A Comparison of SAT Encodings for Acyclicity of Directed Graphs

Neng-Fa Zhou
The City University of New York, NY, USA
Relational AI, Berkeley, CA, USA

Ruiwei Wang
National University of Singapore, Singapore

Roland H. C. Yap
National University of Singapore, Singapore

Abstract

Many practical applications require synthesizing directed graphs that satisfy the acyclic constraint along with some side constraints. Several methods have been devised for encoding acyclicity of directed graphs into SAT, each of which is based on a cycle-detecting algorithm. The leaf-elimination encoding (LEE) repeatedly eliminates leaves from the graph, and judges the graph to be acyclic if the graph becomes empty at a certain time. The vertex-elimination encoding (VEE) exploits the property that the cyclicity of the resulting graph produced by the vertex-elimination operation entails the cyclicity of the original graph. While VEE is significantly smaller than the transitive-closure encoding for sparse graphs, it generates prohibitively large encodings for large dense graphs. This paper reports on a comparison study of four SAT encodings for acyclicity of directed graphs, namely, LEE using unary encoding for time variables (LEE-u), LEE using binary encoding for time variables (LEE-b), VEE, and a hybrid encoding which combines LEE-b and VEE. The results show that the hybrid encoding significantly outperforms the others.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning; Computing methodologies → Planning and scheduling; Hardware → Theorem proving and SAT solving

Keywords and phrases Graph constraints, Acyclic constraint, SAT encoding, Graph Synthesis

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.30

Funding R. Wang and R. H. C. Yap's work was partly suported by NUS grant T1 251RES2219.

1 Introduction

Many practical combinatorial problems require synthesizing acyclic directed graphs, such as utility networks, planning problems [9], Markov networks [2], neural networks, and Bayesian networks [1, 8]. As there are side constraints involved beyond acyclic constraints, these problems are more complicated than simply checking if a synthesized subgraph is acyclic. In this paper, we focus on how to solve such problems with SAT solvers by encoding the acyclic constraint into SAT. While efficient algorithms exist for checking the acyclicity of directed graphs, it is still unknown which method performs well for encoding acyclicity of directed graphs into SAT.

Several methods have been devised for encoding acyclicity of directed graphs into SAT, each of which is based on a cycle-detecting algorithm. The leaf-elimination encoding (LEE), which is basically the same as the tree-reduction encoding [4] and the binary labeling encoding [6], is inspired by the leaf-elimination algorithm. The algorithm repeatedly eliminates leaves from the graph, and judges the graph to be acyclic if the graph becomes empty at a certain time.

Another approach is the transitive-closure encoding for directed graphs, which relies on the proposition that a graph's transitive closure preserves the graph's acyclicity. However the transitive closure encoding can be prohibitively large. For that reason, GraphSAT, which is an SMT solver integrating acyclicity checking into SAT solving, has been proposed [4]. Recently, the vertex-elimination encoding (VEE) [10] was proposed, which exploits the property that the cyclicity of the resulting graph produced by the vertex-elimination operation [11] entails the cyclicity of the original graph. VEE can be significantly smaller than the transitive-closure encoding for sparse graphs. However, for dense graphs, it is asymptotically the same as the transitive-closure encoding.

This paper first presents a comparison study of three SAT encodings for acyclicity of directed graphs, namely, LEE using unary encoding for time variables (LEE-u), LEE using binary encoding for time variables (LEE-b), and VEE. This paper then proposes a hybrid encoding, which combines the strengths of LEE-b and VEE. The hybrid encoding starts with VEE, and switches to LEE-b when the resulting graph after vertex elimination becomes dense. The experimental results show that the hybrid encoding significantly outperforms the others.

2 Preliminaries

The constraint $acyclic\_d(V, E)$ takes a base directed graph $G = (V, E)$, where V is a set of vertices, and E is a set of directed edges. The task is to synthesize a acyclic subgraph of G. More precisely, each vertex in V has a binary variable, called a *characteristic variable*, associated with it, which is 1 iff the vertex is in the subgraph to be synthesized. Each edge in E also has an associated characteristic variable, which indicates if the edge is in the subgraph. The constraint $acyclic\_d(V, E)$ is true if the subgraph of G determined by the characteristic variables is acyclic.

In the following, we use the notation (u, v) to denote a directed edge from vertex u to vertex v. The function $b(x)$ returns the characteristic variable of x. A vertex v is called an *in-vertex* if $b(v) = 1$, and an edge $e = (v_1, v_2)$ is called an *in-edge* if $b(e) = 1$. For each edge $e = (v_1, v_2)$ in E, $b(e) \rightarrow b(v_1) \wedge b(v_2)$, meaning that if an edge is in the subgraph, then both end vertices connected by the edge must be in the subgraph as well.

For a directed graph $G = (V, E)$, the function $nbs^-(v)$ returns the set of *in-neighbors* connected to v by incoming edges in the base graph:

$$nbs^-(v) = \{u \mid (u, v) \in E\}$$

and the function $nbs^+(v)$ returns the set of *out-neighbors* connected to v by outgoing edges in the base graph:

$$nbs^+(v) = \{u \mid (v, u) \in E\}.$$

A vertex v is said to be *peripheral* if either it has no in-neighbors or it has no out-neighbors. In particular, a peripheral vertex v is called a *leaf* in this paper if it has no out-neighbors. Note that singleton vertices that are not connected to any other vertices are treated as leaves.

3 The Leaf-Elimination Encoding (LEE)

Given a directed graph, the leaf-elimination algorithm detects cycles by repeatedly eliminating leaves from the graph from time 0 to a certain maximum time. If the graph is empty at the maximum time, then the graph is acyclic; otherwise, the graph is cyclic.

The maximum time is determined by the longest path in the graph. As the graph that comprises a list of linearly connected vertices has the longest path, the maximum time is upper bounded by n, where $n = |V|$. While finding the longest path in a graph is NP-hard, a tight upper bound can be obtained in some cases.

There is a straightforward CSP (Constraint Satisfaction Problem) model for the acyclicity constraint, which mimics the leaf-elimination algorithm for detecting cycles. For each vertex, a variable, called a *time variable*, is utilized to indicate the time at which the vertex becomes a leaf and is removed from the graph. The domain of the time variables is $0..m$, where m is the maximum time. The constraints ensure that only leaves can be removed at each time, and the graph must be empty at time m. If the graph is cyclic, then the CSP model is unsatisfiable. The encodings of the CSP model into SAT are called *leaf-elimination encodings* (LEE). Different methods can be utilized to encode time variables, such as unary encoding (also called direct encoding [3]) and binary encoding (also called log encoding [5]). When binary encoding is used for time variables, LEE, called LEE-b, is compact and can encode large graphs. However, it is well known that binary encoding has weak propagation strength [7], yet, the results in Section 6 show that the binary encoding pays off.

3.1 LEE-u

LEE-u (similar to tree reduction in [10]) employs a matrix of binary variables A of size $n \times m$, where $n = |V|$, the number of vertices in the base graph, and m is the maximum time. The entry $A_{v,t}$ is 1 if and only if vertex v has been eliminated by time t. The encoding imposes the following constraints on the variables:

$$\text{For } v \in V\text{: } \textstyle\sum_{u \in nbs^+(v)} b((v,u)) = 0 \leftrightarrow A_{v,0} \tag{1}$$

$$\text{For } v \in V,\ t \in 1..m\text{: } A_{v,t-1} \rightarrow A_{v,t} \tag{2}$$

$$\text{For } v \in V,\ t \in 1..m,\ u \in nbs^+(v)\text{: } b((v,u)) \wedge \neg A_{u,t-1} \rightarrow \neg A_{v,t} \tag{3}$$

$$\text{For } v \in V\text{: } A_{v,m} \tag{4}$$

Constraint (1) states that all leaves, i.e., vertices that have no outgoing edges, are eliminated at time 0. Constraint (2) enforces that once a vertex is eliminated, it is eliminated forever. Constraint (3) entails that a vertex cannot be eliminated at time t if any of its out-neighbors is not eliminated at time $t-1$. Constraint (4) forces every vertex to be eliminated at time m.

The correctness of the encoding is supported by the fact that the vertices that occur in a cycle can never be eliminated, i.e. $A_{v,t}$ cannot be equal to 1 for $t \in 1 \dots m$, and constraints (3) and (4) are unsatisfiable for any vertex in a cycle.

The number of variables, besides the characteristic variables, used by LEE-u is the size of A, which is $O(n \times m)$. The number of clauses used by LEE-u, which is dominated by Constraint (3), is $O(n \times m \times d)$, where d is the average degree of the vertices in the base graph.

3.2 LEE-b

LEE-b is a variant of LEE, which encodes time variables using binary encoding. Binary encoding is more compact than unary encoding. It employs a sequence of binary variables for encoding a domain variable. Each combination of values of the binary variables represents a value for the domain variable. If there are holes in the domain, then not-equal constraints are

generated to disallow assigning those hole values to the variable. Also, inequality constraints ($\geq$ and $\leq$) are generated to prohibit assigning out-of-bounds values to the variable if either bound is not $2^k - 1$ for some k. A detailed description of general techniques for binary encodings for domain variables and primitive constraints can be found in [13].

LEE-b uses a variable T_v with the domain $0..m$ for each vertex v in V, where m is the maximum time. LEE-b imposes the following constraints on time variables:

$$\text{For } v \in V\text{: } \textstyle\sum_{u \in nbs^+(v)} b((v,u)) = 0 \leftrightarrow T_v = 0 \tag{5}$$

$$\text{For } v \in V,\ u \in nbs^+(v)\text{: } b((v,u)) \rightarrow T_v > T_u \tag{6}$$

Constraint (5) states that $T_v = 0$ if and only if vertex v is a leaf. Constraint (6) entails that for each directed edge (v,u), $T_v > T_u$, meaning that vertex v is removed after vertex u. The encoding of $T_v > T_u$ can be found in [12].

The correctness of the encoding is supported by the fact that the constraint $T_v > T_u$ is not commutative and constraint (6) is unsatisfiable if u and v occur in a cycle.

The number of variables, besides the characteristic variables, used by LEE-b is $O(n \times log_2(m))$. The number of clauses used by LEE-b is $O(n \times log_2(m) \times d)$, where d is the average degree of the vertices in the base graph.

4 Vertex-Elimination Encoding

The vertex-elimination encoding (VEE) [10] exploits the fact that the sequence of graphs produced by the vertex elimination operation [11] with respect to an elimination ordering preserves the acyclicity of the original graph.

Let O be an elimination ordering, $O = [v_1, v_2, \ldots, v_n]$, and G_0 be the original directed graph, $G_0 = ([v_1, v_2, \ldots, v_n], E_0)$. It is assumed that there are no vertices with self-loops in G_0. VEE produces a sequence of graphs G_1, G_2, …, G_n by eliminating vertices according to the elimination ordering. The *vertex-elimination graph* is the union of the graphs $G^* = G_0 \bigcup G_1 \ldots \bigcup G_n$. Let $G_{i-1} = ([v_i, \ldots, v_n], E_{i-1})$. The graph $G_i = ([v_{i+1}, \ldots, v_n], E_i)$ is obtained by eliminating v_i from G_{i-1}, where

$$\begin{aligned} E_i = E_{i-1} - \\ \{(u, v_i) | u \in nbs^-(v_i)\} - \\ \{(v_i, u) | u \in nbs^+(v_i)\} + \\ \{(u, w) | u \in nbs^-(v_i),\ w \in nbs^+(v_i), u \neq w\}. \end{aligned}$$

The operation eliminates v_i's adjacent edges, and adds the edge (u,w) into E_i for each u in v_i's in-neighbors and each w in v_i's out-neighbors if $u \neq w$ and the edge is not contained in E_i. Each newly added edge (u,w) is attached with a characteristic binary variable $b((u,w))$. In addition, for all u in $nbs^-(v_i)$ and w in $nbs^+(v_i)$ such that $u \neq w$, the variable $b((u,w))$ is entailed by the variables $b((u,v_i))$ and $b((v_i,w))$:

$$b((u,v_i)) \ \wedge \ b((v_i,w)) \ \rightarrow \ b((u,w)). \tag{7}$$

For each u in $nbs^-(v_i)$, if $u \in nbs^+(v_i)$, VEE, after eliminating v_i, generates the following constraint to ensure that there is no cycle between u and v_i in E_{i-1}

$$\neg b((u,v_i)) \ \vee \ \neg b((v_i,u)). \tag{8}$$

Constraints (7) and (8) ensure that the acyclicity of E_i entails the acyclicity of E_{i-1}. Therefore, with the accumulated constraints, E_0 is acyclic if E_i is acyclic by induction on i.

The correctness of VEE is guaranteed by the fact that a cycle of any length in G_0 will lead to a cycle of size 2 in G_{i-1} ($i \in 1..n$), which will make constraint (8) unsatisfiable.

The number of binary variables used by VEE, which is upper bounded by $O(n^2)$, is determined by the number of edges in the original graph and the number of new edges added during the vertex elimination process. The number of clauses generated by VEE is $O(d^2 \times n)$, where $n = |V|$ and d is the average degree of the vertex-elimination graph. In comparison, the transitive closure encoding uses $O(n^2)$ variables and generates $O(n^3)$ clauses [10]. While VEE is significantly more compact than the transitive clause encoding for sparse graphs, it is asymptotically the same as the transitive-closure encoding in the worst case.

5 Hybrid Encoding

The resulting graph obtained after each vertex elimination tends to be more dense than the original graph, and VEE becomes closer to the transitive-closure encoding. Thus, the encoding may become prohibitively large. One idea to alleviate code explosion while harnessing the propagation strengths of VEE is to start with VEE when the graph is sparse, and switch to LEE-b when the graph becomes dense. We call this encoding that combines VEE and LEE-b a *hybrid encoding*.

Formally, given any elimination ordering $[v_1, v_2, \ldots, v_n]$ and a switch position $0 \leq i \leq n$, the hybrid encoding uses the constraints (7) and (8) of VEE to generate the graph G_i by eliminating the vertices $v_1, \cdots, v_i$ from the original graph G_0. Then constraints (5) and (6) of LEE-b are used to encode the acyclicity of G_i.

The correctness of a hybrid encoding is guaranteed by the correctness of VEE and LEE-b. For any $i \in 0 \ldots n$, constraints (7) and (8) ensure that if G_i is acyclic then G_0 is also acyclic. If LEE ensures that G_i is acyclic, and VEE ensures that there are no cycles in G_1, G_2, …, G_{i-1}, then the hybrid encoding also ensures that the graph G_0 is acyclic. The encoding size depends on the encoding sizes of VEE and LEE-b, as well as the switching heuristic.

A concrete hybrid encoding needs to decide when to switch to LEE-b. In our implementation, the hybrid encoding uses the *mindegree* heuristic (selecting a vertex with the smallest total number of incoming and outgoing edges in the graph generated by the vertex elimination). It switches to LEE-b if the current graph $G^c = G_0 \bigcup G_1 \cdots \bigcup G_i$ contains 2.3 times as many edges as the original graph G_0 or the current graph G^c contains more than $30 \times n$ edges based on preliminary experiments, where n is the number of vertices in the original graph G_0.

6 Experimental Results

All the encodings discussed above have been implemented in Picat[1] (version 3.4). Picat provides a state-of-art SAT-based CSP solver. For example, it won two gold medals in the 2022 XCSP solver competition[2] and two silver medals in the 2022 MiniZinc Challenge.[3] Picat encodes constraints into SAT and employs Kissat[4] as the underlying SAT solver.

This section presents the results of an experiment comparing the encodings on the GraphSAT benchmarks.[5] The benchmarks consist of five categories of instances with graph sizes ranging from 15 to 10002 vertices. The GraphSAT benchmarks are modelled with

[1] http://picat-lang.org
[2] https://www.xcsp.org/competitions/
[3] https://www.minizinc.org/challenge2022/results2022.html
[4] https://github.com/arminbiere/kissat
[5] https://users.aalto.fi/~rintanj1/software.html

Table 1 Summary of results.

#Insts	Benchmark	LEE-u	LEE-b	VEE	HYB	Virtual-HYB
26	COMB	28.03	42.54	**0.13**	**0.13**	**0.13**
31	EMPTYCORNER	160.89	1.99	3.84	1.99	**1.51**
36	EMPTYMIDDLE	67.01	6.93	0.92	2.02	**0.87**
9	ESCAPE	379.45	22.74	285.91	23.64	**15.38**
14	ROOMCHAIN	343.35	342.99	15.38	15.34	**11.94**
	TOTAL	16350.64	6423.84	2944.2	565.29	**387.09**
116	**AVE**	140.95	55.38	25.38	4.87	**3.34**
	#SOLVED	94	108	112	**116**	**116**

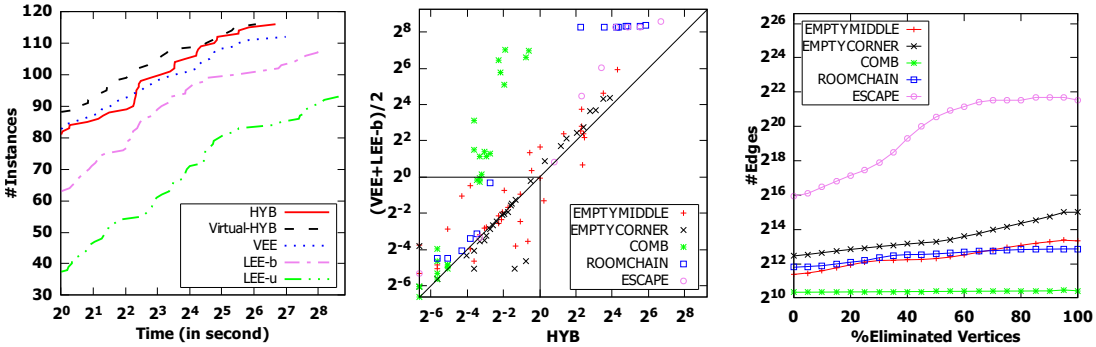
conjunctive-normal-form clauses and some graph constraints encoded as acyclic constraints. These benchmarks are introduced by [9] and have been also used to evaluate implementations of the acyclic constraint in [10]. All the CPU times reported below were measured on Linux Ubuntu with a 3.20GHz and 16G RAM Intel i7-8700 machine. The time limit used was 10 minutes per instance.

Table 1 gives a summary of the experimental results, where the column **#Insts** indicates the number of instances, the column **Benchmark** indicates the benchmark category, and each of the remaining columns indicates the average CPU time taken by an encoding. **Virtual-HYB** is the "virtual best" encoding among 21 different hybrid encodings, each of which uses VEE to eliminate $p \in \{0, 5, \cdots, 100\}$ percent of vertices and then switches to LEE-b. Note that the hybrid encoding with $p = 0$ is equivalent to **LEE-b**, and the hybrid encoding with $p = 100$ is **VEE**. The row **Total** gives the total CPU time for all instances, **AVE** the average CPU time per instance, and **#Solved** the number of solved instances within the time limit. In the experiments, the maximum time m used in **LEE** is n.[6]

It can be seen that, among the four encodings, **HYB** performs the best. **HYB** succeeds on all of the 116 instances, while **LEE-u**, **LEE-b**, and **VEE**, respectively, fail on 22, 8 and 4 instances. It can also be seen that while both **LEE-u** and **LEE-b** are based on the idea of leaf elimination, **LEE-b** is much better than **LEE-u**. On average, **HYB** is 5 times as fast as **VEE**, and 10 times as fast as **LEE-b**. **HYB** is also more robust than **VEE** and **LEE-b**. For each category, **HYB** has similar performance to the best of **LEE-u**, **LEE-b**, and **VEE**. **Virtual-HYB** performs the best on each category, but it is intended as a comparison with a form of virtual best heuristic. The results of **Virtual-HYB** entail that there is potential to improve the hybrid encoding by giving a better heuristic to decide when to switch from **VEE** to **LEE-b**.

Figure 1a gives the runtime distribution of the five encodings. **Virtual-HYB** overall outperforms the other encodings, and **HYB** always solves more instances than **VEE**, **LEE-b** and **LEE-u** when the solving time limit is set to more than 6 seconds. Also as the solving time increases, **HYB** gets close to **Virtual-HYB**. Figure 1b compares the solving time of **HYB** with (**VEE** + **LEE-b**)/2, the average CPU time of **VEE** and **LEE-b**. Each dot in the figure denotes an instance. It can be seen that **HYB** is faster than the average of **VEE** and **LEE-b** on most non-trivial instances (i.e. the instances not solved in 1 second by either **VEE** or **LEE-b**). This illustrates that **HYB** encoding is robust. The performance of **HYB** can be closer to the best between **VEE** and **LEE-b**. Figure 1c shows the average number of

[6] We experimented with optimizing for some special cases, such as removing peripheral vertices, but the difference was negligible.



(a) Runtime distribution. **(b)** The robustness of HYB. **(c)** The average number of edges.

Figure 1 Detailed comparisons between the encodings.

Table 2 A comparison on encoding sizes.

Benchmark	|V|	|E|	LEE-u		LEE-b		VEE		HYB	
			#vars	#cls	#vars	#cls	#vars	#cls	#vars	#cls
COMB005-3	134	1215	24473	208373	13559	86741	4241	13483	4241	13483
COMB010-3	389	3595	170363	1599113	43319	276548	12306	39618	12306	39618
EMPTYCORNER024-1	578	2877	347359	1697K	37951	164478	18990	105021	35753	209184
EMPTYCORNER075-1	5252	26247	fail	fail	427631	1945K	255548	4491K	459127	2680K
EMPTYMIDDLE025-2	1252	10610	fail	fail	155233	878838	75075	883099	159303	1071K
EMPTYMIDDLE030-1	902	4497	834330	4110K	59490	260162	32174	224943	61454	359879
escape08-1	4098	89688	fail	fail	1269K	8047K	fail	fail	1618K	11180K
escape10-1	10002	229252	fail	fail	3675K	23235K	fail	fail	4569K	30856K
ROOMCHAIN005-2	510	4223	282256	2385K	52076	295302	19328	76752	19328	76752
ROOMCHAIN006-3	917	11025	901673	10932K	147290	866044	49244	253186	49244	253186

edges in the vertex-elimination graph of each category generated by using **VEE** to eliminate $p \in \{0, 5, \cdots, 100\}$ percent of vertices. We can see that **VEE** is much faster than **LEE-b** on the categories where the graph size is small and grows slowly, such as the COMB and ROOMCHAIN categories.

Table 2 gives the graph sizes of 10 selected instances and the encoding sizes. The column **|V|** gives the number of vertices, and **|E|** gives the number of edges in the base graph. For each encoding, the table gives the number of variables (#vars) and the number of clauses (#cls) in the generated code. The entry *fail* indicates that the encoder fails to finish generating the encoding, i.e. the encoder runs out of memory or time. **LEE-u** fails on 4 of the instances, **VEE** fails on 2 of the instances, while **LEE-u** and **HYB** succeed on all the instances. Naturally for failed encodings, the SAT solver is never invoked. The results also show that **LEE-u** produces the largest encodings, and for the EMPTYMIDDLE instances the encoder runs out of memory. This may explain why the performance of **LEE-u** is also the worst.

Table 3 gives the CPU time of each run, which includes both the translation time and solving time. The entry *T.O.* indicates that the run does not finish within the time limit. **HYB** succeeds in solving all the 10 instances, while each of the other encoders fails to solve some of the instances. In addition, **HYB** is faster than both **VEE** and **LEE-b** on some instances, such as the EMPTYCORNER075-1 instance. **LEE-b** is significantly faster than **VEE** on instances where **VEE** fails, such as escape08-01.

Table 3 A comparison on CPU times.

Benchmark	LEE-u	LEE-b	VEE	HYB
COMB005-3	7.21	17.36	**0 .08**	**0.08**
COMB010-3	115.17	195.78	**0.59**	**0.59**
EMPTYCORNER024-1	26.86	**0.3**	0.42	0.35
EMPTYCORNER075-1	fail	14.52	24.58	**11.5**
EMPTYMIDDLE025-2	fail	43.15	**5.66**	11.49
EMPTYMIDDLE030-1	26.44	0.84	1.05	**1.03**
escape08-1	fail	**15.96**	fail	29.04
escape10-1	fail	150.98	fail	**101.84**
ROOMCHAIN005-2	T.O.	T.O.	**45.55**	45.79
ROOMCHAIN006-3	T.O.	T.O.	**11.56**	11.61

7 Discussion and Conclusion

This paper compares several SAT encodings for the acyclic constraint on directed graphs. For LEE, this paper compares two encodings of time variables, and shows that the decision on which encoding to select has a great impact on the performance. While LEE-b is compact, it fails on some mid-sized instances due to its weak propagation caused by the binary representation of time variables. Our study also confirms that, while VEE is effective for sparse mid-sized graphs, it generates prohibitively large encodings for larger graphs even when the graphs are sparse.

Most importantly, our study finds that the hybrid encoding, which starts with VEE and switches to LEE-b when the graph becomes dense, significantly outperforms both LEE-b and VEE. The good performance of the hybrid encoding is attributed to its combination of VEE's strong propagation and LEE-b's conciseness. The hybrid encoding clearly advances the state of the art. Ultimately, to solve more difficult problems, scalability is needed. The hybrid encoding will be a new starting point for future researchers and practitioners.

As the idea of hybridization is new in this context, it warrants more investigation. For example, further work needs to be done to find even better heuristics for switching from VEE to LEE-b. Also, the idea of hybridization may also be effective for encoding other graph constraints, such as the reachability constraint.

References

1 Jeremias Berg, Matti Järvisalo, and Brandon Malone. Learning Optimal Bounded Treewidth Bayesian Networks via Maximum Satisfiability. In Samuel Kaski and Jukka Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 86–95, Reykjavik, Iceland, 22–25 April 2014. PMLR. URL: `https://proceedings.mlr.press/v33/berg14.html`.

2 Jukka Corander, Tomi Janhunen, Jussi Rintanen, Henrik J. Nyman, and Johan Pensar. Learning chordal markov networks by constraint satisfaction. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 1349–1357, 2013. URL: `https://proceedings.neurips.cc/paper/2013/hash/c06d06da9666a219db15cf575aff2824-Abstract.html`.

3 Johan de Kleer. A comparison of ATMS and CSP techniques. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, pages 290–296. Morgan Kaufmann, 1989. URL: `http://ijcai.org/Proceedings/89-1/Papers/046.pdf`.

4 Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence – 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2014. doi:10.1007/978-3-319-11558-0_10.

5 Kazuo Iwama and Shuichi Miyazaki. Sat-varible complexity of hard combinatorial problems. In Björn Pehrson and Imre Simon, editors, *Technology and Foundations – Information Processing '94, Volume 1, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August – 2 September, 1994*, volume A-51 of *IFIP Transactions*, pages 253–258. North-Holland, 1994.

6 Mikolas Janota, Radu Grigore, and Vasco M. Manquinho. On the quest for an acyclic graph. *CoRR*, abs/1708.01745, 2017. arXiv:1708.01745.

7 Donald Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.

8 Zhenyu A. Liao, Charupriya Sharma, James Cussens, and Peter van Beek. Finding all bayesian network structures within a factor of optimal. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 – February 1, 2019*, pages 7892–7899. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33017892.

9 Binda Pandey and Jussi Rintanen. Planning for partial observability by SAT and graph constraints. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 190–198. AAAI Press, 2018. URL: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17756.

10 Masood Feyzbakhsh Rankooh and Jussi Rintanen. Propositional encodings of acyclicity and reachability by using vertex elimination. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 – March 1, 2022*, pages 5861–5868. AAAI Press, 2022. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20530.

11 Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976. doi:10.1137/0205021.

12 Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In Marco Gavanelli and John H. Reppy, editors, *Practical Aspects of Declarative Languages – 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*, volume 9585 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2016. doi:10.1007/978-3-319-28228-2_4.

13 Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, pages 671–686, 2017. doi:10.1007/978-3-319-66158-2_43.