Growing Vtrees from Variable Orderings: Novel Static Heuristics for SDD Compilation*

Sander Verwimp[1] **Annegret Seibt**[1] **Vincent Derkinderen**[1]

[1]Department of Computer Science, KU Leuven, Leuven, Belgium

Abstract

Sentential decision diagrams (SDDs) are a tractable target language in knowledge compilation, with applications in probabilistic inference. The size of an SDD compiled from a Boolean formula is governed by a vtree, a binary tree over the formula's variables. Since the number of vtrees grows super-exponentially in the number of variables, heuristics are required to identify vtrees that yield compact SDDs. This paper introduces two new static vtree heuristics that accept Boolean circuits of any form, not just those in conjunctive normal form. The fan-in heuristic adapts the fan-in variable ordering technique to vtrees by traversing the input circuit depth-first and grouping sibling variables into balanced sub-vtrees. The second, interaction graph heuristic, builds a weighted graph that captures the interactions between the circuit's variables, recursively partitions it along minimal cuts, and merges the resulting partitions into a vtree. Both heuristics outperform the static baselines in terms of SDD size and compilation time. A combined variant further reduces SDD size and ranks as the strongest of the evaluated static heuristics.

1 INTRODUCTION

Sentential decision diagrams (SDDs) are a tractable circuit language studied in knowledge compilation [Darwiche, 2011]. They support a wide range of queries in time polynomial in the size of the compiled circuit, enabling applications such as exact probabilistic inference [Chavira and Darwiche, 2008, Derkinderen et al., 2024]. An SDD (in its reduced form) is canonical with respect to a *vtree*, a binary tree over the variables of the input formula. Different vtrees can yield SDDs of vastly different sizes for the same Boolean function [Xue et al., 2012], so finding a small SDD representation reduces to finding a good vtree. The vtree also affects compilation time and memory consumption, such that a poor choice can even prevent compilation from completing in practice, affecting downstream applications.

There are $\frac{(2n-2)!}{(n-1)!}$ distinct vtrees over n variables [Choi and Darwiche, 2013], a count that grows super-exponentially and rules out exhaustive search. Heuristics for selecting a vtree are classified as *static* or *dynamic*. Static heuristics fix a vtree before compilation based on structural properties of the input, while dynamic heuristics refine the vtree during compilation using intermediate SDD sizes as feedback. Dynamic methods generally produce smaller SDDs at the cost of longer compilation times. A good static heuristic can also serve as a strong starting point for dynamic optimisation.

Literature on static vtree heuristics is limited. Most heuristics require the input to be in conjunctive normal form (CNF) [Darwiche, 2011] or to be in an application-specific representation [Choi et al., 2013], which prevents their use for generic Boolean circuits. This paper introduces two new static vtree heuristics that do not have this restriction. Empirical results also show their effectiveness, outperforming the static baselines.

2 SENTENTIAL DECISION DIAGRAMS

To introduce sentential decision diagrams (SDDs), we use the intuitive description of Derkinderen et al. [2020] and copy their example (Figure 1). For a complete, formal exposition, we refer to Darwiche [2011].

An SDD represents a propositional theory. Each SDD node α is either a terminal ($\top$ or $\bot$), a literal (e.g. X or $\neg X$) or a decomposition ($(p_1 \wedge s_1) \vee \cdots \vee (p_n \wedge s_n)$), with X a variable, and p_i and s_i SDDs respectively referred to as a prime and sub of α. From a top-down perspective, the theory α of a decomposition node is partitioned into several branches such that semantically s_i is α conditioned on p_i.

*The code: `https://doi.org/10.48804/MQYNKZ`

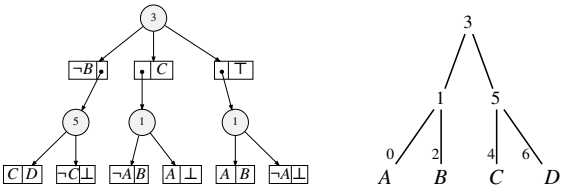

Figure 1: An SDD representing $(A \wedge B) \vee (C \wedge D) \vee (B \wedge C)$ (left) and the vtree used for its construction (right).

Definition 1 (vtree)**.** A vtree for variables $\mathbf{X}$ is a full binary tree whose leaves are in one-to-one correspondence with the variables in $\mathbf{X}$.

A vtree is used to determine the variables to condition on (i.e., those in p_i). Each decomposition node respects a node in the vtree v such that the variables in p_i are all in v^l (left vtree branch) and all variables in s_i are in v^r (right branch).

When each left vtree branch is a leaf node, the vtree is right-linear. In this case, the SDD corresponds to an ordered binary decision diagram (OBDD) [Bryant, 1986]. Such diagrams are guided by a variable ordering, which is an ordered sequence of variables and thus more simple than the hierarchical structure of a vtree. However, both our new heuristics draw inspiration from such established variable ordering heuristics.

3 VTREE HEURISTICS

A static vtree heuristic accepts as input a Boolean circuit comprising a set of *gates* ($\vee$, $\wedge$, $\neg$, `XOR`, `XNOR`, `NOR`, and `NAND`[1]), over a set of Boolean variables $\mathbf{X}$, and outputs a vtree over $\mathbf{X}$.

Three guidelines for constructing good vtrees can be distilled from the literature on static variable ordering and vtree heuristics [Chung et al., 1993, Darwiche, 2011, Choi et al., 2013]:

1. influential variables should appear earlier in the vtree,
2. variables that are topologically close or that strongly interact should be placed close together[2],
3. left (prime) branches in the vtree should be small.

3.1 FAN-IN VTREE HEURISTIC

The *fan-in vtree heuristic* adapts the fan-in variable ordering heuristic for OBDDs [Malik et al., 1988] to vtrees. It performs a depth-first traversal of the input circuit, visiting the children with the deepest sub-circuits first, breaking ties using the chronological order. At each gate, the variables that appear as direct sibling literals are collected into a group. Each variable is assigned only to the first group in which it appears. Each group is then mapped to a balanced sub-vtree, and the sub-vtrees are merged right-linearly in the order in which the groups were produced. Figure 2 illustrates this heuristic.

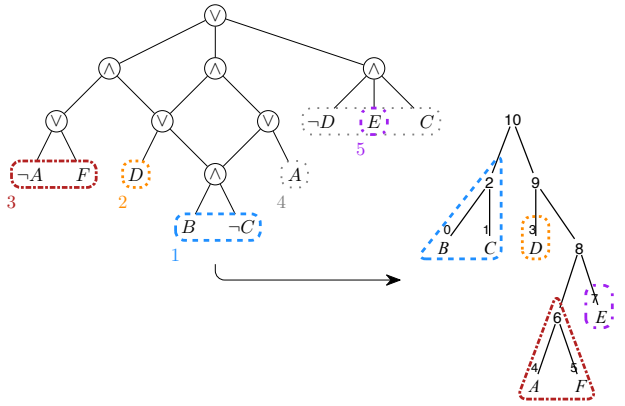

Figure 2: Example of the fan-in vtree heuristic, returning a vtree from an input circuit. This shows the visit order of the literal groups and how they are mapped to balanced sub-vtrees. Grey dotted literals are already in an earlier group.

The fan-in heuristic builds on the intuition that variables deeper in the circuit should be evaluated earlier, and are therefore placed earlier in the vtree (guideline 1). The depth-first traversal places topologically related variables next to each other in the vtree, matching guideline 2. Guideline 3 is automatically considered, as sibling literal groups are typically small: most gates have low fan-in, so the balanced sub-vtrees stay shallow. The resulting vtree is right-leaning with small prime branches.

3.2 INTERACTION GRAPH HEURISTIC

The *interaction graph* (IG) heuristic is loosely inspired by MINCE [Aloul et al., 2001], which recursively partitions a hypergraph representation of a CNF formula along minimal bisections. Lifting the CNF restriction, our IG heuristic adapts the underlying idea to generic Boolean circuits. It first constructs a graph that captures the degree of interaction between variables, then computes an influence score for each variable in the input circuit, and finally partitions the graph along minimal cuts, reorders the partitions by influence, and merges them into a vtree.

Figure 3 illustrates this heuristic on the example circuit. Each of the three steps is modular and admits multiple implementations, so the IG heuristic is best viewed as a family of heuristics rather than a single algorithm. Here, only a few possible implementations are discussed.

The first step constructs the interaction graph whose vertices are the variables and whose edge weights capture the

[1] These additional gate types are included for convenience.

[2] This depends more on the structure of the input circuit rather than the underlying Boolean function.

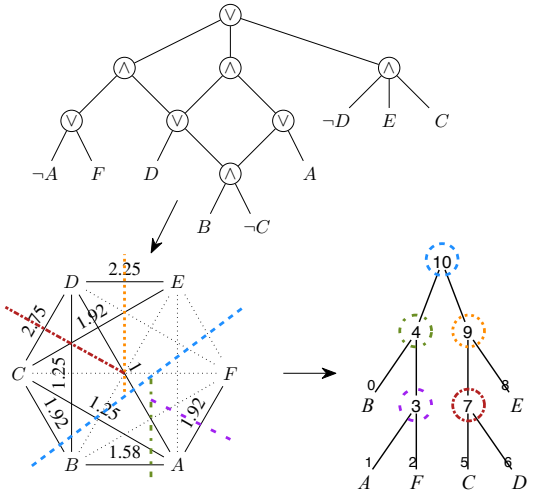

Figure 3: Example of the IG heuristic applied on an input circuit to generate a vtree. The intermediate IG is constructed using the avg-min-distance method, and split using bisections with max-depth influence reordering.

interaction strength between each pair of variables. Three construction methods are considered. The *min-distance* and *average-min-distance* methods derive weights from distances between literals in the circuit. The *sibling* method uses the same sibling literal concept as the fan-in heuristic, with edge weights that reflect how often the variable pair occurs as siblings in the circuit.

The second step assigns each variable an influence score. Two metrics are considered: *max-depth*, which is the depth of the variable's deepest literal, and *literal-count*, which is the number of times the variable occurs as a literal.

The third step recursively splits the interaction graph along minimal cuts using either the Stoer-Wagner minimum cut [Stoer and Wagner, 1997] or the Kernighan-Lin bisection algorithm [Kernighan and Lin, 1970]. By cutting along low-weight edges, both algorithms keep strongly interacting variables within the same partition, per guideline 2. A hybrid approach that combines these two algorithms and additionally bounds the size of the prime branch by a hyperparameter is also implemented, directly enforcing guideline 3. At each level of recursion, the resulting partitions are reordered by descending average influence, which places the most influential variables early in the vtree, thus satisfying guideline 1. The corresponding sub-vtrees are then merged right-linearly.

4 EXPERIMENTS

4.1 SETUP

The two heuristics are evaluated on a benchmark suite of 611 Boolean circuits drawn from four sources: 11 circuits from the ISCAS'85 benchmark set [Brglez and Fujiwara, 1985], 60 circuits obtained by randomly querying Bayesian networks from the bnlearn repository [Scutari, 2022] after translation into ProbLog programs, 270 circuits derived from randomly generated Bayesian networks, and 270 Boolean circuits generated by a custom random algorithm.

Each heuristic produces a vtree, which is then passed together with the original circuit to the PySDD compiler [Meert, 2025]. Two compilation methods are used. Static compilation keeps the vtree fixed throughout, while dynamic compilation starts from the same given vtree but enables PySDD's dynamic vtree heuristic. All experiments have a 10 GB memory limit, with a time limit of 5 minutes for the static method and 10 minutes for the dynamic one.

For each method, the results are aggregated using a *mean-log-ratio* (MLR) score. Let C_h denote the subset of benchmark circuits that a method h successfully compiles. The MLR score of h is then defined as

$$\mathrm{MLR}(h) = \frac{\sum_{c_i \in C_h} \ln \frac{h(c_i)}{\mathcal{H}(c_i)}}{|C_h|} \tag{1}$$

where $\mathcal{H}(c_i)$ is the best result on c_i across all methods under consideration. The score is always positive and a lower value signifies better performance. One drawback is that methods that compile fewer circuits can obtain artificially low MLR scores, so the fraction of successfully compiled circuits must be reported alongside any comparison. A more detailed explanation on this score is given in Appendix A. A different approach is to penalise failures as the PAR2 score does [Balint et al., 2015]. The penalised MLR score (P-MLR) charges failed compilations twice the worst observed result on the corresponding circuit.

Two baselines are used. Both are vtrees built over the naive variable ordering (the order in which variables appear in the input circuit), one with a *right-linear* shape and one with a *balanced* shape. For the interaction graph heuristic, hyperparameter optimisation selects the three configurations listed in Table 1. The configurations IG ⟨`amd`⟩ and IG ⟨`sib`⟩ limit the prime branch size to 4, in line with guideline 3, while IG ⟨`bis`⟩ does not.

IG	Interaction graph	Splitting configuration
⟨`amd`⟩	avg-min-distance	bisection-bisection(4)
⟨`sib`⟩	sibling	mincut-bisection(4)
⟨`bis`⟩	avg-min-distance	bisection

Table 1: Best IG heuristic configurations, each using the max-depth variable influence metric.

The combined heuristic EVAL′ is inspired by the EVAL heuristic [Drechsler, 2002]: it runs both IG ⟨sib⟩ and the fan-in heuristic, and reports the smaller of the two resulting SDDs. We consider both a parallel variant whose compile time is the maximum of the two underlying runs, and a sequential variant whose compile time is their sum.

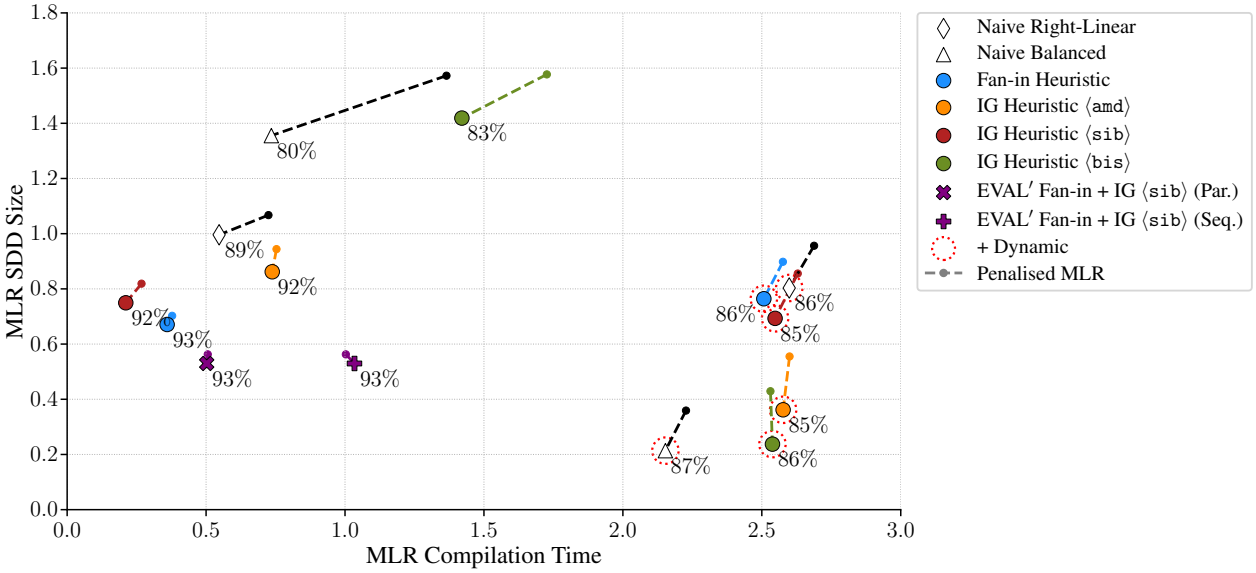

Figure 4: MLR scores of all evaluated methods under static and dynamic compilation. Each point is a method, plotted by its MLR score for compilation time (x-axis) and SDD size (y-axis), connected with a dashed line to the penalised MLR score. Lower is better on both axes. The percentage next to each point is the fraction of benchmark circuits successfully compiled.

4.2 RESULTS

Figure 4 positions all 14 methods in a single plot. For more detailed results, we refer to Table 2 in the appendix.

Static performance. Among the static methods, the fan-in heuristic and IG ⟨sib⟩ achieve the best MLR scores. The fan-in one obtains a slightly lower SDD size MLR (0.67 against 0.75 for IG ⟨sib⟩), while IG ⟨sib⟩ obtains the lowest compilation-time MLR overall (0.21 against 0.36 for fan-in). Both also compile the largest fractions of the benchmark suite among the base heuristics, at 93.29% and 92.31% of the 611 circuits, respectively. Combining the two in EVAL′ pushes the SDD size MLR down further to 0.53, the best static score among the evaluated methods. However, this comes at a slight increase in compilation time.

A broader pattern emerges across the static results. Heuristics that produce *right-leaning* vtrees consistently outperform those that produce balanced ones. For static compilation, this confirms guideline 3, which states that vtrees with small prime branches, like right-leaning vtrees, perform better.

Dynamic performance. The conclusion changes when the new static heuristics are used as starting points for dynamic compilation: the balanced vtrees perform better than the right-leaning vtrees. The strongest performer among the new heuristics is IG ⟨bis⟩, the only one producing balanced vtrees. It approaches the naive balanced baseline on SDD size (0.24 against 0.21) but has a higher compilation-time MLR (2.54 against 2.15). None of the other new heuristics improves on this, so none offers a benefit over the naive balanced vtree as a starting point for dynamic compilation.

Choice of method. The Pareto front in Figure 4 runs from IG ⟨sib⟩ at the faster-but-larger end to the naive balanced vtree with dynamic minimisation at the slower-but-smaller end, with EVAL′ and the fan-in heuristic between the two extremes. The choice of method therefore depends on the available compilation-time budget. When SDD size is the priority and time is abundant, dynamic minimisation from a balanced vtree is the best option. When compilation time is constrained, EVAL′, the fan-in heuristic, and IG ⟨sib⟩ are progressively faster alternatives, at the cost of larger SDDs.

5 CONCLUSION

We introduced two new static vtree heuristics for SDD compilation that accept as input Boolean circuits of any form. The fan-in vtree heuristic adapts a classical OBDD ordering technique to vtrees by traversing the circuit depth-first and grouping sibling variables into balanced sub-vtrees. The interaction graph heuristic instead constructs a weighted graph over the circuit variables, recursively partitions it along minimal cuts, and then reorders and merges the partitions into a vtree. Both heuristics outperform the baselines in static compilation, and the combined variant EVAL′ ranks as the strongest static heuristic in the comparison. When paired with dynamic minimisation, a balanced approach remains preferred. We hypothesize this allows the dynamic search to explore the vtree space more efficiently.

Acknowledgements

This project was funded by the European Union (ERC, DeepLog, 101142702). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. This work was also partially supported by the Internal Funds KU Leuven iBOF/21/075. The authors thank Luc De Raedt for his input and feedback.

References

F.A. Aloul, I.L. Markov, and K.A. Sakallah. Faster SAT and smaller BDDs via common function structure. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 443–448, 2001. doi: 10.1109/ICCAD.2001.968669.

Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. Overview and analysis of the sat challenge 2012 solver competition. *Artificial Intelligence*, 223:120–155, 2015. ISSN 0004-3702.

Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a targeted translator in FORTRAN. 06 1985.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8): 677–691, 1986.

Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7): 772–799, 2008.

Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, page 187–194. AAAI Press, 2013.

Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, ECSQARU'13, page 121–132, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642390906. doi: 10.1007/978-3-642-39091-3_11.

P.-Y. Chung, I.M. Hajj, and J.H. Patel. Efficient variable ordering heuristics for shared ROBDD. In *1993 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1690–1693 vol.3, 1993. doi: 10.1109/ISCAS.1993.394067.

Adnan Darwiche. SDD: a new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, pages 819–826, 2011.

Vincent Derkinderen, Evert Heylen, Pedro Zuidberg Dos Martires, Samuel Kolb, and Luc De Raedt. Ordering variables for weighted model integration. In *UAI*, Proceedings of Machine Learning Research, pages 879–888. AUAI Press, 2020.

Vincent Derkinderen, Robin Manhaeve, Pedro Zuidberg Dos Martires, and Luc De Raedt. Semirings for probabilistic and neuro-symbolic logic programming. *Int. J. Approx. Reason.*, 171:109130, 2024.

R. Drechsler. Evaluation of static variable ordering heuristics for MDD construction [multi-valued decision diagrams]. In *Proceedings 32nd IEEE International Symposium on Multiple-Valued Logic*, pages 254–260, 2002. doi: 10.1109/ISMVL.2002.1011096.

B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970. doi: 10.1002/j.1538-7305.1970.tb01770.x.

S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *[1988] IEEE International Conference on Computer-Aided Design (ICCAD-89) Digest of Technical Papers*, pages 6–9, 1988. doi: 10.1109/ICCAD.1988.122451.

Wannes Meert. The PySDD package. `https://pysdd.readthedocs.io/`, 11 2025. Version: 1.0.6.

Marco Scutari. bnlearn - an R package for bayesian network learning and inference. `https://www.bnlearn.com/bnrepository/`, 11 2022. Accessed: 28th March 2026.

Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, July 1997. ISSN 0004-5411. doi: 10.1145/263867.263872.

Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. *Proceedings of the National Conference on Artificial Intelligence*, 1: 842–849, 01 2012. doi: 10.1609/aaai.v26i1.8221.

Growing Vtrees from Variable Orderings: Novel Static Heuristics for SDD Compilation (Supplementary Material)

Sander Verwimp[1] **Annegret Seibt**[1] **Vincent Derkinderen**[1]

[1]Department of Computer Science, KU Leuven, Leuven, Belgium

A THE MLR SCORE

The MLR score aggregates a per-circuit comparison into a single number. For two methods x and y, a natural per-circuit comparison is the ratio $y(c_i)/x(c_i)$, which is above 1 when x is smaller (better) on circuit c_i and below 1 when y is smaller (better). Taking the logarithm of this ratio makes the comparison symmetric around 0 rather than around 1, so that swapping x and y only flips the sign. Averaging this log-ratio over a set of circuits gives the pairwise score

$$\text{mlr}(x, y) = \frac{\sum_{c_i \in C_x \cap C_y} \ln \frac{y(c_i)}{x(c_i)}}{|C_x \cap C_y|}, \tag{2}$$

where C_x and C_y denote the circuits successfully compiled by x and y respectively. A positive value indicates that x performs better overall, and a negative value indicates the opposite. Unlike a simple count of circuits where one method beats the other, this pairwise score also accounts for the size of the difference.

Comparing every pair of methods directly in this way becomes cumbersome once more than a handful of methods are involved. The MLR score in Equation 1 avoids this by scoring each method h against a hypothetical optimal method $\mathcal{H}$ instead, where $\mathcal{H}(c_i)$ is the best result achieved by any method under consideration on circuit c_i. Since $\mathcal{H}(c_i) \leq h(c_i)$ by construction, $\text{MLR}(h) \geq 0$ for every method, and a lower score means the method is closer to the optimum. Provided that all methods under consideration successfully compile a sufficiently large and overlapping fraction of the benchmark circuits, the difference between two MLR scores approximates their pairwise mlr score:

$$\text{MLR}(y) - \text{MLR}(x) \approx \text{mlr}(x, y). \tag{3}$$

This means the methods in Figure 4 can be compared pairwise by simply reading off the difference between their MLR scores, rather than computing a separate pairwise score for each pair. This approximation degrades when a method compiles only a small and unrepresentative subset of circuits, which is why the fraction of successfully compiled circuits is reported alongside every MLR score in Table 2.

*The code: `https://doi.org/10.48804/MQYNKZ`

B EXPERIMENTAL RESULTS

Method	Success (%)	Timeout (%)	OOM (%)	Equal to $\mathcal{H}$ (%) Size	Time	MLR Size	Time	P-MLR Size	Time
Naive Balanced	*80.03*	*18.49*	1.47	*0.82*	**30.77**	1.36	0.73	1.57	1.37
Naive Right-linear	89.20	5.40	5.40	14.08	12.60	1.00	0.55	1.07	0.72
Fan-in Heuristic	**93.29**	**4.91**	1.80	4.42	**21.77**	0.67	**0.36**	0.70	**0.38**
IG Heuristic ⟨`amd`⟩	91.65	**0.65**	*7.69*	2.45	3.60	0.86	0.74	0.94	0.75
IG Heuristic ⟨`sib`⟩	92.31	**1.80**	5.89	2.78	**23.08**	0.75	**0.21**	0.82	**0.27**
IG Heuristic ⟨`bis`⟩	82.65	9.82	7.53	1.15	*0.00*	*1.42*	1.42	*1.58*	1.73
EVAL′ (Par.)	**93.45**	5.07	1.47	7.20	0.33	0.53	**0.50**	0.56	**0.51**
EVAL′ (Seq.)	**93.45**	5.07	1.47	7.20	*0.00*	0.53	1.03	0.56	1.00
Naive Balanced (Dyn.)	87.40	12.44	**0.16**	**21.77**	1.15	**0.21**	2.15	**0.36**	2.23
Naive Right-linear (Dyn.)	85.60	14.40	**0.00**	7.53	0.33	0.80	*2.60*	0.96	*2.69*
Fan-in Heuristic (Dyn.)	85.76	14.24	**0.00**	7.36	*0.00*	0.76	2.51	0.90	2.58
IG Heuristic ⟨`amd`⟩ (Dyn.)	84.94	7.86	7.20	**17.02**	*0.00*	**0.36**	2.58	**0.56**	2.60
IG Heuristic ⟨`sib`⟩ (Dyn.)	85.43	10.15	4.42	11.62	*0.00*	0.69	2.55	0.86	2.63
IG Heuristic ⟨`bis`⟩ (Dyn.)	85.76	7.04	7.20	**20.95**	0.65	**0.24**	2.54	**0.43**	2.53

Table 2: Main metrics for all evaluated methods. The best three values in every column are bold and the worst is italicised. Columns report the fraction of circuits compiled successfully, the fractions of failures by timeout and out-of-memory error, the fraction of circuits on which the method matched the hypothetical optimum $\mathcal{H}$, and the MLR and penalised MLR scores for SDD size and compilation time.