E-Graphs as Circuits, and Optimal Extraction via Treewidth*

Glenn Sun, Yihong Zhang, and Haobin Ni

University of Washington

Abstract

We demonstrate a new connection between e-graphs and Boolean circuits. This allows us to adapt existing literature on circuits to easily arrive at an algorithm for optimal e-graph extraction, parameterized by treewidth, which runs in $2^{O(w^2)}\text{poly}(w, n)$ time, where w is the treewidth of the e-graph. Additionally, we show how the circuit view of e-graphs allows us to apply powerful simplification techniques, and we analyze a dataset of e-graphs to show that these techniques can reduce e-graph size and treewidth by 40-80% in many cases. While the core parameterized algorithm may be adapted to work directly on e-graphs, the primary value of the circuit view is in allowing the transfer of ideas from the well-established field of circuits to e-graphs.

1 Introduction

E-graphs are a type of directed graph with an equivalence relation on its nodes that can be used to compactly represent exponentially many equivalent expressions. In recent years, this capability has found e-graphs to have many applications in formal methods, compilers, and automated reasoning communities ([JNR02], [TSTL09], [STL11], [WNW+21]).

One important problem with e-graphs is extraction: from the compact representation, how does one pick a minimum cost expression? E-graph extraction is known to be NP-hard [Ste11], so applications of e-graphs often use suboptimal techniques like greedy algorithms [WNW+21] to extract one expression out of an e-graph. If one is interested in exact optimal extraction, integer linear programming (ILP) is sometimes used [YPW+21], but there has been little research into specialized algorithms to solve extraction optimally.

Common algorithmic techniques for NP-hard optimization problems include approximation algorithms and parameterized algorithms. It turns out that extraction is also hard to approximate to any constant factor [GLP24], so it is natural to turn to parameterized algorithms: algorithms that are efficient after a particular parameter is chosen to be fixed. One commonly used parameter is *treewidth*, a measure of how "close" to a tree the graph is (a survey is available in [Bod06]).

We make three key observations:

1. E-graphs that appear in practice often have low treewidth, so it is a good choice for parameterization. We quantify this in Section 4.2.

*This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-1836724, CCF-2006359, CNS-2232339, and CCF-2312195. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

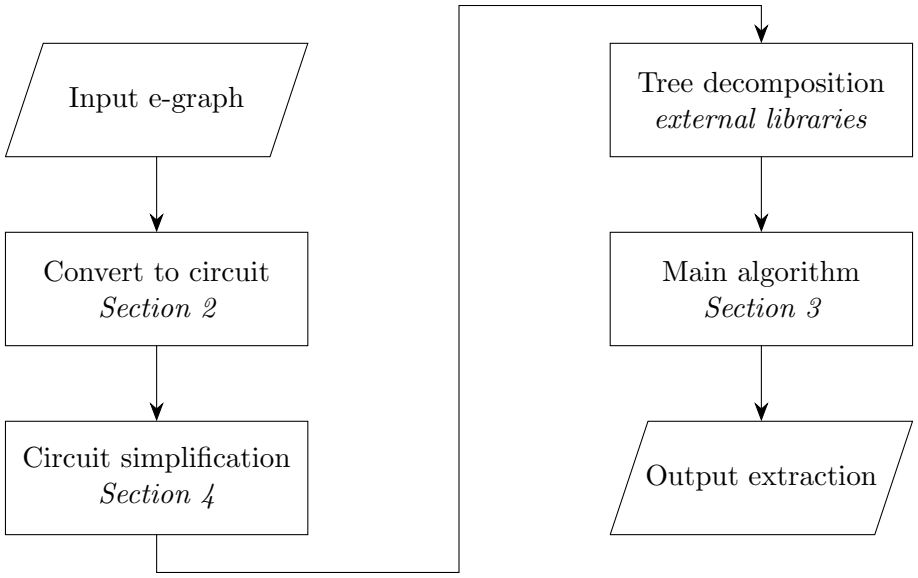

Figure 1: Overall algorithm pipeline and article organization

2. E-graphs may be considered a certain class of *monotone circuits*, meaning a Boolean circuit with only AND and OR gates (no NOT gates). This allows us to draw on existing literature about treewidth-based algorithms for circuits in order to solve extraction.

3. The circuit view also allows us to apply circuit simplification, which is more broadly studied and flexible than simplifying e-graphs directly.

More specifically, we will show in Section 2 that with the appropriate translation between e-graphs and circuits, the extraction problem is nothing more than the weighted monotone circuit satisfiability problem, with the one caveat that our circuits may have cycles. In Section 3, we then draw on an existing algorithm for weighted *acyclic* monotone circuit satisfiability given by Kanj, Thilikos, and Xia [KTX17]. Their algorithm parameterizes on treewidth as desired, and we make only one minor change to take care of cycles. In Section 4, we discuss simplification rules that make the main algorithm more practical.

Simultaneously and independently of our own efforts, Goharshady, Lam, and Parreaux [GLP24] also gave a solution to the e-graph extraction problem, also through parameterization by treewidth. They do not use circuits, but our main algorithms are actually very similar. They include a few additional extensions and optimizations as well.

Our main contribution is the connection between circuits and e-graphs, which not only allows us to use existing circuit algorithms but also employ simplification techniques more effectively. Given that our main algorithm is extremely similar to the one given in [GLP24], we will omit to give our own practical implementation, evaluation against existing extraction methods, and proofs of correctness for the main algorithm, and direct the interested reader to their paper. While we do have a publicly available implementation[1], our focus is on the circuit connection and simplification.

[1] `https://github.com/glenn-sun/egg-extraction-gym/tree/glenn-treewidth/src/extract/treewidth`

2 Circuits and e-graphs

We first show the equivalence of circuits and e-graphs. For the reader who is already familiar with e-graphs, you may find Fig. 2 sufficient to give the intuition of the equivalence.

Definition 2.1. A *weighted cyclic monotone circuit* (henceforth "circuit") is a directed graph with additional information $G = (V, E, V_{out}, g, c)$. The set $V_{in} \subseteq V$ is the set of *inputs*, defined to be the set of vertices with in-degree 0. The set $V_{out} \subseteq V$ is the set of *outputs*. (These are the nodes whose values we are interested in, which may or may not have out-degree 0.) Finally, $g : V \setminus V_{in} \to \{\text{AND}, \text{OR}\}$ is the gate type function and $c : V_{in} \to \mathbb{R}$ is a cost function. For any $U \subseteq V_{in}$, we denote $c(U) = \sum_{u \in U} c(u)$. ⌟

Because our circuits have cycles, we need to be a bit more precise than usual about the semantics of the circuit. In particular, there may be undefined behavior on certain inputs.

Definition 2.2. A function $\alpha : V \to \{0, 1\}$ is a (valid total) *evaluation* of G if for all $u \in V$, denoting the inputs to u as $\{v_1, \ldots, v_k\}$, we have $\alpha(u) = g(u)(\alpha(v_1), \ldots, \alpha(v_k))$. The evaluation *satisfies* G if $\alpha(u) = 1$ for all $u \in V_{out}$. It *minimally satisfies* G if for every proper subset $A \subsetneq \{u \in V : \alpha(u) = 1\}$, the function

$$\alpha|_A(u) = \begin{cases} \alpha(u) & \text{if } u \in A \\ 0 & \text{if } u \notin A \end{cases}$$

is not a valid evaluation satisfying G. We denote $G[\alpha]$ to be the subgraph of G induced by the set $\{u \in V : \alpha(u) = 1\}$. We say that α is *acyclic* if $G[\alpha]$ is acyclic. ⌟

The key fact that allows us to have well-defined semantics with cyclic circuits is the following:

Proposition 2.3. *An acyclic evaluation α is uniquely determined by its value on the inputs.* ⌟

Proof. Let β be an acyclic evaluation which agrees with α on V_{in}, we will show that $\alpha(u) = 1$ iff $\beta(u) = 1$. The forward and backward directions are identical. Let us treat the forward direction.

Recall that $\alpha(u) = 1$ iff $u \in G[\alpha]$. Because $G[\alpha]$ is acyclic, take the vertices in topological order. The base cases are the elements of $G[\alpha]$ with in-degree 0; note that these must have in-degree 0 in G because gates cannot be 1 without at least one 1 input. Hence β agrees with α here by hypothesis.

For the inductive step, to show that $\beta(u) = 1$, take cases based on the gate type of u. If u is an AND gate, because $\alpha(u) = 1$, all of u's inputs in G belonged to $G[\alpha]$, so β is 1 there by induction, and the only valid choice for $\beta(u)$ is 1. If u is an OR gate, a similar argument applies. □

Next, let us draw the connection between e-graphs and circuits.

Definition 2.4. An *e-graph* is a structure $\mathcal{G} = (N, \mathcal{C}, \mathcal{E}, \mathcal{C}_{out}, c)$, where N is a set of *e-nodes*, $\mathcal{C}$ is a partition of N into *e-classes*, $\mathcal{E} \subseteq N \times \mathcal{C}$ is a directed edge relation, $\mathcal{C}_{out} \subseteq \mathcal{C}$ is the set of *output classes*, and $c : N \to \mathbb{R}$ is a cost function. For any $M \subseteq N$, we denote $c(M) = \sum_{u \in M} c(u)$. ⌟

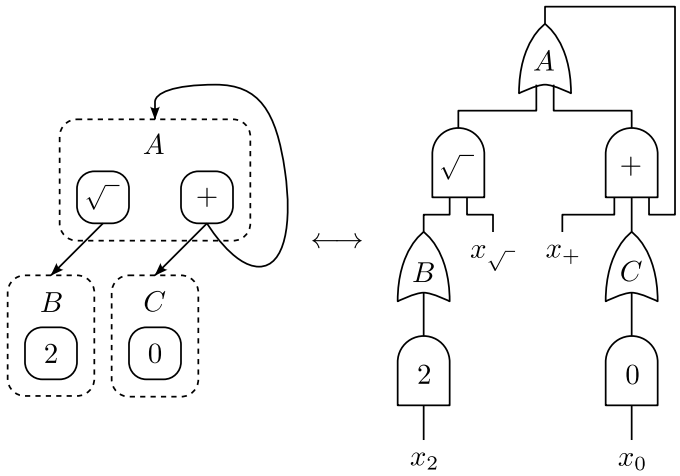

Figure 2: An example of converting e-graphs into circuits. Note that the arrows conventionally point to dependencies in e-graphs, but signals flow in the opposite direction in a circuit, so we flip the arrows. Furthermore, the extraction $A \mapsto \sqrt{}$, $B \mapsto 2$ corresponds to the evaluation where everything on the left and the OR gate A are all 1, and the rest are 0. The cyclic extraction $A \mapsto +$, $C \mapsto 0$ corresponds to the cyclic evaluation where everything on the right and the OR gate A are all 1, and the rest are 0. The evaluation where everything is 1 has no corresponding extraction because A can only choose one e-node in an e-graph; however, such an evaluation is not minimal.

Definition 2.5. Let $\mathrm{dom}(\varphi) \subseteq \mathcal{C}$. An *extraction* of an e-graph is a choice function $\varphi : \mathrm{dom}(\varphi) \to N$ (that is, $\varphi(C) \in C$ for all $C \in \mathrm{dom}(\varphi)$) which additionally has that whenever $C \in \mathrm{dom}(\varphi)$, the inputs to $\varphi(C)$ are all in $\mathrm{dom}(\varphi)$ as well. The extraction is *satisfying* if $C \in \mathrm{dom}(\varphi)$ for all $C \in \mathcal{C}_{out}$. It is *minimally satisfying* if for every proper subset $\mathcal{A} \subsetneq \mathrm{dom}(\varphi)$, the function $\varphi|_{\mathcal{A}}$ is not a satisfying extraction. A *selected path* in φ is a finite list of e-classes $C_1, \ldots, C_k$ such that for all $1 \leq i \leq k-1$, we have $(\varphi(C_i), C_{i+1}) \in \mathcal{E}$. The extraction is *acyclic* if there are no selected paths from a class to itself. ⌟

Our main observation is that every e-graph can be represented as a circuit in such a way that its semantics are equivalent. For an example, see Fig. 2.

Proposition 2.6. *Given an e-graph $\mathcal{G} = (N, \mathcal{C}, \mathcal{E}, \mathcal{C}_{out}, c)$, construct a monotone circuit $G = (V, E, V_{out}, g, c)$ by converting every e-class into an OR gate, every e-node into an AND gate, and then flip every edge. Additionally, create one input for every e-node, and attach it to its corresponding AND gate, with cost set equal to the cost of the e-node.*

Then, there exists an acyclicity-preserving bijection between minimal satisfying extractions of $\mathcal{G}$ and minimal satisfying evaluations of G. ⌟

The proof is a tedious checking of these definitions that does not require any external results. The details are contained in Appendix A. With this observation, in order to solve the extraction problem for e-graphs, it suffices to solve the weighted satisfiability problem for (potentially cyclic) circuits.

3 The main dynamic programming algorithm

3.1 Preliminaries

Our main inspiration is Proposition 4.7 of [KTX17], which solved the weighted minimum satisfiability problem for acyclic monotone circuits by parameterizing on treewidth. Based on the reduction illustrated in the previous section, the only additional algorithmic contribution that we need to make is to describe why the cyclic nature of the graph is not a problem and how we enforce the extraction result to be acyclic. For completeness, we will recap the full algorithm (with slightly different notations from the original paper).

The main technique is called treewidth, or tree decomposition. This is a classical technique; see Chapter 7 of [CFK+15] for more information.

Definition 3.1. Given a undirected graph $G = (V, E)$, a tree decomposition of G is a tree $\mathcal{T} = (\mathcal{X}, \mathcal{E})$, whose vertices are subsets of V (called *bags*), satisfying:

1. For all $\{u, v\} \in E$, there exists a bag $X \in \mathcal{X}$ such that $u, v \in X$.
2. For all $v \in V$, the subgraph of $\mathcal{T}$ induced by the bags that contain v forms a tree.

The *width* of a tree decomposition is the size of the largest bag minus one. The *treewidth* of a graph is the smallest width of any tree decomposition. If $\mathcal{T}$ is rooted, we write T_X for the union of all bags underneath $X \in \mathcal{X}$ (including X). A *nice tree decomposition* is one in which every bag X is one of 4 kinds:

1. Leaf bag: $X = \emptyset$.
2. Insert bag: $X = Y \cup \{u\}$, where Y is the unique child of X and $u \in V \setminus Y$.
3. Forget bag: $X = Y \setminus \{u\}$, where Y is the unique child of X and $u \in Y$.
4. Join bag: X has two children, which contain exactly the same vertices as X. ⌟

A nice tree decomposition can be computed from a tree decomposition in linear time: pick any bag to serve as the root, then for all children of a bag, first forget and insert the difference between the child and the current bag, then join all of the copies of the current bag, and repeat.

The core feature of tree decomposition is that as you walk up the tree, the current bag cuts the original graph into two disconnected pieces: the set of vertices that you have already seen (including those that are forgotten) and the set of vertices that you have not seen yet. This property allows us to do dynamic programming and focus only on the current bag.

One last definition that we will use in the algorithm is that of a partial evaluation since we would like to build up our evaluations at each bag incrementally. We will use the following definition with $U = T_X$.

Definition 3.2. A (locally valid) partial evaluation of a circuit is a function $\alpha : U \to \{0, 1\}$ where $U \subseteq V$, and for all $u \in U$, denoting the inputs to u as $\{v_1, \ldots, v_k, w_1, \ldots, w_\ell\}$ where each $v_i \in U$ and each $w_j \notin U$, there exist $b_1, \ldots, b_\ell \in \{0, 1\}$ such that $\alpha(u)$ is $g(u)$ (AND or OR) applied to $(\alpha(v_1), \ldots, \alpha(v_k), b_1, \ldots, b_\ell)$. ⌟

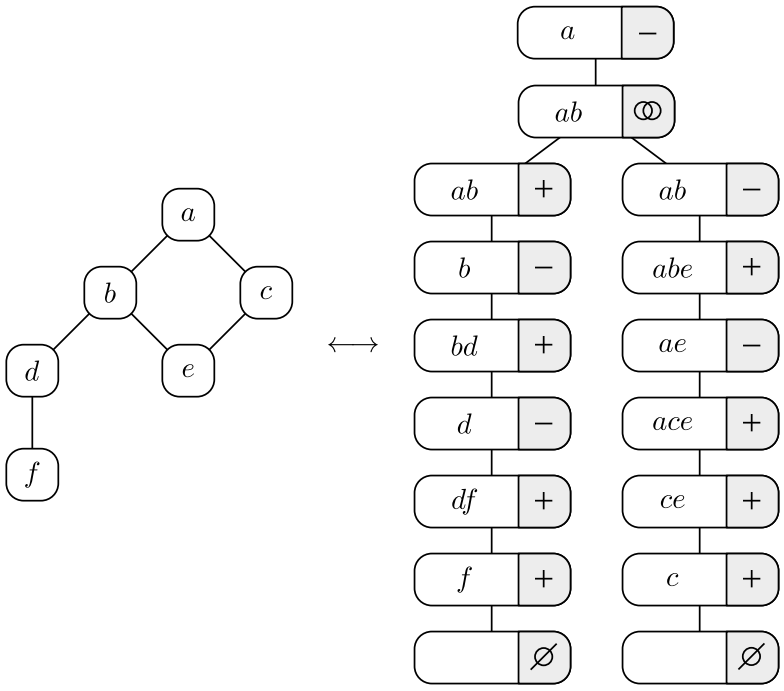

Figure 3: A nice tree decomposition of a graph, where each node is annotated whether it is a leaf, insert, forget, or join node. This graph has treewidth 2.

3.2 The algorithm

We are given a weighted cyclic monotone circuit $G = (V, E, V_{out}, g, c)$, and are tasked to compute a minimum cost satisfying evaluation. The algorithm sketch is as follows:

1. Add a new AND gate to G, with every vertex in V_{out} as inputs, and call it u_{out}.
2. Compute a tree decomposition of the undirected underlying graph of G.
3. Compute a nice tree decomposition rooted at any bag containing u_{out}.
4. We do dynamic programming. At every bag, for every possible *summary* of a partial evaluation, the program will remember the minimum cost partial evaluation producing that summary. The summary of a partial evaluation $\alpha : T_X \to \{0, 1\}$ at bag X has three parts:
   - First, there is the restriction $\alpha|_X$.
   - Second, there is the map $\text{known}_\alpha|_X$, where $\text{known}_\alpha(u) = 1$ if and only if $\alpha(u) = 1$ and that this fact can be deduced from the value of α on the inputs to u. For example, if u is an AND gate and one of its inputs is not in $\text{dom}(\alpha)$, then $\text{known}_\alpha(u) = 0$.
   - Lastly, there is $G[\alpha]^+[X]$, the subgraph of the transitive closure of $G[\alpha]$ induced by X. In other words, which elements in X have paths between them in $G[\alpha]$? This is what allows us to correctly handle acyclicity, remembering which vertices in the bag are connected, even when the connections themselves have already been forgotten from the bag, and is the only change from [KTX17].

We break into cases depending on the type of bag.

1. Leaf bag: We map the empty summary to the empty evaluation.
2. Insert bag ($X = Y \cup \{u\}$): For every partial evaluation α remembered at Y, we attempt to extend it with $\alpha(u) = 0$ and $\alpha(u) = 1$. If these extensions are valid and acyclic, compute their new summaries and remember them if they have the smallest cost for their summary so far.
3. Forget bag ($X = Y \setminus \{u\}$): For every partial evaluation α remembered at Y, restrict the entire summary to X and remember the lowest cost evaluations per summary.
4. Join bag: Denote this bag as X with children Y and Z (even though as sets $X = Y = Z$). By property 2 of tree decompositions, T_Y and T_Z intersect only at X. Therefore, for all remembered $\alpha : T_Y \to \{0, 1\}$ and $\beta : T_Z \to \{0, 1\}$, as long as they agree on X, they can be merged into a new evaluation on T_X. If it is valid and acyclic, compute its summary and remember it if it has the smallest cost for its summary so far.

5. At the root, output the minimum cost evaluation producing the summary corresponding to $\alpha(u_{out}) = 1$.

The discussion within the algorithm gave some ideas to why $G[\alpha]^+[X]$ is necessary in the summary, but it remains to motivate known_α. Consider an OR gate u such that all but one of its children have been forgotten. Without known_α, we might keep only the evaluation where all of its children are set to 0, because that is cheaper, forcing us to pick the last child to continue to this line, even if it is suboptimal.

The running time of this algorithm is roughly $2^{O(w^2)}\text{poly}(w, n)$, where w is the treewidth and $n = |V|$. The precise coefficients and polynomial degree are dependent on the data structures in the implementation, but the largest term comes from there being at most $2^{O(w^2)}$ distinct summaries for each bag, since the transitive graph has $O(w^2)$ edges. For more details, see [KTX17] or [GLP24].

Note that when implementing this algorithm, instead of actually remembering the best partial evaluation per summary, it is faster to just remember a pointer to the previous summary that produced it. At the end of the algorithm, one can walk back and recover the full evaluation using these pointers. Depending on the precise data structures, another small optimization could be to combine $\alpha|_X$ and $\text{known}_\alpha|_X$ into a ternary-valued collection in the summary, since known_α is only recorded for true vertices.

4 Circuit simplification

4.1 Rules

One natural way to improve the speed of our algorithm is to simplify the instances directly. We found in our testing that this helps dramatically. The idea of preprocessing e-graphs to simplify them before extraction is not new (see for example, the repository at [Han24]), but the circuit view of e-graphs makes this process easier, more transparent, and more powerful.

Finding the most compact representation of a Boolean circuit is often known as circuit minimization. Circuit minimization is well-known to be NP-hard, but there is an abundance of existing software to quickly attempt a best effort, such as SIS [SSL+92] among others.

Note that in order to use off-the-shelf software, it must support cycles. Though our particular semantics are unique, all simplifications valid for *sequential circuits* are valid for us, a type of circuits fundamental in hardware. Sequential circuits are cyclic circuits where the "undefined" behavior is explicitly specified by propagation delay and keeping track of the circuit's state. A minimizer for sequential circuits will additionally ensure that these stateful behaviors are preserved, which we can just ignore—it must also preserve behaviors independent of state, as acyclic extractions are. Thus, we may want to do some additional simplification afterwards specific to our own extraction semantics, but such software may be used as a first step.

To show some basic ideas, as well as to highlight properties of extraction to make simplifications beyond what is possible from off-the-shelf software, we implemented several of our own heuristic rules. For all of the rules below, let $G = (V, E, V_{out}, g, c)$ be a monotone circuit. These rules are by no means an exhaustive list of all possible simplifications, they are only an exploratory list of the kinds of rules that may be beneficial.

Proposition 4.1. *When applying each of the following rules to a weighted cyclic monotone circuit, the optimal acyclic evaluation is either retained or efficiently recoverable.*

1. (Remove unreachable) For all $u \in V$, if there does not exist a path from u to some $v \in V_{out}$, then it is safe to remove u from V.

2. (Contract indegree one) Suppose $u \in V$ has indegree 1, in particular $(v, u) \in E$. Then it is safe to contract the edge (v, u). The new vertex has the same gate type as v.

3. (Contract same gate) Suppose $v \in V$ has outdegree 1, in particular $(v, u) \in E$, and suppose $g(v) = g(u)$. Then it is safe to contract the edge (v, u). The new vertex has the same gate type as v and u.

4. (Same gate no shortcut) Suppose $(v, u) \in E$ and there exists a path $(v, w_1), (w_1, w_2), \ldots, (w_{n-1}, w_n), (w_n, u) \in E$ such that $g(v) = g(w_1) = \cdots = g(w_{n-1}) = g(u)$. Then it is safe to delete (v, u).

5. (Factoring) Suppose we have $(w, v_i), (v_i, u) \in E$ where $g(v_i) = \text{OR}$ for $2 \leq i \leq n$, and $g(u) = \text{AND}$. Then it is safe to delete all of these edges and replace them with two new vertices a and b, where $g(a) = \text{AND}$ and $g(b) = \text{OR}$, with the edges (v_i, a) for all i, (a, b), (w, b), and (b, u). The rule may also apply with AND and OR swapped.

6. (Remove lone OR loops) Suppose $u \in V$ is an OR gate and $v_1, \ldots, v_n \in V$ are AND gates, and $(u, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, u) \in E$. Then it is safe to delete v_n.

7. (Collect variables) Suppose $u_1, u_2 \in V$ are variables with the same out-neighborhood, all of which are AND gates. Then it is safe to merge u_1 and u_2 into a new variable with the same out-neighborhood, with cost the sum of the originals. ⌟

Proof. 1. The optimal acyclic evaluation is minimal, so it does not set true any vertices that do not affect the output.

2. Suppose $u \in V$ has indegree 1, in particular $(v, u) \in E$. Then it is safe to contract the edge (v, u). The new vertex has the same gate type as v.

3. Because AND and OR are associative, when two of the same gate are adjacent and the subexpression is not reused in other situations, it is an equivalent circuit to merge the two gates.

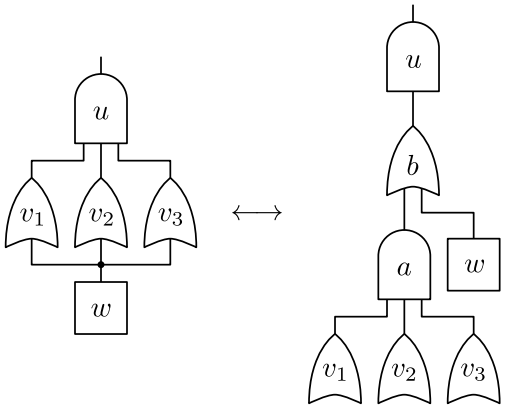

Figure 4: The factoring rule. All gates may have additional inputs/outputs, which are preserved and not depicted here. The square denotes any vertex.

---

4. One may check for both AND and OR that after deleting the edge, the dependency is still maintained through the path.

5. This rule is nothing more than observing how $(w \vee x_1) \wedge \cdots \wedge (w \vee x_n) = w \vee (x_1 \wedge \cdots \wedge x_n)$, generalized slightly.

   (See Fig. 4 for a visualization of this rule. We note that this is the only rule that may increase the size of the circuit. However, it is generally beneficial to apply this because it reduces the number of cycles in the underlying undirected graph, which generally reduces the treewidth.)

6. Suppose v_n is true. Then by induction, every v_i is true, as well as u. Then, this evaluation has a true cycle. So, the optimal acyclic evaluation must have v_n false, and we can delete it.

7. Simply note that an evaluation that sets u_1 to true but u_2 to false is not minimal, so u_1 and u_2 must be both false or both true. □

Some of these rules are purely based on the circuit structure (rules 2, 3, 4, 5). These were rules that we arbitrarily decided to implement, but in future work, one might consider replacing with an existing general purpose circuit minimization tool. This is one potential benefit of circuits that we have yet to explore. Other rules are more specific to extraction (rules 1, 6, 7). We note that some rules can be generalized to all extraction algorithms, such as rule 1.

Although there are often equivalent rules that operate on e-graphs directly without translating to circuits, circuits can generally be smaller (not to mention unlocking the ability to harness existing software). Recall that a direct translations of an e-graph will always result in an alternating AND/OR pattern, with every AND gate having outdegree 1 and one variable as an in-neighbor. Rules like rule 2 and rule 7 break this structure, and they are valid because the algorithm works for all monotone circuits.

Source	no. of e-graphs	avg. $\|V\|$	avg. degree
babble	173	5336.8	2.6
egg	28	4276.5	4.1
eggcc-bril	36	20329.5	2.7
flexc	14	23620.7	3.4
fuzz	18	126.0	4.0
rover	9	21303.4	6.8
tensat	10	57969.9	3.3

Table 1: Basic characteristics of "extraction gym" dataset after circuit conversion.

4.2 Simplification evaluation

To evaluate our simplification rules, we applied them to a large set of e-graphs from various sources, collected in the "extraction-gym" benchmarking suite [Goo24]. These are e-graphs that were generated by real projects, such as a e-graph based general purpose compiler ("eggcc-bril"), a compiler for specialized hardware ("flexc"), a fuzzer for automated testing ("fuzz"), and several other sources. A basic summary of the dataset is given in Table 1.

Note that the main algorithm's correctness only relies on having a valid tree decomposition of any width—a larger width only affects the running time. Thus, although we unfortunately found that the vast majority of e-graphs in this test set were too large for computing exact tree decomposition, approximate tree decomposition algorithms suffice. We used the "arboretum" Rust library, which implements various heuristics but primarily relies on the classical minimum degree heuristic to compute an upper bound on treewidth.

In order to compare the effects of our simplification scheme, such an upper bound is also a more realistic measure to compare than the true treewidth, since the important quantity is the width of the tree decomposition available to our algorithm. In our implementation, we apply each of the rules in a loop until we reach a fixed point. The effect of simplification on treewidth and $|V|$ is shown in Fig. 5, and these effects are quantified in Table 2.

Note that every source produces e-graphs in a different way, which can dramatically affect which optimizations are more effective. E-graphs from some sources, like "eggcc-bril", demonstrated extraordinary simplification with 65% reduction in treewidth and 97% reduction in $|V|$, whereas e-graphs from other sources like "babble" or "flexc" demonstrated negligible improvement in treewidth (or even slight degradation due to the non-exact tree decomposition algorithm), although $|V|$ continues to be reduced substantially, by 60% or more in all but one collection.

We note that although the treewidth of most e-graphs in our dataset is small, sparsity is likely to be the only contributing factor to low treewidth in these data, not any deeper features of the e-graph generation process. This is because the linear relationship between $|V|$ and treewidth is the exact relationship predicted by random graphs of constant average degree: In a random graph where each edge has probability c/n of appearing for $c > 1$, the treewidth of the graph is $\Omega(n)$ [LLO11] and upper bounded by tn for some $t < 1$ [WLCX11].

Lastly, we note that these simplification methods bring many more e-graphs into the range in which treewidth-based methods are faster than existing methods. Though we did not develop an efficient implementation ourselves, [GLP24] claims that their implementation of treewidth-based extraction is faster than ILP solvers for most e-graphs with treewidth under 10, at least when not required to output acyclic extractions. Few e-graphs in our

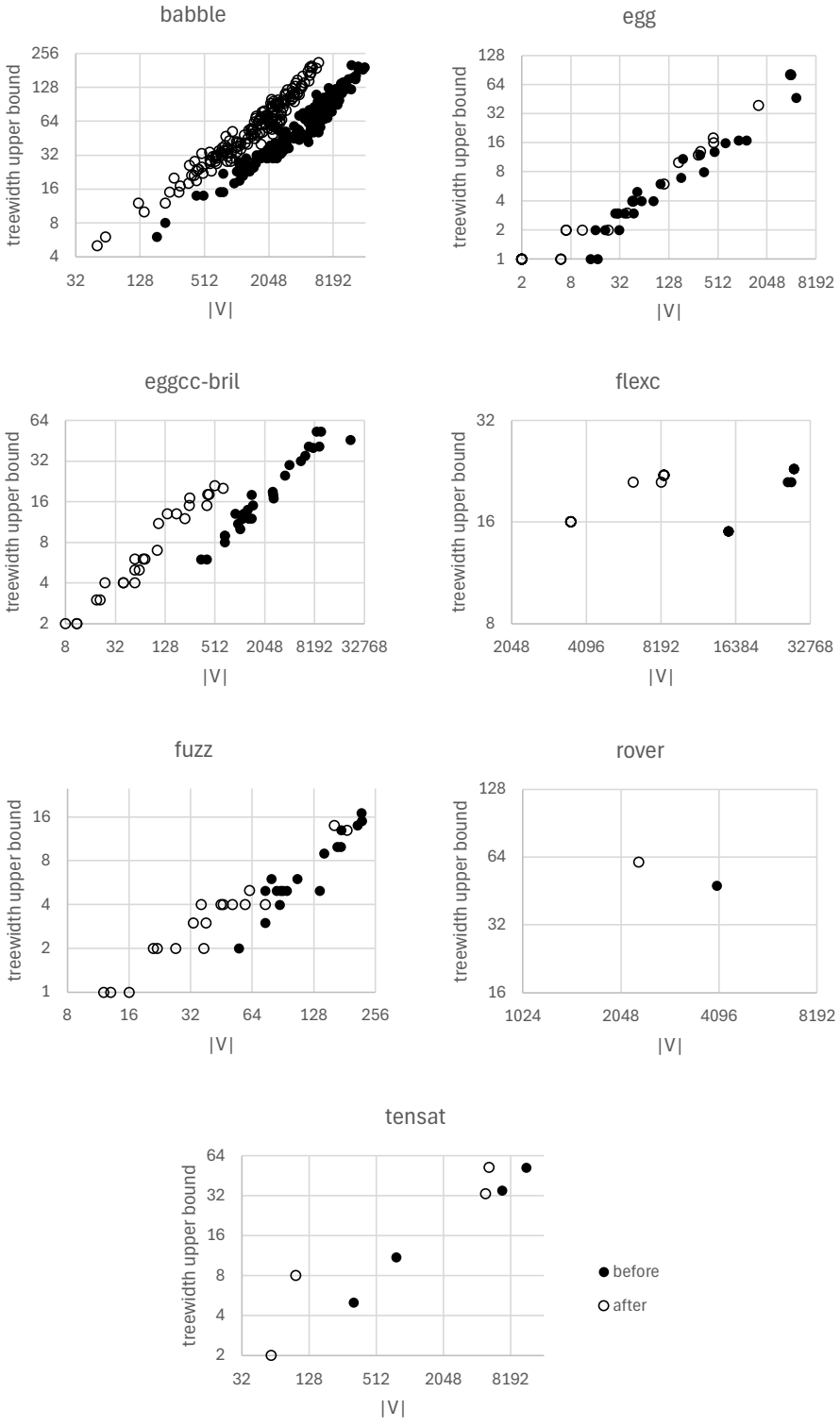

Figure 5: Treewidth and $|V|$ before and after applying all simplification rules. Note that there are some artifacts due to heuristics used in tree decomposition, e.g. in "rover". A few examples continued to time out after 15 seconds and are omitted from these data.

Source	avg. Δ treewidth	avg. $\Delta\|V\|$	avg. $\Delta\|E\|$	% timeout
babble	-5%	-64%	-57%	1%
egg	-40%	-72%	-80%	7%
eggcc-bril	-65%	-97%	-96%	17%
flexc	1%	-74%	-81%	7%
fuzz	-46%	-60%	-73%	0%
rover	27%	-42%	-72%	89%
tensat	-23%	-63%	-64%	60%

Table 2: Results of simplification, using heuristic approximate tree decomposition, so some instances may falsely confuse an actual rise in treewidth for an instance that just happens to be more difficult for the heuristic. A few examples continued to time out after 15 seconds and are omitted from these data.

---

test set had treewidth under 10 before simplification, but especially for the egraphs in the collections "eggcc-bril" and "fuzz", a significant fraction of them have treewidth under 10 after simplification.

5 Future directions

Many open questions remain regarding extraction and treewidth. The largest open question in our mind relates to more general cost functions than the simple additive ones we have considered here. This is the one area where ILP solvers can never work—by nature of being integer *linear* programs, they are not suitable for other cost functions.

In general, every e-node can be associated with a local cost function, which depends on the costs of its children. For example, an e-node representing the operation "do my child again" could have cost $c(x) = x$, in other words it would copy the cost of its child. Such cost functions present a challenge to this present algorithm because we do not enforce the children to be discovered before the parent. In other words, we have to decide whether or not "do my child again" is cheap, without knowing the cost of the child, in order to decide whether or not to keep that partial evaluation.

One potential solution would be to represent costs abstractly, and keep all minimal cost solutions, not just one minimum cost one. In this example, we would define a symbolic variable x denoting the unknown cost, and set the cost of "do my child again" to x. If an evaluation has cost x and another evaluation with the same summary has, for example, cost 7, we would keep both, and then evaluations with cost $2x$, $x + 5$, or 8 could be thrown away. When the vertex associated with x is inserted, these expressions would update based on the cost of x that we now know. However, this would massively increase the running time of the algorithm, and would generally not be polynomial with fixed treewidth. It is an interesting open direction to hand such general cost functions efficiently.

We note that [GLP24] analyzed the main algorithm (that we share) closely, and identified some criteria slightly more general than additive cost functions that actually work automatically. However, those criteria still exclude nodes like "do my child again", so there remains work to be done.

A second open direction is to investigate more closely if particular methods of e-graph creation lead to smaller treewidth, and then design e-graph creation methods (or *saturation*,

as it is often called) that minimize treewidth from the start. Since treewidth is often calculated with heuristic algorithms, it would be interesting to determine if certain saturation methods could even be designed with a particular treewidth heuristic in mind to further improve efficiency.

A Proof of Proposition 2.6

Proposition A.1. *Let $\mathcal{G} = (N, \mathcal{C}, \mathcal{E}, \mathcal{C}_{out}, c)$ be an e-graph and define $G = (V, E, V_{out}, g, c)$ as follows:*

1. *Let $V = \{x_u : u \in N\} \cup \{\wedge_u : u \in N\} \cup \{\vee_C : C \in \mathcal{C}\}$.*
2. *Let*

$$E = \{(\wedge_u, \vee_C) : u \in C \in \mathcal{C}\} \cup \{(\vee_C, \wedge_u) : (u, C) \in \mathcal{E}\} \cup \{(x_u, \wedge_u) : u \in N\} \quad (\star)$$

3. *Let $V_{out} = \{\vee_C : C \in \mathcal{C}_{out}\}$.*
4. *Let g map each $\wedge_u$ to AND and each $\vee_C$ to OR.*
5. *Let $c(x_u) = c(u)$.*

Then the following map is a bijection between minimal satisfiable extractions of $\mathcal{G}$ and minimal satisfiable evaluations of G. We map an extraction φ to the function α defined for all $u \in C \in \mathcal{C}$:

$$\alpha(x_u) = \alpha(\wedge_u) = \mathbf{1}(\varphi(C) = u) \qquad \alpha(\vee_C) = \mathbf{1}(C \in \mathrm{dom}(\varphi)) \quad (\dagger)$$

The bijection also preserves acyclicity and cost. ⌟

Proof. First, we need to show that α is a minimal satisfying evaluation.

- To show that α is valid, we check the gate functions using $(\star)$ and $(\dagger)$.
  1. If $\alpha(\wedge_u) = 0$, then x_u is an input and $\alpha(x_u) = 0$, so we are good.
  2. If $\alpha(\wedge_u) = 1$, then $\alpha(x_u) = 1$, and the other inputs are $\vee_C$ for all $(u, C) \in \mathcal{E}$. By definition of extraction, $C \in \mathrm{dom}(\varphi)$, so $\alpha(\vee_C) = 1$ as required.
  3. If $\alpha(\vee_C) = 0$, then $C \notin \mathrm{dom}(\varphi)$, so for all $u \in C$, we have $\alpha(\wedge_u) = 0$ as required.
  4. If $\alpha(\vee_C) = 1$, then $C \in \mathrm{dom}(\varphi)$ and $\varphi(C) = u$ for some $u \in C$. Hence $\alpha(\wedge_u) = 1$ as required.
- To show that α is minimally satisfying, it is certainly satisfying from the definitions of α and V_{out}, so it remains to show minimality. Suppose for contradiction that there exists a proper subset $A \subsetneq \{i \in V : \alpha(i) = 1\}$ such that $\alpha|_A$ is a satisfying evaluation.

  Let $\mathcal{A} = \{C \in \mathcal{C} : \alpha|_A(\vee_C) = 1\}$. We will show that $\mathcal{A} \subsetneq \mathrm{dom}(\varphi)$ yet $\varphi|_{\mathcal{A}}$ is satisfying, contradicting the fact that φ is minimal satisfying.

  First, note that $\mathcal{A}$ is a subset of $\mathrm{dom}(\varphi)$ because $\alpha|_A(\vee_C) = 1$ implies $C \in \mathrm{dom}(\varphi)$ by $(\dagger)$. Next, we show that it is a proper subset. Because A is a proper subset of $\{i \in V : \alpha(i) = 1\}$, there is a true vertex (by α) outside of A.

1. If it is $\vee_C$, then $\alpha|_A(\vee_C) = 1$ and C is a satisfied class outside of $\mathcal{A}$.
2. If it is $\wedge_u$, by our construction (†), no siblings of $\wedge_u$ are true. So if $\wedge_u \notin A$ and C denotes the class containing u, by validity of $\alpha|_A$, we have $\alpha|_A(\vee_C) = 0$, and C is a satisfied class outside of $\mathcal{A}$.
3. If it is x_u, by validity of $\alpha|_A$, the vertex $\wedge_u$ must be false in $\alpha|_A$, and $\wedge_u$ must be true in α by (†). Therefore, we are in case (2) and can repeat the argument to find the desired C.

Now to show that $\varphi|_{\mathcal{A}}$ is a satisfying extraction. First, to show that it is a valid extraction, if $\varphi(C)$ depends on $C_1, \ldots, C_k$, by validity of $\alpha|_A$ it must be that $\alpha|_A(\vee_{C_i}) = 1$ for each i, so $C_i \in \mathcal{A}$ as required for an extraction. The extraction is satisfying because $\alpha|_A$ being satisfying implies $\alpha|_A(\vee_C) = 1$ for all $C \in \mathcal{C}_{out}$, and hence $C \in \mathcal{A}$ as required.

To prove that the map is a bijection, we define the inverse. Given a minimal satisfying total evaluation α, let $\varphi(C) = u$ if and only if $\alpha(\wedge_u) = 1$ for some $u \in C$. The inverse map is a well-defined choice function because for every C, there exists at most one $u \in C$ such that $\alpha(\wedge_u) = 1$. This follows from minimality of α: if $\alpha(\wedge_u) = \alpha(\wedge_v) = 1$ for $u, v \in C$, then taking $A = V \setminus \{\wedge_u, x_u\}$ would allow $\alpha|_A$ to be a satisfying evaluation, noting that the only output of $\wedge_u$ is $\vee_C$. Now we need to show that φ is a minimal satisfying extraction.

- To show that φ is an extraction, simply note that whenever $\varphi(C)$ is defined, by the above construction $\alpha(\wedge_{\varphi(C)}) = 1$, so by validity of α and ($\star$), all dependencies C_i must have $\alpha(\vee_{C_i}) = 1$. Again by validity, this means that for each C_i, at least one $u_i \in C_i$ must have $\alpha(\wedge_{u_i}) = 1$, so $C_i \in \mathrm{dom}(\varphi)$ and we are done.

- To show that φ is satisfying, simply note that because α is satisfying, $\alpha(\vee_C) = 1$ for all $C \in \mathcal{C}_{out}$. Each $\vee_C$ can only be 1 if at least one of its children $\wedge_u$ is evaluated to 1 where $u \in C$, so $C \in \mathrm{dom}(\varphi)$ and we are done.

  To show minimality, suppose for contradiction that there exists a proper subset $\mathcal{A} \subsetneq \mathrm{dom}(\varphi)$ such that $\varphi|_{\mathcal{A}}$ is a satisfying extraction. Consider $A = \{x_u, \wedge_u, \vee_C : \varphi|_{\mathcal{A}}(C) = u\}$. We will show that $\alpha|_A$ is a satisfying evaluation with $A \subsetneq V$.

  To show that $\alpha|_A$ is a valid evaluation, we need to check the gate functions.

  1. If $\alpha|_A(\wedge_u) = 0$, then either $\alpha(\wedge_u) = 0$, in which case this is valid by validity of α, or the class of u did not belong to $\mathcal{A}$, in which case we also have $\alpha|_A(x_u) = 0$, which suffices for validity.
  2. If $\alpha|_A(\wedge_u) = 1$, then where C is the class of u, we have $C \in \mathcal{A}$. Because $\varphi|_{\mathcal{A}}$ an extraction, we conclude that the children $C_1, \ldots, C_k$ of u also belong to $\mathcal{A}$, so combined with the fact that α is valid, we conclude that $\alpha|_A(\vee_{C_i}) = 1$ as desired. Again because α is valid and $\wedge_u \in A$ if and only if $x_u \in A$, we also have $\alpha|_A(x_u) = 1$ to conclude.
  3. If $\alpha|_A(\vee_C) = 0$, then either $\alpha(\vee_C) = 0$, in which this is valid by validity of α, or $C \notin \mathcal{A}$, in which case there is no $u \in C$ for which $\wedge_u \in A$. Hence $\alpha|_A(\wedge_u) = 0$ for all $u \in C$, which suffices for validity.
  4. If $\alpha|_A(\vee_C) = 1$, then we have $C \in \mathcal{A}$. Then where $u = \varphi|_{\mathcal{A}}(C) = \varphi(C)$, we have $\wedge_u \in A$, and hence $\alpha|_A(\wedge_u) = \alpha(\wedge_u) = 1$, which suffices for validity.

As mentioned above, $\alpha|_A$ is satisfying because α is satisfying and $\vee_C \in A$ for all $C \in \mathcal{C}_{out}$, because $\varphi|_{\mathcal{A}}$ is satisfying. Then $A \subsetneq V$ because if $C \in \mathrm{dom}(\varphi) \setminus \mathcal{A}$, then $\alpha(\vee_C) = 1$ whereas $\alpha|_A(\vee_C) = 0$.

Lastly, to show that inverse map is truly an inverse, we need to show that transforming φ to α to φ is the identity, which is obvious, and that transforming α to φ to α is identity, for which it suffices to note that α is entirely determined by its values on $\wedge_u$: vertices $\vee_C$ only have vertices of type $\wedge_u$ as inputs, so they are determined, and we must have $\alpha(x_u) = \alpha(\wedge_u)$, because $\alpha(x_u) = 0$ with $\alpha(\wedge_u) = 1$ is not valid and $\alpha(x_u) = 1$ with $\alpha(\wedge_u) = 0$ is not minimal (take $A = V \setminus \{x_u\}$).

To show that the bijection preserves acyclicity, we need to show both directions:

- Suppose φ is acyclic. To show that $G[\alpha]$ is acyclic, it suffices to show that every directed cycle in G has at least one vertex on which α is 0. The only possible directed cycles of G occur as alternations of edges of type $(\wedge_u, \vee_C)$ and $(\vee_C, \wedge_u)$. So let $\wedge_{u_1}, \vee_{C_1}, \ldots, \wedge_{u_k}, \vee_{C_k}, \wedge_{u_{k+1}} = \wedge_{u_1}$ be a cycle in G, where $u_i \in C_i$ and $(u_{i+1}, C_i) \in \mathcal{E}$ for all i, and it would suffice to find u_i such that $\alpha(\wedge_{u_i}) = 0$. Because φ is acyclic, there must be some u_i for which $\varphi(C_i) \neq u_i$ (otherwise $C_1, \ldots, C_k, C_1$ is a cycle). Therefore by (†), $\alpha(\wedge_{u_i}) = 0$ as desired.

- Suppose α is acyclic and suppose for contradiction that $C_1, \ldots, C_k = C_1$ forms a cycle in φ. Then for each i, denoting $u_i = \varphi(C_i)$, we have that $\alpha(\wedge_{u_i}) = 1$ and hence $\alpha(\vee_{C_i}) = 1$. But then $\wedge_{u_1}, \vee_{C_1}, \ldots, \wedge_{u_{k-1}}, \vee_{C_{k-1}}, \wedge_{u_1} = \wedge_{u_k}$ is a cycle in $G[\alpha]$, a contradiction.

Lastly, for the cost, we simply note that by the bijection, $x_u = 1$ if and only if $\varphi(C) = u$ where C is the class of u, so this is clear. □

References

[Bod06] Hans L. Bodlaender. Treewidth: characterizations, applications, and computations. In *Proceedings of the 32nd International Conference on Graph-Theoretic Concepts in Computer Science*, WG'06, page 1–14, Berlin, Heidelberg, 2006. Springer-Verlag.

[CFK+15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[GLP24] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. Fast and optimal extraction for sparse equality graphs. *To appear in Proceedings of the ACM on Programming Languages*, 2024.

[Goo24] Egraphs Good. Extraction gym, 2024. `https://github.com/egraphs-good/extraction-gym`.

[Han24] Trevor Hansen. faster-ilp extractor, 2024. `https://github.com/egraphs-good/extraction-gym/pull/16`.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 304–314, New York, NY, USA, 2002. Association for Computing Machinery.

[KTX17] Iyad Kanj, Dimitrios M. Thilikos, and Ge Xia. On the parameterized complexity of monotone and antimonotone weighted circuit satisfiability. *Information and Computation*, 257:139–156, 2017.

[LLO11] Choongbum Lee, Joonkyung Lee, and Sang-il Oum. Rank-width of random graphs. *Journal of Graph Theory*, 70(3):339–347, June 2011.

[SSL+92] Ellen Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho W. Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis : A system for sequential circuit synthesis. *CTIT technical reports series*, 1992.

[Ste11] Michael Benjamin Stepp. *Equality saturation: engineering challenges and applications*. PhD thesis, USA, 2011. AAI3482452.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 737–742, Berlin, Heidelberg, 2011. Springer-Verlag.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44(1):264–276, January 2009.

[WLCX11] Chaoyi Wang, Tian Liu, Peng Cui, and Ke Xu. A note on treewidth in random graphs. In Weifan Wang, Xuding Zhu, and Ding-Zhu Du, editors, *Combinatorial Optimization and Applications*, pages 491–499, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[YPW+21] Yichen Yang, Mangpo Phitchaya Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *ArXiv*, abs/2101.01332, 2021.