A Tractable Class of Model Counting Problems

Denis Deratani Mauá
Instituto de Matemática e Estatística
Universidade de São Paulo
São Paulo, Brazil
Email: denis.maua@usp.br

Fabio Gagliardi Cozman
Escola Politécnica
Universidade de São Paulo
São Paulo, Brazil
Email: fgcozman@usp.br

***Abstract*—We develop a polynomial-time algorithm for model counting of formulas in monotone conjunctive normal form where the clauses can be partitioned in two sets: any two clauses in different sets share exactly one variable; any two clauses in the same set have the same number of variables and share no variables. Our algorithm reduces the problem to edge cover counting problems that can be solved by dynamic programming efficiently. We also comment on extensions and applications of these results, such as solving weighted model counting with uniform weights and computing the number of satisfying interpretations of DL-Lite sentences.**

I. Introduction

"Model counting" usually refers to the problem of counting the number of satisfying truth-value assignments of a given Boolean formula. Many problems in artificial intelligence and combinatorial optimization can be either specialized to or generalized from model counting. For instance, propositional satisfiability (i.e., the problem of deciding whether a satisfying truth-value assignment exists) is a special case of model counting; probabilistic reasoning in graphical models such as Bayesian networks can be reduced to a weighted variant of model counting [1], [2]; validity of conformal plans can be formulated as model counting [3]. Thus, characterizing the theoretical complexity of the problem is both of practical and theoretical interest.

In unrestricted form, the problem is complete for the class #P, which contains the integer-valued functions that can be computed by counting the number of accepting paths in a non-deterministic Turing machine running in polynomial time [4]. Even very restrictive versions of the problem are complete for #P. For example, the problem is #P-complete even when the formulas are in conjunctive normal form with two variables per clause, there is no negation, and the variables can be partitioned into two sets such that no clause contains two variables in the same block [5]. The problem is also #P-complete when the formula is monotone and each variable appears at most twice, or when the formula is monotone, the clauses contain two variables and each variables appears at most k times for any $k \geq 5$ [6]. A few tractable classes have been found: for example, Roth [7] developed an algorithm for counting the number of satisfying assignments of formulas in conjunctive normal form with two variables per clause, each variable appearing in at most two clauses. Relaxing the constraint on the number of variables per clauses takes us back to intractability: model counting restricted to formulas in conjunctive normal form with variables appearing in at most two clauses is #P-complete [8].

Researchers have also investigated the complexity with respect to the graphical representation of formulas. Computing the number of satisfying assingments for monotone formulas in conjunctive normal form, with at most two variables per clause, with each variable appearing at most four times is #P-complete even when the primal graph (where nodes are variables and an edge connects variables that coappear in a clause) is bipartite and planar [6]. The problem is also #P-complete for monotone conjunctive normal form formulas whose primal graph is 3-regular, bipartite and planar. In fact, even deciding whether the number of satisfying assignments is even (i.e., counting *modulo two*) in conjunctive normal form formulas where each variable appears at most twice, each clause has at most three variables, and the incidence graph (where nodes are variables and clauses, and edges connect variables appearing in clauses) of the formula is planar is known to be NP-hard by a randomized reduction [9]. Interestingly, counting the number of satisfying assignments *modulo seven* (!) of that same class of formulas is polynomial-time computable [10].

In this paper, we present another class of tractable model counting problems defined by its graphical representation. In particular, we develop a polynomial-time algorithm for formulas in monotone conjunctive normal form whose clauses can be partitioned into two sets such that (i) any two clauses in the same set have the same number of variables which are not shared between them, and (ii) any two clauses in different sets share exactly one variable. These formulas lead to intersection graphs (where nodes are clauses, and edges connect clauses which share variables) which are bipartite complete. We state our result in the language of edge coverings; the use of a graph problem makes communication easier with no loss of generality. Even though the class of formulas we consider is somewhat narrow, we expect that future work can build on the results presented here and relax some of the assumptions.

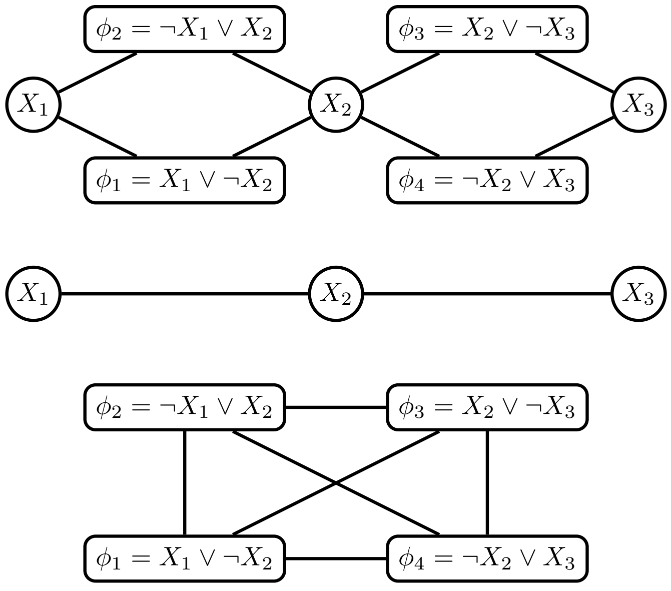
The rest of the paper is organized as follows. Section II presents the basics of model counting and the particular class of problems we consider. Section III introduces the problem of counting edge covers in black-and-white graphs, and establishes its correspondence with model counting problems. Our main result, a polynomial-time algorithm for counting edge covers of a certain class of black-and-white graphs, is shown in Section IV under the assumption that graphs contain dangling edges (which is explained in Section III). This restriction is lifted in Section V, where we develop an algorithm for graphs with no dangling edges. We then comment on possible extensions of the algorithms in Section VI, and its equivalence with the problem of counting the number of satisfying interpretations in DL-Lite formulas in Section VII. Our final remarks are given in Section VIII.

II. Model Counting

A propositional (or Boolean) variable X_i represents a statement that can be either true ($X_i = 1$) or false ($X_i = 0$). A literal is either a single variable (X_i) or its negation ($\neg X_i$). A clause is a disjunction of literals, for example, $X_2 \vee \neg X_4 \vee X_5$. We say that two clauses do not intersect if the variables in one clause do not appear in the other. If X is the largest set of variables that appear in two clauses, we say that the clauses intersect (at X). For instance, the clauses $X_1 \vee X_2 \vee X_3$ and $\neg X_2 \vee \neg X_4$ intersect at $\{X_2\}$. A clause containing k variables is called a k-clause, and k is called the size of the clause. A formula in conjunctive normal form (CNF) is a conjunction of clauses, for instance, $(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_3)$. A k-CNF formula contains only j-clauses, with $j \leq k$. The degree of a variable in a CNF formula is the number of clauses in which either the variable or its negation appears. A CNF formula where every variable has degree at most two is said *read-twice*. If any two clauses intersect in at most one variable, the formula is said *linear*. The formula $(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_3)$ is a linear read-twice 2-CNF containing two 2-clauses that intersect at X_1. The degree of X_1 is two, while the degree of either X_2 or X_3 is one. A formula is *monotone* if no variable appears negated, such as in $X_1 \vee X_2$.

We can graphically represent the dependencies between variables and clauses in a CNF formula in many ways. The *incidence graph* of a CNF formula is the bipartite graph with variable-nodes and clause-nodes. The variable-nodes correspond to variables of the formula, while the clause-nodes correspond to clauses. An edge is drawn between a variable-node and a clause-node if and only if the respective variable appears in the respective clause. The *primal graph* of a CNF formula is a graph whose nodes are variables and edges connect variables that co-appear in some clause. The primal graph can be obtained from the incidence graph by deleting clause-nodes (along with their edges) and pairwise connecting their neighbors. The *intersection graph* of a CNF formula is the graph whose nodes correspond to clauses, and an edge connects two nodes if and only if the corresponding clauses intersect. The intersection graph can be obtained from the incidence graph by deleting variable-nodes and pairwise connecting their neighbors. Figure 1 shows examples of graphical illustrations of a Boolean formula. We represent clauses as rectangles and variables as circles.

A truth-value assignment (or simply assignment) is a function $\sigma : \{X_1, \ldots, X_n\} \to \{0, 1\}$ mapping each variable X_i to a true or false value $\sigma(X_i)$. A CNF formula ϕ is satisfied by an assignment σ (written $\sigma \models \phi$) if each clause contains either a nonnegated variable X_i such that $\sigma(X_i) = 1$ or a negated variable X_j such that $\sigma(X_j) = 0$. In this case, we say that σ is a model of ϕ. For monotone CNF formulas, this condition simplifies to the existence of a variable X_i in each clause for which $\sigma(X_i) = 1$. Hence, monotone formulas are always satisfiable (by the trivial model that assigns every variable the value one). The *model count* of a formula ϕ is the number $Z(\phi) = |\{\sigma : \sigma \models \phi\}|$ of models of the formula. The *model counting problem* is to compute the model count of a given CNF formula ϕ.



$\phi_2 = \neg X_1 \vee X_2$
$\phi_3 = X_2 \vee \neg X_3$
X_1
X_2
X_3
$\phi_1 = X_1 \vee \neg X_2$
$\phi_4 = \neg X_2 \vee X_3$
X_1
X_2
X_3
$\phi_2 = \neg X_1 \vee X_2$
$\phi_3 = X_2 \vee \neg X_3$
$\phi_1 = X_1 \vee \neg X_2$
$\phi_4 = \neg X_2 \vee X_3$

Fig. 1. Graphical illustrations of the formula $(X_1 \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (X_2 \vee \neg X_3) \wedge (\neg X_2 \vee X_3)$. Top: incidence graph. Middle: primal graph. Bottom: intersection graph.

In this work, we consider linear monotone CNF formulas whose intersection graph is bipartite complete, and such that all clauses in the same part have the same size. These assumptions imply that each variable appears in at most two clauses (hence the formula is read-twice). We call CNF formulas satisfying all of these assumptions linear monotone clause-bipartite complete (LinMonCBPC) formulas. Under these assumptions, we show that model counting can be performed in quadratic time in the size of the input. It is our hope that in future work some of these assumptions can be relaxed. However, due to the results mentioned in the introduction, we do not expect that much can be relaxed without moving to #P-completeness.

The set of model counting problems generated by LinMonCBPC formulas is equivalent to the following problem. Take integers m, n, M, N such that $N > n > 0$ and $M > m > 0$, and compute how many $\{0,1\}$-valued matrices of size M-by-N exist such that (i) each of the first m rows has at least one cell with value one, and (ii) each of the first n columns has at least one cell with value one. Call A_{ij} the value of the ith row, jth column. The problem is equivalent to computing the number of matrices $A_{M \times N}$ with $\sum_{j=1}^{N} A_{ij} > 0$, for $i = 1, \ldots, m$, and $\sum_{i=1}^{M} A_{ij} > 0$, for $j = 1, \ldots, n$. This problem can be encoded as the model count of the CNF formula whose clauses are

$$
\begin{gathered}
A_{11} \vee A_{12} \vee \cdots \vee A_{1n} \vee \cdots \vee A_{1N}, \\
A_{21} \vee A_{22} \vee \cdots \vee A_{2n} \vee \cdots \vee A_{2N}, \\
\vdots \\
A_{m1} \vee A_{n2} \vee \cdots \vee A_{mn} \vee \cdots \vee A_{1N}, \\
A_{11} \vee A_{21} \vee \cdots \vee A_{m1} \vee \cdots \vee A_{M1}, \\
\vdots \\
A_{1n} \vee A_{2n} \vee \cdots \vee A_{mn} \vee \cdots \vee A_{MN}.
\end{gathered}
$$

The first m clauses are the row constraints, while the last

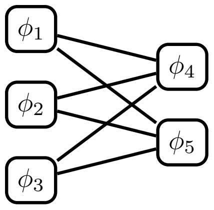


Fig. 2. Intersection graph for the LinMonCBPC formula described in the text.

n clauses are the columns constraints. The row constraints have size n, and the column constraints have size m. The ith row constraint intersects with the jth column constraint at the variable A_{ij}. For example, given integers $m = 3, n = 2, M = 5, N = 6$, the equivalent model counting problem has clauses

$$\begin{aligned}
\phi_1 &: A_{11} \vee A_{12} \vee A_{13} \vee A_{14} \vee A_{15} \vee A_{16},\\
\phi_2 &: A_{21} \vee A_{22} \vee A_{23} \vee A_{24} \vee A_{25} \vee A_{26},\\
\phi_3 &: A_{31} \vee A_{32} \vee A_{33} \vee A_{34} \vee A_{35} \vee A_{36},\\
\phi_4 &: A_{11} \vee A_{21} \vee A_{31} \vee A_{41} \vee A_{51},\\
\phi_5 &: A_{12} \vee A_{22} \vee A_{32} \vee A_{42} \vee A_{52}.
\end{aligned}$$

The intersection graph of that formula is show in Figure 2. Note that for the complexity of both problems be equivalent we must have the integers in the matrix problem be given in unary notation (otherwise building the equivalent formula takes time exponential in the input).

III. Counting Edge Covers and Its Connection to Model Counting

A *black-and-white graph* (bw-graph) is a triple $G = (V, E, \chi)$ where (V, E) is a simple undirected graph and $\chi : V \to \{0, 1\}$ is binary valued function on the node set (assume 0 means white and 1 means black).[1] We denote by $E_G(u)$ the set of edges incident in a node u, and $N_G(u)$ the open neighborhood of u (i.e., not including u). Let $G = (V, E, \chi)$ be a bw-graph. An edge $e = (u, v) \in E$ can be classified into one of three categories:[2]

- **free edge:** if $\chi(u) = \chi(v) = 0$;
- **dangling edge:** if $\chi(u) \neq \chi(v)$; or
- **regular edge:** if $\chi(u) = \chi(v) = 1$.

In the graph in Figure 3(b), the edge (f, g) is a dangling edge while the edge (g, j) is a free edge. The edge (f, g) in the graph in Figure 3(a) is a regular edge.

An *edge cover* of a bw-graph G is a set $C \subseteq E$ such that for each node $v \in V$ with $\chi(v) = 1$ there is *at least one* edge $e \in C$ incident in it. An edge cover for the graph in Figure 3(a) is $\{(a, d), (d, g), (e, g), (f, g), (h, j)\}$. We denote by $Z(G)$ the number of edge covers of a bw-color graph G. Computing $Z(G)$ is #P-complete [13], and admits an FPTAS [11], [12].

[1]In [11] and [12], graphs are uncolored, but edges might contain empty endpoints. These are analogous to white node endpoints in our terminology. We prefer defining coloured graphs and allow only simple edges to make our framework close to standard graph theory terminology.

[2]The classifications of edges given here are analogous to those defined in [11], [12], but not fully equivalent. Regular edges are analogous to the *normal edges* defined in [11], [12].

Consider a LinMonCBPC formula and let (L, R, E_{LR}) be its intersection graph, where L and R are the two partitions. Call s_L and s_R the sizes of a clause in L and R, respectively (by construction, all clauses in the same part have the same size), and let $k_L = s_L - |R|$ and $k_R = s_R - |L|$. The value of $k_L + k_R$ is the number of variables that appear in a single clause. Since the graph is bipartite complete, $k_L, k_R \geq 0$. Obtain a bw-graph $G = (V_1 \cup V_2 \cup V_3 \cup V_4, E, \chi)$ such that

1) $V_1 = \{1, \ldots, k_L\}$, $V_2 = L$, $V_3 = R$ and $V_4 = \{1, \ldots, k_R\}$;
2) All nodes in $V_1 \cup V_4$ are white, and all nodes in $V_2 \cup V_3$ are black;
3) There is an edge connecting (u, v) in E for every $u \in V_1$ and $v \in V_2$, for every $(u, v) \in E_{LR}$, and for every $u \in V_3$ and $v \in V_4$.

We call $\mathcal{B}$ the family of graphs that can obtained by the procedure above. Figure 3(a) depicts an example of a graph in $\mathcal{B}$ obtained by applying the procedure to the formula represented in the Figure 2. By construction, for any two nodes $u, v \in V_i$, $i = 1, \ldots, 4$, it follows that $N_G(u) = N_G(v)$ and $(u, v) \notin E$. The following result shows the equivalence between edge covers and model counting.

Proposition 1. *Consider a LinMonCBPC formula ϕ and $G = (V_1, V_2, V_3, V_4, E, \chi)$ be a corresponding bw-graph in $\mathcal{B}$. Then number of edge covers of G equals the model counting of ϕ, that is, $Z(G) = Z(\phi)$.*

Proof: Let u_i denote the node in G corresponding to a clause ϕ_i in ϕ. Label each edge (u_i, v_j) for $\phi_i \in L$ and $\phi_j \in R$ with the variable corresponding to the intersection of the two clauses. For each $\phi_i \in L$, label each dangling edge (u, u_i) incident in u_i with a different variable that appears only at ϕ_i. Similarly, label each dangling edge (u_j, u) with a different variable that appears only at $\phi_j \in R$. Note that the labeling function is bijective, as every variable in ϕ labels exactly one edge of G.

Now consider a satisfying assignment σ of ϕ and let C be set of edges labeled with the variables X_i such that $\sigma(X_i) = 1$. Then C is an edge cover since every clause (node in G) has at least one variable (incident edge) with $\sigma(X_i) = 1$ and the corresponding edge is in C. To show the converse holds, consider an edge cover C for G, and construct an assignment such that $\sigma(X_i) = 1$ if the edge labeled by X_i is in C and $\sigma(X_i) = 0$ otherwise. Then σ satisfies ϕ, since for every clause ϕ_i (node u_i) there is a variable in ϕ_i with $\sigma(X_i)$ (incident edge in u_i in C). Since there are as many edges as variables, the correspondence between edge covers and satisfying assignment is one-to-one. ■

IV. A Dynamic Programming Approach to Counting Edge Covers

In this section we derive an algorithm for computing the number of edge covers of a graph in $\mathcal{B}$. Let e be an edge and u be a node in $G = (V, E, \chi)$. We define the following operations and notation:

- **edge removal:** $G - e = (V, E \setminus \{e\}, \chi)$.
- **node whitening:** $G - u = (V, E, \chi')$, where $\chi'(u) = 0$ and $\chi'(v) = \chi(v)$ for $v \neq u$.

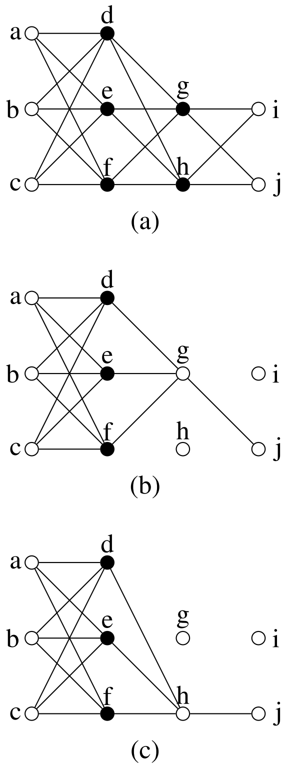


Fig. 3. (a) A graph G in $\mathcal{B}$. (b) The graph $G - E_G(h) - (i,g) - h - g$. (c) The graph $G - E_G(g) - (i,h) - g - h$.

Note that these operations do not alter the node set, and that they are associative (e.g., $G - e - f = G - f - e$, $G - u - v = G - v - u$, and $G - e - u = G - u - e$). Hence, if $E = \{e_1, \cdots, e_d\}$ is a set of edges, we can write $G - E$ to denote $G - e_1 - \cdots - e_d$ applied in any arbitrary order. The same is true for node whitening and for any combination of node whitening and edge removal. These operations are illustrated in the examples in Figure 3.

The following result shows that the number of edge covers can be computed recursively on smaller graphs:

Proposition 2. *Let $e = (u, v)$ be a dangling edge with u colored black. Then:*

$$Z(G) = 2Z(G - e - u) - Z(G - E_G(u) - u) \,.$$

Proof: There are $Z(G - e - u)$ edge covers of G that contain e and $Z(G - e)$ edge covers that do not contain e. Hence, $Z(G) = Z(G - e - u) + Z(G - e)$. Now, consider the graph $G' = G - e - u$. There are $Z(G - e)$ edge covers of G' that contain at least one edge of $E_{G'}(u)$ and $Z(G - E_G(u) - u)$ edge covers that contain no edge of $E_{G'}(u)$. Thus $Z(G - e - u) = Z(G - e) + Z(G - E_G(u) - u)$. Substituting $Z(G - e)$ in the first identity gives us the desired result. ■

Free edges and isolated white nodes can be removed by adjusting the edge count correspondingly:

Proposition 3. *We have:*

1) *Let $e = (u, v)$ be a free edge of G. Then $Z(G) = 2Z(G - e)$.*
2) *If u is an isolated white node (i.e., $N_G(u) = \emptyset$) then $Z(G) = Z(G - u)$.*

Proof: (1) If C is an edge cover of $G - e$ then both C and $C \cup \{e\}$ are edge covers of G. Hence, the number of edge covers containing e equals the number $Z(G - e)$ of edge covers not containing e. (2) Every edge cover of G is also an edge cover of $G - u$ and vice-versa. ■

We can use the formulas in Propositions 2 and 3 to compute the edge cover count of a graph recursively. Each recursion computes the count as a function of the counts of two graphs obtained by the removal of edges and whitening of nodes. Such a naive approach requires an exponential number of recursions (in the number of edges or nodes of the initial graph) and finishes after exponential time. We can transform such an approach into a polynomial-time algorithm by exploiting the symmetries of the graphs produced during the recursions. In particular, we take advantage of the invariance of edge cover count to isomorphisms of a graph, as we discuss next.

We say that two bw-graphs $G = (V, E, \chi)$ and $G' = (V', E', \chi')$ are *isomorphic* if there is a bijection γ from V to V' (or vice-versa) such that (i) $\chi(v) = \chi'(\gamma(v))$ for all $v \in V$, and (ii) $(u, v) \in E$ if and only if $(\gamma(u), \gamma(v)) \in E'$. In other words, two bw-graphs are isomorphic if there is a color-preserving renaming of nodes that preserves the binary relation induced by E. The function γ is called an *isomorphism* from V to V'. The graphs in Figures 3(b) and 3(c) are isomorphic by an isomorphism that maps g in h and maps any other node into itself. If C is an edge cover of G and γ is an isomorphism between G and G', then $C' = \{(\gamma(u), \gamma(v)) : (u, v) \in C\}$ is an edge cover for G' and vice-versa. Hence, $Z(G) = Z(G')$. The following result shows how to obtain isomorphic graphs with a combination of node whitenings and edge removals.

Proposition 4. *Consider a bw-graph G with nodes $v_1, \ldots, v_n$ such that $N_G(v_1) = \cdots = N_G(v_n) \neq \emptyset$ and $\chi_G(v_1) = \cdots = \chi_G(v_n)$. For any node $w \in N_G(v_1)$, mapping $\gamma : \{v_1, \ldots, v_n\} \to \{v_1, \ldots, v_n\}$, and nonnegative integers k_1 and k_2 such that $k_1 + k_2 \leq n$ the graphs $G' = G - E_G(v_1) - \cdots - E_G(v_{k_1}) - (w, v_{k_1+1}) - \cdots - (w, v_{k_1+k_2}) - v_1 - \cdots - v_{k_1+k_2}$ and $G'' = G - E_G(\gamma(v_1)) - \cdots - E_G(\gamma(v_{k_1})) - (w, \gamma(v_{k_1+1})) - \cdots - (w, \gamma(v_{k_1+k_2})) - \gamma(v_1) - \cdots - \gamma(v_{k_1+k_2})$ are isomorphic.*

Proof: Let γ' be the bijection on the nodes of G that extends γ, that is, $\gamma'(u) = u$ for $u \notin \{v_1, \ldots, v_n\}$ and $\gamma'(u) = \gamma(v_i)$, for $i = 1, \ldots, n$. We will show that γ' is an isomorphism from G' to G''. First note that $\chi_G(u) = \chi_G(\gamma(u))$ for every node u. The only nodes that have their color (possibly) changed in G' with respect to G are the nodes $v_1, \ldots, v_{k_1+k+2}$, and these are white nodes in G'. Likewise, the only nodes that would (possibly) changed color in G'' were $\gamma(v_1), \ldots, \gamma(v_{k_1+k_2})$ and these are white in G''. Hence, $\chi_{G'}(u) = \chi_{G''}(\gamma(u))$ for every node u.

Now let us look at the edges. First note that since $N_G(v_i)$ is constant through $i = 1, \ldots, n$, G' and G'' have the same number of edges. Hence, it suffices to show that for each edge (u, v) in G' the edge $(\gamma'(u), \gamma'(v))$ is in G''. The only edges modified in obtaining G' and G'' are, respectively, those incident in $v_1, \ldots, v_{k_1+k_2}$ and in $\gamma(v_1), \ldots, \gamma(v_{k_1+k_2})$. Consider an edge (u, v) where $u, v \notin \{v_1, \ldots, v_n\}$ (hence not in $E_G(v_i)$ for any i). If $(u, v) = (\gamma'(u), \gamma'(v))$ is in G' then it is also in G''. Now consider an edge (u, v_i) in G where $u \notin \{w, v_{k_1+1}, \ldots, v_n\}$ and $k_1 < i \leq k_1 + k_2$. Then (u, v_i) is in G' and $(\gamma'(u), \gamma'(v_i))$ is in G''. Note that u could be in $N_G(v_i)$ for $k_1 + k_2 < i \leq n$. ■

According to the proposition above, the graphs in Figures 3(b) and 3(c) are isomorphic by a mapping from g to h (and with $w = i$). Hence, the number of edge covers in either graph is the same.

The algorithms RightRecursion and LeftRecursion described in Figures 4 and 5, respectively, exploit the isomorphisms described in Proposition 4 in order to achieve polynomial-time behavior when using the recursions in Propositions 2 and 3. Either algorithm requires a base white node w and integers k_1 and k_2 specifying the recursion level (with the same meaning as in Proposition 4). Unless $k_1 + k_2$ equals the number of neighbors of w in the original graph, a call to either algorithm generates two more calls to the same algorithm: one with the graph obtained by removing edge (w, v_h) and whitening v_h, and another by removing edges $E(v_h)$ and whitening v_h. Assume that $|V_2| \geq |V_3|$ (if $|V_3| > |V_2|$ we can simply manipulate node sets to obtain an isomorphic graph satisfying the assumption). The RightRecursion algorithm first checks whether the value for the current recursion level has been already computed; if yes, then it simply returns the cached value; otherwise it uses the formula in Proposition 2 (and possibly the isomorphism in Proposition 4) and generates two calls of the same algorithm on smaller graphs (i.e. with fewer edges) to compute the edge cover counting for the current graph and stores the result in memory. The recursion continues until the recursion levels equates with the number of nodes in V_3, in which case it checks for free edges and isolated nodes, removes them and computes the correction factor 2^k, where k is the number of free edges, and calls the algorithm LeftRecursion to start a new recursion. At this point the graph in the input is bipartite complete and contains only nodes in V_1 and V_2. The latter algorithm behaves very similarly to the former except at the termination step. When all neighbors v_h of w have been whitened the graph no longer contains black nodes, and the corresponding edge cover count can be directly computed using the formulas in Proposition 3. Note that a different cache function must be used when we call LeftRecursion from RightRecursion (this can be done by instantiating an object at that point and passing it as argument; we avoid stating the algorithm is this way to avoid cluttering).

Note that the algorithms do not use the color of nodes, which hence does not need to be stored or manipulated. In fact the node whitening operations ($-v_h$ or $-u_h$) performed when calling the recursion are redundant and can be neglected without altering the soundness of the procedure (we decided to leave these operations as they make the connection with Proposition 2 more clear).

Figure 6 shows the recursion diagram of a run of RightRecursion. Each box in the figure represents a call of the algorithm with the graph drawn as input. The left child of each box is the call RightRecursion$(G - (v_h, w) - v_h, w, k_1, k_2 + 1)$, and the right child is the call RightRecursion$(G - E_G(v_h) - v_h, w, k_1 + 1, k_2)$. For instance, the topmost box represents RightRecursion$(G_0, w, 0, 0)$, which computes $Z(G_0)$ as the sum of $2Z(G_1)$ and $-Z(G_{24})$, which are obtained, respectively, from the calls corresponding to its left and right children. The number of the graph in each box corresponds to the order in which each call was generated. Solid arcs represent non cached calls, while dotted arcs indicate cached calls. For instance, by the time RightRecursion$(G_{24}, w, 1, 0)$ is called, RightRecursion$(G_{13}, w, 0, 0)$ has already been computed so the value of $Z(G_{13})$ is simply read from memory and returned. When called in the graph in the top, with to rightmost node w, and integers $k_1 = k_2 = 0$, the algorithm computes the partition function $Z(G_0)$ as the sum of $2Z(G_1)$ and $-Z(G_{24})$, where G_1 is obtained from the removal of edge (v_1, w) and whitening of v_1, while G_{24} is obtained by removing edges $E_{G_1}(v_1)$ and whitening of v_1. The recursion continues until all incident edges on w have been removed, at which point it removes free edges and isolated nodes and calls LeftRecursion. The recursion diagram for the call of LeftRecursion$(G_4, w, 0, 0)$ where w is the top leftmost node of G_4 in the figure is shown in Figure 7. The semantics of the diagram is analogous. Note

```
1: if Cache($w, k_1, k_2$) $> 0$ then
2:   return Cache($w, k_1, k_2$)
3: else
4:   if $k_1 + k_2 < m$ then
5:     Let $h \leftarrow k_1 + k_2 + 1$
6:     Cache($w, k_1, k_2$) $\leftarrow$ $2 \times$ RightRecursion($G - (v_h, w) - v_h, w, k_1, k_2 + 1$) $-$ RightRecursion($G - E_G(v_h) - v_h, w, k_1 + 1, k_2$)
7:     return Cache($w, k_1, k_2$)
8:   else
9:     Let $k = |\{(u, v) : u \in V_4\}|$ be the number of free edges
10:    Remove any edges with an endpoint in $V_4$ and all the resulting isolated nodes
11:    Set $V_1 \leftarrow V_1 \cup V_3$, $V_3 \leftarrow \emptyset$
12:    if $V_1$ is empty then
13:      return 0
14:    end if
15:    Select an arbitrary $w' \in V_1$
16:    return $2^k \times$ LeftRecursion($G, w', 0, 0$)
17:  end if
18: end if
```

Fig. 4. Algorithm RightRecursion: Takes a graph $G = (V_1, V_2, V_3, V_4, E)$ with $V_3 = \{v_1, \ldots, v_m\}$, $m > 0$, a node $w \in V_4$, and nonnegative integers k_1 and k_2; outputs $Z(G)$.

```
1: if Cache($w, k_1, k_2$) is undefined then
2:   if $k_1 + k_2 < n$ then
3:     Let $h \leftarrow k_1 + k_2 + 1$
4:     Cache($w, k_1, k_2$) $\leftarrow$ $2 \times$ LeftRecursion($G - (u_h, w) - u_h, k_1, k_2 + 1$) $-$ LeftRecursion($G - E_G(u_h) - u_h, k_1 + 1, k_2$)
5:   else
6:     Cache($w, k_1, k_2$) $\leftarrow 2^{|E|}$
7:   end if
8: end if
9: return Cache($w, k_1, k_2$)
```

Fig. 5. Algorithm LeftRecursion: Takes a bipartite graph $G = (V_1, V_2, E)$ with $V_2 = \{u_1, \ldots, u_n\}$, $n > 0$, a node $w \in V_1$, nonnegative integers k_1 and k_2; outputs $Z(G)$.

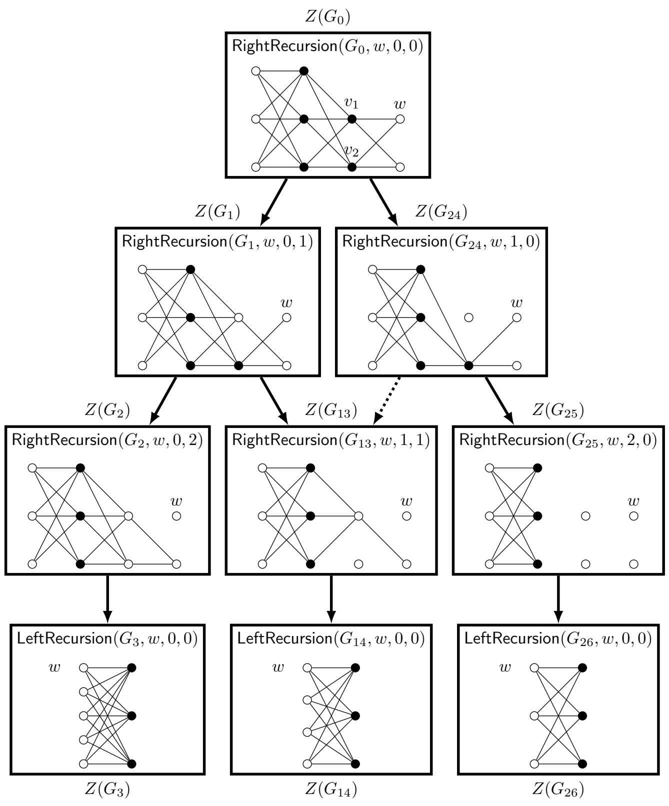

$Z(G_0)$
RightRecursion$(G_0, w, 0, 0)$
v_1
w
v_2
$Z(G_1)$
$Z(G_{24})$
RightRecursion$(G_1, w, 0, 1)$
RightRecursion$(G_{24}, w, 1, 0)$
w
w
$Z(G_2)$
$Z(G_{13})$
$Z(G_{25})$
RightRecursion$(G_2, w, 0, 2)$
RightRecursion$(G_{13}, w, 1, 1)$
RightRecursion$(G_{25}, w, 2, 0)$
w
w
w
LeftRecursion$(G_3, w, 0, 0)$
LeftRecursion$(G_{14}, w, 0, 0)$
LeftRecursion$(G_{26}, w, 0, 0)$
w
w
w
$Z(G_3)$
$Z(G_{14})$
$Z(G_{26})$

Fig. 6. Simulation of RightRecursion$(G_0, w, 0, 0)$.

that the recursion of LeftRecursion eventually reaches a graph with no black nodes, for which the edge cover count can be directly computed (in closed-form).

In these diagrams, it is possible to see how the isomorphisms stated in Proposition 4 are used by the algorithms and lead to polynomial-time behavior. For instance, in the run in Figure 6, the graph G_{13} is not the graph obtained from G_{24} by removing edge (v_2, w) and whitening v_2 but instead is isomorphic to it. Note that both G_{13} and its isomorphic graph obtained as the left child of G_{24} were obtained by one operation of edge removal $-(w, v_i)$ and one operation of neighborhood removal $-E(v_i)$, plus node whitenings of v_1 and v_2. Hence, Proposition 4 guarantees their isomorphism.

The polynomial-time behavior of the algorithms strongly depends on caching the calls (dotted arcs) and exploiting known isomorphisms. For instance, in the run in Figure 6, the graph G_{13} is not the graph obtained from G_{24} by removing edge (v_2, w) and whitening v_2 but instead is isomorphic to it. Note that both G_{13} and its isomorphic graph obtained as the left child of G_{24} were obtained by one operation of edge removal (w, v_i) and one operation of neighborhood removal $E(v_i)$, plus node whitenings of v_1 and v_2. Hence, Proposition 4 guarantees their isomorphism.

Without the caching of computations, the algorithm would perform exponentially many recursive calls (and its corresponding diagram would be a binary tree with exponentially many nodes). The use of caching allows us to compute only one call of RightRecursion for each configuration of k_1, k_2 such that $k_1 + k_2 \leq n$, resulting in at most $\sum_{i=0}^{n}(i + 1) = (n + 1)(n + 2)/2 = O(n^2)$ calls for RightRecursion, where $n = |V_3|$. Similarly, each call of LeftRecursion requires at most $\sum_{i=0}^{m}(i+1) = (m+1)(m+2)/2 = O(m^2)$ recursive calls for LeftRecursion, where $m = |V_2|$. Each call to RightRecursion with $k_1 + k_2 = n$ generates a call to LeftRecursion (there are $n + 1$ such configurations). Hence, the overall number of recursions (i.e., call to either function) is

$$\frac{(n+1)(n+2)}{2} + (n+1)\frac{(m+1)(m+2)}{2} = O(n^2 + n \cdot m^2)\,.$$

This leads us to the following result.

Theorem 1. *Let G be a graph in $\mathcal{B}$ with $w \in V_4 \neq \emptyset$. Then* RightRecursion$(G, w, 0, 0)$ *outputs $Z(G)$ in time and memory at most cubic in the number of nodes of G.*

Proof: Except when $k_1 + k_2 = n$, RightRecursion calls the recursion given in Proposition 2 with the isomorphisms in Proposition 4 (any graph obtained from G by k_1 operations $-E_G(v_i)$ and k_2 operations $-(w, v_i)$ are isomorpohic). For k_1+k_2, any edge left connecting a node in V_3 and a node in V_4 must be a free edge (since all nodes in V_4 have been whitened), hence they can be removed according to Proposition 3 with the appropriate correction of the count. By the same result, any isolated node can be removed. When the remaining nodes in V_3 are transfered to V_1, the resulting graph is bipartite complete (with white nodes in one part and black nodes in the other). Hence, we can call LeftRecursion, which is guaranteed to compute the correct count by the same arguments.

The cubic time and space behavior is due to RightRecursion and LeftRecursion being called at most $O(n^2)$ and $O(nm^2)$, respectively, and by the fact that each call consists of local operations (edge removals and node whitenings) which take at most linear time in the number of nodes and edges of the graph. ∎

V. Graphs With No Dangling Edges

The algorithm RightRecursion requires the existence of a dangling edge. Now it might be that the graph contains no white nodes (hence no dangling edges), that is, that G is bipartite complete graph for $V_2 \cup V_3$. The next result shows how to decompose the problem of counting edge covers in smaller graphs that either contain dangling edges, or are also bipartite complete.

Proposition 5. *Let G be a bipartite complete bw-graph with all nodes colored black and $e = (u, v)$ be some edge. Then $Z(G) = 2Z(G - e - u - v) - Z(G - E_G(v) - v) - Z(G - E_G(u) - u) - Z(G - E_G(u) - E_G(v) - u - v)$.*

Proof: The edge covers of G can be partitioned according to whether they contain the edge e. The number of edge covers that contain e is not altered if we color both u and v white. Thus, $Z(G) = Z(G - e - u - v) + Z(G - e)$. Let $e_1, \ldots, e_n$ be the edges incident in u other than e, and $f_1, \ldots, f_m$ be the edges incident in v other than v. We have that $Z(G - e - u - v) = Z(G - e - u - v) + Z(G - E_G(u) - u) + Z(G - e) + Z(G - E_G(u) - E_G(v) - u - v)$. Substituting $Z(G - e)$ into the first equation obtains the result. ∎

In the result above, the graphs $G - e - u - v$, $G - E_G(v) - v$ and $G - E_G(u) - u$ are in $\mathcal{B}$ and contain dangling edges,

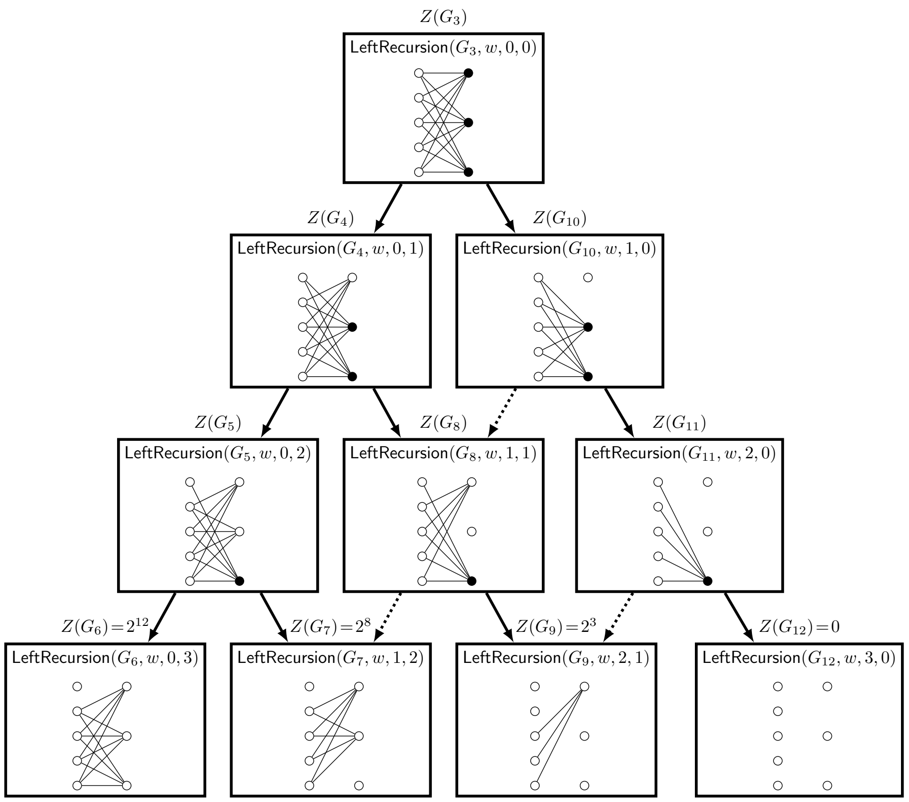

$Z(G_3)$
$\mathsf{LeftRecursion}(G_3, w, 0, 0)$
$Z(G_4)$
$Z(G_{10})$
$\mathsf{LeftRecursion}(G_4, w, 0, 1)$
$\mathsf{LeftRecursion}(G_{10}, w, 1, 0)$
$Z(G_5)$
$Z(G_8)$
$Z(G_{11})$
$\mathsf{LeftRecursion}(G_5, w, 0, 2)$
$\mathsf{LeftRecursion}(G_8, w, 1, 1)$
$\mathsf{LeftRecursion}(G_{11}, w, 2, 0)$
$Z(G_6) = 2^{12}$
$Z(G_7) = 2^8$
$Z(G_9) = 2^3$
$Z(G_{12}) = 0$
$\mathsf{LeftRecursion}(G_6, w, 0, 3)$
$\mathsf{LeftRecursion}(G_7, w, 1, 2)$
$\mathsf{LeftRecursion}(G_9, w, 2, 1)$
$\mathsf{LeftRecursion}(G_{12}, w, 3, 0)$

Fig. 7. Simulation of $\mathsf{LeftRecursion}(G_3, w, 0, 0)$.

while the graph $G - E_G(u) - E_G(v) - u - v$ is bipartite complete. Note that Proposition 4 can be applied to show that altering the edges on which the operations are applied lead to isomorphic graphs. A very similar algorithm to LeftRecursion, implementing the recursion in the result above in polynomial-time can be easily derived.

VI. Extensions

Previous results can be used beyond the class of graphs $\mathcal{B}$. For instance, the algorithms can compute the edge cover count for any graph that can be obtained from a graph G in $\mathcal{B}$ by certain sequences of edge removals and node whitenings, which includes graphs not in $\mathcal{B}$. Graphs that satisfy the properties of the class $\mathcal{B}$ except that every node in V_2 (or V_4 or both) are pairwise connected can also have their edge cover count computed by the algorithm (as this satisfies the conditions in Proposition 4). Another possibility is to consider graphs which can be decomposed in graphs $\mathcal{B}$ by polynomially many applications of Proposition 2.

We can also consider more general forms of counting problems. A simple mechanism for randomly generating edge covers is to implement a Markov Chain with starts with some trivial edge cover (e.g. one containing all edges) and moves from an edge cover X_t to an edge cover X_{t+1} by the following Glauber Dynamics-type move: (1) Select an edge e uniformly at random; (2a) if $e \notin X_t$, make $X_{t+1} = X_t \cup \{e\}$ with probability $\lambda/(1+\lambda)$; (2b) if $e \in X_t$ and if $X_t \setminus \{e\}$ is an edge cover, make $X_{t+1} = X_t \setminus \{e\}$ with probability $1/(1+\lambda)$; (2c) else make $X_{t+1} = X_t$. The above Markov chain can be shown to be ergodic and to converge to a stationary distribution which samples an edge cover C with probability $\lambda^{|C|}$ [14], [15]. When $\lambda = 1$, the algorithm performs uniform sampling of edge covers. A related problem is to compute the total probability mass that such an algorithm will assign to sets of edge covers given a bw-graph G, the so-called *partition function*: $Z(G, \lambda) = \sum_{C \in \mathsf{EC}(G)} \lambda^{|C|}$, defined for any real $\lambda > 0$, where $\mathsf{EC}(G)$ is the set of edge covers of G. For $\lambda = 1$ the problem is equivalent to counting edge covers. This is also equivalent to weighted model counting of LinMonCBPC formulas with uniform weight λ.

The following results are analogous to Propositions 2 and 3 for computing the partition function:

Proposition 6. *The following assertions are true:*

1) *Let $e = (u, v)$ be a free edge of G. Then $Z(G) = (1+\lambda)Z(G - e)$.*
2) *If u is an isolated white node (i.e., $N_G(u) = \emptyset$) then $Z(G) = Z(G - u)$.*
3) *Let $e = (u, v)$ be a dangling edge with u colored black. Then $Z(G) = (1+\lambda)Z(G - e - u) - Z(G - E_G(u) - u)$.*

Hence, a straightforward modification of algorithms RightRecursion and LeftRecursion (modifying the weights by which the the recursive calls are multiplied) enables the algorithms to compute the partition function of graphs in $\mathcal{B}$ (or equivalently, the partition function of LinMonCBPC formulas).

VII. Model Counting of DL-Lite Formulas

The original motivation for us studying the problem of model counting of LinMonCBPC formulas was to count satisfying interpretations of DL-Lite sentences, which is a class of description logics which efficient reasoning services [16], [17]. We now summarize some points about this problem, as it is an example of application for our results.

Descriptions Logics are usually decidable fragments of first-order logic that can be used to specify and reason about ontologies. Indeed, the OWL language, which has been adopted as the standard for the semantic web, is largely based on description logics. The DL-Lite logic is a simple description logic developed to enable efficient reasoning with large amounts of linked and knowledge-enriched data. As with all description languages, the basic components of DL-Lite are sets of symbols for concepts, roles and constants, and set operations such as intersection ($\sqcap$), complementation ($\neg$) and existential quantification ($\exists$). Concepts represent properties of individuals, while roles represent relationships between two individuals. Mathematically, concepts and roles are, respectively, unary and binary relations. As an example, the definition of a father as man who is a parent of someone can be expressed in DL-Lite as $\mathsf{Man} \sqcap \exists \mathsf{parentOf}$. DL-Lite also adopts *inverse roles*: if r is a role then its inverse r^{-1} is a binary relation such that $(y, x) \in \mathsf{r}^{-1}$ if and only if $(x, y) \in \mathsf{r}$. For example, the definition of a son can be expressed as $\mathsf{Man} \sqcap \exists \mathsf{parentOf}^{-1}$. A DL-Lite sentence is inductively defined such that a concept name and its complement is a sentence, $\exists \mathsf{r}$ and $\exists \mathsf{r}^{-1}$ are sentences for every role r, and if ϕ and ψ are sentences then $\phi \sqcap \psi$ are also sentences.

The semantics of a DL-Lite sentence is given by a domain Δ, which contains a number of individuals, and an interpretation $\mathcal{I}$, which maps each constant to an individual in the domain, each concept to a set of individuals, and each role to a set of pairs of individuals. The interpretation thus assigns which individuals have certain properties (and which do not), and which relationships hold (and which do not). An interpretation is extended to a formula inductively: $a \in \mathcal{I}(\neg \mathsf{C})$ if $a \notin \mathcal{I}(\mathsf{C})$, $a \in \mathcal{I}(\mathsf{C} \sqcap \mathsf{D})$ if $a \in \mathcal{I}(\mathsf{C}) \cap \mathcal{I}(\mathsf{D})$, $a \in \mathcal{I}(\exists \mathsf{r})$ if there is $b \in \Delta$ such that $(a, b) \in \mathcal{I}(\mathsf{r})$ and $a \in \mathcal{I}(\exists \mathsf{r}^{-1})$ if there is $b \in \Delta$ such that $(b, a) \in \mathcal{I}(\mathsf{r}^{-1})$. A sentence DL-Lite ϕ is satisfied by an interpretation if $\mathcal{I}(\phi) \neq \emptyset$. In a recent paper, we showed that counting the number of satisfying interpretations of a DL-Lite sentence can be reduced to model counting of LinMonCBPC formulas [18].

VIII. Conclusion

Model counting is the problem of computing the number of satisfying assignments of Boolean formulas, with applications in several fields. The problem cannot be solved in polynomial time (unless #P collapses to P); even though tractable subclasses of the problem have been found, finding restrictions that make the problem tractable and yet interesting is difficult.

In this work, we developed a polynomial-time algorithm for solving the problem in the restrictive class of monotone conjunctive normal formulas where the clauses can be partitioned in two sets such that the clauses in each set have the same number of variables, no two clauses in the same set share a variable, and every clause in one set shares exactly one variable with every clause in the other set. We showed that such problem can be efficiently reduced to counting edge covers in a specific type of graphs, which can then be solved by dynamic programming in quadratic time. The formulas we consider can be used to compute the number of satisfying interpretations in knowledge expressed in DL-Lite, a popular description logic.

It would be interesting to investigate whether it is possible to relax any of the assumptions without leading into #P-complete problems. This is left to future work.

Acknowledgment

The second author is partially supported by CNPq.

References

[1] F. Bacchus, S. Dalmao, and T. Pitassi, "Solving #SAT and Bayesian inference with backtracking search," *Journal of Artificial Intelligence Research*, vol. 34, pp. 391–442, 2009.

[2] A. Darwiche, *Modeling and reasoning with Bayesian networks.* Cambridge University Press, 2009.

[3] H. Palacios, B. Bonet, A. Darwhice, and H. Geffner, "Pruning conformant plans by counting models on compiled d-DNNF representations," in *Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, 2005, pp. 141–150.

[4] L. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal of Computing*, vol. 8, pp. 410–421, 1979.

[5] J. Provan and M. Ball, "The complexity of counting cuts and of computing the probability that a graph is connected," *SIAM Journal of Computing*, vol. 12, pp. 777–788, 1983.

[6] S. P. Vadhan, "The complexity of counting in sparse, regular and planar graphs," *SIAM Journal of Computing*, vol. 31, no. 2, pp. 398–427, 2001.

[7] D. Roth, "On the hardness of approximate reasoning," *Artificial Intelligence*, vol. 82, pp. 273–302, 1996.

[8] R. Bubley and M. Dyer, "Graph orientations with no sink and an approximation for a hard case of #SAT," in *Proc. of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1997, pp. 248–257.

[9] M. Xia and W. Zhao, "#3-regular bipartite planar vertex cover is #P-complete," in *Theory and Applications of Models of Computation*, ser. Lecture Notes in Computer Science, 2006, vol. 3959, pp. 356–364.

[10] L. Valiant, "Accidental algorithms," in *Proc. of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006, pp. 509–517.

[11] C. Lin, J. Liu, and P. Lu, "A simple FPTAS for counting edge covers." in *Proc. of 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2014, pp. 341–348.

[12] J. Liu, P. Lu, and C. Zhang, "FPTAS for counting weighted edge covers," in *Proc. of the 22nd Annual European Symposium on Algorithms (ESA)*, 2014, pp. 654–665.

[13] J.-Y. Cai, P. Lu, and M. Xia, "Holographic reduction, interpolation and hardness," *Computational Complexity*, vol. 21, no. 4, pp. 573–604, 2012.

[14] M. Bordewich, M. Dyer, and M. Karpinski, "Path coupling using stopping times," in *Proc. of the 15th International Symposium Fundamentals of Computation Theory (FCT)*, 2005, pp. 19–31.

[15] I. Bezáková and W. Rummler, "Sampling edge covers in 3-regular graphs," in *Proc. of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2009, pp. 137–148.

[16] D. Calvanese, G. De Giacomo, M. Lembo, D. abd Lenzerini, and R. Rosati, "DL-lite: Tractable description logics for ontologies," in *Proc. of the AAAI Conference (AAAI)*, 2005, pp. 602–607.

[17] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyashev, "The DL-lite family and relations," *Journal of Artificial Intelligence Research*, vol. 36, pp. 1–69, 2009.

[18] D. Mauá and F. Cozman, "DL-lite Bayesian networks: A tractable probabilistic graphical model," in *Proc. of the 9th International Conference on Scalable Uncertainty Models (SUM)*, 2015, (accepted).