Algebras for tree decomposable graphs[⋆]

Roberto Bruni[1] [0000−0002−7771−4154] ✉, Ugo Montanari[1] [0000−0002−6204−8670], and Matteo Sammartino[2] [0000−0003−1456−2242]

[1] Dipartimento di Informatica, Università di Pisa, Italia
{bruni,ugo}@di.unipi.it

[2] Royal Holloway University of London, University College London, London, UK
matteo.sammartino@rhul.ac.uk

Abstract. Complex problems can be sometimes solved efficiently via recursive decomposition strategies. In this line, the *tree decomposition* approach equips problems modelled as graphs with tree-like parsing structures. Following Milner's *flowgraph algebra*, in a previous paper two of the authors introduced a *strong network* algebra to represent open graphs (up to isomorphism), so that homomorphic properties of open graphs can be computed via structural recursion. This paper extends this graphical-algebraic foundation to tree decomposable graphs. The correspondence is shown: (i) on the algebraic side by a *loose network* algebra, which relaxes the restriction reordering and scope extension axioms of the strong one; and (ii) on the graphical side by Milner's *binding bigraphs*, and *elementary* tree decompositions. Conveniently, an interpreted loose algebra gives the evaluation complexity of each graph decomposition. As a key contribution, we apply our results to dynamic programming (DP). The initial statement of the problem is transformed into a term (this is the secondary optimisation problem of DP). Noting that when the scope extension axiom is applied to reduce the scope of the restriction, then also the complexity is reduced (or not changed), only so-called canonical terms (in the loose algebra) are considered. Then, the canonical term is evaluated obtaining a solution which is locally optimal for complexity. Finding a global optimum remains an NP-hard problem.

Keywords: tree decomposition · bigraphs · graph algebras.

1 Introduction

Many quite relevant complex graph problems become easy for specific classes of graphs. Usually these graphs are equipped with a suitable recursive structure which allows to compute the solution by problem reduction. The typical structure studied in the literature is *tree decomposition* [14,18,3,4,7,11]. Another suggestive approach is to consider a hyperedge replacement grammar [5], where the structure of a derived graph is its derivation tree.

⋆ Research supported by MIUR PRIN 201784YSZ5 *ASPRA*, by Univ. of Pisa PRA_2018_66 *DECLWARE*, and by EPSRC Grant *CLeVer* (EP/S028641/1).

In [17], a *strong network* algebra, called *Soft Constraint Evaluation Problems* (SCEP), is introduced (see also [12]). The algebra has operations of parallel composition, node restriction and permutation and, in particular, has the axioms of restriction reordering and of scope extension. Equivalence classes of strong terms correspond exactly to open graphs up to isomorphism, which thus can be seen as standard representatives of the classes. Consequently, homomorphic properties of open graphs can be conveniently computed via structural recursion. While two strongly equivalent graphs evaluate to the same result by construction, it may happen that the complexities of their evaluations be vastly different. To represent explicitly similar additional information, it is convenient to introduce a finer graphical-algebraic initial pair of models. For instance, an algebra for the computational complexity of problems should fit the new axiomatisation.

In this paper, we choose *elementary* tree decompositions, a simple variant of the classical tree decomposition approach, as the reference graphical model. Interestingly, for obtaining an adequate algebraic model it is enough to eliminate the axioms of restriction reordering and of scope extension from the strong specification. The resulting specification is called *loose*. In [17], an alternative version of algebraic model was chosen, by eliminating only the axiom of scope extension, and called *weak*. The present axiomatisation is needed in order to achieve a tighter correspondence with tree decompositions.

Here we also consider Milner's *bigraphs* [16], a widely studied graphical model for process calculi. Once equipped with a suitable signature, bigraphs can be put into bijective correspondence with equivalence classes of loose terms: the *link* graph represents the variables, and the *place* graph the nesting of restrictions.

As in the strong case, the existence of graphical standard representatives for the initial algebra makes it easy to define interesting algebras of the class. For instance, the evaluation complexity of a term can be easily computed by its interpretation within a simple loose algebra. Notably, the reverse application of the scope extension axiom (i.e. aimed to reduce the scope of the restriction) reduces, or does not change, the evaluation complexity of a term. Thus minimal complexity must be achieved by terms which are fully reduced with respect to the extension axiom (they are called *canonical*) and search for optimal evaluations can be restricted to canonical terms. To take advantage of this property, we define a type system where only canonical terms are typeable.

An algebra of graphs of special interest naturally arises in the case of dynamic programming (DP) [1]. DP usually consists of minimising a cost function F of variables X while keeping variables Y as parameters, i.e. $F(Y) = \min_X F(X,Y)$. Typically, function F has the form

$$F(X,Y) = F_1(X_1,Y_1) + ... + F_n(X_n,Y_n).$$

where each function F_1,...,F_n is dependent only on a few variables. The key issue is how the variables in X and Y are used in F_1,...,F_n. The sharing structure can be conveniently represented by a hypergraph F where nodes are variables and hyperedges are labelled by functions F_i, represented as multidimensional tables.

The evaluation procedure corresponds to compute by structural recursion the optimal cost of F, where values are multidimensional tables representing intermediate functions, parallel composition is sum, constants are hyperedge labels and restriction with respect to a variable x, $[\![(x)F(X,Y)]\!]$, corresponds to eliminate variable x in table $F(X,Y)$: $\min_x [\![F(X,Y)]\!]$. In conclusion, the solution of an optimisation problem via dynamic programming consists of two steps: (i) find a canonical loose term of low complexity for F; and (ii) evaluate the term.

To compute the complexity, it is enough to define a loose homomorphism where parallel composition $F_1|F_2$ is the max of the complexities of F_1 and F_2 and of the number of free variables of $F_1|F_2$, a constant is the number of variables in the corresponding hyperedge, and restriction is the identity. Notice that since table handling is typically of exponential complexity with respects to dimension, the complexity of a sequence of steps is assumed to be just the max.

The computational cost of (ii) typically depends on the chosen term, but not on the values stored in the tables. Thus the chosen term determines the complexity of step (ii). This property allows to separate the two optimisation procedures: the first step is called the *secondary optimisation problem* of DP [2]. Unfortunately, the secondary problem is NP hard [21], thus typically it is convenient to solve it exactly only if the evaluation must be executed many times.

Structure of the paper. In Section 2 we briefly recall some basic notions and some results from [17]. The original contribution starts in Section 3, where we present the loose network specification and draw the graphical-algebraic correspondence with binding bigraphs. Section 4 focuses on dynamic programming, tree decomposition and canonical form, showing how to move from one to the other. There we also define the (loose) algebra we introduced above for computing the evaluation complexity. A simple type system characterises canonical forms. Finally, it is shown that all and only canonical forms are computed by an algorithm based on bucket elimination. Concluding remarks are in Section 5.

2 Background

Notation. Given a set V we denote by $V^\star$ the set of (finite) sequences over V and we let $|\cdot|$ return the length of a sequence. Given a function $f\colon V_1 \to V_2$ we overload the symbol f to denote also its lifting $f\colon V_1^\star \to V_2^\star$, defined elementwise.

Hypergraphs. A *ranked alphabet* $\mathcal{E}$ is a set where each element $e \in \mathcal{E}$ has an arity $\mathsf{ar}(e) \in \mathbb{N}$. A *labelled hypergraph* over a ranked alphabet $\mathcal{E}$ is a tuple $G = (V_G, E_G, a_G, lab_G)$, where: V_G is the set of vertices (also called nodes); E_G is the set of (hyper)edges; $a_G\colon E_G \to V_G^\star$ assigns to each hyperedge e the sequence of nodes attached to it; $lab_G\colon E_G \to \mathcal{E}$ is a labeling function, assigning a label to each hyperedge e such that $|a_G(e)| = \mathsf{ar}(lab_G(e))$.

Given two hypergraphs G_1 and G_2 over $\mathcal{E}$, a *homomorphism* between them is a pair of functions $h = (h_V\colon V_{G_1} \to V_{G_2}, h_E\colon E_{G_1} \to E_{G_2})$ preserving connectivity and labels, namely: $h_V \circ a_{G_1} = a_{G_2} \circ h_E$ and $lab_{G_2} \circ h_E = lab_{G_1}$. We say that G_1 and G_2 are *isomorphic*, denoted $G_1 \cong G_2$, whenever there exists a homomorphism between them which is a component-wise isomorphism. We write $G_1 \uplus G_2$ for the component-wise disjoint union of G_1 and G_2.

Permutation algebras. Given a countable set of variables $\mathbb{V}$, we write $\mathit{Perm}(\mathbb{V})$ for the set of finite permutations over $\mathbb{V}$, i.e., bijective functions $\pi\colon \mathbb{V} \to \mathbb{V}$. A *permutation algebra* is an algebra for the signature comprising all finite permutations and the formal equations $x\ \mathsf{id} = x$ and $(x\ \pi_1)\ \pi_2 = x\ (\pi_2 \circ \pi_1)$.

2.1 Strong network algebras

In [15] Milner introduced an *algebra of flowgraphs* defined by simple axioms. Here we introduce an *algebra of networks* that has essentially the same axioms, but that exploits a nominal structure for nodes. Hereafter we fix a ranked alphabet $\mathcal{E}$ and a countable set of variables $\mathbb{V}$. We also assume functions $\mathsf{var}\colon \mathcal{E} \to \mathbb{V}^\star$ (with $\mathsf{ar}(A) = |\mathsf{var}(A)|$, for all $A \in \mathcal{E}$), assigning a tuple of *distinct* canonical variables to each symbol of the alphabet. We require $\mathsf{var}(A) \cap \mathsf{var}(B) = \emptyset$ whenever $A \neq B$.

We explicitly equip hypergraphs with an interface, specifying which nodes allow them to interact when composed. We call these hypergraphs *networks*.

Definition 1 (Concrete network). *A concrete network is a pair $I \blacktriangleright G$ of a hypergraph G without isolated nodes*[3] *such that $V_G \subseteq \mathbb{V}$, and a set $I \subseteq V_G$.*

Every edge e in a network can be connected to the same node multiple times. This can be understood as having a variable substitution σ mapping the tuple of canonical variables $\mathsf{var}(\mathit{lab}_G(e))$ to the actual variables $a_G(e)$ to which e is connected.

Two networks $I_1 \blacktriangleright G_1$ and $I_2 \blacktriangleright G_2$ are *isomorphic* whenever $I_1 = I_2$ and there exists an isomorphism $\iota\colon G_1 \to G_2$ such that $\iota_{|I_1} = \mathsf{id}_{I_1}$.

Definition 2 (Abstract network). *An* abstract *network is an isomorphism-class of a concrete network.*

Intuitively, abstract networks are taken up to α-conversion of non-interface nodes. We write $I \triangleright G$ to denote the abstract network that corresponds to the equivalence class of the concrete network $I \blacktriangleright G$.

Concrete networks can be seen as terms of an algebraic specification which we call *strong network specification*, where free variables correspond to the interface, and variable restriction (written $(x)P$) is used to declare local (non-interface) nodes (x is local to P in $(x)P$). Following well-known algebraic descriptions of nominal calculi [10], terms will carry a permutation algebra structure. This enables an algebraic treatment of variable binding, together with associated notions of scope, free/bound variables and α-conversion.

The syntax of networks is given by the following grammar:

$$P, Q := A(\tilde{x}) \ \mid\ P|Q \ \mid\ (x)P \ \mid\ P\pi \ \mid\ \mathsf{nil}$$

where $A \in \mathcal{E}$, $\pi \in \mathit{Perm}(\mathbb{V})$, $x \in \mathbb{V}$, $\tilde{x} \in \mathbb{V}^\star$ and $|\tilde{x}| = \mathsf{ar}(A)$. The free variables $\mathsf{fv}(P)$ of P are the unrestricted ones, and are defined by recursion as expected.

[3] In the network specification below, nodes are introduced as support of the permutation algebra of hyperarcs. Isolated nodes would require additional items with singleton support of little use in our model.

$(\mathbf{AX}_{|})$

$P|Q \equiv_s Q|P \qquad (P|Q)|R \equiv_s P|(Q|R) \qquad P|\mathsf{nil} \equiv_s P$

$(\mathbf{AX}_{(x)})$

$(x)(y)P \equiv_s (y)(x)P \quad (x)\mathsf{nil} \equiv_s \mathsf{nil}$

$(\mathbf{AX}_{\alpha})$

$(x)P \equiv_s (y)P[x \mapsto y] \qquad (y \notin \mathsf{fv}(P))$

$(\mathbf{AX}_{SE})$

$(x)(P|Q) \equiv_s (x)P \mid Q \qquad (x \notin \mathsf{fv}(Q))$

$(\mathbf{AX}_{\pi})$

$P\,\mathsf{id} \equiv_s P \qquad (P\pi')\pi \equiv_s P(\pi \circ \pi')$

$(\mathbf{AX}^{p}_{\pi})$

$A(x_1,\ldots,x_n)\pi \equiv_s A(\pi(x_1),\ldots,\pi(x_n)) \qquad \mathsf{nil}\,\pi \equiv_s \mathsf{nil} \qquad (P|Q)\pi \equiv_s P\pi \mid Q\pi$

$((x)P)\pi \equiv_s (\pi(x))(P\pi)$

Fig. 1: Axioms of strong networks.

The *atom* $A(\tilde{x})$ represents an A-labelled hyperedge, connecting the nodes $\tilde{x}$, possibly with repetitions. The *parallel composition* $P|Q$ represents the union of networks P and Q, possibly sharing some nodes. The *restriction* $(x)P$ represents a network where x is local, and hence cannot be shared. The *permutation* $P\pi$ is P where its free variables have been renamed according to π. As usual in permutation algebras, π is not a capture-avoiding substitution, but just a renaming of all global and local names that appear in the term. The constant nil represents the empty graph. We say that a term P is nil-free if nil is not a subterm of P.

We now introduce a *strong* network specification, which, as opposed to the loose one, shown later, identifies more terms.

Definition 3 (Strong network specification). *The* strong network specification *consists of the syntax given above, subject to the axioms of Fig. 1.*

The operator $|$ forms a commutative monoid $(\mathbf{AX}_{|})$. Restrictions can be α-converted $(\mathbf{AX}_{\alpha})$, reordered and removed whenever their scope is nil $(\mathbf{AX}_{(x)})$. The scope of restricted variables can be narrowed to terms where they occur free by the scope extension axiom $(\mathbf{AX}_{SE})$. Axioms for permutations say that identity and composition behave as expected $(\mathbf{AX}_{\pi})$ and that permutations distribute over syntactic operators $(\mathbf{AX}^{p}_{\pi})$. Permutations replace all names bijectively, including the bound ones.

Example 1. Consider the terms

$$(x)(y)(z)(\ A(x,y) \mid B(y,z)\ ) \qquad \text{and} \qquad (y)(\ (x)A(x,y) \mid (z)B(y,z)\ )$$

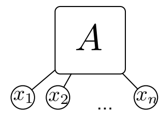
They are proved to be (strong) equivalent by exploiting $(\mathbf{AX}_{(x)})$ to switch the order of restrictions on x and y and then $(\mathbf{AX}_{SE})$ (twice) to move the restrictions on x and z inside parallel composition.

An *s-algebra* $\mathcal{A}$ is a set together with an interpretation $\mathsf{op}^{\mathcal{A}}$ of each operator op. The set of freely generated terms modulo the axioms of Fig. 1 is an *initial* algebra. By initiality, for any such term P there is a unique interpretation $[\![P]\!]^{\mathcal{A}}$ of P as an element of $\mathcal{A}$.

In [17] we show that abstract networks form an initial s-algebra. Hence, we have a unique evaluation of abstracts networks into any other s-algebra.

Definition 4 (Initial s-algebra). *The initial s-algebra $\mathcal{N}$ consists of abstract networks, and the following interpretation of operations:*

$$A^{\mathcal{N}}(x_1, x_2, \dots, x_n) =$$



$$\mathsf{nil}^{\mathcal{N}} = \emptyset \rhd 0_G$$

$$(I \rhd G)\pi^{\mathcal{N}} = \pi(I) \rhd G_\pi \qquad (x)^{\mathcal{N}}(I \rhd G) = I \setminus \{x\} \rhd G$$

$$I_1 \rhd G_1 |^{\mathcal{N}} I_2 \rhd G_2 = I_1 \cup I_2 \rhd G_1 \uplus_{I_1,I_2} G_2$$

where: G_π is G where each node v is replaced with $\pi(v)$; $G_1 \uplus_{I_1,I_2} G_2$ is the disjoint union of G_1 and G_2 where nodes in $I_1 \cup I_2$ with the same name are identified; and 1_G is the empty hypergraph.

Permutations in the specification allow computing the set of free variables, called *support*, in any s-algebra.

Definition 5 (Support). *Let $\mathcal{A}$ be an s-algebra. We say that a finite $X \subset \mathbb{V}$* supports *$P \in \mathcal{A}$ whenever $P\pi = P$, for all permutations π such that $\pi|_X = \mathsf{id}_X$. The (minimal) support $\mathsf{supp}(P)$ is the intersection of all sets supporting P.*

It is important to note that $\mathsf{supp}(I \rhd G) = I$.

2.2 Tree decomposition

A decomposition of a graph can be represented as a *tree decomposition* [14,18,3,4,7,11], i.e., a tree where each node is a piece of the graph. Following [17], we introduce a notion of *rooted tree decomposition.* Recall that a *rooted tree* $T = (V_T, E_T)$ is a set of nodes V_T and a set of edges $E_T \subseteq V_T \times V_T$, such that there is a *root*, i.e. a node $r \in V_T$:

- with no ingoing edges: there are no edges (v, r) in E_T;
- such that, for every $v \in V_T$, $v \neq r$, there is a unique path from r to v.

Definition 6 (Rooted tree decomposition). *A* rooted tree decomposition *of a hypergraph G is a pair $\mathcal{T} = (T, X)$, where T is a rooted tree and $X = \{X_t\}_{t \in V_T}$ is a family of subsets of V_G, one for each node of T, such that:*

1. *for each node $v \in V_G$, there exists a node t of T such that $v \in X_t$;*
2. *for each hyperedge $e \in E_G$, there is a node t of T such that $a_G(e) \subseteq X_t$;*
3. *for each node $v \in V_G$, let $S_v = \{t \mid v \in X_t\}$, and $E_v = \{(x, y) \in E_T \mid x, y \in S_v\}$; then (S_v, E_v) is a rooted tree.*

We gave a slightly different definition of tree decomposition: the original one refers to a non-rooted, undirected tree. In our dynamic programming application it is convenient to model hierarchical problem reductions as rooted structures. All tree decompositions in this paper are rooted, so we will just call them tree decompositions, omitting "rooted". Of course the above definition includes trivial decompositions, like the one with a single node, or the one where $X_t = V_G$ for every node t. They will be ruled out by the notion of *elementary* tree decomposition on which our contribution is centred (see Section 4.3).

Tree decompositions are suited to decompose networks: we require that interface variables are located at the root.

Definition 7 (Decomposition of a network). *The decomposition of a network $I \rhd G$ is a decomposition of G rooted in r, such that $I \subseteq X_r$.*

3 Loose Specification

In strong network specifications the order and positions of restrictions are immaterial. However, the order and positions in which restrictions appear in a term provide some sort of parsing structure for the underlying network. In this section we relax some axioms of the strong algebra to make explicit the hierarchical structure in the network. This is achieved by showing a tight correspondence between terms of the relaxed algebra and Milner's binding bigraphs [13]. In the next sections we will show that the same correspondence can be extended to characterise some special kinds of tree decompositions, called elementary, and also the output produced by an algorithm based on bucket elimination [9].

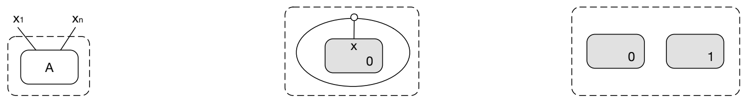
Definition 8 (Loose network specification). *The* loose network specification *is the strong one without axioms* $(\mathbf{AX}_{SE})$ *and* $(\mathbf{AX}_{(x)})$.

The removal of axioms $(\mathbf{AX}_{(x)})$ means that the order in which restrictions are applied is recorded in each equivalence class. The removal of axiom $(\mathbf{AX}_{SE})$ means that the hierarchy imposed by a restriction on name x is not permeable to all hyperarcs, even those that are not attached to x, in the sense that the axioms do not allow to freely move them down and sideways to the restriction on x. We write $P \equiv_l Q$ if P and Q are in the same loose equivalence class.

3.1 Initial loose algebra of binding bigraphs

Our first result shows that, in the same way as the algebra of strong terms offers a syntax for open graphs, the algebra of loose terms offers a syntax for a well-known model of structured hypergraphs, called binding bigraphs [13,8].

We recall that bigraphs are structures where two dimensions coexist: one related to a tree of nested components, mimicking the structure of a term; and another related to the sharing of names. The first is called place graph, the second link graph. The nodes of the graph are labelled by so-called controls that



(a) $A^{\mathcal{B}}(x_1, ..., x_n)$ (b) $(x)^{\mathcal{B}}\_$ (c) $\_\,|^{\mathcal{B}}\_$

Fig. 2: The loose algebra of binding bigraphs

fix their type and arity. A set of controls gives the signature of a class of bigraphs. See [8] for the exact definitions.

In binding bigraphs, places can act as binders for names and the scope rule guarantees that whenever a name is linked to some component then it is either a free name or one bound to some parent of the (place of the) component.

Graphically a bigraph consists of some *roots* (dashed boxes) where *nodes* (solid boxes) can be nested inside (according to the place graph). Each node is labelled by a control that indicates the number of ports for linkage of the node. Binding ports are denoted by circular attachments. Bigraphs can also contain *sites* (grey boxes) that represent some holes where other bigraphs can be plugged in. Sites are numbered, starting from 0. Bigraphs have also *names* (denoted by $x, y, z, ...$) that are local if introduced by some binding port or global otherwise. Ports and names are linked by lines, (according to the link graph).

The tensor product $A \otimes B$ of two bigraphs corresponds to put them side by side, while the composition $A \circ B$ of two bigraphs is defined when the number of holes in A matches with the number of roots in B and it corresponds to plug each root of B in the corresponding hole of A.

In the following we mostly consider *ground* bigraphs, i.e., without sites, with a unique root and with just global outer names. As explained in the Appendix, we take (lean support) equivalence classes of concrete bigraphs, up-to graph isomorphism, renaming of local names and presence of unused names.

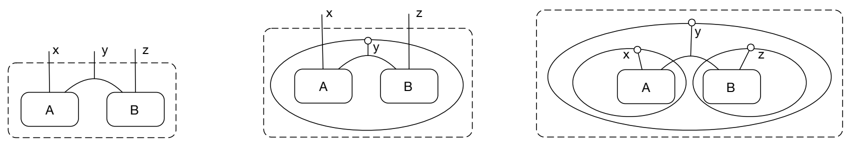
The correspondence between loose terms and binding bigraphs is obtained by taking one control A (drawn as a rounded box) for each constant $A(\tilde{x})$ with binding arity $\mathsf{ar}_b(A) = 0$ and free arity $\mathsf{ar}_f(A) = |\tilde{x}|$ and one control ν (drawn as an oval) for each binding expression (x), with $\mathsf{ar}_b(\nu) = 1$ and $\mathsf{ar}_f(\nu) = 0$.

Definition 9 (The loose algebra of binding bigraphs). *The l-algebra of binding bigraphs consists of (lean-support equivalence classes of) ground binding bigraphs (i.e., with one root and no sites). The symbols of the signature are mapped to the binding bigraphs as shown in the table in Fig. 2 (permutations are just applied to rename the global ports of the graph) and term substitution corresponds to bigraph composition.*

Some examples of interpretations are in Fig 3.

Binding bigraphs offer a convenient model for the loose network specification.

Proposition 1. *Binding bigraphs form an initial loose network algebra.*



(a) $A(x,y)|B(y,z)$ (b) $(y)(A(x,y)|B(y,z))$ (c) $(y)(((x)A(x,y))|(z)B(y,z))$

Fig. 3: Some examples of bigraphs

4 Dynamic programming

The structure of a number of optimisation problems can be conveniently represented as dynamic programming (DP) networks, where hyperedges correspond to atomic subproblems and nodes to (possibly shared) variables. Costs of subproblems are summed up and restricted variables are assigned optimal values.

In this section we will first show how optimisation problems are represented in our framework. We will then turn our attention to the *secondary optimisation problem*, i.e., the problem of finding a decomposition into subproblems of minimal complexity. This is a problem of paramount practical importance for DP.

We have so far given two equivalent ways of decomposing a network: l-terms and binding bigraphs. We will introduce a notion of evaluation complexity for l-terms, and we will characterise local optima as *l-canonical* terms. Then we will establish a formal connection with another way of decomposing DP problems, namely (*elementary*) tree decompositions.

Finally, we will show that the well-known bucket elimination algorithm (see, e.g., [19, 5.2.4]) precisely corresponds to computing the l-canonical form of a term w.r.t. a given ordering on restricted variables. Leveraging the algebraic representation of networks as terms, and the correspondence between l-terms and tree decompositions, this result provides us with a way to compute a locally optimal decomposition of a network in three equivalent ways: as a l-canonical term, a binding bigraph, or a tree decomposition.

4.1 Networks as optimisation problems

We now introduce an s-algebra of *cost functions*, where networks are evaluated to solutions of the corresponding optimisation problem. We fix a domain $\mathbb{D}$ of values for variables. Then an element of the s-algebra is a cost function $\varphi\colon (\mathbb{V} \to \mathbb{D}) \to \mathbb{R}_\infty$ that given an assignment of values to variables returns its cost. To interpret all terms we assume that an interpretation is given of each symbol $A \in \mathcal{E}$ as a cost function $\mathsf{func}_A\colon (\mathbb{V} \to \mathbb{D}) \to \mathbb{R}_\infty$ such that, for any $\rho, \rho'\colon \mathbb{V} \to \mathbb{D}$, $\rho_{|\mathsf{var}(A)} = \rho'_{|\mathsf{var}(A)}$ implies $\mathsf{func}_A\,\rho = \mathsf{func}_A\,\rho'$, i.e., func_A only depends on the canonical variables of A.

Definition 10 (S-algebra of cost functions). *The s-algebra $\mathcal{V}$ consists of cost functions $\varphi\colon (\mathbb{V} \to \mathbb{D}) \to \mathbb{R}_\infty$ and the following interpretation of operations,* *for any $\rho\colon \mathbb{V} \to \mathbb{D}$:*

$$A^{\mathcal{V}}(\tilde{x})\rho = \mathsf{func}_A(\rho \circ \sigma) \qquad \mathsf{nil}^{\mathcal{V}}\rho = 1$$

$$((x)^{\mathcal{V}}\phi)\rho = \min\{\phi\rho[x \mapsto d]\}_{d\in\mathbb{D}} \qquad (\phi\pi^{\mathcal{V}})\rho = \phi(\rho \circ \pi) \qquad (\phi_1|^{\mathcal{V}}\phi_2)\rho = \phi_1\rho + \phi_2\rho$$

where $\sigma\colon \mathbb{V} \to \mathbb{V}$ is a substitution mapping $\mathsf{var}(A)$ to $\tilde{x}$ component-wise and expressing the connection between the canonical vertices of an A-labelled hyperedge and the actual nodes of the graph it is connected to.

Example 2. Consider the term $P = (y)(x)(z)(\ A(x,y)\ |\ B(y,z)\ )$. This is evaluated as the optimisation problem consisting in minimising the sum of the cost functions for A and B w.r.t. to all the variables. Explicitly:

$$[\![P]\!]^{\mathcal{V}} = \lambda\rho.\min\left\{\mathsf{func}_A(\rho[x \mapsto d_1, y \mapsto d_2]) + \mathsf{func}_B(\rho[y \mapsto d_2, z \mapsto d_3])\right\}_{d_1,d_2,d_3\in\mathbb{D}}$$

which, since func_A and func_B only depends on $\{x,y\}$ and $\{y,z\}$, respectively, is a single value that does not depend on ρ.

Although all terms for the same network have the same evaluation in any algebra, different ways of computing such an evaluation, represented as different terms, may have different computational costs.

We make this precise by introducing a notion of evaluation complexity.

4.2 Evaluation complexity

We define the complexity of a term P as the maximum "size" of elements of an algebra $\mathcal{A}$ computed while inductively constructing $[\![P]\!]^{\mathcal{A}}$, the size being given by the number of variables in the support. Intuitively, a step of DP consists of solving a subproblem parametrically with respect to a number n of variables. Thus if the number of possible values of a variable is $|\mathbb{D}|$, then the number of cases to consider for solving the subproblem is $|\mathbb{D}|^n$, namely it is exponential with the number n of parameters of the subproblem. As a consequence, we can approximate the complexity of the whole problem with the complexity of the hardest subproblem.

In our algebraic representation, DP problems are terms P of the strong algebra interpreted as functions $\varphi\colon (\mathbb{V} \to \mathbb{D}) \to \mathbb{R}_\infty$ and their cost is just the evaluation of P. Solving a subproblem corresponds to evaluating a restriction operator $(x)P$ of a term P, while the parameters are the variables in its support. On the other hand, the cost of a single case is again proportional to $|\mathbb{D}|$: we fix the parameters and we optimize with respect to the values of the restricted variable. Thus the complexity of evaluating $(x)P$ is $|\mathsf{fv}(P)|$. Notice that it represents the space and time complexity of the problem. In fact, it correctly coincides with the dimension of the matrix needed to represent the function $\varphi\colon (\mathbb{V} \to \mathbb{D}) \to \mathbb{R}_\infty$ corresponding to $[\![P]\!]^{\mathcal{A}}$. The key observation is that if we take two strong equivalent terms P_1 and P_2, they will necessarily evaluate to the same cost function in $(\mathbb{V} \to \mathbb{D}) \to \mathbb{R}_\infty$, but they will not have necessarily the same complexity.

Example 3. Consider the following terms:

$$P = (y)(x)(z)(\ A(x,y) \mid B(y,z)\ ) \qquad Q = (y)(\ (x)A(x,y) \mid (z)B(y,z)\ ).$$

Although $P \equiv_s Q$, these terms have different complexities. The term P has complexity 3 because, when evaluating it in any algebra, one has to evaluate $A(x,y)|B(y,z)$, and then solve it w.r.t. all its variables. Intuitively, $A(x,y)|B(y,z)$ is the most complex subproblem one considers in P, with 3 variables. Instead, the complexity of Q is 2, because its evaluation requires solving $A(x,y)$ and $B(y,z)$ w.r.t. x and z, which are problems with 2 variables.

Given a term in the strong algebra, the problem of finding a (syntactical) term with minimal complexity corresponds to the secondary optimisation problem of DP. In [17] we have inductively defined a complexity function for terms.

Definition 11. *Given a term P, its complexity $\langle\langle P\rangle\rangle$ is defined as follows:*

$$\langle\langle P|Q\rangle\rangle = \max\{\langle\langle P\rangle\rangle, \langle\langle Q\rangle\rangle, |\mathsf{fv}(P|Q)|\} \qquad \langle\langle (x)P\rangle\rangle = \langle\langle P\rangle\rangle \qquad \langle\langle P\pi\rangle\rangle = \langle\langle P\rangle\rangle$$
$$\langle\langle A(\tilde{x})\rangle\rangle = |\mathsf{set}(\tilde{x})| \qquad \langle\langle \mathsf{nil}\rangle\rangle = 0$$

Complexity is well-defined for loose terms but not for strong terms, as applying $(\mathbf{AX}_{SE})$ may change the complexity (see [17]).

Lemma 1. *Given $(x)(P|Q)$, with $x \notin \mathsf{fv}(Q)$, we have $\langle\langle (x)P|Q\rangle\rangle \leq \langle\langle (x)(P|Q)\rangle\rangle$.*

We now classify l-terms according to their complexity. We say an l-term is *pure* if every subterm $(x)P$ is such that $x \in \mathsf{fv}(P)$.

Definition 12 (L-normal and l-canonical forms). *An l-term is in* l-normal *form whenever it is of the form*

$$(\tilde{x})(\ A_1(\tilde{x}_1) \mid A_2(\tilde{x}_2) \mid \ \dots \ \mid A_n(\tilde{x}_n)\ )$$

A l-term is in l-canonical *form whenever the directed form of* $(\mathbf{AX}_{SE})$

$$(x)(P|Q) \to (x)P \mid Q \qquad (x \notin \mathsf{fv}(Q))$$

cannot be applied to it. For both forms, we assume that they are pure and that nil *sub-terms are removed via* $(\mathbf{AX}_{|})$*.*

L-normal forms have *maximal* complexity. A term can be s-equivalent ($\equiv_s$) to several l-normal forms, all with the same complexity: they differ for the order in which restrictions are applied.

Example 4. The l-term $(z)(x)(y)(A(x,y)|B(y,z))$ is in l-canonical form, while the l-term $(x)(y)(z)(A(x,y)|B(y,z))$ is not, because the axiom $(\mathbf{AX}_{SE})$ is applicable to restrict the scope of x as in $(x)(y)(A(x,y)|(z)B(y,z))$.

Due to Lemma 1, l-canonical forms are local minima of complexity w.r.t. the application of strong axioms minus $(\mathbf{AX}_{(x)})$. In fact, $(\mathbf{AX}_{(x)})$ may enable further applications of $(\mathbf{AX}_{SE})$, and lead to a further complexity reduction, as shown in [17] for weak terms. This phenomenon is exemplified in Example 4, where bringing (z) closer to the parallel via restriction swaps makes the term suboptimal. By forbidding $(\mathbf{AX}_{(x)})$, we considerably simplify the algorithm for computing local complexity optima, and we recover a full correspondence with the bucket elimination algorithm [9], as we shall see later.

4.3 Elementary tree decompositions

In this section we establish a correspondence between two ways of decomposing problems that admit a graph model: loose terms and tree decompositions. We first introduce the novel notion of *elementary* tree decomposition (e.t.d.).

Definition 13 (Elementary tree decomposition). *A tree decomposition (T, X) for a network $I \rhd G$ is* elementary *whenever $|X_r \setminus I| \leq 1$ and, for all non-root nodes t of T, $|X_t \setminus X_{\mathsf{parent}(t)}| = 1$.*

We can now adapt the translation function from tree decompositions to terms from [17] to elementary tree decomposition (*e.t.d.* for short).

Definition 14 (From elementary tree decompositions to l-terms). *Let $\mathcal{T} = (T, X)$ be an e.t.d. for $I \rhd G$. For each node t of T, let $E(t) \subseteq E_G$ and $V(t) \subseteq V_G$ be sets of nodes and edges of G such that $e \in E(t)$ (resp. $v \in V(t)$) if and only if t is the closest node to the root of T such that $\alpha_G(e) \subseteq X_t$ (resp. $v \in X_t$). Let $\tau(t)$ be recursively defined on nodes of T as follows:*

$$\tau(t) = (x)(A_1(\tilde{x}_1)|A_2(\tilde{x}_2)|\dots|A_n(\tilde{x}_n)|\tau(t_1)|\tau(t_2)|\dots|\tau(t_k))$$

where $E(t) = \{e_1, e_2, \dots, e_n\}$, $x \in V(t)\backslash I$ (we drop the restriction if $V(t)\backslash I = \emptyset$), $lab_G(e_i) = A_i$, $\alpha_G(e_i) = \tilde{x}_i$, and $t_1, t_2, \dots, t_k$ are the children of t in T. Then we define $\mathsf{term}(\mathcal{T}) \coloneqq \tau(r)$, where r is the root of $\mathcal{T}$.

It is immediate to observe that each e.t.d. is mapped to an l-term. In fact, at each node we add at most one restriction, which results in an ordering over restricted names. In general the mapping is not injective, as the following example shows.

Example 5. Consider the l-term $(x)(A(x, y)|(z)B(y, z))$. Then $\mathsf{term}(\_)$ maps both the elementary tree decompositions depicted in Figure 4a to it.

We shall define a converse mapping, from pure l-terms to e.t.d.s. In the following we exploit the fact that every pure l-term is congruent to the form

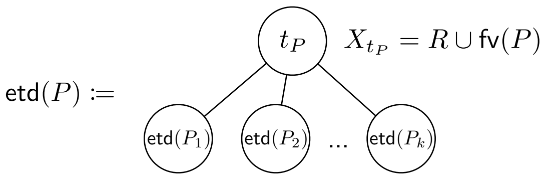
$$(R)\,(A_1(\tilde{x}_1)|\dots|A_1(\tilde{x}_n)|P_1|\dots|P_k)$$

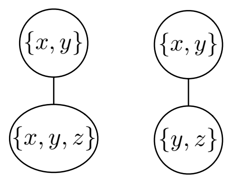
where $|R| \leq 1$ and it occurs free in the rest of the term if non-empty, $n, k \geq 0$, and P_i are l-terms of the same form with a top level restriction.

Definition 15 (From pure l-terms to elementary tree decompositions). *Given a pure l-term P such that*

$$P \equiv_l (R)\,(A_1(\tilde{x}_1)|\dots|A_n(\tilde{x}_n)|P_1|\dots|P_k)$$

the corresponding e.t.d. $\mathsf{etd}(P)$ is recursively defined as follows:





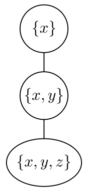


Fig. 4: E.t.d.s for Examples 5 (a) and 6 (b). For simplicity, we have annotated each node t with the corresponding set X_t.

$$\frac{}{A(\tilde{x}) : \mathsf{set}(\tilde{x})} \qquad \frac{P : X}{P\pi : \pi(X)} \qquad \frac{P : X \cup \{z\}}{(z)P : \mathsf{fv}((z)P)} \qquad \frac{P_1 : X_1 \quad P_2 : X_2}{P_1|P_2 : X_1 \cap X_2}$$

Fig. 5: Type rules for l-canonical terms

The place graph of the binding bigraph $P^{\mathcal{B}}$ is tightly connected to $\mathsf{etd}(P)$. In fact, if we remove from the place graph all leaves whose controls are atoms A we get the tree structure of (controls that are) restrictions. Then each node of the graph can be tagged with the names of the ports that are used by nested controls and we get $\mathsf{etd}(P)$. The free variables of P will also appear in the root.

Lemma 2. *We have that* $\mathsf{etd}(P)$ *is an elementary tree decomposition of* $[\![P]\!]^{\mathcal{N}}$.

As for $\mathsf{term}(\_)$, this translation is not injective (see the following example).

Example 6. Consider the l-terms $P = (x)(y)(A(x,y)|(z)B(x,y,z))$ and $Q = (x)(y)(z)(A(x,y)|B(x,y,z))$. Then $\mathsf{etd}(P) = \mathsf{etd}(Q)$ is the e.t.d. in Fig. 4b.

The translation is injective if we restrict the domain to l-canonical terms.

Proposition 2. *Different l-canonical terms give rise to different e.t.d.s.*

We conjecture that $\mathsf{term}(\_)$ is left-inverse to $\mathsf{etd}(\_)$ on l-canonical terms.

4.4 A type system for l-canonical terms

We use a type system to characterise l-canonical forms. Type judgements are of the form $P : X$ where X denotes the set of names of P that can be restricted immediately on top of P. The typing rules are in Fig. 5. Those for atoms and for permutations are trivial.

The rule for restriction deserves some explanation. Let P be a term such that the names in $X \cup \{z\}$ can be restricted, then for the term $(z)P$ all remaining names $\mathsf{fv}((z)P)$ can also be restricted. This is because taken any name $x \in \mathsf{fv}((z)P) \setminus X$, in the l-term $(x)(z)P$ we cannot swap x with z in order to apply the axiom $(\mathbf{AX}_{SE})$. The fact that $(\mathbf{AX}_{SE})$ cannot be applied to z is guaranteed by z being one of the names that can be restricted on top of P (see premise).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A(x,y):\{x,y\}} \qquad \dfrac{\dfrac{}{B(y,z):\{y,z\}}}{(z)B(y,z):\{y\}}}{A(x,y)\mid(z)B(y,z):\{y\}}}{(y)(\ A(x,y)\mid(z)B(y,z)\ ):\{x\}}}{(x)(y)(\ A(x,y)\mid(z)B(y,z)\ ):\emptyset} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A(x,y):\{x,y\}}}{(x)A(x,y):\{y\}} \qquad \dfrac{\dfrac{}{B(y,z):\{y,z\}}}{(z)B(y,z):\{y\}}}{(x)A(x,y)\mid(z)B(y,z):\{y\}}}{(y)(\ (x)A(x,y)\mid(z)B(y,z)\ ):\emptyset}$$

Fig. 6: Two type derivations

$$\begin{aligned}
R_0(x,z,y) &\triangleq A(x,y)|B(x,z)\\
R_{i+1}(x,z,y) &\triangleq (v)(R_i(x,z,v)|R_i(v,z,y))\\
FW_k(x,y) &\triangleq (z)(R_k(x,z,y)|B(y,z))\\
W_k(x,y) &\triangleq FW_k(x,y)|A(y,x)
\end{aligned}$$

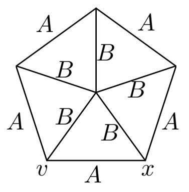


Fig. 7: The wheel network $W_2(v,x)$

The rule for parallel composition follows a simple criterion: only names that are in common between all constituents can be restricted. If a name z appears in P_1 but not in P_2, then obviously the term $(z)(P_1|P_2)$ is not in l-canonical form.

Typing is preserved by the axioms of the loose network specification, in the sense that for all nil-free terms $P \equiv_l Q$ and type X, if $P : X$, then $Q : X$.

Proposition 3. *For any P and type X, if $P : X$ then P is in l-canonical form.*

Proposition 4. *If $P \neq \mathsf{nil}$ is in l-canonical form, then $P : X$ for some X.*

Example 7. The term $(x)(y)(A(x,y) \mid (z)B(y,z))$ is in l-canonical form as witnessed by the derivation in Fig. 6, left. Also the term $(y)(\ (x)A(x,y) \mid (z)B(y,z)\ )$ is in l-canonical form (see Fig. 6, right). The term $(x)(y)(z)(A(x,y)|B(y,z))$ is not typeable, because $A(x,y)|B(y,z) : \{y\}$ and $z \notin \{y\}$ thus the subterm $(z)(A(x,y)|B(y,z))$ is not typeable. In fact it is not in l-canonical form because the axiom $\mathbf{(AX}_{SE}\mathbf{)}$ can be applied to restrict the scope of z to $B(y,z)$.

Example 8. Consider the wheel example from [17] in Fig. 7 (see [20] for a graph grammar presentation). Then $R_0(x,z,y) : \{x\}$, but $R_1(x,z,y) = (v)(R_0(x,z,v)|R_0(v,z,y))$ is not typeable, as $R_0(x,z,v)|R_0(v,z,y) : \emptyset$. We can give an alternative typeable definition of wheels where all terms are l-canonical:

$$\begin{aligned}
R_0(x,z,y) &\triangleq (v)(A(x,v)|B(v,z)|A(v,y)) : \{x,z,y\}\\
R_{i+1}(x,z,y) &\triangleq (v)(R_i(x,z,v)|R_i(v,z,y)|B(v,z)) : \{x,z,y\}\\
FW_k(x,y) &\triangleq (z)(R_k(x,z,y)|B(x,z)|B(y,z)) : \{x,y\}\\
W_k(x,y) &\triangleq FW_k(x,y)|A(y,x) : \{x,y\}
\end{aligned}$$

Input: l-term $(R)M$ in normal form.
Output: l-term P in canonical form.

```
1: $P \leftarrow (R)M$
2: while $R \neq \emptyset$ do
3:     $x \leftarrow \max R$
4:     take largest $M' \subseteq M$ such that $x$ occurs free in all elements of $M'$
5:     $P' \leftarrow (x)M'$
6:     $M \leftarrow M \setminus M' \cup \{P'\}$
7:     $R \leftarrow (R \setminus \{x\})$
8: return $M$
```

Fig. 8: Bucket elimination algorithm for l-terms.

4.5 Computing l-canonical forms

We now give an algorithm to compute an l-canonical form of a term. This is based on *bucket elimination* (see, e.g., [19, 5.2.4]), also known as adaptive consistency.

We briefly recall the bucket elimination algorithm. Given a network representing an optimisation problem, and a total order over its variables, sub-problems are partitioned into *buckets*: each sub-problem is placed into the bucket of its last variable in the ordering. At each step, the bucket of a variable x is eliminated by creating a new sub-problem involving all the variables in the bucket different from x. This new problem is put into the bucket of its last variable, and the process is iterated.

In [17] we have extended the algorithm to modify the ordering of variables in the attempt of reducing the size of subproblems. This required an additional backtracking step. Here we show that l-canonical forms can be computed via the ordinary bucket elimination algorithm, suitably adapted to l-terms.

Our algorithm is shown in Figure 8. Here putting a constraint in the bucket of its last variable amounts to applying the scope extension axiom, and eliminating a variable amounts to restricting it. The algorithm takes as input an l-term in normal form $(R)M$, represented by a totally ordered set of variables R (recall that restricted variables are assumed to be distinct), and a multiset of atomic terms M. The algorithm first picks the max variable in the total order (line 3), then it partitions the input l-term into subterms according to whether x occurs free or not (line 4), and from the former it creates a new term P' where x is restricted. It then adds P' to the remaining terms of P, removes x from the total order R, and iterates the process for the resulting term.

Example 9. We now show an example of execution. We will run the algorithm with the following l-normal term as input:

$$(a < b < c < d)\ \{A_1(c,d), A_2(d,b), A_3(d,a), A_4(b,a)\}$$

On the first iteration, line 3 picks d, and line 4 will select $M' = \{A_1(c,d), A_2(d,b), A_3(d,a)\}$. Then R and M in line 6 and 7 are

$$(a < b < c) \qquad \{A_4(b,a),\ (d)\{A_1(c,d), A_2(d,b), A_3(d,a)\}\ \}$$

At the next iteration, line 3 picks c, and line 6 and 7 give the following:

$$(a < b) \qquad \{A_4(b,a),\ (c)(d)\{A_1(c,d), A_2(d,b), A_3(d,a)\}\}$$

Despite c occurring only in $A_1(c,d)$, the presence of (d), which is greater in the ordering, prevents (c) from permeating the parallel composition. After two more iterations, where the variables b and a are processed, the output is:

$$(a)(b)(A_4(b,a) \mid (c)(d)(A_1(c,d) \mid A_2(d,b) \mid A_3(d,a)))$$

The algorithm outputs all and only the l-canonical forms for a given term.

Proposition 5. *Given a term P, a term $C \equiv_s P$ is l-canonical if and only if there is an l-normal form for P which, if provided as input to the bucket elimination algorithm, outputs C.*

5 Conclusion

Along the graphical-algebraic correspondence introduced by Milner [15] for networks and flow algebras, in [17] two of the authors have studied the connections between tree decompositions and certain weak network algebras. In this paper the correspondence of the two former models and, in addition, of a version of Milner's bigraphs [16], has been fully formalised introducing a small variation of weak network algebras, called loose network algebras. Milner's flow algebras also fit in the schema as strong algebras. The algebraic treatment is instrumental for conveniently expressing important computational properties of the complex problem at hand. In this line, we examine the very relevant case of graphical optimisation problems solved via dynamic programming. We show that the solution of a problem corresponds to the evaluation of a strong term, while the cost of such a computation is obtained by evaluating the same term in a loose algebra. It is also shown that reducing a loose term w.r.t. the axiom of scope expansion produces a modified loose term of better or equal complexity. The notion of l-canonical form for loose terms captures local minima of complexity and is tightly related to particular kinds of tree decompositions, called elementary. We also define a type system for checking if a term is in l-canonical form. Finally, when a total ordering is imposed on the variables of a term, a term in l-canonical form can be uniquely derived by applying Dechter's bucket algorithm [9].

The problem of how to represent parsing trees for (hyper)graphs has been studied in depth in the literature. We mention the work of Courcelle on graph algebras [6]. Here tree decompositions are used to bound the complexity of checking graph properties, and a term encoding allows employing automata-theoretic tools. The focus is different than ours, and investigating connections is left for future work. Other approaches based on tree decompositions are [14,18,3,4,7,11]. Here we provide an algebraic characterisation of tree decompositions with locally minimal complexity. The application of bigraphs to dynamic programming is also novel. We plan to study the relation between loose network specifications and graph grammars for hyperedge replacement [5], where the structure of a derived graph is its derivation tree: it seems that our approach has a simpler compositional structure, and an up-to-date foundation for name handling.

References

1. R. Bellman. Some applications of the theory of dynamic programming - A review. *Operations Research*, 2(3):275–288, 1954.
2. U. Bertelè and F. Brioschi. On non-serial dynamic programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.
3. C. Blume, H. J. S. Bruggink, M. Friedrich, and B. König. Treewidth, pathwidth and cospan decompositions with applications to graph-accepting tree automata. *J. Vis. Lang. Comput.*, 24(3):192–206, 2013.
4. H. L. Bodlaender and A. M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.
5. D. Chiang, J. Andreas, D. Bauer, K. M. Hermann, B. K. Jones, and K. Knight. Parsing graphs with hyperedge replacement grammars. In *Proc. of ACL'13, Vol. 1: Long Papers*, pages 924–932. The Association for Computer Linguistics, 2013.
6. B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2012.
7. B. Courcelle and M. Mosbah. Monadic second-order evaluations on tree-decomposable graphs. *Theor. Comput. Sci.*, 109(1&2):49–82, 1993.
8. T. C. Damgaard and L. Birkedal. Axiomatizing binding bigraphs. *Nord. J. Comput.*, 13(1-2):58–77, 2006.
9. R. Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003.
10. F. Gadducci, M. Miculan, and U. Montanari. About permutation algebras, (pre)sheaves and named sets. *H.-O. and Symbolic Comput.*, 19(2-3):283–304, 2006.
11. V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proc. of UAI'04*, pages 201–208. AUAI Press, 2004.
12. N. Hoch, U. Montanari, and M. Sammartino. Dynamic programming on nominal graphs. In *Proc. of GaM@ETAPS'15*, volume 181 of *EPTCS*, pages 80–96, 2015.
13. O. H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical Report 580, University of Cambridge, 2004.
14. T. Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.
15. R. Milner. Flowgraphs and flow algebras. *J. ACM*, 26(4):794–818, 1979.
16. R. Milner. Bigraphical reactive systems. In *Proc. of CONCUR'01*, volume 2154 of *LNCS*, pages 16–35. Springer, 2001.
17. U. Montanari, M. Sammartino, and A. Tcheukam Siwe. Decomposition structures for soft constraint evaluation problems: An algebraic approach. In *Graph Transformation, Specifications, and Nets - In Memory of H. Ehrig*, pages 179–200, 2018.
18. N. Robertson and P. D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.
19. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.
20. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
21. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.