

# Notes on Validated Model Counting

## Version of September 16, 2022

Randal E. Bryant

Computer Science Department  
Carnegie Mellon University, Pittsburgh, PA, United States

### 1 Notation

Let  $X = \{x_1, x_2, \dots, x_n\}$  be a set of Boolean variables. An *assignment* is a function  $\alpha$  assigning Boolean values to the variables:  $\alpha : X \rightarrow \{0, 1\}$ . We can also view an assignment as a set of *literals*  $\{l_1, l_2, \dots, l_n\}$ , where each literal  $l_i$  is either  $x_i$  or  $\bar{x}_i$ , corresponding to the assignments  $\alpha(x_i) = 1$  or 0, respectively. We denote the set of all assignments over these variables as  $\mathcal{U}$ .

#### 1.1 Boolean Functions

A *Boolean function*  $f : 2^X \rightarrow \{0, 1\}$  can be characterized by the set of assignments for which the function evaluates to 1:  $\mathcal{M}(f) = \{\alpha \mid f(\alpha) = 1\}$ . Let **1** denote the Boolean function that assigns value 1 to every assignment, and **0** denote the assignment that assigns value 0 to every assignment. These are characterized by the universal assignment set  $\mathcal{U}$  and the empty assignment set  $\emptyset$ , respectively.

From this we can define the *negation* of function  $f$  as the function  $\neg f$  such that  $\mathcal{M}(\neg f) = \mathcal{U} - \mathcal{M}(f)$ . We can also define the conjunction and disjunction operations over functions  $f_1$  and  $f_2$  as characterized by the sets  $\mathcal{M}(f_1 \wedge f_2) = \mathcal{M}(f_1) \cap \mathcal{M}(f_2)$  and  $\mathcal{M}(f_1 \vee f_2) = \mathcal{M}(f_1) \cup \mathcal{M}(f_2)$ .

For assignment  $\alpha$  and a Boolean formula  $\phi$  over  $X$ , we use the notation  $\alpha[\phi/x_i]$  to denote the assignment  $\alpha'$ , such that  $\alpha'(x_j) = \alpha(x_j)$  for all  $j \neq i$  and  $\alpha'(x_i) = \alpha(\phi)$ , where  $\alpha(\phi)$  is the value obtained by evaluating formula  $\phi$  with each variable assigned the value given by  $\alpha$ . In particular, the notation  $\alpha([\bar{x}_i/x_i])$  indicates the assignment in which the value assigned to  $x_i$  is negated, while others remain unchanged.

A Boolean function  $f$  is said to be *independent* of variable  $x_i$  if every  $\alpha \in \mathcal{M}(f)$  has  $\alpha([\bar{x}_i/x_i]) \in \mathcal{M}(f)$ . The *dependency set* of  $f$ , denoted  $D(f)$  consists of all variables  $x_i$  for which  $f$  is *not* independent.

#### 1.2 Separable Cost Functions

Let  $\mathcal{Z}$  denote the elements of a commutative ring. A *separable cost function*  $\sigma : X \rightarrow \mathcal{Z}$  assigns a value from the ring to each variable. We extend this function by defining the cost of literal  $\bar{x}_i$  as  $\sigma(\bar{x}_i) = 1 - \sigma(x_i)$ , the cost of an assignment as  $\sigma(\alpha) = \prod_{l_i \in \alpha} \sigma(l_i)$ , and the cost of a function  $f$  as  $\sigma(f) = \sum_{\alpha \in \mathcal{M}(f)} \sigma(\alpha)$ .

For ring element  $a$ , we use the notation  $\sim a$  to denote  $1 - a$ .

**Example 1a:** Let  $\mathcal{Z}$  be the set of rational numbers of the form  $a \cdot 2^b$  where  $a$  and  $b$  are integers. Let  $\sigma(x_i) = 1/2$  for all variables  $x_i$ . The cost of every assignment is then  $1/2^n$ , and the cost of a function is its *density*, denoted  $\rho(f)$ . That is, the density of  $f$ , for which  $0 \leq \rho(f) \leq 1$ , is the fraction of assignments for which  $f$  evaluates to 1, with  $\rho(\mathbf{0}) = 0$  and  $\rho(\mathbf{1}) = 1$ . The density of a function  $f$  can be scaled by  $2^n$  to compute the total number of models  $|\mathcal{M}(f)|$ . This is the core task of model counting. Using density as the metric, rather than the number of models, has the advantage that it does not vary when the function is embedded in a larger domain  $X' \supseteq X$ .

**Example 1b:** Let  $\mathcal{Z}$  be the set of rational numbers. Assign a *weight*  $w(x_i)$  to each variable  $x_i$  such that  $0 \leq w(x_i) \leq 1$  and let  $\sigma(x_i) = w(x_i)$ . This implements *weighted model counting* under the restrictions that: 1) the weight of an assignment equals the product of the weights of its literals, and 2) the weight of a variable  $x_i$  and its negation  $\bar{x}_i$  sum to 1.

**Example 1c:** Let  $\mathcal{Z}$  be the set of rational numbers. Assign *weights* to each literal. In particular, for each variable  $x_i$ , define weights  $w(x_i)$  and  $w(\bar{x}_i)$ , such that  $\beta(x_i) \doteq w(x_i) + w(\bar{x}_i) \neq 0$ . We can convert this to the formulation of Example 1b by using weight  $w'(x_i) = w(x_i)/\beta(x_i)$  and then scaling the final result by  $\prod_{1 \leq i \leq n} \beta(x_i)$ . Scaling eliminates the need to have the weight of each variable and its negation sum to 1. This form is sufficiently general to handle the benchmarks used in weighted model counting track of the Model Counting Competition. Using  $w(x_i) = w(\bar{x}_i) = 1$  for each variable  $x_i$ , gives the standard model counting of Example 1a.

**Example 1d:** Restrict the rational numbers of Example 1c to be of the form  $a \cdot 2^b \cdot 5^c$ , where  $a$ ,  $b$ , and  $c$  are integers. This class of numbers can exactly represent any value written in base-10 decimal notation, and it is closed under addition and multiplication. It can therefore be used to exactly encode the weights used in the weighted model counting track of the Model Counting Competition, as well as any results that are to be generated. In terms of implementation, both  $b$  and  $c$  can be encoded as fixed-length integers, leaving only  $a$  requiring an arbitrary-precision representation. Manipulating numbers with this representation could be simpler and more efficient than using full rational arithmetic.

**Example 2:** Let  $\mathcal{Z}$  be a finite field with  $z = |\mathcal{Z}| \geq 2n$ , and let  $\mathcal{H}$  be the set of functions mapping elements of  $X$  to elements of  $\mathcal{Z}$ . For two distinct functions  $f_1$  and  $f_2$  and a randomly chosen  $h \in \mathcal{H}$ , the probability that  $h(f_1) \neq h(f_2)$  will be at least  $(1 - \frac{1}{z})^n \geq (1 - \frac{1}{2n})^n > 1/2$ . Therefore, these functions can be used as part of a randomized algorithm for equivalence testing [1].

### 1.3 Computing Cost Functions

Three key properties of separable cost functions make it possible, in some cases, to compute the cost of a Boolean formula without enumerating all of its satisfying solutions.

**Proposition 1 (Negation).** For separable cost function  $\sigma$  and Boolean function  $f$ :  $\sigma(\neg f) = 1 - \sigma(f) = \sim \sigma(f)$ .

**Proposition 2 (Variable-Partitioned Conjunction).** For separable cost function  $\sigma$  and Boolean functions  $f_1$  and  $f_2$  such that  $D(f_1) \cap D(f_2) = \emptyset$ :  $\sigma(f_1 \wedge f_2) = \sigma(f_1) \cdot \sigma(f_2)$ .

We use the notation  $f_1 \wedge_v f_2$  to denote the conjunction of  $f_1$  and  $f_2$  under the condition that  $f_1$  and  $f_2$  are defined over disjoint sets of variables.

**Proposition 3 (Assignment-Partitioned Disjunction).** *For separable cost function  $\sigma$  and Boolean functions  $f_1$  and  $f_2$  such that  $\mathcal{M}(f_1) \cap \mathcal{M}(f_2) = \emptyset$ :  $\sigma(f_1 \vee f_2) = \sigma(f_1) + \sigma(f_2)$ .*

We use the notation  $f_1 \vee_a f_2$  to denote the disjunction of  $f_1$  and  $f_2$  under the condition that  $f_1$  and  $f_2$  hold for mutually exclusive assignments.

## 2 Partitioned-Operation Graphs (POGs)

Computing the cost function for a Boolean formula becomes straightforward when the formula contains only the operations  $\wedge_v$ ,  $\vee_a$ , and  $\neg$ , as is demonstrated by Propositions 1–3. We define a *partitioned-operation graph* (POG) as a direct-acyclic graph representation of such formulas. Representing formulas as a graph enables sharing sub-formulas, yielding a more compact representation.

### 2.1 POG Definition

**Table 1.** Recursive Definition of POGs and their Costs

$P$	Restrictions	$D(P)$	$\mathcal{M}(P)$	$\sigma(P)$
$0$	None	$\emptyset$	$\emptyset$	$0$
$1$	None	$\emptyset$	$\mathcal{U}$	$1$
$x_i$	None	$\{x_i\}$	$\{\alpha   \alpha(x_i) = 1\}$	$\sigma(x_i)$
$\neg P_1$	None	$D(P_1)$	$\mathcal{U} - \mathcal{M}(P_1)$	$\sim \sigma(P)$
$\wedge_v \{P_1, \dots, P_k\}$	$D(P_i) \cap D(P_j) = \emptyset, 1 \leq i < j \leq k$	$\bigcup_{1 \leq i \leq k} D(P_i)$	$\bigcap_{1 \leq i \leq k} \mathcal{M}(P_i)$	$\prod_{1 \leq i \leq k} \sigma(P_i)$
$P_1 \vee_a P_2$	$\mathcal{M}(P_1) \cap \mathcal{M}(P_2) = \emptyset$	$D(P_1) \cup D(P_2)$	$\mathcal{M}(P_1) \cup \mathcal{M}(P_2)$	$\sigma(P_1) + \sigma(P_2)$

The set of POGs over a set of variables  $\{x_1, x_2, \dots, x_n\}$  can be defined recursively, as is shown in Table 1. Each POG  $P$  has an associated dependency set  $D(P)$  and an associated set of models  $\mathcal{M}(P)$ . A key property of a Boolean formula represented by POG  $P$  is that, for any separable cost function  $\sigma$ , the cost of the formula  $\sigma(P)$  can be computed with a linear number of ring operations, as is shown in the final column of Table 1.

POGs are inspired by the DDNNF graphs devised by Darwiche for encoding Boolean functions, a process he refers to as “knowledge compilation.” POGs generalize DDNNF in two ways:

- It allows negation of arbitrary nodes in the graph, not just variables. It is therefore not a negation-normal form. We will see examples where this allows a more compact representation.
- It allows arbitrary arguments to a  $\vee_a$  operation, as long as they have disjoint satisfying assignments. By contrast, for each disjunction node in a DDNNF, there must be a variable  $x$  such that the assignments for one argument have  $x = 1$  while those for the other argument to have  $x = 0$ .

Both of these generalizations allow more flexibility in the POG representation while maintaining the ability to easily compute their cost for separable cost functions.

## 2.2 POG Normalization

**Table 2.** Normalization Rules

$\neg 0$	$\rightarrow$	1	$\neg 1$	$\rightarrow$	0
$\neg \neg P$	$\rightarrow$	$P$			
$P \wedge_v 0$	$\rightarrow$	0	$0 \wedge_v P$	$\rightarrow$	0
$P \wedge_v 1$	$\rightarrow$	$P$	$1 \wedge_v P$	$\rightarrow$	$P$
$P \vee_a 0$	$\rightarrow$	$P$	$0 \vee_a P$	$\rightarrow$	$P$
$P \vee_a 1$	$\rightarrow$	1	$1 \vee_a P$	$\rightarrow$	1

Table 2 shows a list of *normalizing* transformations to simplify a POG. These eliminate extra negations and remove constant terms, such that constants only occur in POGs when representing constant functions **0** and **1**.

## 2.3 Encoding the ITE Operation

The if-then-else (ITE) operation arises when converting the CNF representation of a formula into a POG, both for bottom-up approaches based on decision diagrams, and for top-down approaches based on CDCL. For functions  $f_1$ ,  $f_2$ , and  $f_3$ , we define  $ITE(f_1, f_2, f_3) = (f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$ . Observe that the  $\vee$  operation in this expression satisfies the requirements for  $\vee_a$ , since the first argument can only yield 1 for assignments that yield 1 for  $f_1$ , while the second can only yield 1 for assignments that yield 0 for  $f_1$ . Therefore, the only condition imposed on an expansion of *ITE* into the allowed POG operations is that the dependency set for  $f_1$  must be disjoint from those of  $f_2$  and  $f_3$ .

Table 3 shows different ways to encode an ITE operation into POG operations. All of these require the dependency set of the argument  $P_1$  to be disjoint from those of arguments  $P_2$  and  $P_3$ . The first row shows the most general case, requiring one  $\vee_a$  and two  $\wedge_v$  operations. The other rows show special cases, where one or more argument is a constant. Of these, the final two rows are particularly noteworthy. They make use of DeMorgan’s Laws to convert disjunctions into conjunctions. In particular, for Boolean

**Table 3.** Encodings of the ITE Operation

ITE Form	Encoding
$ITE(P_1, P_2, P_3)$	$(P_1 \wedge_v P_2) \vee_a (\neg P_1 \wedge P_3)$
$ITE(1, P_2, P_3)$	$P_2$
$ITE(0, P_2, P_3)$	$P_3$
$ITE(P_1, 1, 0)$	$P_1$
$ITE(P_1, 0, 1)$	$\neg P_1$
$ITE(P_1, P_2, 0)$	$P_1 \wedge_v P_2$
$ITE(P_1, 0, P_3)$	$\neg P_1 \wedge_v P_3$
$ITE(P_1, 1, P_3)$	$\neg(\neg P_1 \wedge_v \neg P_3)$
$ITE(P_1, P_2, 1)$	$\neg(P_1 \wedge_v \neg P_2)$

functions  $f_1$  and  $f_2$ , we can write  $ITE(f_1, 1, f_2)$  as  $f_1 \vee f_2$ , and by DeMorgan's Laws as  $\neg(\neg f_1 \wedge \neg f_2)$ . Similarly  $ITE(f_1, f_2, 1) = \neg f_1 \vee f_2 = \neg(f_1 \wedge \neg f_2)$ . These conjunctions can then be encoded with  $\wedge_v$  operations, since their arguments will have disjoint dependency sets.

### 3 Proof Framework for Cost Functions

The CRAT clausal proof framework provides a means to express a checkable proof that a Boolean formula, given in conjunctive normal form, is logically equivalent to a POG. Once this equivalence has been established, the POG can form the basis for computations enabled by the representation, including trusted model counting

The CRAT format draws its inspiration from the LRAT format for Boolean formulas and the QRAT format for quantified Boolean formulas (QBF). The following are its key properties:

- In addition to explicit clause additions and deletions, the proof contains declarations of  $\wedge_v$  and  $\vee_a$  operations.
  - These declarations implicitly add extension variables and their defining clauses to the proof. This is the only means for generating extension variables or adding blocked clauses to the proof.
  - The checker tracks the dependency set for every input and extension variable. When an extension variable is introduced based on the  $\wedge_v$  operation, the dependency sets of its arguments must be disjoint. The resulting extension variable has a dependency set equal to the union of those of its arguments.
  - Declaring a  $\vee_a$  operation requires a sequence of clauses providing a RUP proof that the arguments are mutually exclusive. This sequence can either be provided explicitly or inferred by the proof checker.
  - Boolean complement is provided implicitly by allowing the arguments of the  $\vee_a$  and  $\wedge_v$  operations to be literals and not just variables.
- Clauses can also be added when they satisfy the RUP property, with respect to a sequence of existing clauses. This sequence can be either supplied explicitly or inferred by the proof checker.

- Deleting clauses requires proving that the resulting formula is not weaker.
- For an input clause or a clause declared by RUP addition, its deletion must be accompanied by a sequence of remaining clauses providing a RUP proof of the clause. This sequence can either be provided explicitly or inferred by the proof checker.
- The clauses defining a  $\wedge_v$  or  $\vee_a$  operation are implicitly deleted by deleting the operation. This can only be done when the only undeleted clauses containing references to the associated extension variable are those implicitly defined when the operation was declared. This implies that there can be no undeleted operations having the operation result as an argument.

### 3.1 Syntax

**Table 4.** CRAT Step Types.  $C$ : clause identifier,  $L$ : literal,  $V$ : variable

Rule				Description
$C$	i	$L^* 0$		Input clause
$C$	a	$L^* 0$	$C^+ 0$	Add RUP clause. Explicit hint
$C$	a	$L^* 0$	$* 0$	Add RUP clause. Inferred hint
	dc	$C$	$C^+ 0$	Delete RUP clause. Explicit hint
	dc	$C$	$* 0$	Delete RUP clause. Inferred hint
$C$	p	$V L^{2+} 0$		Declare $\wedge_v$ operation
$C$	s	$V L L$	$C^+ 0$	Declare $\vee_a$ operation. Explicit hint
$C$	s	$V L L$	$* 0$	Declare $\vee_a$ operation. Inferred hint
	do	$V$		Delete operation.

Table 4 shows the declarations that can occur in a CRAT file. As with other clausal proof formats, a variable is represented by a positive integer  $v$ , with the first ones being input variables and successive ones being extension variables. Literal  $l$  is represented by a signed integer, with  $-v$  being the complement of variable  $v$ . Each clause is indicated by a positive integer identifier  $C$ , with the first ones being the IDs of the input clauses and successive ones being the IDs of added clauses. Clause identifiers must be totally ordered, such that clause  $C$  can only reference clauses  $C'$  such that  $C' < C$ . Clause identifiers need not be consecutive.

The first set of proof rules are similar to those in other clausal proofs. Our syntax optionally allows input clauses to be listed with a rule of type i. Clauses can be added via RUP addition (command a), with the sequence of antecedent clauses (the “hint”) either provided explicitly or to be inferred (indicated by the character ‘\*’) by the proof checker. Similarly for clause deletion (command dc).

The declaration of a  $\wedge_v$  operation has the form:

$$i \quad p \quad v \quad l_1 \quad l_2 \quad \dots \quad l_k \quad 0$$

where  $i$  is a new clause ID,  $v$  is a positive integer that does not correspond to any previous variable, and  $l_1, l_2, \dots, l_k$  is a sequence of  $k$  ( $k \geq 2$ ) integers representing literals of existing variables. This declaration implicitly causes  $k + 1$  clauses to be added to the proof:

ID	Clause
$i$	$v \quad -l_1 \quad -l_2 \quad \dots \quad -l_k$
$i+1$	$-v \quad l_1$
$i+2$	$-v \quad l_2$
	$\dots$
$i+k$	$-v \quad l_k$

The dependency sets for the arguments represented by each pair of literals  $l_{j_1}$  and  $l_{j_2}$ , for  $1 \leq j_1 < j_2 \leq k$ , must be disjoint.

The declaration of a  $\vee_a$  operation has the form:

$$i \quad s \quad v \quad l_1 \quad l_2 \quad H \quad 0$$

where  $i$  is a new clause ID,  $v$  is a positive integer that does not correspond to any previous variable, and  $l_1$  and  $l_2$  are signed integers representing literals of existing variables. Hint  $H$  can consist either of the single character  $*$ , or it can be a sequence of clause IDs. This declaration implicitly causes three clauses to be added to the proof:

ID	Clause
$i$	$-v \quad l_1 \quad l_2$
$i+1$	$v \quad -l_1$
$i+2$	$v \quad -l_2$

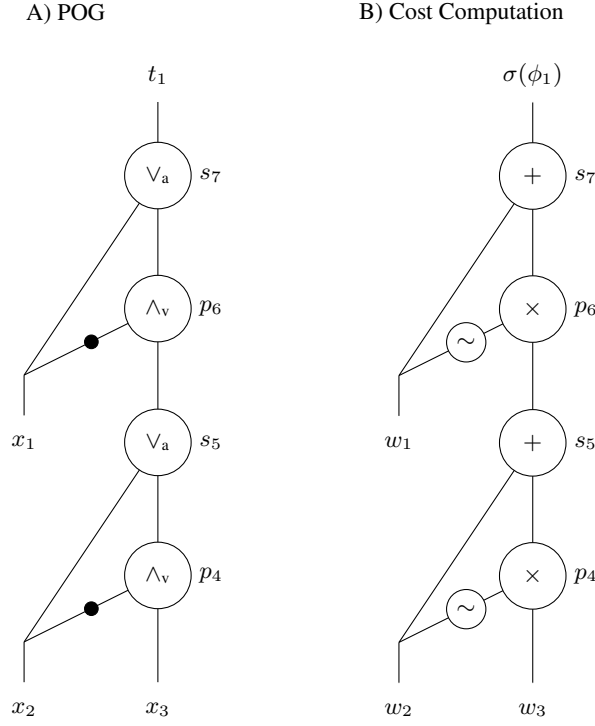
The explicit or implied hints must provide a RUP proof of the clause  $-l_1 \vee -l_2$ .

Finally, operator deletion ( $\text{do}$ ) specifies which operator to delete via the extension variable associated with the operator. Doing so for operator  $v$  requires that there is no (undeleted) RUP clause containing a literal of  $v$ . Implicitly, that means that there is no (undeleted) operator  $u$  having a literal of  $v$  as one of its inputs. The defining clauses for the operation are also deleted.

### 3.2 Semantics

A CRAT proof follows the same general form as a QRAT dual proof [3]—one that ensures that each clause addition and each clause deletion preserves equivalence. With CRAT, however, clauses are defined both explicitly and implicitly. Starting with the set of input clauses, the proof consists of a sequence of steps that both add and delete clauses. Each addition must be truth preserving and each deletion must be falsehood preserving. At the end, all input clauses must have been deleted, and among the remaining clauses there must be only a single unit clause consisting of some variable or its complement. Except for trivial cases, the final literal will be an extension variable or its complement. That literal will indicate the root of the POG as the (possibly negated) output of a POG operation. Working from that root backward, the POG can be extracted from the CRAT file.

### 3.3 Example 1



**Fig. 1.** POG #1 for Formula  $\phi_1 = x_1 \vee x_2 \vee x_3$  and its Cost Computation

As an illustration, consider the Boolean formula  $\phi_1 = x_1 \vee x_2 \vee x_3$ , represented by a single clause. We cannot directly use the  $\vee_a$  operation to form these disjunctions, since the sets of assignments satisfying the individual literals are not disjoint. Instead, we must decompose this formula into a sequence of operations. Figure 1A shows one such decomposition. The subscripts of the variables and the operator labels correspond to the numbers of the input and extension variables in the CRAT proof. Edges marked with dots indicate Boolean negation.

The conjunction of  $\bar{x}_2$  and  $x_3$  can be computed as  $p_4 = \bar{x}_2 \wedge_v x_3$ , since the literals have disjoint dependency sets. We can then express the disjunction  $x_2 \vee x_3$  as  $s_5 = x_2 \vee_a p_4$ . A similar process forms the disjunction  $x_1 \vee x_2 \vee x_3$  by first forming the product  $p_6 = \bar{x}_1 \wedge_v s_5$  and the final sum  $s_7 = x_1 \vee_a p_6$ .

The logical representation can readily be converted into a formula for computing  $\sigma(\phi_1)$ , the cost of formula  $\phi_1$ , given a weight  $w_i$  for each variable  $x_i$  for  $1 \leq i \leq 3$ . This is illustrated in Figure 1B. Note how the Boolean negations become  $\sim$  operations. This formula is valid for any cost function.



A). CRAT File Contents

File line											Explanation			
1	i	1	2	3	0						Input clause			
2	p	4	-2	3	0						Declare $p_4 = \bar{x}_2 \wedge_v x_3$			
5	s	5	2	4		3	0				Declare $s_5 = x_2 \vee_a p_4$			
8	p	6	-1	5	0						Declare $p_6 = \bar{x}_1 \wedge_v s_5$			
11	s	7	1	6		9	0				Declare $s_7 = x_1 \vee_a p_6$			
14	a	7	0			13	12	8	7	6	2	1	0	Assert unit clause $[s_7]$
	dc	1				4	5	10	11	14	0			Delete input clause

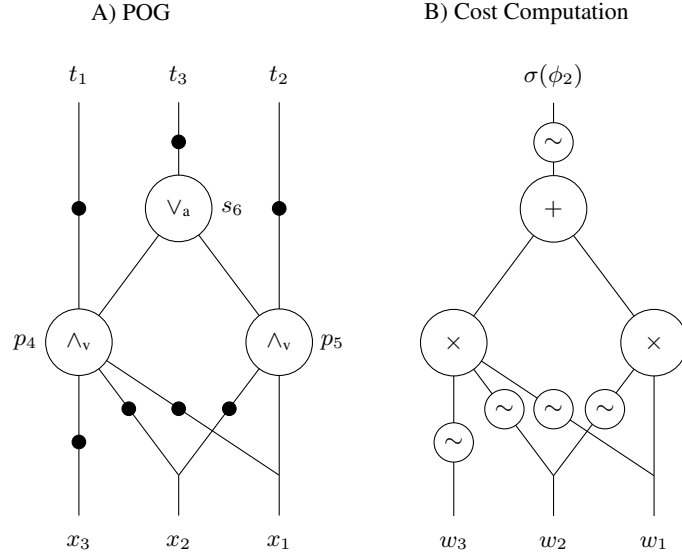
B). Proof Clauses

Clause										Explanation		
1	1	2	3	0							Input clause	
2	4	2	-3	0							Defining clauses for $p_4$	
3	-4	-2	0									
4	-4	3	0									
	-2	-4	0		3	0					Mutual exclusion proof for $s_5$	
5	-5	2	4	0							Defining clauses for $s_5$	
6	5	-2	0									
7	5	-4	0									
8	6	1	-5	0							Defining clauses for $p_6$	
9	-6	-1	0									
10	-6	5	0									
	-1	-6	0		9	0					Mutual exclusion proof for $s_7$	
11	-7	1	6	0							Defining clauses for $s_7$	
12	7	-1	0									
13	7	-6	0									
14	7	0			13	12	7	6	2	1	0	Assert unit clause $[s_7]$
	dc	1			4	5	10	11	14	0		Delete input clause

**Fig. 2.** CRAT file #1 for formula  $x_1 \vee x_2 \vee x_3$ , and the resulting set of proof clauses

Figure 2A shows an annotated version of the CRAT file for this example, while 2B shows the sequence of clauses that constitute the proof, including those defined implicitly, and those required for mutual exclusion checks. Clause 1 is the input clause. Clauses 2–13 are added explicitly as the defining clauses for the four operations. Also shown are the required mutual exclusion proofs for the two sum operations. Proof clause 14 adds the unit clause for the extension variable  $s_7$ . We refer to the literal representing formula  $\phi_1$  as  $t_1$ , and we therefore have  $t_1 = s_7$ . The unit clause indicates that extension variable  $s_7$  will evaluate to 1 for any assignment that satisfies the formula. We can write this as  $\phi_1 \models t_1$ . The deletion step at the end turns this around, showing that  $t_1 \models \phi_1$ , and therefore the input clause can be deleted. This completes a proof that  $t_1$  is logically equivalent to the input formula.

### 3.4 Example 2A



**Fig. 3.** POG #2A for Formula  $\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$  and its Cost Computation

As a more complex example, consider the Boolean formula  $\phi_2$  given by the conjunction of clauses  $C_1 = x_1 \vee x_2 \vee x_3$  and  $C_2 = \bar{x}_1 \vee x_2$ . With this example, we also demonstrate the use of DeMorgan's Laws to provide a more compact encoding of the formula, similar to the use of these laws when encoding ITE operations in Table 3.

Figure 3 shows a POG representing the formula, and Figure 4 shows the associated CRAT file and proof clauses. This proof was generated via a bottom-up strategy, such as would be created using BDDs. It creates POG representations of the input clauses and then forms their conjunction. In the proof, unit clauses are generated for the repres-

A). CRAT File Contents

		Proof line	Explanation
1	i	1 2 3 0	Input clause $C_1$
2	i	-1 2 0	Input clause $C_2$
3	p	4 -1 -2 -3 0	Declare $p_4 = \bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3$
7	a	-4 0            6 5 4 1 0	Assert $t_1 = \bar{p}_4 = x_1 \vee x_2 \vee x_3$
	dc	1                3 7 0	Delete clause $C_1$
8	p	5 1 -2 0	Declare $p_5 = x_1 \wedge \bar{x}_2$
11	a	-5 0            10 9 2 0	Assert $t_2 = \bar{p}_5 = \bar{x}_1 \vee x_2$
	dc	2                8 11 0	Delete clause $C_2$
12	s	6 4 5            4 9 0	Declare $s_6 = \bar{t}_1 \vee_a \bar{t}_2$
15	a	-6 0            7 11 12 0	Assert $t_3 = \bar{s}_6 = t_1 \wedge t_2$
	dc	7                13 15 0	Delete $t_1$
	dc	11               14 15 0	Delete $t_2$

B). Proof Clauses

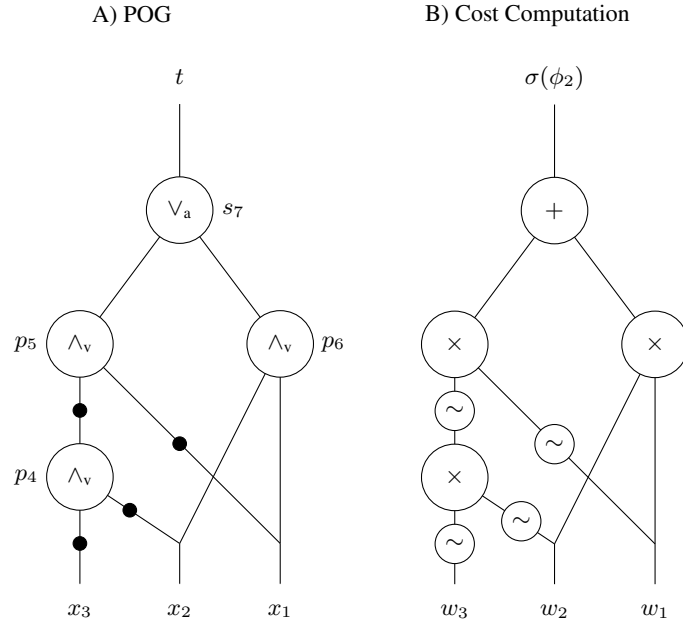
		Clause	Explanation
1		1 2 3 0	Input clause $C_1$
2		-1 2 0	Input clause $C_2$
3		4 1 2 3 0	Defining clauses for $p_4$
4		-4 -1 0	
5		-4 -2 0	
6		-4 -3 0	
7		-4 0            6 5 4 1 0	Assert $t_1 = \bar{p}_4 = x_1 \vee x_2 \vee x_3$
	dc	1                3 7 0	Delete clause $C_1$
8		5 -1 2 0	Defining clauses for $p_5$
9		-5 1 0	
10		-5 -2 0	
11		-5 0            10 9 2 0	Assert $t_2 = \bar{p}_5 = \bar{x}_1 \vee x_2$
	dc	2                8 11 0	Delete clause $C_2$
		-4 -5 0            4 9 0	Mutual exclusion proof for $s_6$
12		-6 4 5 0	Defining clauses for $s_6$
13		6 -4 0	
14		6 -5 0	
15		-6 0            7 11 12 0	Assert $t_3 = \bar{s}_6 = t_1 \wedge t_2$
	dc	7                13 15 0	Delete $t_1$
	dc	11               14 15 0	Delete $t_2$

**Fig. 4.** CRAT file #2A for formula  $\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$ , and the resulting set of proof clauses

tations of the input clauses, and then the input clauses are deleted. These intermediate unit clauses are used to justify a unit clause for the final root, and then they are deleted.

Using DeMorgan's Laws,  $C_1$  can be written as  $t_1 = \neg[\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3]$ , and this can be expressed using the  $\wedge_v$  operation  $p_4$ , shown in Figure 3A. (Note that  $t_1$  in this case is logically equivalent to root  $t_1$  in the POG of Figure 1A, but the use of negation enables it to be represented with one operation rather than four.) Similarly,  $C_2$  can be written as  $t_2 = \neg[x_1 \wedge \bar{x}_2]$ , and this can be represented by  $\wedge_v$  operation  $p_5$ . Terms  $t_1$  and  $t_2$  are asserted as unit clauses on proof lines 7 and 11, allowing the input clauses to be deleted. The conjunction  $t_1 \wedge t_2$  can be written as  $t_3 = \neg[\bar{t}_1 \vee \bar{t}_2]$ , represented by  $\vee_a$  operation  $s_6$ . Term  $t_3$  is asserted as a unit clause on proof line 15. Based on this, the unit clauses for terms  $t_1$  and  $t_2$  can be deleted. Term  $t_3$  then becomes the unique root of the cost computation shown in Figure 3B.

### 3.5 Example 2B



**Fig. 5.** POG #2B for Formula  $\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$  and its Cost Computation

Figure 5 shows an alternate POG representing the same formula  $\phi_2$  as in Example 2A, and Figure 6 shows the associated CRAT proof. This proof was generated via a top-down strategy, such as would be created using a model counter based on CDCL. It starts by splitting on variable  $x_1$ . Clause  $C_1$  is trivially satisfied when  $x_1$  is assigned 1, and clause  $C_2$  becomes the clause  $C'_2 = x_2$ . On the other hand, clause  $C_2$  is trivially

A). CRAT File Contents

	File line	Explanation
1	1 2 3 0	Input clause $C_1$
2	i -1 2 0	Input clause $C_2$
3	p 4 -2 -3 0	Declare $p_4 = \bar{x}_2 \wedge_v \bar{x}_3$
6	p 5 -1 -4 0	Declare $p_5 = \bar{x}_1 \wedge_v \bar{p}_4$
9	p 6 1 2 0	Declare $p_6 = x_1 \wedge_v x_2$
12	s 7 5 6 7 10 0	Declare $s_7 = p_5 \vee_a p_6$
15	a 1 -4 0 4 5 1 0	Justify $\bar{x}_1 \rightarrow \bar{p}_4$
16	a 1 7 0 13 6 15 0	Justify $\bar{x}_1 \rightarrow s_7$
17	a -1 7 0 14 9 2 0	Justify $x_1 \rightarrow s_7$
18	a 7 0 16 17 0	Justify unit clause $t = s_7$
	dc 1 3 8 10 12 18 0	Delete $C_1$
	dc 2 11 7 12 18 0	Delete $C_2$

B). Proof Clauses

	File line	Explanation
1	1 2 3 0	Input clause $C_1$
2	-1 2 0	Input clause $C_2$
3	4 2 3 0	Defining clauses for $p_4$
4	-4 -2 0	
5	-4 -3 0	
6	5 1 4 0	Defining clauses for $p_5$
7	-5 -1 0	
8	-5 -4 0	
9	6 -1 -2 0	Defining clauses for $p_6$
10	-6 1 0	
11	-6 2 0	
	-5 -6 7 10 0	Mutual exclusion proof for $s_7$
12	-7 5 6 0	Defining clauses for $s_7$
13	7 -5 0	
14	7 -6 0	
15	1 -4 0 4 5 1 0	Justify $\bar{x}_1 \rightarrow \bar{p}_4$
16	1 7 0 13 6 15 0	Justify $\bar{x}_1 \rightarrow s_7$
17	-1 7 0 14 9 2 0	Justify $x_1 \rightarrow s_7$
18	7 0 16 17 0	Justify unit clause $t = s_7$
	dc 1 3 15 0	Delete $C_1$
	dc 2 11 7 12 18 0	Delete $C_2$

**Fig. 6.** CRAT file #2B for formula  $\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$ , and the resulting set of proof clauses

satisfied when  $x_1$  is assigned 0, and clause  $C_1$  becomes  $C'_1 = x_2 \vee x_3$ . Clause  $C'_1$  can be represented by POG operation  $\neg[\bar{x}_2 \wedge_v \bar{x}_3]$ , with this operation labeled  $p_4$  in Figure 5A. The splitting on variable  $x_1$  can be rejoined as  $ITE(x_1, x_2, \bar{p}_4)$ , and this ITE operation can be expressed using the product operations  $p_5$  and  $p_6$ , joined by the sum operation  $t = s_7$ .

The two POGs shown in Figures 3A and 5A have similar structure. They have very different negation patterns, but they are logically equivalent. Their associated cost computations (Figures 3B and 5B) also yield the same results for arbitrary weights  $w_1$ ,  $w_2$ , and  $w_3$ .

The CRAT proof for the top-down approach follows a different pattern than does the proof based on a bottom-up construction. It does not create any intermediate unit clauses. Instead, it constructs a proof that the root  $t = s_7$  holds as a unit clause by splitting into the two assignments for variable  $x_1$ . The proof that  $\bar{x}_1 \rightarrow s_7$  (proof line 16) builds on the proof that  $\bar{x}_1 \rightarrow \bar{p}_4$  (proof line 15), which derives from clause  $C_1$ . The proof that  $x_1 \rightarrow x_7$  (proof line 17) derives from clause  $C_2$ . These two are combined to yield the unit clause  $s_7$  (proof line 18). Given this unit clause, the two input clauses can then be deleted.

## 4 Looking Ahead

### 4.1 Implementing Certified Counters

Given an arbitrary CNF formula, we can use BDD operations to generate a POG representation. The proof generation can follow the methods we have used for generating unsatisfiability proofs of Boolean formulas [4] and dual proofs of quantified Boolean formulas [3]. Each BDD node can be expressed as an ITE operation and make use of the encodings shown in Table 3.

A second class of model counters proceeds top-down, based on the CDCL framework. These choose a splitting variable  $x$  and recursively construct POGs  $P_1$  and  $P_0$  for the two assignments to the variable. These are combined as  $ITE(x, P_1, P_0)$ , using one of the encodings of  $ITE$  shown in Table 3. Like CDCL, the top-down algorithm can make use of unit propagation, conflict detection, and clause learning. It can also make use of *variable* partitioning. That is, suppose for some partial assignment to the variables, the input clauses decompose into two or more sets over disjoint variables. Then the POG for each of these partitions can be generated separately, and these are joined via the  $\wedge_v$  operation.

## 5 Exploiting Shared Subgraphs

### References

1. Blum, M., Chandra, A.K., Wegman, M.N.: Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters* **10**(2), 80–82 (18 March 1980)
2. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers* **35**(8), 677–691 (1986)

```

p cnf 4 5
-1 3 -4 0
-1 -3 4 0
1 3 -4 0
1 -3 4 0
-1 -2 0

```

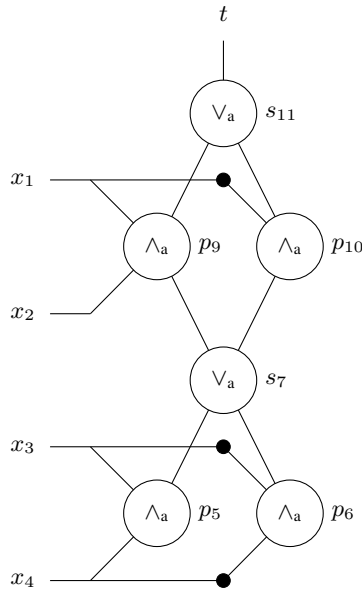
**Fig. 7.** CNF representation of Formula #4

```

c Node P5 = AND(var-3, var-4)
12 p 5 3 4 0
c Node P6 = AND(-var-3, -var-4)
15 p 6 -3 -4 0
c Node S7 = OR(P5, P6)
18 s 7 5 6 13 16 0
c Node P9 = AND(var-1, -var-2, S7)
21 p 9 1 -2 7 0
c Node P10 = AND(-var-1, S7)
25 p 10 -1 7 0
c Node S11 = OR(P9, P10)
28 s 11 9 10 22 26 0

```

**Fig. 8.** Declarations of operations to represent Formula #4



**Fig. 9.** POG for Formula #4 yielding shared subgraph  $s_7$

```

c Shadow clause [3, -4]
6 p 12 -3 4 0
c Implicit declarations:
c 6 a 12 3 -4 0
c 7 a -12 -3 0
c 8 a -12 4 0
c Shadow clause [-3, 4]
9 p 13 3 -4 0
c Implicit declarations:
c 9 a 13 -3 4 0
c 10 a -13 3 0
c 11 a -13 -4 0

```

**Fig. 10.** Declarations of operations to represent lemma arguments

```

c Proof of Lemma S7
31 a 12 13 -3 4 0 * 0
32 a 12 13 3 -4 0 * 0
33 a -3 7 12 13 0 31 12 19 0
34 a 7 12 13 0 33 32 15 20 0
c Apply Lemma S7. Context = [1, -2]
35 a -1 2 -12 0 7 8 1 0
36 a -1 2 -13 0 10 11 2 0
37 a -1 2 7 0 35 36 34 0
c Apply Lemma S7. Context = [-1]
38 a 1 -12 0 7 8 3 0
39 a 1 -13 0 10 11 4 0
40 a 1 7 0 38 39 34 0
c Assert ITE at node S11 (root)
41 a -1 11 0 5 37 21 29 0
42 a 11 0 41 40 25 30 0

```

**Fig. 11.** Assertions leading to unit clause for  $t = s_7$



```

c Delete all but final asserted clause
dc 41 5 37 21 29 0
dc 40 38 39 34 0
dc 39 10 11 4 0
dc 38 7 8 3 0
dc 37 35 36 34 0
dc 36 10 11 2 0
dc 35 7 8 1 0
dc 34 33 32 15 20 0
dc 33 31 12 19 0
dc 32 * 0
dc 31 * 0
c Delete input clauses
dc 1 13 17 18 24 26 28 42 0
dc 2 14 16 18 24 26 28 42 0
dc 3 22 13 17 18 27 28 42 0
dc 4 22 14 16 18 27 28 42 0
dc 5 23 26 28 42 0

```

**Fig. 12.** Clause deletions for Formula #4

3. Bryant, R.E., Heule, M.J.H.: Dual proof generation for quantified Boolean formulas with a BDD-based solver. In: Conference on Automated Deduction (CADE). LNAI, vol. 12699, pp. 433–449 (2021)
4. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2021)