

$$\exists M \in G_{egraph}.\ T(M) \quad [\text{ minimise } cost(M)\ ]$$

The gap here is clear: in ASP, the search space is defined by non-deterministic rules rather than the definite rules enabling monotone evaluation in Datalog, so the congruence closure is to be achieved differently, not as naturally as in Datalog. ASP's role is then not merely extraction after the fact, but potentially a richer integration: its CDCL engine can prune saturation by short-circuiting rule firing when a sufficiently good M is already reachable, and its weak constraints can guide *which* rewrites are worth exploring. This points toward a single, cost-aware saturation-plus-extraction pipeline rather than a strict two-phase decomposition.

5.4 Equivalence Checking as Incremental Search

Equivalence checking admits a different instantiation that falls *outside* the static generate-and-test quantification above. Rather than searching over a fixed space G_{egraph}, checking asks whether s and t can be merged by *some* finite sequence of rule firings—a question whose answer depends on the evolving state of the e-graph itself.

This maps naturally to incremental solving: each step extends the e-graph by one bounded increment of rule applications and checks whether $[s] = [t]$ holds in the current state. Crucially, each intermediate state G_k is not a raw accumulation of derived terms but is *compressed* by congruence closure—equivalent terms are merged and represented once, so the search horizon grows in the space of equivalence classes rather than individual terms. Formally, let G_k denote the congruence-closed e-graph after k increments; the checking problem becomes:

$$\exists k.\ [s]_{G_k} = [t]_{G_k}$$

with termination as soon as the condition is met—or when $G_k = G_{k-1}$ (fixpoint reached without witnessing the merge, certifying $s \not\equiv_E t$).

This is structurally analogous to bounded model checking: each increment deepens the rewriting horizon by one step, and the equivalence plays the role of the safety property.

5.5 How EqSat might enhance ASP?

In the scenarios above, full equality saturation may be unnecessary—or infeasible within resource bounds. Instead, CDCL-based ASP systems can evolve the search space on the fly, firing rewrites lazily only when they are needed to improve or certify a candidate solution. The more ambitious reverse direction is to use an e-graph to enhance ASP: an "ASP modulo equivalence" workflow where congruence closure provides additional propagation and pruning during stable-model search. How to expose and exploit such congruence information effectively inside current ASP solving remains an open question.

References

[1] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012 (LIPIcs)*, Agostino Dovier and Vítor Santos Costa (Eds.). 211–221. doi:10.4230/LIPICS.ICLP.2012.211

[2] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2012. Multi-threaded ASP solving with clasp. *Theory Pract. Log. Program.* 12, 4-5 (2012), 525–545. doi:10.1017/S1471068412000166

[3] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. 2024. Fast and Optimal Extraction for Sparse Equality Graphs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 2551–2577. doi:10.1145/3689801

[4] Mike He, Haichen Dong, Sharad Malik, and Aarti Gupta. 2023. Improving term extraction with acyclic constraints. (2023).

[5] Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. 2023. How to Build Your Own ASP-based System?! *Theory Pract. Log. Program.* 23, 1 (2023), 299–361. doi:10.1017/S1471068421000508

[6] Michael Stepp. 2011. *Equality saturation : engineering challenges and applications.* Ph. D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/85f640cc

[7] Glenn Sun, Yihong Zhang, and Haobin Ni. 2024. E-Graphs as Circuits, and Optimal Extraction via Treewidth. *CoRR* abs/2408.17042 (2024). arXiv:2408.17042 doi:10.48550/ARXIV.2408.17042

[8] Jiaqi Yin, Zhan Song, Chen Chen, Yaohui Cai, Zhiru Zhang, and Cunxi Yu. 2025. e-boost: Boosted E-Graph Extraction with Adaptive Heuristics and Exact Solving. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2025.* 1–9. doi:10.1109/ICCAD66269.2025.11240719

[9] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492. doi:10.1145/3591239

[10] Philip Zucker. [n. d.]. Answer Set Programming for E-Graph DAG Extraction. https://www.philipzucker.com/asp-for-egraph-extraction/. [Online; accessed 15-4-2026].