Modular Decomposition for Logic

An interactive tool for the modular decomposition of graphs

Remy Seassau

Under the supervision of
Lutz Strassburger

A thesis presented for the degree of
Bachelor of Science

Laboratoire d'Informatique de l'X
École Polytechnique, Institut Polytechnique de Paris
Palaiseau, France

Modular Decomposition for Logic

An interactive tool for the modular decomposition of graphs

Remy Seassau

Abstract

There exists a well-known correspondence between cographs and logical formulae. This relationship has been explored, extended, and inverted in such a way that we are now interested in logics built on graphs. Modular decomposition is a graph-theoretical method for the decomposition of the structure of a graph. It has proven to be an essential tool for the construction and use of these logics. We seek to provide a tool for the interactive creation, modification, and modular decomposition of graphs. The tool should also allow for the creation, modification and recomposition into graphs of modular decompositions.

We present in this report a tool for the interactive creation, modification and modular decomposition of graphs. We implement in Ocaml a naive, quartic-time complexity algorithm for the decomposition as it is sufficiently fast for human-readable purposes. Using the browser as our GUI [1] , we use an Ocaml to JavaScript compiler to run the algorithm directly in the browser. We support both directed and undirected graphs as different logics rely on different notions of graphs. We have thus built a practical tool for the interactive modular decomposition of graphs and a prototype for enabling logicians to work digitally on logics with graphs as formulae.

Keywords — *Logic, Modular Decomposition, Proof Theory, Graph Theory*

[1] The tool can be accessed as a webpage `http://www.lix.polytechnique.fr/Labo/Lutz.Strassburger/modular_decomposition/` or compiled from the source code and used locally (see Section 6.2)

Contents

1 Motivation and Related work **3**
- 1.1 Logic and Graphs . . . 3
- 1.2 Modular Decomposition Algorithms . . . 4

2 Modular Decomposition of Graphs **5**
- 2.1 Graphs . . . 5
- 2.2 Modules . . . 6
- 2.3 From Formulae to Graphs . . . 8

3 The Algorithm **11**
- 3.1 Connective Compression . . . 11
- 3.2 Maximal Module Compression . . . 13
- 3.3 Tree Recovery . . . 14

4 Our Implementation **16**
- 4.1 Graphs . . . 16
- 4.2 Modular Decomposition Trees . . . 17
- 4.3 State and Subsets . . . 18
- 4.4 From Condensed Graph to MDT . . . 18
- 4.5 From a MDT to its Graph . . . 19

5 Interactivity Features **21**
- 5.1 Interacting with the Graph . . . 21
  - 5.1.1 Mouse inputs . . . 22
  - 5.1.2 Keyboard inputs . . . 22
  - 5.1.3 Buttons . . . 22
- 5.2 Interacting with the Decomposition Tree . . . 23
  - 5.2.1 Mouse Inputs . . . 23
  - 5.2.2 Keyboard inputs . . . 24
  - 5.2.3 Buttons . . . 24
- 5.3 Additional Features . . . 24

6 Running the code **26**
- 6.1 Calling OCaml Functions from the Browser . . . 26
- 6.2 Compiling the Code . . . 27

7 Conclusion **28**
- 7.1 Future work . . . 28

References **30**

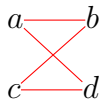
A Examples **32**

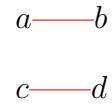
1 | Motivation and Related work

This paper presents a tool written for the user to interact with graphs and their structure through the graphs' modular decomposition. We explain in the introduction the motivation behind such a tool and our approach to implementing an algorithm for modular decomposition. We then formally define the notions of graphs and logic in order to motivate the project further. Afterwards, we give the algorithm that we use for modular decomposition before detailing our implementation. Finally, we introduce the tool's full features, followed by some technical indications on how to run the project and conclude.

1.1 Logic and Graphs

There exists a well known correspondence between logical formulae and a certain class of graphs called cographs [1]. Indeed, consider the two following logical formulae $P = (a \parr c) \otimes (b \parr d)$ and $Q = (a \otimes b) \parr (c \otimes d)$. These formulae can be represented by the following graphs:

G_P: 

G_Q: 

This correspondence has motivated the definition of a proof system on cographs [19]. Work has been done to generalize proof systems on cographs to directed cographs [21, 20]. This idea was then generalized further when the cograph condition was dropped and the *graphic proof system* (GS) was created [1]. Work is currently being done by L. Strassburger and others to generalize GS even further by extending the proof system from undirected graphs to directed graphs. This idea of directed logical connective is implemented in the logic BV (*basic system V* [13, 23]), which introduces a non-commutative logical connective called "before" ($\lhd$).

In order to reason about such proofs systems on graphs, we would like to study the structure of the "formulae" which in our case correspond to nested (or composed) graphs. Unlike in an ordinary logical formula, our logical connectives are now graphs and n-ary. We thus use the modular decomposition trees of our graphs

to study their structure, identifying children of a parent in the tree to subformulas of a formula.

1.2 Modular Decomposition Algorithms

There exists several algorithms for modular decomposition as it is a well studied concept in graph theory [14]. Although linear time complexity algorithms exist for the modular decomposition of both directed [16] and undirected [22] graphs, we decide to use a "naive" algorithm which is quartic in the number of nodes in the graph. The reason for this is double. Firstly, existing linear time algorithms, although simpler than older linear algorithms, remain much more complicated to reason about than the naive algorithm. Using the quartic algorithm makes it easy to reason about and maintain. Secondly, the intended purpose of this tool is to allow the user to visually play around with graphs. The graphs in question should thus be of reasonable size (although there is no size restriction on them), and in practice, we do not experience any significant computation time. We base our algorithm on the first paper on modular decomposition, published in the 1970s [15].

We split the project into two distinct parts that need only interact at discrete moments in time: the modular decomposition algorithm and the interactions with graphs. Given the nature of the structures we intend to work on, we would like to be able to efficiently reason about our code and thus choose the strongly-typed functional programming language Ocaml. Advantages of this language choice are the potential merging with other projects (for example, interactive theorem proving projects such as [6]) and a decrease in technical debt at the PARTOUT team at INRIA.

For the interaction with graphs, we choose to use the browser as our GUI and base ourselves on the Cytoscape.js JavaScript library [10]. Running our interface in the browser means that we can openly host the project for anyone to use while also allowing anyone who wishes to run a local version to download the source code and build the project themselves. We allow our modular decomposition algorithm to be run from the browser using the `Js_of_Ocaml` compiler [24], which allows us to compile Ocaml bytecode to Javascript, which is runnable in the browser.

2 | Modular Decomposition of Graphs

2.1 Graphs

An **undirected graph** $G = (V_G, E_G)$ is an ordered pair where V_G is a set of vertices and E_G is a set of unordered pairs of elements of V_G. E_G defines an irreflexive and symmetric binary relation that corresponds to the edges in the graph. A **directed graph** $G = (V_G, E_G)$ is similarly defined with the exception that E_G is a set of *ordered pairs* that defines an irreflexive binary relation (not symmetric). In the following, we say graph to mean both an undirected or directed graph and we specify which we mean when necessary. We restrict ourselves to graphs with finite vertex sets.

Definition 2.1.1. The edges of a graph define four possible relations on an ordered pair of vertices (v_1, v_2):

- there is an edge from v_1 to v_2 but not from v_2 to v_1 ($v_1 \curvearrowright v_2$)
- there is an edge from v_2 to v_1 but not from v_1 to v_2 ($v_1 \curvearrowleft v_2$)
- there is an edge from v_1 to v_2 and from v_2 to v_1 ($v_1 — v_2$)
- there is no edge between v_1 and v_2 ($v_1 \cdots v_2$)

We thus denote R_{edges} the set $\{—, \cdots, \curvearrowright, \curvearrowleft\}$ of **edge relations** on an ordered pair of vertices.

We choose to identify the edge pairs (v_1, v_2) of an undirected graph with the $v_1 — v_2$ relation since the pair is unordered. This provides an injection from undirected to directed graphs, where every edge pair (v_1, v_2) in the undirected graph is mapped to a pair of edges $(v_1, v_2), (v_2, v_1)$ in the directed graph. As such, the edge pairs of an undirected graph define the $—$ and $\cdots$ relations, whereas a directed graph can define any of the edge relations.

We say that for some vertex $v \in V$, the set $\{ v_i \mid v_i \in V, v \curvearrowright v_i \text{ or } v — v_i \}$ is the set of **successors** of v, denoted $\text{suc}(v)$. Similarly, $\{ v_i \mid v_i \in V, v_i \curvearrowright v \text{ or } v — v_i \}$ is the set of **predecessors** of v, denoted $\text{pred}(v)$. If the graph is undirected, we have for all $v \in V$ that $\text{suc}(v) = \text{pred}(v)$.

Definition 2.1.2. Given a graph G, the complement of G, denoted $\bar{G}$ is defined as

$$(V_G, \{(v_1, v_2) \mid v_1, v_2 \in V_G, v_1 \neq v_2, (v_1, v_2) \notin E_G\})$$

Definition 2.1.3. Given a set L of labels, we say that a graph G is ***L*-labelled** if there exists an injective function $l_G \colon V_G \to L$. We write $l_G(v)$ to denote the label of a vertex $v \in V_G$. If L contains only atoms, we say that G is an atomic graph.

A graph G' is a subgraph of a graph G iff $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$. We write $G' \subseteq G$ to denote that G' is a subgraph of G.

Definition 2.1.4. We say that G' is an **induced subgraph** of G if

$$V_{G'} \subseteq V_G \quad \text{and} \quad \forall v_1, v_2 \in V_{G'}, (v_1, v_2) \in E_{G'} \iff (v_1, v_2) \in E_G$$

Specifically, we say that G' is the subgraph of G induced by $V_{G'}$.

2.2 Modules

A module M corresponds to a subgraph where all of the vertices inside M relate to the other vertices outside of M in the same way. We are in particular interested in maximal modules, which are the modules that are not contained in any module other than the whole graph. Indeed, a module is to a graph what a subformula is to a formula.

Definition 2.2.1. A **module** of a graph G is an induced subgraph $M = (V_M, E_M)$ of G such that

$$\forall x, y \in V_M, \forall z \in V_G \setminus V_M, \forall R \in R_{edges}, xRz \iff yRz$$

where R is any of the four possible edge relations previously defined.

A module M in G is **maximal** if for all modules M' of G with $M' \neq G$ we have $M \subseteq M' \implies M = M'$. A module is **trivial** if $V = \varnothing$, V is a singleton $\{v\}$ or $V_M = V_G$.

Definition 2.2.2. A graph G is **prime** if and only if $|V_G| \geq 2$ and all modules of G are trivial.

We note and name here the graphs of size 2 (which by definition, are all prime),

(2.2.1)

We specify two additional prime graph names: P_n for $n \geq 4$ and S_n for $n \geq 3$, where P_n is the line graph of n vertices with the — relation and S_n is the line graph of n vertices with the ⌒ relation.

Example 2.2.1.

P_4: S_4:

We will see later that prime graphs can be viewed as generalized non-decomposable n-ary connectives [12, 3], where n is the number of vertices of the prime graph.

Definition 2.2.3. Let G be a graph with n vertices $V_G = \{v_1, \ldots, v_n\}$ and let $H_1, \ldots, H_n$ be n graphs such that $\forall i, j < n, i \neq j \implies V_{H_i} \cap V_{H_j} = \varnothing$. The **composition of $\mathbf{H_1}, \ldots, \mathbf{H_n}$ via G** is the graph $G(|H_1, \ldots, H_n|)$ where each vertex v_i of G has been replaced by the graph H_i. We thus have

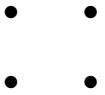
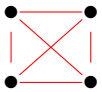
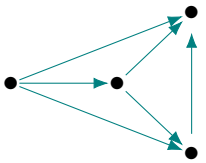
$$V_{G(|H_1,\ldots,H_n|)} = \bigcup_{i=1}^{n} V_{H_i}$$

$$E_{G(|H_1,\ldots,H_n|)} = \left(\bigcup_{i=1}^{n} E_{H_i} \right) \cup \{ (h_i, h_j) \mid h_i \in V_{H_i}, h_j \in V_{H_j}, (v_i, v_j) \in E_G \}$$

The previous definition allows us to view graphs as operators on graphs. We thus define for all $n \geq 2$ the $\parr_n$, $\otimes_n$ and $\triangleleft_n$ operators to correspond to the following prime graphs:

- $\parr_n$: The graph of n vertices with no edges
- $\otimes_n$: The graph of n vertices where all edges are connected
- $\triangleleft_n$: The graph of n vertices where the edges form a transitive chain going through all of the vertices

Example 2.2.2.

$\parr_4$:  $\otimes_4$:  $\triangleleft_4$: 

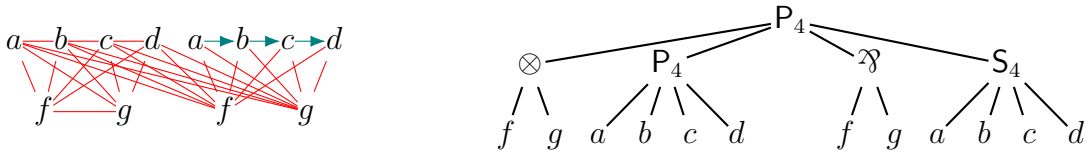
From this point forward, we omit the $_n$ index from the above graphs/operators when it is clear from context.

Theorem 1 ([15, 8])**.** *Let G be a graph such that $|V_G| = n \geq 2$. Then, there are non-empty graphs $H_1, \ldots, H_n$ and a prime graph P such that $G = P(|H_1, \ldots, H_n|)$.*

Corollary 1.1. *Each graph admits a **modular decomposition** by prime graphs, creating a tree where the leaves are labelled with vertices and nodes are labelled with prime graphs. Such a decomposition is unique [7] modulo associativity of $\parr$, $\otimes$ and $\triangleleft$.*

By abuse of notation, we represent these **modular decomposition trees** (MDT) by drawing the labels of vertices instead of the vertices themselves. We write $\parr$, $\otimes$ and $\triangleleft$ for the n-ary prime connectives that extend the prime graphs of size 2 and we write P for arbitrary prime graphs.

Example 2.2.3 (A Graph and its Modular Decomposition Tree [2, page 9])**.**



Definition 2.2.4. Let G be a graph, $V \subseteq V_G$ be a set of vertices and $v_{new} \notin V_G$ be a vertex. The graph obtained by **compressing** (or **condensing**) V into v_{new} is the graph G_{new} where

$$\begin{aligned} V_{G_{new}} &= (V_G \setminus V) \cup \{v_{new}\} \\ E_{G_{new}} &= (E_G \setminus \{(v_1, v_2) \mid v_1 \in V \text{ or } v_2 \in V\}) \\ &\quad \cup \{(v_{new}, v_2) \mid v_2 \in V_G \setminus V, v_1 \in V, (v_1, v_2) \in E_G\} \\ &\quad \cup \{(v_1, v_{new}) \mid v_1 \in V_G \setminus V, v_2 \in V, (v_1, v_2) \in E_G\} \end{aligned}$$

G_{new} is thus the graph obtained by replacing the subgraph of G induced by V by a single vertex, keeping the edges going in and out of the subgraph on the new vertex.

2.3 From Formulae to Graphs

In the following section, we consider undirected graphs and introduce the correspondence between logic formulae and graphs. We define a formula through the following grammar:

Formula Grammar

⟨formula⟩ ::= ⟨atom⟩ | ∘ | ⟨par⟩ | ⟨tensor⟩ | ¬ ⟨formula⟩
⟨par⟩ ::= (⟨formula⟩ ⅋ ⟨formula⟩)
⟨tensor⟩ ::= (⟨formula⟩ ⊗ ⟨formula⟩)
⟨atom⟩ ::≈ `[a-z][a-z]*`

Here, we use the logical connectives $\parr$ (par) and $\otimes$ (tensor) from linear logic [11]. We have also equipped our grammar with the unit $\circ$ and the negation $\neg$. Logically, the negation is defined inductively by setting $\neg\neg a = a$ for a an atom and defining the De Morgan laws for two formulae ϕ and φ:

$$\neg(\phi \parr \varphi) = (\neg\phi \otimes \neg\varphi) \quad \text{and} \quad \neg(\phi \otimes \varphi) = (\neg\phi \parr \neg\varphi)$$

Additionally, the unit is self-dual, i.e. $\neg\circ = \circ$. We thus consider from now on that the negation has been inductively applied from all formulas to their sub-formulas such that only atoms are (singly) negated. Given a set $\mathcal{A}$ of atoms, we define its completion $\Omega_{\mathcal{A}}$ as $\mathcal{A} \cup \{\neg a \mid a \in \mathcal{A}\}$.

We also note that given formulas ϕ, φ and γ, the following isomorphisms on formulae hold:

- Associativity: $\phi \parr (\varphi \parr \gamma) \equiv (\phi \parr \varphi) \parr \gamma, \quad \phi \otimes (\varphi \otimes \gamma) \equiv (\phi \otimes \varphi) \otimes \gamma$
- Commutativity: $\phi \parr \varphi \equiv \varphi \parr \phi, \quad \phi \otimes \varphi \equiv \varphi \otimes \phi$
- Identity: $\phi \parr \circ \equiv \phi, \quad \phi \otimes \circ \equiv \phi$

Recall now the previously defined prime graphs $\parr$ and $\otimes$ (2.2.1). For these prime graphs we employ the notation $\parr(|G, H|) = G \parr H$ and $\otimes(|G, H|) = G \otimes H$ and treat these compositions as operations on graphs. We extend these operations to labelled graphs by unifying the labelling functions' domains and codomains. We can now inductively define the graph associated to a formula ϕ.

Definition 2.3.1. Given a formula ϕ and the set of atoms $\mathcal{A}_\phi$ appearing in ϕ, we define its **associated graph** $[\![\phi]\!] = (V_{[\![\phi]\!]}, E_{[\![\phi]\!]})$, labelled by $\Omega_{\mathcal{A}_\phi}$, according to the structure of ϕ:

$$
\begin{aligned}
[\![\circ]\!] &\to (\varnothing, \varnothing) && (2.3.1)\\
[\![a]\!] &\to (\{v_i\}, \varnothing), l_{[\![\phi]\!]}(v_i) = a && (2.3.2)\\
[\![\neg a]\!] &\to (\{v_i\}, \varnothing), l_{[\![\phi]\!]}(v_i) = \neg a && (2.3.3)\\
[\![\varphi \parr \Phi]\!] &\to [\![\varphi]\!] \parr [\![\Phi]\!] && (2.3.4)\\
[\![\varphi \otimes \Phi]\!] &\to [\![\varphi]\!] \otimes [\![\Phi]\!] && (2.3.5)
\end{aligned}
$$

Here, when we write a in a formula, such as in the LHS of (2.3.2), we mean an element $a \in \mathcal{A}$. When we write v_i in the vertex set of a graph, such as in the RHS of (2.3.2), we are uniquely identifying the atom a on the LHS with v_i, while also labelling v_i with a: $l_{[\![\phi]\!]}(v_i) = a$. This uniqueness is important so that no atom is lost during the graph operations $\parr$ and $\otimes$.

Definition 2.3.2. A **cograph** is a graph that may be constructed recursively as follows:

- For any vertex a, $(\{a\}, \varnothing)$ is a cograph
- Given a family of cographs $G_1, \ldots, G_n$, $(\bigcup_{i=1}^n V_{G_i}, \bigcup_{i=1}^n E_{G_i})$ is a cograph
- Given a cograph G, $\bar{G}$ is a cograph

Remark 2.3.1. The cographs correspond exactly to the graphs that are P_4-free, that is to say the graphs that do not have P_4 as an induced subgraph of a set of four vertices.

Theorem 2 ([13, 17]). *Let G be a graph, there exists a formula ϕ such that $[\![\phi]\!] = G$ if and only if G is a cograph.*

This theorem tells us that standard proof systems defined on formulas that correspond to our grammar are also proof systems on cographs. This fact is well known and we do not go into the details of the inference rules of the proof system. We do, however, require some notion of implication for such a proof system, which we define using the notion of dual or negation of a graph.

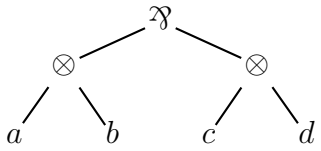
Definition 2.3.3. For an $\Omega_{\mathcal{A}}$-labelled graph G, the **dual** of G is denoted $\neg G = (V_{\neg G}, E_{\neg G})$ and has the same edge and vertex sets as $\bar{G} = (V_G, E_{\bar{G}})$, the complement of G. The label of $v \in V_{\neg G}$ is then the dual of the label of v in G, that is to say: $l_{\neg G}(v) = \neg l_G(v)$. $\neg G$ is then also an $\Omega_{\mathcal{A}}$-labelled graph.

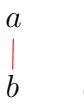
We can now define the (linear) **implication** in the usual way: between two graphs $G \multimap H$ is defined as $\neg G \parr H$. We can now extend our proof system on cographs to a proof system on graphs called GS [1] (*graphical proof system*). The proof system on cographs mentioned section corresponds to multiplicative linear logic (MLL) with mix [5, 9, 11]. GS then extends MLL (what is provable in MLL is also provable in GS) and constitutes a valid proof system once equipped with the proper inference rules [1].

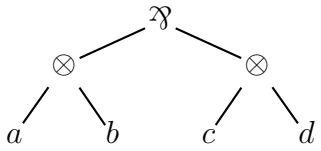
Finally, notice the correspondence between a formula's term tree and the modular decomposition of a graph:

Example 2.3.1.

$\phi \coloneqq (a \otimes b) \parr (c \otimes d) \quad \rightarrow$



$G \coloneqq$  $\rightarrow$



Indeed, the MDT of a graph G corresponds exactly to the formula that G represents. The modules are then the subformulas and the vertices correspond to the atoms through a labelling function. This paradigm of MDT-as-formula extends past MLL and continues to apply for logic on directed cographs [20], logic on general graphs [1] and in work currently being done by L. Strassburger regarding logic on general directed graphs.

3 | The Algorithm

We base our implementation on the algorithm detailed in the first paper on modular decomposition [15]. Although the original algorithm contains an error (see Remark 3.2.1), we retain the original idea and extend it to directed graphs. The final algorithm thus functions on both directed and undirected graphs.

The modular decomposition algorithm proceeds by successively compressing the graph until only one vertex is left. This is done in two distinct steps. We first detect the prime graphs of the form $\mathcal{Y}$, $\otimes$ and $\triangleleft$ and compress them until there are none in the graph. We then look for the smallest maximal modules and compress them:

Algorithm 1: Modular Decomposition Overview

Input : An atomic graph, graph
Output : The graph condensed module-by-module to a single vertex

while *length(V_{graph}) > 1* **do**
  `/* Compress all` $\mathcal{Y}$`, ` $\otimes$ `and` $\triangleleft$ `until there are none */`
  do
    prevGraph = graph;
    graph = Compress $\mathcal{Y}$, $\otimes$ and $\triangleleft$; `// Algorithm 2`
  while *prevGraph* $\neq$ *graph*;
  graph = Compress smallest maximal modules ; `// Algorithm 3`
return *graph*;

3.1 Connective Compression

Let G be a graph. We use the following method to identify modules of size 2. let $v_1, v_2 \in V_G$, if $\text{suc}(v_1) \setminus \{v_2\} = \text{suc}(v_2) \setminus \{v_1\}$ and $\text{pred}(v_1) \setminus \{v_2\} = \text{pred}(v_2) \setminus \{v_1\}$ then v_1 and v_2 share a module. Then:

- if v_1—v_2 then v_1 and v_2 are both under the same $\otimes$ node
- if v_1 ····· v_2 then v_1 and v_2 are both under the same $\mathcal{Y}$ node

- if $v_1 \frown v_2$ or $v_2 \frown v_1$ then v_1 and v_2 are both under the same $\triangleleft$ node

To compress the modules corresponding to the $⅋$, $\otimes$ and $\triangleleft$ connectives, we iterate over all pairs of vertices and compare their successors and their predecessors. If they share the same successors and predecessors, we know that they belong to the same module and we can use the edge relation they share to determine what type of connective they correspond to. The following algorithm details the procedure:

Algorithm 2: Compression of $⅋$, $\otimes$ and $\triangleleft$ connectives

Input : A graph
Output : The graph where all the connective modules have been compressed

```
modules = emptySet;
addedVertices = emptySet;
for v_i ∈ V_graph do
    for v_j ∈ V_graph, v_j ≠ v_i do
        if shareModule(v_i, v_j) ;                    // Remark 3.1.1
        then
            if v_i ∈ addedVertices then
                addToSameModule(modules, v_i, v_j);
                addedVertices.add(v_j);
            else if v_j ∈ addedVertices then
                addToSameModule(modules, v_j, v_i);
                addedVertices.add(v_i);
            else
                addedVertices.add(v_i);
                addedVertices.add(v_j);
                if v_i ∈ suc(v_j) then
                    if v_j ∈ suc(v_i) then
                        modules.add(Tensor([v_i, v_j]));
                    else
                        modules.add(Before([v_i, v_j]));
                else if v_j ∈ suc(v_i) then
                    modules.add(Before([v_j, v_i]));
                else
                    modules.add(Par([v_i, v_j]));
```

Remark 3.1.1. In Algorithm 2, we know that two vertices v_i and v_j share the same connective module when $\mathrm{suc}(v_i) \setminus \{v_j\} = \mathrm{suc}(v_j) \setminus \{v_i\}$ and $\mathrm{pred}(v_i) \setminus \{v_j\} = \mathrm{pred}(v_j) \setminus \{v_i\}$.

Note that the procedure described in Algorithm 2 may miss the transitivity of

some configurations (for example, it only detects the first two vertices in a before operation since those are the only ones that are detected by `shareModule()`). To remediate this, we compress nodes during the creation of the tree (see subsection 3.3).

3.2 Maximal Module Compression

Once we have removed all the potential connectives from our graph, we are tasked with finding the smallest maximal prime subgraphs. This requires us to first find all of the maximal prime subgraphs, before selecting the smallest ones. In order to find all of the maximal prime subgraphs, it is easiest to compute the maximal prime subgraphs containing a given edge. Doing this for all edges, choosing the smallest prime subgraph associated to an edge containing a specific vertex, then removing the largest of two prime subgraphs when they intersect gives us a list of disjoint maximal prime subgraphs, which we can then compress. The procedure is described in Algorithm 3.

Algorithm 3: Smallest Maximal Module Compression

Input : A graph G
Output : G' obtained by compressing the smallest maximal modules of G

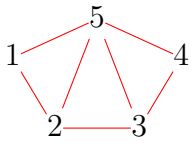
```
smallestMaxModules = array[length(V_G)];
possibleMaxModules = array[length(V_G)][emptySet];
for (v_i, v_j) ∈ E_G do
    possibleMaxModules[i].add(maxModule({v_i, v_j}));
    possibleMaxModules[j].add(maxModule({v_i, v_j}));

for v_i ∈ V_G do
    smallestMaxModules[i] = smallest(possibleMaxModules[i]);

res = set(smallestMaxModules);
for M_i ∈ smallestMaxModules do
    for m_j ∈ M_i, m_j ≠ m_i do
        if |M_j| ≥ |M_i| then
            res.remove(M_j);
        else
            res.remove(M_i);

for vset ∈ res do
    G = compress(G, vset);
return G;
```

Finding the maximal module containing a set (the `maxModule()` function) is

done by looking at which vertices are related by one of the edge relations to some but not all of the vertices in the set. Note that this holds for both directed and undirected graphs. We repeat this procedure while we keep finding new vertices and add them to our set. This yields the maximal module containing our original set, which in Algorithm 3 is a set of vertices connected by an edge.

Remark 3.2.1. In the original paper on the modular decomposition of graphs [15], the algorithm we adapted into Algorithm 3 is incorrect. Indeed, when comparing the size of the smallest maximal modules, the original article added the constraint that we only consider vertices $m_j \in M_i$ such that $j > i$. This condition causes errors when i is the last index but the module M_i is not minimal.

Example 3.2.1 (Counterexample to the original algorithm)**.**



If the vertices of the previous graph are indexed by their label, then running the original algorithm on it fails as it fails to see that the smallest maximal module defined on 5 *is larger than the smallest maximal modules defined on the other vertices.*

3.3 Tree Recovery

During compression, we need to preserve the structure of every prime subgraph being compressed (otherwise we are left with a single vertex we cannot unroll after the algorithm is done). In order to do so, we keep a global `state` variable containing a hash table which maps from the ids to vertices. We can them simply store any vertex we are about to compress as we know that once it is compressed it will not be changed by the algorithm any further.

Once the compression algorithm has finished, we are then left with two elements from which we wish to recover our MDT: a compressed graph with a single vertex and the `state` with its hash table. This is sufficient for us to unfold our condensed graph into a tree naturally (see Figure 4.3 for the correspondence from the condensed graph vertices to the tree nodes). We start from our single-vertex graph and get its `id`. We then find the corresponding vertex v and compute its MDT according to Algorithm 4.

Algorithm 4: Tree Unfolding Algorithm

Input : A vertex v of the condensed graph
Output : The MDT of the graph with root v

```
rec unfoldTree(v):
    /* Get the set of vertices that were condensed into v    */
    childVertices = children(v);
    successors = emptyList;
    for child in childVertices do
        successors.push(unfoldTree(child));
    node = createNode(v, successors);
    /* node depends on the connective of v                   */
    return node
```

4 | Our Implementation

From the previous section, we know to convert a graph to its MDT, our program finds the prime graphs composing our graph, compressing them into single vertices. We do this recursively until we have obtained a graph with a single vertex. The last vertex becomes the root of our MDT, which we can then "unfold". This procedure leaves us with several design choices/questions. What type do we define to represent a graph? How do we make this type work with compressing vertices? How do we represent the modular decomposition tree? We detail the various types we use during this process in the following sections.

4.1 Graphs

In order to define graphs, we first need to define vertices. Keeping in mind our characterisation of the modular decomposition, we know that the vertices of our graph will be labelled with a (possibly negated) atom. In order to represent vertices, it is thus sufficient to use a label from some set of labels corresponding to atoms, a polarisation to indicate whether the atom is negated, and a unique identifier.

```
type atom = {
    label : string;
    pol : bool;
}
```

```
type node =
  | Atom of atom
  | Tensor of ISet.t
  | Par of ISet.t
  | Before of int list
  | Prime of IMap.t
```

Figure 4.1: Definition of graph nodes

A compressed vertex must contain the information of the structure that we just compressed and we thus use the `node` type defined in Figure 4.1, paired with a unique id to represent a vertex:

```
type vertex = { connective : node; id : int; }
```

Now that we have defined out vertex, we can defined our graph in the following

way:

```
type graph = {
    nodes : VSet.t;
    edges : VMap.t;
    edges_from : VMap.t;
 }
```

Where `VSet.t` is the type of sets of vertices and `VMap.t` is the type of maps (dictionaries) with vertices as keys and sets of vertices as values. The representation of a directed graph is as follows: `nodes` is the set of vertices of our graph; `edges` is a map from vertices to their successors; `edges_from` is a map from vertices to their predecessors. If the graph is undirected, we simply use `edges` to represent the nodes connected to a vertex.

4.2 Modular Decomposition Trees

We would like to use standard recursive types to define our MDT. However, we run into a problem when representing the prime nodes of our graph. We would like our prime nodes to represent a graph structure where the nodes in the graph are MDTs. But MDTs should themselves be allowed to carry prime graphs as nodes. Although it is possible to create such a structure with mutually recursive types, we find that the structure is quickly bloated and somewhat difficult to work with. We opt instead to represent the structure of the prime graphs as follows:

```
type id_graph = { nodes : int list; edges : (int * int) list; }
```

Here, we start using lists and tuples instead of sets and maps as the main operations we want to do on these structures are serializing and deserializing. Thus, `nodes` is a list of identifiers corresponding to the identifiers of the children of the prime node in the tree. Similarly, `edges` is the list of edges between the children of the prime node in the tree. Although we still require mutually recursive types, they are now much lighter, and we can define our MDT type as shown in Figure 4.2.

```
type connective =
  | Atom of atom
  | Tensor of tree list
  | Par of tree list
  | Before of tree list
  | Prime of id_graph *
     (tree list)
```

```
and tree = {
    connective :
       connective;
    id: int;
    }
```

Figure 4.2: MDT type definition

We do not include a type for the empty tree as it adds overhead without providing any modelling benefits. We consider the tree lists of `Tensor` and `Par` to be unordered whereas in the the tree list of `Before` carries with it the order of application. Indeed, `Before [id1; id2; id3]` corresponds to the graph $\triangleleft(|G_3, G_2, G_1|)$. The head of the list is thus the last element in the transitive chain.

4.3 State and Subsets

During compression, we need to store somewhere that a compressed vertex v corresponds to such or such prime graph in order to ensure that we do not lose the structure of our graph. We use a state variable with a hash table that takes identifiers as keys and maps them to the vertex with the corresponding `id`. We also store the total number of vertices in order to ensure that vertices created during compression have unique ids:

```
type state = {
  mutable total_vertices : int;
  id_map : (int, Vertex.t) Hashtbl.t;
}
```

Before compression, we need to not only store the set of vertices we are intending on compressing but also mark the set depending on the type of prime graph we are currently detecting. As such, we define

```
type subset =
  | Singleton of vertex
  | Clique of VSet.t (* Corresponds to Tensor *)
  | Before of vertex list
  | IndSet of VSet.t (* Correspons to Par *)
```

4.4 From Condensed Graph to MDT

Once we have totally compressed our graph, we are ready to unfold it into a modular decomposition tree. This is done by the means of Algorithm 4, where we get a vertex from it's `id` using the `state.id_map`. We find the children of a given vertex v using the information stored in v`.node`. The procedure we follow then depends on the type of v`.node`:

- v`.node` = `Atom atom`: return the tree composed of a single node corresponding to atom
- v`.node` = `Tensor iset`: recursively compute a list of subtrees from the ids in iset, if any of the subtrees are also tensors, replace them in our subtree list with the elements of their own subtree list

- v`.node` = `Par iset`: recursively compute a list of subtrees from the ids in iset, if any of the subtrees are also pars, replace them in our subtree list with the elements of their own subtree list
- v`.node` = `Before int list`: recursively compute a list of subtrees from the ids in int list, if any of the subtrees are also befores, replace the subtree in the successor list with the elements of the subtree's subtree list
- v`.node` = `Prime imap`: create an `id_graph` from `imap` and recursively compute the list of subtrees corresponding to the nodes of `id_graph`

```
type node =
    | Atom of atom
    | Tensor of ISet.t
    | Par of ISet.t
    | Before of int list
    | Prime of IMap.t
```

$\longrightarrow$

```
type connective =
    | Atom of Graph.atom
    | Tensor of tree list
    | Par of tree list
    | Before of tree list
    | Prime of id_graph *
      (tree list)
```

Figure 4.3: Comparison of graph node and tree node connective types

Recursively continuing this procedure gives us a natural way of constructing the MDT.

4.5 From a MDT to its Graph

The structure of the MDT allows us to use a simple recursive algorithm to construct the corresponding graph [18]. We need only recursively find the vertices corresponding to the successors of a node in the tree before drawing edges between them as specified by the connective of the node of the tree. For example, we show in Figure 4.5 how we deal with the $\triangleleft$ connective.

```
    ...
| Before tl ->
      let nel = List.map tl ~f:(tree_to_graph_r) in
      let nodes, edges = List.fold nel ~init:(Set.empty
         (module Graph.Vertex), [])
        ~f:(fun (vsetacc, elacc) (vset, el) ->
          let vertices = Set.union vsetacc vset in
          let edge_base = el @ elacc in
          let edges = join_sets ~symmetric:false vset vsetacc
              in
          vertices, edges @ edge_base)
      in
      nodes, edges
    ...
```

Figure 4.4: Transformation from Before tree node to Before graph vertex

5 | Interactivity Features

5.1 Interacting with the Graph

The main purpose of our tool is to allow the exploration of graphs and their structure. To this end, we allow the user to draw vertices and edges; to delete vertices and edges and to move vertices around the canvas. The following section may serve as a user manual for these interactions.

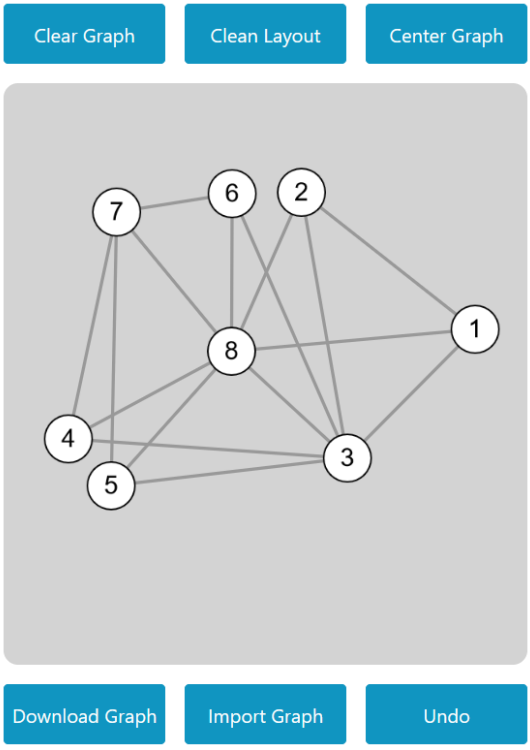


Figure 5.1: Cy1, the canvas corresponding to graph interaction

5.1.1 Mouse inputs

Although the cytoscape.js library supports tactile interactions, we only define behaviour for inputs using a mouse or track-pad.

Action	Command
Selecting a vertex or an edge	Shift + Click to successively select elements. Shift + Click + Drag to create a selection box.
Moving around the canvas	Click + Drag on the background.
Moving a vertex or selection	Click + Drag on one of the elements.
Deselecting elements	Click the background
Drawing Edges	Select source(s) and click a target vertex.
Negating a selection	Right Click one of the elements.

5.1.2 Keyboard inputs

Action	Command
Creating a new vertex	Type an alphanumeric character while hovering over the desired location of the vertex.
Remove a selection	Pressing backspace will remove the current selection.

5.1.3 Buttons

The graph canvas is equipped with several buttons

- *Clear Graph*: Removes all vertices from the graph (adds them to the undo queue)
- *Clean Layout*: Uses a physics simulation to compute a force-directed layout called fCose [4]. Is randomized and will not produce the same result when run multiple times.
- *Center Graph*: Centers the viewport on the graph.
- *Download Graph*: Downloads the graph serialized into Json.
- *Import Graph*: Allows the user to upload a json file describing the graph they would like to work on. Clears the previous graph before adding the new one.
- *Undo*: Undoes the previous graph topology change, whether that be the addition or deletion of elements.

5.2 Interacting with the Decomposition Tree

Since we are interested in the manipulation of the structure of graphs, we allow the user convert from a modular decomposition tree to a graph.

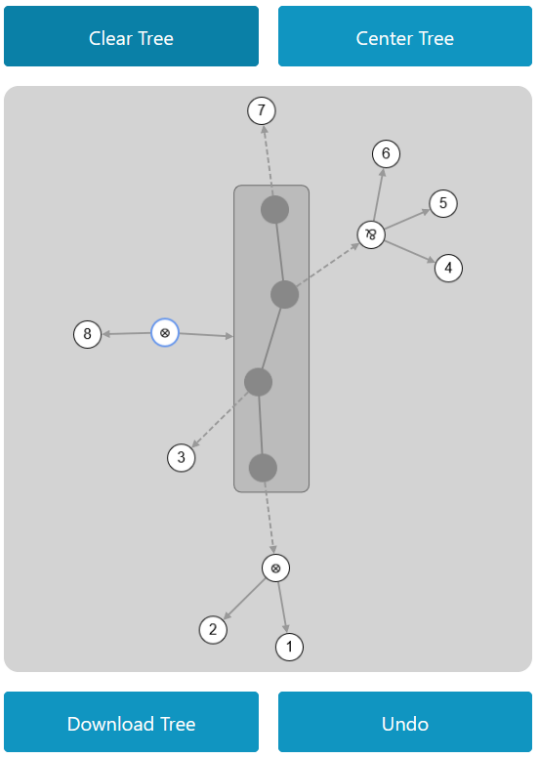


Figure 5.2: Cy2, the canvas corresponding to tree interaction

5.2.1 Mouse Inputs

Action	Command
Selecting a node or an edge	Shift + Click to successively select elements. Shift + Click + Drag to create a selection box.
Moving around the canvas	Click + Drag on the background.
Moving a node or selection	Click + Drag on one of the elements.
Deselecting elements	Click the background
Drawing Edges	Select a single source and click a target vertex, only allowed edges will be drawn.
Negating a selection	Right Click one of the elements.
Setting the root	Double Click on a node.
Add a node to a compound	Select the compound and Click on the node

5.2.2 Keyboard inputs

Action	Command
Creating a new leaf	Type an alphanumeric character while hovering over the desired location of the leaf.
Create a new node	Type one of the characters given by Figure 5.3
Removing a selection	Pressing backspace will remove the current selection.
Adding to a compound	Select a compound and add a node in the usual way

"&" : ⅋ "*" : ⊗ ">" : ◁ "^" : compound

Figure 5.3: Character to node mapping

5.2.3 Buttons

The tree canvas is equipped with several buttons

- *Clear Tree*: Removes all nodes from the tree (adds them to the undo queue)
- *Center Tree*: Centers the viewport on the tree.
- *Download Tree*: Downloads the tree serialized into Json.
- *Undo*: Undoes the previous tree topology change, whether that be the addition or deletion of elements.

5.3 Additional Features

We provide with the implementation three additional buttons. The user may toggle whether they wish to work on *directed or undirected* graphs. The user may *compute the modular decomposition* of the graph in Cy1 and display it as a MDT in Cy2 with the *Get Modular Decomposition* button. The user may *compute the graph* corresponding to the tree in Cy2 and display it in Cy1 with the *Get Graph* button. Note that a tree will only be read if all of the nodes are connected to the root.

When downloading a graph or a modular decomposition tree, we serialize to Json according to Figure 5.4 and Figure 5.5 respectively.

```
{
  "nodes": [
    {
      "id": 1,
      "label": "1",
      "polarisation": true
    },
    ⋮
  ],
  "edges": [
    {
      "source": 1,
      "target": 2
    },
    ⋮
  ]
}
```

Figure 5.4: Serialization of a graph

```
{
  "node": { "connective": "tensor", "id": 12 },
  "successors": [
    {
      "node": { "connective": "atom", "id": 8 },
      "successors": []
    },
    ⋮
  ]
}
```

Figure 5.5: Serialization of a tree

6 | Running the code

6.1 Calling OCaml Functions from the Browser

As previously mentioned, the frontend of the project is implemented with the Cytoscape.js library. We make use here of two canvases (or `cytoscape` objects) to represent the full graph in one (cy1) and the modular decomposition tree in the other (cy2). We use the default interactions provided by the library to select vertices and edges, and to grab and moves vertices.

We implement directly in JavaScript all the interactions that remain within a single canvas, such as adding and deleting vertices, drawing edges, etc. We thus restrict the communication between the JavaScript library and the Ocaml code to a minimum and are left with the following interactions between the two:

1. Reading and drawing on the graph canvas
2. Reading and drawing on the MDT canvas
3. Calling graph decomposition and tree recomposition from the frontend

Thankfully, the Js_of_Ocaml package makes it quite easy to interact between JS and Ocaml. For example, we are able to call arbitrary JavaScript functions from Ocaml, including Cytoscape.js functions. For example, the two following lines of code thus return the same object.

```
(* Ocaml *)
let root_arr = cy##nodes (Js.string ".root")

/* JavaScript */
let root_arr = cy.nodes(".root")
```

We then expose the following functions from Ocaml to allow their use from JavaScript:

- `decompose()`: Read the graph in cy1, compute its modular decomposition tree, draw the tree in cy2

- `recompose()`: Read the tree in cy2, compute the corresponding graph, draw the graph in cy1
- `isPrime(graph)`: Given a graph in the format of a cytoscape collection, return whether it is prime or not

Choosing this format means that, once compiled, all the necessary algorithms can be run in JavaScript by the browser. The experience from the user's point of view is then the same as opening any other html file.

6.2 Compiling the Code

Given that the project is designed to run on a server **or** locally, the reader may be interested in building and running the program themselves. The code is available at `https://github.com/remyjck/modular_decomposition` and can be cloned using

```
git clone https://github.com/Remyjck/modular_decomposition.git
```

To build the code, we require the Ocaml Package Manager (opam). We also require the installation of Ocaml 4.13 or higher. Once these prerequisites are installed, install the ocaml libraries used in the project by running the following command from inside the repository.

```
opam install . ——deps—only
```

Once the libraries have been installed, the project can be built using:

```
dune build src/main.bc.js
```

This produces a JavaScript file that is used by `index.html` to run the modular decomposition algorithm. All that remains is then to open `index.html` with a browser to run the project.

Some unit tests are included in the project, to run them, it is necessary to first build the full project. This can be done in one command:

```
dune build && dune runtest
```

The above works for Unix-based systems (Linux, MacOS) but compiling on windows may prove difficult. The easiest solution is to use the Windows Subsystem for Linux (WSL) and compile the project from there.

7 | Conclusion

We have created a tool that allows the user to interactively build graphs in order to compute their modular decomposition. This tool works for both directed and undirected graphs that are labelled by either atoms or their negation. We also allow the creation of modular decomposition trees in order to compute their graphs. Additional features provided are the possibility to upload and download graphs and modular decomposition trees in Json format.

The interactivity features are intended to enable logicians to easily build and modify graphs without having to serialize them by hand in order to view them. We thus hope that this tool will be of use for future research on logic and graphs. Indeed, the notions presented in sections 1.1 and 2.3 are only the foundations of the relation between graphs and logical connectives.

We can also note that we have not found any online tools for modular decomposition other than this tool. It may thus be used not only for logic but also for graph theory in general, whether for practical or educational purposes.

7.1 Future work

There are several ways in which the tool can be extended. There are always improvements to the GUI that can be made, such as extending the tool to work with touchscreens and allowing vertex re-labelling. Some more important possible future changes are the implementation of a linear time modular decomposition algorithm [22] and the possibility to view the modular decomposition as nested graphs (not unfolding the compressed graph into a modular decomposition tree, rather drawing compound nodes where the root of the tree is the parent of all compound nodes).

This tool may also serve as a first prototype for working with logics on graphs. A natural predecessor would be a tool for applying inference rules on graphs (such as the inference rules found in GS). To this end, the code for the modular decomposition could certainly be reused as well as some interactivity elements.

Acknowledgments

This bachelor thesis was conducted under the supervision of Lutz Strassburger, whom I would like to extensively thank for his guidance and structuring of the project. I would like to also thank the whole PARTOUT team at INRIA for the warm welcome they offered me. In particular, I would like to thank Bahar Carabetta for making the administrative parts of my internship as smooth as possible. Finally, I would like to specifically thank Giti Omidvar for helping me around the office during the early days of the thesis and Prof. Kaustuv Chaudhuri for his technical advice.

References

[1] M. Acclavio, R. Horne, and L. Straßburger. Logic beyond formulas: a proof system on graphs. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, pages 38–52, Saarbrücken Germany. ACM, July 8, 2020.

[2] M. Acclavio, R. Horne, and L. Strassburger. An analytic propositional proof system on graphs, 2020.

[3] M. Acclavio and R. Maieli. Generalized connectives for multiplicative linear logic. In *CSL 2020 - 28th EACSL annual conference on Computer Science Logic*, volume 152, 6:1–6:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Jan. 2020.

[4] H. Balci and U. Dogrusoz. Fcose: a fast compound graph layout algorithm with constraint support. *IEEE Transactions on Visualization and Computer Graphics*:1–1, 2021.

[5] G. Bellin. Subnets of proof-nets in multiplicative linear logic with mix. *Mathematical Structures in Computer Science*, 7(6):663–669, 1997.

[6] K. Chaudhuri. Subformula Linking for Intuitionistic Logic with Application to Type Theory. In *CADE 2021 - 28th International Conference on Automated Deduction*, volume 12699, pages 200–216. Springer International Publishing, July 2021.

[7] M. Chein, M. Habib, and M. C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37(1):35–50, Jan. 1, 1981.

[8] A. Ehrenfeucht, T. Harju, and G. Rozenberg. *The Theory of 2-Structures: A Framework for Decomposition and Transformation of Graphs*. WORLD SCIENTIFIC, Aug. 1999.

[9] A. Fleury and C. Retoré. The mix rule. *Mathematical Structures in Computer Science*, 4(2):273–285, 1994.

[10] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311, Jan. 15, 2016.

[11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[12] J.-Y. Girard. Multiplicatives. In G. Lolli, editor, *Logic and Computer Science: New Trends and Applications*, pages 11–34. Rosenberg & Sellier, 1987.

[13] A. Guglielmi. A system of interaction and structure. *ACM Trans. Comput. Logic*, 8(1):1–es, Jan. 2007.

[14] M. Habib and C. Paul. A survey on algorithmic aspects of modular decomposition, 2009.

[15] L. James, R. Stanton, and D. Cowan. Graph decomposition for undirected graphs. *Utilitas Mathematica*, Jan. 1, 1972.

[16] R. M. McConnell and J. P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 536–545, USA. Society for Industrial and Applied Mathematics, Jan. 23, 1994.

[17] R. Möhring. *Computationally tractable classes of ordered sets*. In Jan. 1989, pages 105–193.

[18] C. Papadopoulos and K. Vogklis. Drawing graphs using modular decomposition. In volume 11, Sept. 2005.

[19] C. Retoré. Handsome Proof-nets: R&B-Graphs, Perfect Matchings and Series-parallel Graphs. Research Report RR-3652, INRIA, 1999.

[20] C. Retoré. Pomset Logic as a Calculus of Directed Cographs. Research Report RR-3714, INRIA, 1999.

[21] C. Retoré. Pomset logic: a non-commutative extension of classical linear logic. In P. de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 300–318, Berlin, Heidelberg. Springer Berlin Heidelberg, 1997.

[22] M. Tedder, D. Corneil, M. Habib, and C. Paul. Simple, linear-time modular decomposition. *arXiv:0710.3901 [cs]*, Mar. 20, 2008. arXiv: 0710.3901.

[23] A. Tiu. A system of interaction and structure ii: the need for deep inference. *ArXiv*, abs/cs/0512036, 2006.

[24] J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. *Softw. Pract. Exper.*, 44(8):951–972, Aug. 2014.

A | Examples

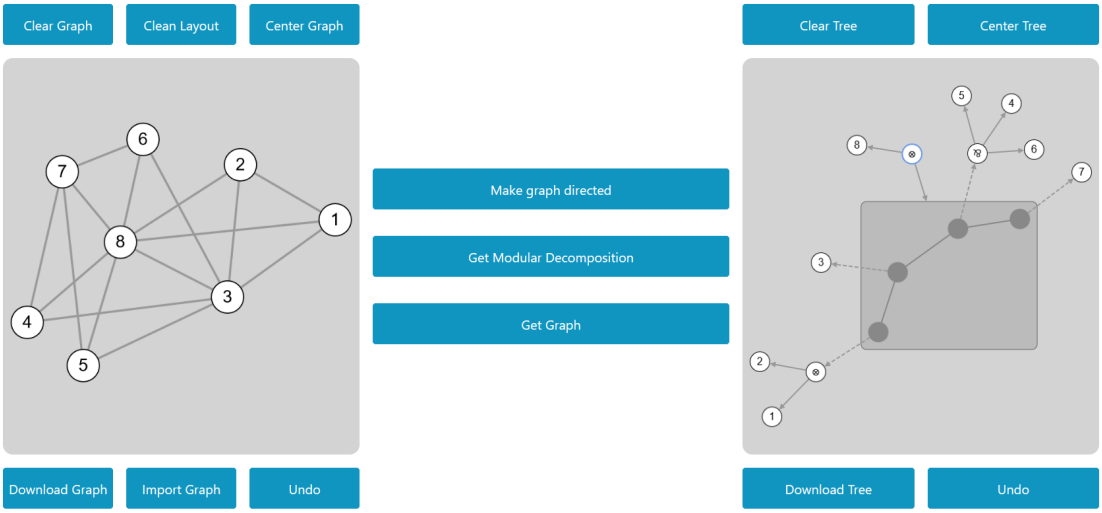


Figure A.1: Modular Decomposition of an Undirected Graph

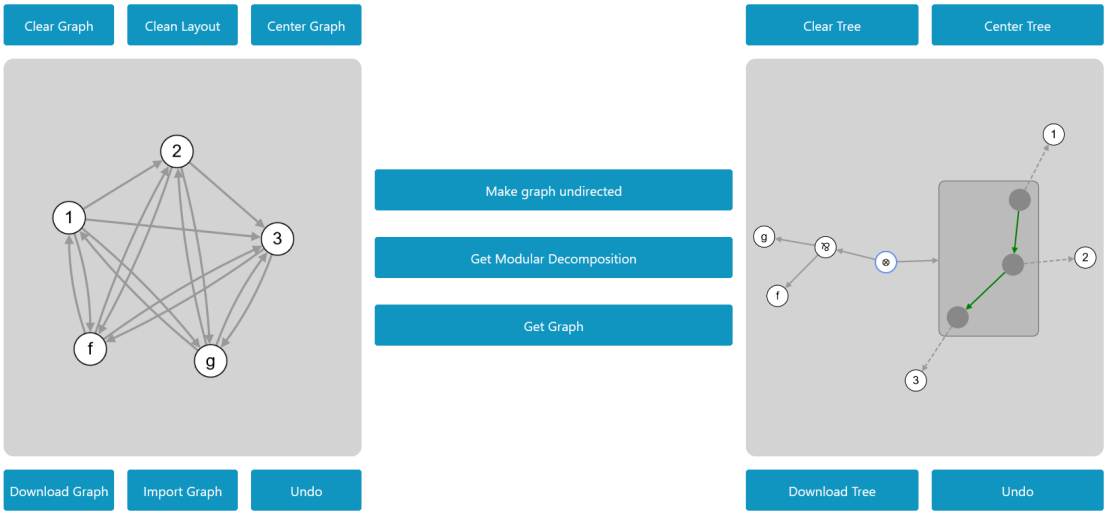


Figure A.2: Modular Decomposition of a Directed Graph