UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Data analysis through graph decomposition

Marie Ely Piceno

Data analysis through graph decomposition

A thesis presented for the degree of
Doctor of Philosophy

Marie Ely Piceno

Advisor:
José Luis Balcázar

Department of Computer Science
Universitat Politècnica de Catalunya
Barcelona, Spain

Contents

List of Figures

Chapter 1

Introduction and preliminaries

1.1 Introduction

This work is a development within the field of data mining and data visualization. Thus, in order to analyze and explore data, we study some graph decomposition methods. Our work consists in the search for relationships between items on data to identify co-occurrence patterns through the construction and decomposition of graphs, the so-called Gaifman graph and its variations. Throughout this thesis we argue the usefulness of Gaifman graphs on first-order relational structures as an exploratory data analysis tool, we provide the theory that supports our work and we explain the algorithmics implemented.

As a definition of data mining we have: "data mining is the process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems. It is the analysis step of the knowledge discovery in databases process" [1]. In turn, the basic flow of steps that compose the KDD process are selection, preprocessing, transformation, data mining and integration/evaluation [17].

On the other hand, "data visualization is the presentation of data in a pictorial or graphical format, and a data visualization tool is the software that generates this presentation. Data visualization provides users with intuitive means to interactively explore and analyze data, enabling them to effectively identify interesting patterns, infer correlations and causalities, and supports sense-making activities" [7]. As examples of data visualization we have Pyra-

[1] https://en.wikipedia.org/wiki/Data_mining

midViz [23] or arulesViz [20] that aim at explaining visually frequent patterns or association rules.

Highly frequent co-occurrences of data items are often of interest in data analysis, and they have been a target for several sorts of data mining frameworks for decades now, on all types of data. Despite the large amount of existing literature, current notions do not really reach end users, mainly due to the difficulty of finding explanatory schematic descriptions. In fact, one of the main setbacks is that, mathematically speaking, the results of these notions of data analysis are in spaces of great dimensionality, and their reductions to the 2D or 3D frames available so far almost never offer a sufficient interpretation. However, graphic descriptions add greatly to the interpretability of the results of the data analysis in many fields. Through this work we demonstrate that our approach is convenient in the analysis of data, specifically in the analysis of medical diagnostic data.

We propose the use of Gaifman graphs and the application of a decomposition mechanism on them based on the so-called 2-structures to obtain a hierarchical co-occurrence visualization.

The structure of the thesis is as follows. In this chapter we provide a general view of preliminaries and the notation used.

In Chapter 2, based on the intuition that the results of this decomposition could be related to the closure space obtained from the so-called modular implications and clan implications, we develop their relationship where we introduce the construction of this set of implications and we explain the decomposition graph as a variant of the closure space associated to this set. This provides a fundamental study of the process behind our proposal, by explaining how graph decompositions fit a specific view of closure spaces, allowing us to connect our approach to more standard tools in Data Analysis, such as closure and implication mining.

In the next chapter, Chapter 3, we show the algorithm developed by us to get the corresponding graph decomposition, the theorems that support our algorithms and we give the details of the software implementation. Also we comment on a couple of related algorithms, showing by examples the differences with our algorithm.

Chapter 4 is a compendium of some of the decomposition results applied on Gaifman graph variations from different datasets using different parameters. While in Chapter 5 we show the result of applying the 2-structure decomposition method on different Gaifman graph variants of a medical diagnostic dataset.

Finally, in Chapter 6 we talk about the profit of applying this method but also we talk about restrictions that we found and some of possible expansions that can be made or considered.

1.1.1 Publications

Throughout the development of this thesis we have had some publications. There are references to them in most of the chapters of this thesis. We list them below.

- *Decomposition of quantitative Gaifman graphs as a data analysis tool*, Advances in Intelligent Data Analysis XVII - 17th International Symposium, IDA 2018, 's-Hertogenbosch, The Netherlands [5].
- *A graphical tool for the interpretation of medical data*, ACM Celebration of Women in Computing: womEncourage 2019, Rome, Italy [34].
- *Hierarchical visualization of co-occurrence patterns on diagnostic data*, 32nd IEEE International Symposium on Computer-Based Medical Systems, CBMS 2019, Cordoba, Spain [6]. Obtained the Best-student paper award.
- An extended version that includes both previous papers was invited to a special issue of the Computational Intelligence Journal with the name: *Co-occurrence patterns in diagnostic data* [27].

There is an additional publication, which in fact was our first publication, however because the focused changed through its development, it is not reference in any of the chapters. This publication is:

- *Relative entailment among probabilistic implications*, Logical methods in Computer Science, February 2019 [2].

Additionally, we have another publication that right now is in a submission process to a journal: *Hierarchical visualization of co-occurrence patterns as clans in generalized Gaifman graphs* [4]. It contains the theory behind the connection of modular decomposition and its generalization, clan decomposition, to closure and implication mining, that is, the results related to the Chapter 2.

Meanwhile, in [5] we expose the bases of our approach, we first apply it on a Titanic dataset as a way to encourage its use, the decomposition results are shown in the present chapter as motivation.

In the rest of the publications, we show the results of working with variations of Gaifman graphs from more elaborated datasets, such as the medical diagnostic dataset among others. All these results are shown in Chapters 4 and 5. To be precise, in [6] we show some results of working with linear and exponential Gaifman graph variants, while, in [34] we add the analysis of the shortest path Gaifman graph variant. There is an additional contribution, K-means Gaifman graph variant, which has not been sent for publication yet but whose results of applying it on the medical diagnostic dataset are also shown in Chapter 5.

1.2 General preliminaries

Throughout this work we call universe, denoted as $\mathcal{U}$, some fixed set of atomic elements or items. An itemset is a subset of the universe. To represent subsets of items we use capital letters, and to represent single items we use lower case letters.

Let X and Y be two sets, if X is a subset of Y, denoted as $X \subseteq Y$, each element of X is also an element of Y, they can even be the same sets. To denote the subset X cannot be equal to Y, that is, X is a proper subset of Y, we use $X \subset Y$.

For practicality, in order to represent the union of X, Y, we use the juxtaposition XY.

We say that two sets *overlap* if neither is a subset of the other, but they are not disjoint. That is:

Definition 1.1. *X and Y overlap if the three sets $X \cap Y, X \setminus Y$, and $Y \setminus X$ are not empty.*

1.3 Implications and partial implications

Association mining [8] is a method of data analysis. It studies the relation between two itemsets, this relation is commonly called partial implication. To select interesting partial implications, constraints on various association quality measures are used, most of the times constraints on their confidence and on their support.

Definition 1.2. *The support of an itemset X in a dataset $\mathcal{D}$ is the number of transactions that contain X in the dataset, $\mathcal{D}_X$, divided by the cardinality of the dataset, n: $S_{\mathcal{D}}(X) = \frac{\mathcal{D}_X}{n}$.*

From a dataset $\mathcal{D}$, the universe $\mathcal{U}$ will be conformed by those items in the transactions, when we work with transactional databases, or will be conform by all possible attribute values, when we work with relational databases; we will go in detail latter in Section 1.5. Let $X, Y \subseteq \mathcal{U}$ be two itemsets, by definition we may deduce that $S_{\mathcal{D}}(X) \leq S_{\mathcal{D}}(Y)$ when $Y \subseteq X$. The support of the rule $X \to Y$ in a dataset $\mathcal{D}$, is again the number of transactions where XY appear together in the dataset, $\mathcal{D}_{XY}$, divided by the cardinality of the dataset: $S_{\mathcal{D}}(X \to Y) = S_{\mathcal{D}}(XY) = \frac{\mathcal{D}_{XY}}{n}$.

Definition 1.3. *The confidence of a rule $X \to Y$ in a dataset $\mathcal{D}$ is defined as: $C_{\mathcal{D}}(X \to Y) = \frac{S_{\mathcal{D}}(XY)}{S_{\mathcal{D}}(X)}$,*

An *implication* is a rule with confidence equal to 1, denoted as $X \Rightarrow Y$. Implications are directly related to conjunctions of Horn clauses, so they have a closure space associated. A Horn clause, in fact a definite Horn clause, is a clause disjunction of possibly negated propositional variables, with exactly one non-negated variable. We represent Horn formulas in implicational form by grouping together into a single expression $X \Rightarrow Y$ all the Horn clauses with the same set X of negated attributes, each contributing their positive attribute to Y. For sets of items X, Y and Z, Z satisfies the implication $X \Rightarrow Y$, denoted as $Z \models X \Rightarrow Y$, if either $X \nsubseteq Z$ or $XY \subseteq Z$.

For a set of implications $\mathcal{B}$:

- $Z \models \mathcal{B}$ means $Z \models \wedge_{\mathcal{B}}(X \Rightarrow Y)$.
- $\mathcal{B} \models X \Rightarrow Y$ means for every $Z \models \mathcal{B}$ we have $Z \models X \Rightarrow Y$.

The connection of classical implications and closure spaces runs as follows: Given $\mathcal{B}$ a set of implications, the closure $\overline{X}$ of a set X is the largest set Y such that $\mathcal{B}$ logically entails $X \Rightarrow Y$; whereas, if we are given a closure operator, we can axiomatize it by the set of implications $\{X \Rightarrow Y : Y \subseteq \overline{X}, X \subseteq \mathcal{U}\}$ or, equivalently, any set of implications that entails exactly this set. Thus $\mathcal{B} \models X \Rightarrow Y$ if and only if $Y \subseteq \overline{X}$.

A set is a *closed set* if it coincides with its closure, and the closure space is the family of all the closed sets. The fact, well-known in logic and knowledge representation, that Horn theories are exactly those closed under bitwise intersection of propositional models leads to a strong connection with closure spaces, where closure under intersection always holds [13] [21]. A basic fact from the theory of closure spaces is that the closure operator is characterized by three properties:

- Extensivity: $X \subseteq \overline{X}$
- Idempotency: $\overline{\overline{X}} = \overline{X}$
- Monotonicity:if $X \subseteq Y$ then $\overline{X} \subseteq \overline{Y}$

For references and supporting facts of all our claims so far, see the discussions in [41] or its early version `https://arxiv.org/abs/1411.6432v2`.

Definition 1.4. *$\overline{X}$ is a strong closure if, for all other closures $\overline{Y}$, $\overline{X}$ and $\overline{Y}$ do not overlap. That is, either $\overline{X}$ and $\overline{Y}$ are disjoint, or a subset of one another.*

1.4 Graphs

A graph is a structure determined by a set of vertices and a set of edges. It could be a directed graph or an undirected graph. An undirected graph $\mathcal{G}$ is described by a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ that define a symmetric relation on the vertices, that is, let $x, y \in \mathcal{V}$, if $(x, y) \in \mathcal{E}$, thus $(y, x) \in \mathcal{E}$.

The complement graph $\mathcal{G}'$ of a graph $\mathcal{G}$, is a graph defined on the same set of vertices where the edges are determined by those pair of vertices that are not adjacent in $\mathcal{G}$.

Definition 1.5. *In a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, two vertices $x, y \in \mathcal{V}$ are adjacent if $(x, y) \in \mathcal{E}$.*

P_4 is the graph consisting of a path on 4 vertices with 3 edges; its complement is also P_4 [9]. As an example, Figure 1.1 shows two different P_4 graphs in one of them we make explicit its complement showing that it is also a P_4.

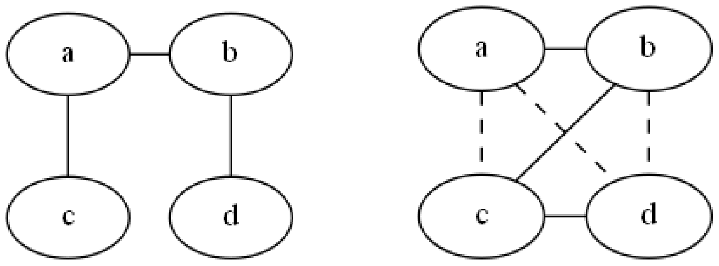


Figure 1.1: Example: P4 graphs.

1.5 Gaifman graphs of relational structures

The theory of Gaifman graphs was developed in the 1980's; it is a quite natural theoretical construction that can be applied to any relational structure [24]. A relational structure consists of a domain and a set of relations. As we know, relational databases, often composed of multiple relations (or: "tables"), have been a key figure in the theory and practice of Computation for decades. It is possible to construct its associated Gaifman graph, since the conceptual essence of the relational database model [24] is the first order relational structure.

Given a first order relational structure $\{\mathcal{R}_i\}_{i\in I}$ where the values that appear in the tuples of the relations $\mathcal{R}_i$ come from a fixed universe $\mathcal{U}$, its corresponding Gaifman graph has the elements of $\mathcal{U}$ as vertices, and the edges (x, y), for $x \neq y$, are determined exactly when x and y appear together in some tuple $t \in \mathcal{R}_i$ for some $\mathcal{R}_i$.

Thus, it could be applied directly on relational dataset where the relations $\mathcal{R}_i$ will be the tables in the database, the tuples $t \in \mathcal{R}_i$ will be those rows in the tables and the set of vertices $\mathcal{U}$ will be determined by all possible attribute values, and the edge (x, y) will be determined exactly when the attribute values x and y appear together in some row.

Example 1.6. *Let us consider a very small relational dataset on the universe* $\mathcal{U} = \{a_0, a_1, b_0, b_1, b_2, b_3\}$ *conformed by the tuples:*

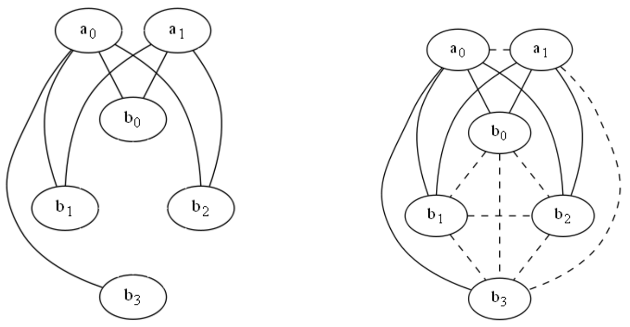


Figure 1.2: Graph of Example 1.6: Standard Gaifman graph and its natural completion.

$$\begin{aligned}
t_0 &: \; a_0 b_0 \\
t_1 &: \; a_0 b_1 \\
t_2 &: \; a_0 b_2 \\
t_3 &: \; a_0 b_3 \\
t_4 &: \; a_1 b_0 \\
t_5 &: \; a_1 b_1 \\
t_6 &: \; a_1 b_2
\end{aligned}$$

The Gaifman graph that represents its co-occurrences is shown in the left of Figure 1.2. According to the definition, the vertices are all the possible attribute values and the edges link pair of attributes values that appear together in some row. An alternative drawing, that will fit the 2-structure-based approach described shortly, is the natural completion of the Gaifman graph, shown in the right of Figure 1.2. Extending the definition of natural completion of a graph to Gaifman graph we have:

Definition 1.7. *The natural completion of a Gaifman graph $\mathcal{G}$ is the Gaifman graph plus an equivalence relation among its absent edges, getting a complete graph with two equivalences classes: the items that sometimes appear together in some tuple of some $\mathcal{R}_i$ are joined by an edge in one equivalence class, while the items that never appear together are joined by edges in another equivalence class.*

In most of our examples we use solid lines to join items that appear together while the items that never appear together are joined by broken lines.

We also may extend, in a natural way, the construction of the Gaifman graph from a transactional dataset. Transactional datasets, also known as "market-basket datasets" [25], consist of a sequence of transactions, each of which consists, in turn, of a set of items. Then, the Gaifman graph from a transactional dataset will have as set of vertices $\mathcal{U}$ all the possible items found in the transactions, and the edge (x, y) will be determined exactly when the items x and y appear together in some transaction.

If we want to see a transactional dataset as a relational dataset, each item will be an attribute with binary values: the presence or absence of it in the original transaction. In most practical cases, those zeros representing absence of an item abound, yet we are only interested in items being jointly present, so the zeros are not informative. This is the reason of using the Gaifman graph transactional construction as we have just described.

Example 1.8. *Let us consider a very small dataset, quite similar to that shown in our work [5], on the universe $\mathcal{U} = \{a, b, c, d, e\}$ conformed by the transactions:*

$$\begin{array}{rc} t_1: & abc \\ t_2: & ae \\ t_3: & acd \\ t_4: & de \end{array}$$

The Gaifman graph that represents its information is shown in left of Figure 1.3.

According to the definition, each vertex of the graph represents an item, and edges link pairs that appear together in some transaction. Its natural completion Gaifman graph variant is shown in center of Figure 1.3, where items that sometimes appear together are joined by solid lines, while items that never appear together are joined by broken lines, again, leading to a complete graph with two classes of edges.

Gaifman graphs record co-occurrence (or lack of it) among every pair of attribute values, or items, of the universe $\mathcal{U}$, but there are cases where it is useful to have quantitative information about the co-occurrences. We will

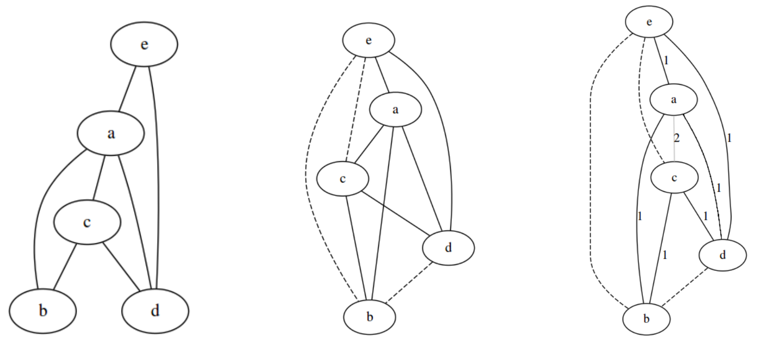


Figure 1.3: Graph of Example 1.8: Standard Gaifman graph, its natural completion and the labeled variant.

propose, in Chapter 3, several natural ideas to enhance their symbolic capabilities through different ways of considering this quantitative information. As a first approach, we have its labeled variation.

Definition 1.9. *The labeled Gaifman graph variation of a Gaifman graph, is a graph where the edges are labeled according to the multiplicity of the items that they connect, that is, according to the number of transactions that contain the pair of items linked by the edge.*

The right of Figure 1.3 shows the labeled Gaifman graph variant of the Example 1.8. In an intuitive way we represent the different labels with different colors. In this way, some pairs do not appear together and the corresponding edges are labeled with zero and represented by dashed edges, those pairs that appear together exactly once are labeled 1 and represented by black lines, and the pair that appears two times is labeled 2 and represented by a gray line.

Hence, a clique in a standard Gaifman graph would group items that, pairwise, appear together somewhere in the relational structure: co-occurrence patterns; a clique in its complement would reveal an incompatibility pattern. Of course, finding maximal cliques is NP-complete; but there are less demanding ways to study graphs that identify efficiently both sorts of patterns in a recursive decomposition: namely, the modular decomposition and its generalization, the decomposition of 2-structures.

1.6 Modular decomposition

The modular decomposition was first described in the 1960's by Gallai [18][2]; it has been rediscovered many times and described under many different names, for a survey see [19] and [31][3]. The modular decomposition of a graph is a process that consists of decomposing the graph into sets of vertices, nowadays called *modules.*

Definition 1.10. *Given a graph, a set X of vertices is a module if, for each vertex $z \notin X$, either every member of X is connected with z or every member of X is not connected with z.*

Note that the set of all vertices is a module. Also, each vertex by itself and the empty set are vacuously modules, commonly called *trivial modules*; we will systematically ignore the empty module.

As an intuition, we consider the presence or absence of an edge as the way one vertex *sees another.* Thus, we will often resort to expressions like "a vertex *sees in a different way* two other vertices" when it is connected to one of them and disconnected from the other one; or, in the same case, we may say that "one vertex *can distinguish* between other two vertices".

Proposition 1.11. *Permuting absence and presence of all edges leaves the same set of modules.*

That is, a graph and its complement have the same modules. The previous proposition comes from the fact that the permutation of the edges, to connect those disconnected edges and disconnect those connected edges, does not change the ways vertices distinguish each other; that is, we give the same interpretation to a graph and to its complement.

Let $\mathcal{M}$ be a module and x, y be elements in $\mathcal{M}$, by definition every $z \notin \mathcal{M}$ may not distinguish between them. Thus, if z may distinguish between them, z must be in $\mathcal{M}$:

Proposition 1.12. *Let x and y be elements of a module $\mathcal{M}$: if item z sees in a different way x and y then, necessarily, $z \in \mathcal{M}$.*

Proposition 1.13. *Let X and Y be overlapping sets, if both are modules, then also $X \cap Y$, $X \setminus Y$, and $Y \setminus X$ are modules.*

[2]An English translation is available as [26]

[3]See also `https://en.wikipedia.org/wiki/Modular_decomposition`

Let X and Y be modules, thus every $z \notin X$ may not distinguish the elements of X and every $z \notin Y$ may not distinguish the elements of Y. If $X \cap Y \neq \emptyset$, those elements in the intersection cannot be distinguish by any $z \notin X$ and also by any $z \notin Y$, that is, they cannot be distinguish by any $z \notin X \cap Y$, thus, $X \cap Y$ is a module. $X \setminus Y$ is equal to the $x \in X$ such that $x \notin X \cap Y$, since X is a module the only z that could distinguish the elements of $X \setminus Y$ are those $z \in X \cap Y$. Suppose $X \setminus Y$ is not a module thus, there is $z \in X \cap Y$ that may distinguish two arbitrary $x_0, x_1 \in X \setminus Y$ thus $(z, x_0) \neq (z, x_1)$. As $z \in Y$ and Y is a module, for all of the remaining $z_i \in Y$: $(x_0, z_i) = (x_0, z)$ and also $(x_1, z_i) = (x_1, z)$. Thus, will be exist $z_i \notin X$ such that $(z_i, x_0) \neq (z_i, x_1)$, contradicting the fact that X is a module.Therefore, $X \setminus Y$ is a module. The same for $Y \setminus X$.

A main interest of the notion of module is that all the vertices of a module can be collapsed into a single vertex without ambiguity with respect to how to connect it to the rest of the vertices: the new vertex gets connected to y if all the members of the module were connected to y, and remains disconnected if all the members were disconnected. Clearly, the definition given of module is what is needed for this process to be applied without hesitation about whether the new vertex should or should not be connected to some external vertex y. More generally, the same considerations apply if we consider two disjoint modules:

Proposition 1.14. *If X, Y are disjoint modules then either $\forall x \in X \forall y \in Y, (x, y) \in \mathcal{E}$ or $\forall x \in X \forall y \in Y, (x, y) \notin \mathcal{E}$*

Nothing forbids modules to intersect each other; in that case, though, collapsing one module into a single vertex may affect the other. In order to avoid side effects, it is customary to restrict oneself to so-called *strong modules* [18] (see also [26]).

Definition 1.15. *A module $\mathcal{M}$ is a strong module of a graph if it does not overlap any other module; that is, for all other modules $\mathcal{M}'$ of the graph, either $\mathcal{M} \cap \mathcal{M}' = \emptyset$ or they are subsets of one another.*

Since strong modules can be proper subsets of other modules, we can obtain a hierarchical decomposition of the graph, commonly called *decomposable set family* [29], that can be pictured in a tree-like form. These facts have been studied in a very wide bibliography, see [19] and the references there.

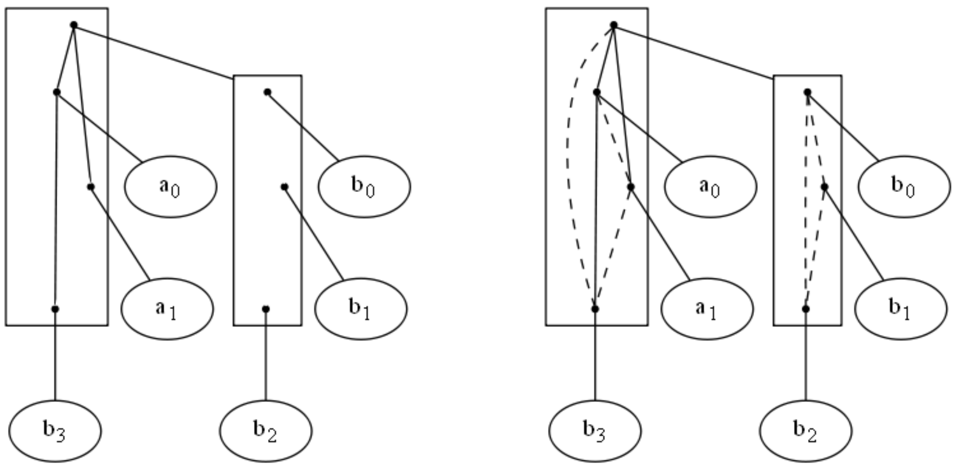


Figure 1.4: Graph of Example 1.6: Standard graph decomposition and its natural completion decomposition.

Given a graph, we can focus on its maximal strong modules; it is known that each vertex belongs to exactly one of them [19]. Thus, one or more (or even all) of these maximal strong modules can be collapsed into a single vertex.

Definition 1.16. *The coarsest quotient graph of a given graph $\mathcal{G}$ is obtained by collapsing all the maximal strong modules of $\mathcal{G}$.*

Then, a tree-like structure arises from the fact that each of these modules, taken as a set of vertices, is actually a subgraph that can be recursively decomposed, in turn, into maximal strong modules, generating views of subsequent internal structures given by their respective coarsest quotient graphs. We display the decomposition tree while labeling each node (a strong module) with its corresponding coarsest quotient graph, and connect visually each collapsed vertex to the subtree decomposing the corresponding module. Of course, the root of the tree is the coarsest quotient of the whole graph.

Let us see some examples of the modular decomposition method application.

The modular decomposition of the standard Gaifman graph in the left of Figure 1.2 for the Example 1.6, is shown in Figure 1.4. And also, the decomposition of its natural completion is shown in Figure 1.4.

As a generality, boxes corresponds to sets that are strong modules, and dots within them to subsets that are also strong modules. All along the whole decomposition, trivial (single-item) modules are indicated by a link to the vertex they consist of, represented with an elliptic node; nontrivial ones are linked instead to a new box describing the internal structure of the module, in terms of the strong modules it has again as proper subsets.

At the rightmost box, three of the b values are connected by broken lines, which means that they never co-occur together: indeed, as different values of the same attribute, no row can have two of them. In the other box, the top node is a condensed version of the module formed by b_0, b_1, and b_2, and it is connected with solid lines to both a_0 and a_1, meaning that all the ways of pairing one of the a's with one of these b's do appear in the data. Like the attribute values b_i, the values a_i do not appear together in any row and, if we are decomposing a natural completion, they appear connected by a broken line. More interestingly, we could have expected to see the attribute value b_3 in the same module of the rest of the values b_i; however, the fact that it appears in the higher module points out to us that, unlike the other b_i values, there is not any co-occurrence of b_3 with a_1. Of course, there are no co-occurrences of b_3 with the other b_i's either. That is, the items b_0, b_1, and b_2 "behave equally": all are connected to a_0 and a_1 and all are disconnected to b_3; this is why they conform a module. But b_3 cannot join them since it "behaves differently" with respect to a_0 and a_1, as it co-occurs with a_0 in the data but not with a_1.

The modular decomposition of the standard Gaifman graph in the left of Figure 1.3 for the Example 1.8, is displayed in the left Figure 1.5. Also, the decomposition of its natural completion is shown in right of Figure 1.5.

One can see, in the top of the figure, the root of the tree collapsing the maximal strong modules a and one nontrivial maximal strong module conformed by $\{b, c, d, e\}$: all vertices not in the module (that is, vertex a) are connected to each vertex inside of the module through edges with the same color, by solid lines. Any other candidate turns out not to be a strong module. For instance, any set including a and b but excluding e is not a module, as e distinguishes between a and b; then, any set including b and e must including c and d. We end up including all the vertices, that is, becoming a trivial module. Analogous reasoning applies if we start by pairing a with any other vertex.

That is the way this decomposition tells us, in the particular case of our very small Example 1.8, that item a in our given relation does not distinguish

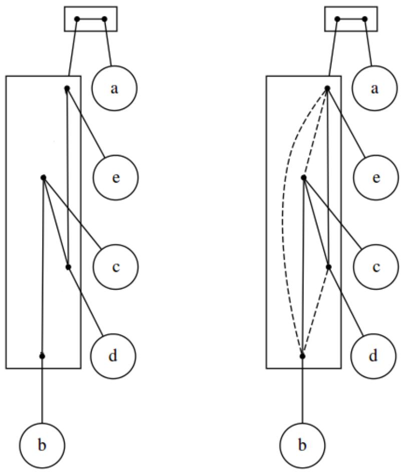


Figure 1.5: Graph of Example 1.8: Standard graph decomposition and its natural completion decomposition.

all the others, in the sense that it shares some tuple with each one of the rest, while the others do not offer any specific pattern in this respect.

The type of the strong modules is determined by the way the members of its coarsest quotient are connected. To maintain consistency, for some notions we will be using terminology corresponding to the theory of 2-structures (Section 1.7). Given a graph if all of its vertices are connected, or if all of its vertices are disconnected, we say that it is *complete*, otherwise it is *primitive*. As an alternative definition, a graph is primitive if, for any two vertices in the graph, there is a third one that is connected with just one of them. By convention, graphs of size 1 or 2 are considered complete.

This, if the coarsest quotient graph of a strong module is a complete graph, the strong module is a *complete module*, otherwise it is a *primitive module*. For example, in Figure 1.5 the module conformed by $\{b, c, d, e\}$ is a primitive module since the edges are not in the same equivalence class, while in the top of the figure we find a complete clan having just one edge.

On the basis of the definition of module, we can state (see [19]):

Proposition 1.17. *Primitive graphs (and thus also their complements) always have induced P_4 subgraphs [37]. Hence, for a primitive module, its corresponding coarsest quotient graph consists of at least four maximal strong*

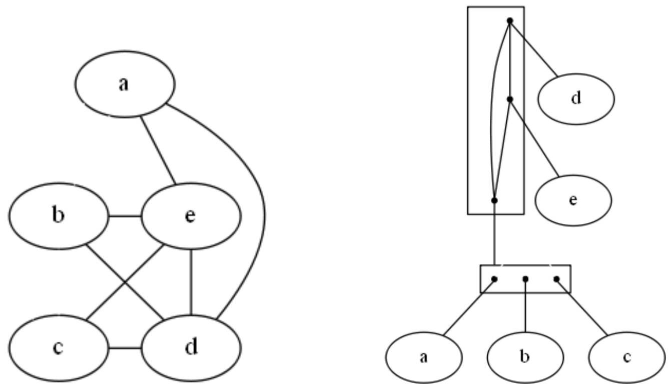


Figure 1.6: Example: A graph and its modular decomposition.

modules.[4]

Among other terms, primitive modules are sometimes called neighborhood modules. Modular decomposition theory distinguishes fully disconnected complete graphs (or "parallel" modules) and fully connected ones (or "series" modules), leading to often duplicated arguments and definitions because both cases fulfill the same role. Indeed, recall from Proposition 1.11 that presence or absence of edges can be swapped with no change in the modular structure: that's why fully connected and fully disconnected modules are treated similarly here. Hence, we prefer the view of naming them both "complete".

The decomposition of the Gaifman graphs in Figure 1.2 shown in Figure 1.4, shows two modules. One of them a primitive module conformed by $\{a_0, a_1, b_3\}$, where we may see the implied P_4, both for the initial standard graph (solid lines) and for its complement (broken lines). The other module is a complete module conformed by the items $\{b_0, b_1, b_2\}$.

The right of the Figure 1.5 shows the decomposition Gaifman graph of the natural completion Gaifman graph variant in the center of the Figure 1.3. It shows the implied P_4 in its primitive module, both for the initial standard graph and for its complement.

Another example is that one on Figure 1.6, it shows the original standard graph (connected and disconnected vertices) and its modular decomposition. It is easy to check that $\{a, b, c\}$ conforms a module: both d and e are con-

[4]See also `https://en.wikipedia.org/wiki/Cograph`

nected to each of them. The other possible nontrivial modules are $\{a, b, c, d\}$, $\{a, b, c, e\}$, $\{d, e\}$, $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$; they are not strong: each of them intersects others. Instead, $\{a, b, c\}$ is a strong module as it does not overlap any other module. The root is the fully connected coarsest quotient graph where $\{a, b, c\}$, the single nontrivial maximal strong module, has been collapsed to a single vertex. Each of the three vertices $\{\{a, b, c\}, d, e\}$ is connected to the module they represent: two are maximal strong but trivial ones, and the largest one decomposes itself again into three trivial modules.

Now, let us see the result of applying it on a famous, and relatively small dataset often used for teaching introductory data analysis courses: the Titanic dataset. Among several existing incarnations of the Titanic dataset, we employ a very simple one, prepared by Radford Neal in `http://www.cs.toronto.edu/~delve/data/titanic/desc.html` that, for each of the 2201 people on board the well-known ship, records the traveling class (`1st` class, `2nd` class, `3rd` class, `crew` member), age (discretized into `adult` or `child`), sex (`female` or `male`), and whether or not the person survived. Its modular decomposition is depicted in Figure 1.7.

The modules sex and survival are clear and intuitive, as they are different possible values for the same attribute, they never appear together, but happen to have the same set of neighbors. Likewise, one might expect a module with four alternative values of traveling class, namely, `1st`, `2nd`, `3rd` or `crew`. However, the closest such module is a complete, fully disconnected graph that only includes actual passenger classes that, of course never appear together. Instead, the value `crew` appears in the parent ages module, a small primitive graph where, of course, being an `adult` is into compatible with being a `child`, and both are compatible with all the traveling classes, however, `crew` co-occurs only with `adult`. Thus we are told that, of course, the crew included no children, a fact that we might overlook in a non-systematic analysis. That is, even if the traveling classes and the crew item are employed as values in the same column, the data tell us, through our decomposition procedure, that they have different semantics.

This small primitive graph is actually P_4 (Proposition 1.17), its coarsest quotient graph collapses four maximal strong modules, thus they must be the 4-vertex path. We will be seeing P_4 often again below.

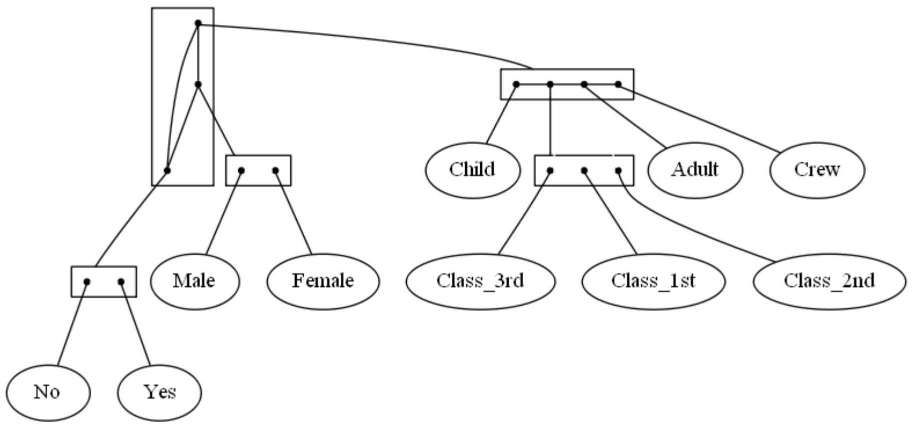


Figure 1.7: Modular decomposition: Titanic dataset graph.

1.7 2-structures and clan decomposition

According to the type of analysis that we would like to do, we can work with different variations of the Gaifman graphs since there are cases where it is useful to have quantitative information about the co-occurrences.

For a previous example, Example 1.8, we may see its labeled Gaifman graph variation in the right of Figure 1.3. As a result we have a graph with three classes of edges. While the notion of modular decomposition [18] is enough to be applied on standard Gaifman graphs, and even on their natural completion, it is insufficient to handle adequately other variations of Gaifman graphs. Therefore, we will work with a more general notion, 2-structures and their clans [16], that naturally generalize modular decompositions to more than two equivalence classes of edges.

The 2-structure concept can be studied in a number of ways; for example, somewhat unrelated to our study, a notion of "region" gives rise to a connection with Petri nets [3].

Definition 1.18. *A 2-structure consists of a finite not empty set of vertices $\mathcal{U}$, called domain, and an equivalence relation $\mathcal{E} \subseteq ((\mathcal{U} \times \mathcal{U}) \times (\mathcal{U} \times \mathcal{U}))$.*

In specific, we work with a special kind of 2-structures, the so called *symmetric 2-structures* [16] where all the equivalence classes are symmetric. An equivalence class is symmetric if all of its edges are symmetric, that is, the equivalence class $\mathcal{E}_i$ on a set $\mathcal{U}$ is symmetric if for all $x, y \in \mathcal{U} : (x, y) \in$

$\mathcal{E}_i \Leftrightarrow (y,x) \in \mathcal{E}_i$. Since in the 2-structures that we use the edges represent co-occurrences, is the same to say that x co-occurs with y than y co-occurs with x. Then, the equivalence classes of co-occurrences are symmetric equivalence classes. Hence, in the following when we talk about 2-structures we will be referring to symmetric 2-structures.

To visualize the graph we will employ the common, very graphical and intuitive representation of coloring in the same way edges belonging to the same equivalence class. Extending the distinguish concept, we will say that a node x *distinguishes* nodes y and z if the edge $(x,y) \in \mathcal{E}_i$ and the edge $(x,z) \in \mathcal{E}_j$, being $\mathcal{E}_i \neq \mathcal{E}_j$. Alternatively, as in modules, we say that y and z "are *seen* in a different way" by x.

For a 2-structure given by a set of vertices $\mathcal{U}$, let X be a nonempty subset of the domain $\mathcal{U}$, the induced substructure by X is defined as a 2-structure obtained from the original one by the restriction of the equivalence relation on the subset X, that is from the original 2-structure we only take those edges involving the nodes in X.

The notion now corresponding to a module is called a *clan*, for each $z \notin X$, z may not distinguish the elements of X. Formally:

Definition 1.19. *Given a 2-structure defined on $\mathcal{U}$, a set of vertices $X \subseteq \mathcal{U}$ is a clan if, for all $z \notin X$ and arbitrary different nodes $x, y \in X$, the edge from z to x is in the same equivalence class than the edge from z to y.*

Again, as in modules, there are some *trivial clans*, those clans are: the empty set, the singleton elements of the domain and the domain by itself.

Strong clans allow us to decompose a 2-structure into a tree-like form.

Definition 1.20. *A clan $\mathcal{C}$ is a strong clan of a 2-structure if it does not overlap any other clan.*

It is easy to see that, if two strong clans are disjoint, then all the edges between the elements of the clans are in the same equivalence class. Let X and Y be two disjoint sets of vertices, we may define an edge $\mathcal{E}_i$ between X and Y, $X, Y \subset \mathcal{U}$, like $(X,Y) \in \mathcal{E}_i$ if $\{(x,y) \in \mathcal{E}_i : \forall x \in X, \forall y \in Y\}$. Thus, two clans are connected, in the sense that all the respective pairs of vertices (one from each clan) are.

Similarly to strong modules, the strong clans provide us with a decomposable set family that can be pictured in a tree-like form, by displaying every strong clan as a child of the smallest strong clan that properly contains it.

Thus, as with modules, strong clans can be collapsed into single vertices without any ambiguity about how the 2-structure looks like after the collapse. The corresponding notion of *coarsest quotient* 2-structure follows by the same procedure as with modules:

Definition 1.21. *The coarsest quotient of a 2-structure is obtained by collapsing all the maximal strong clans.*

According to the internal 2-structure of a clan, it could be classified as a *complete* clan or as a *primitive* clan. In a complete clan all the edges are in the same equivalence class while in a primitive clan there are nothing but trivial clans. Originally, strong clans were called prime clans. However, in the context of modular decompositions, the adjective prime has received other usages in the literature (as use it to denote primitive clans), we have deemed better to avoid that adjective.

If the 2-structure is not symmetric, thus akin to a directed graph, it may exhibit a third basic component in its tree decomposition, *linear 2-structures* [16]. As we previously said, we only work with undirected graphs, since the equivalence classes described by the edges in Gaifman graphs are symmetric, so this component does not appear in our work. Mathematical theorems supporting all our claims on 2-structures are provided in [16].

1.8 Gaifman graphs as 2-structures

We have already indicated that it is possible to see a Gaifman graph as a 2-structure by its natural completion, Definition 1.7, where all the absent edges will be in the same equivalence class while the existing edges will belong to the other equivalence class. In this way we get a complete graph with two equivalence classes.

Taking the example of the Titanic dataset, the Figure 1.8 shows the decomposition of its natural completion Gaifman graph, where broken edges represent pairs of items that never appear together in any transaction, whereas solid lines join items that appear together in at least one transaction.

Thus, in the case of the natural completion Gaifman graph variant we have a 2-structure, while in the cases of the labeled Gaifman graph variant to have all the edges labeled by their multiplicity produce many equivalence classes and leads to many multiplicities so, often, no nontrivial clan shows

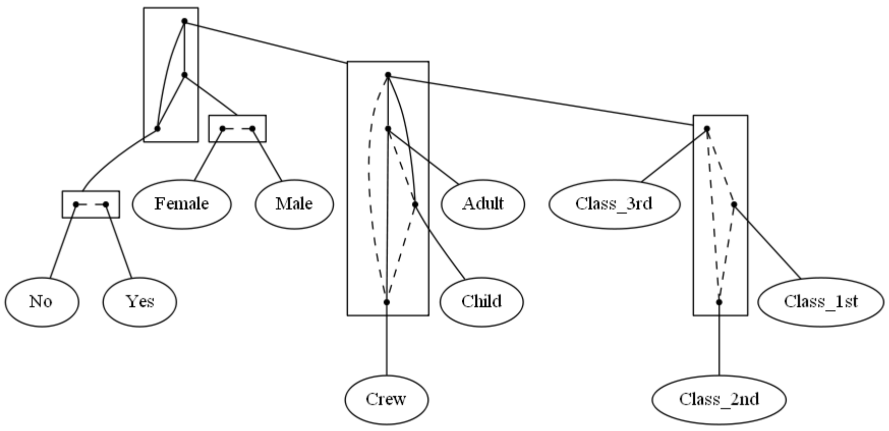


Figure 1.8: Clan decomposition: Titanic dataset natural completion graph.

up. In order to avoid it, we apply the following discretization methods (some of them introduced in our early works [5], [6], [34]).

As first discretization method we have the thresholded Gaifman graph variant. In the thresholded Gaifman graph variant we may assign a lower threshold or/and an upper threshold. The resulting Gaifman graph is a 2-structure with two equivalence classes: one of them for the labeled edges that satisfy the thresholds, and the other one, for the labeled edges below the lower threshold or/and over the upper threshold. If the label of the edge is below than the lower threshold, is consider that the linked vertices co-occur not enough. While, if the label of the edge is over than the upper threshold, is consider that the linked vertices are out of the range.

For the following discretizations we need some notation. Let $\mathcal{U}$ be the domain of a 2-structure, let x, y be vertices on it and let $c_{x,y}$ be the label for the vertices x and y. That is, $c_{x,y}$ denotes the quantity of co-occurrences of x and y. Let i represent the equivalence class $\mathcal{E}_i$.

In the linear Gaifman graph variant the edge (x, y) is in the equivalence class $\mathcal{E}_i$, $(x, y) \in \mathcal{E}_i$, if and only if $i = \lceil c_{x,y}/n \rceil$, being n the assigned interval size.

For the exponential version of the Gaifman graph we explored several formulas to determined the most precise one, that is a formula where we are not losing information, since in one of our versions we were losing the edges with just one co-occurrence. So, in our final version, the edge (x, y) is in the equivalence class $\mathcal{E}_i$, $(x, y) \in \mathcal{E}_i$, if and only if $i = \lceil \log_2(c_{x,y} + 1) \rceil$.

In the shortest path variant of the Gaifman graph, the edge (x, y) is in the equivalence class $\mathcal{E}_i$, $(x,y) \in \mathcal{E}_i$, if and only if i is the minimum distance between the vertices x and y in the standard Gaifman graph version.

We may combine the linear, exponential and shortest path Gaifman graph variants with the thresholded variant. In these cases we do not get just two equivalence classes as in the cases of the regular version of the thresholded variant, instead we just "disconnect" those values that are out of the threshold being able to get isolated vertices and more than one connex component. Thus, to combine the thresholded variant with the rest of the Gaifman graph variants does not change the equivalence classes defined on the edges with labels that do not fall out of the threshold.

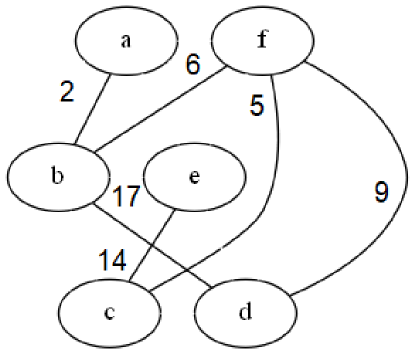
When we work with linear and exponential Gaifman graph variants, the threshold is on the multiplicities of the edges. Whereas, that for the shortest path Gaifman graph variant the threshold is on the path length between the vertices, that is, we are only interested in those paths whose lengths are within the thresholds.

Let us see an example.

Example 1.22. *In the left of Figure 1.9 is the labeled Gaifman graph variant of some data. Assume for the linear Gaifman graph variant we assign 10 as interval size then, the resulting graph is shown in the right of Figure 1.9. As one can see it has three different equivalence classes, while, the exponential Gaifman graph variation, left of Figure 1.10, has five equivalence classes, and in the shortest path Gaifman graph variation there are four equivalence classes. In this example, we can see how the edges may belong to different equivalence classes according to the Gaifman graph variant used.*

For example, in the linear variation the edges (b,d) and (c,e) are in the same equivalence class, while in the exponential variant they are in different equivalence classes. Another example are the edges (b,c) and (b,e), they are in the same equivalence class in the linear and in the exponential Gaifman graph variants, while in the shortest path variant they are in different equivalence classes since the size path to go from b to c is 2 and the size path to go from b to e is 3.

The Figure 1.11 shows the thresholded Gaifman graph variant for the standard and linear Gaifman graphs and the Figure 1.12 shows the thresholded variant for the exponential and the shortest path Gaifman graphs. We give the same threshold to the standard, linear and exponential Gaifman graph variations in order to more easily compare the graphs, being 5 the lower threshold and 15 the upper threshold. For the shortest path Gaifman graph variation



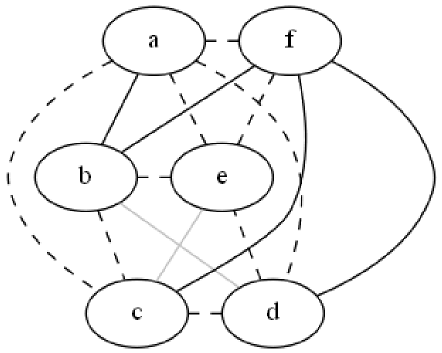
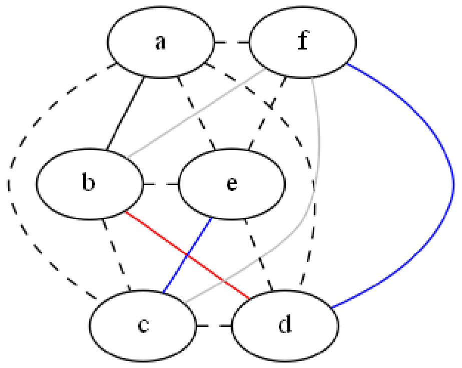


Figure 1.9: Graph of Example 1.22: Labeled and linear Gaifman graph variations.



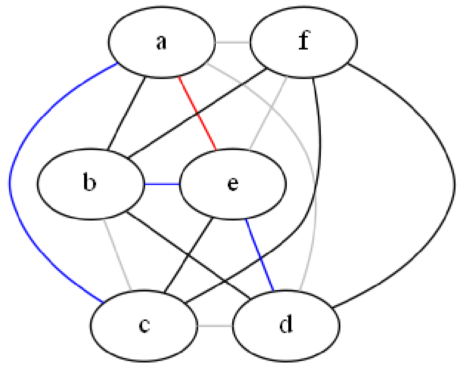


Figure 1.10: Graph of Example 1.22: Exponential and shortest path Gaifman graph variations.

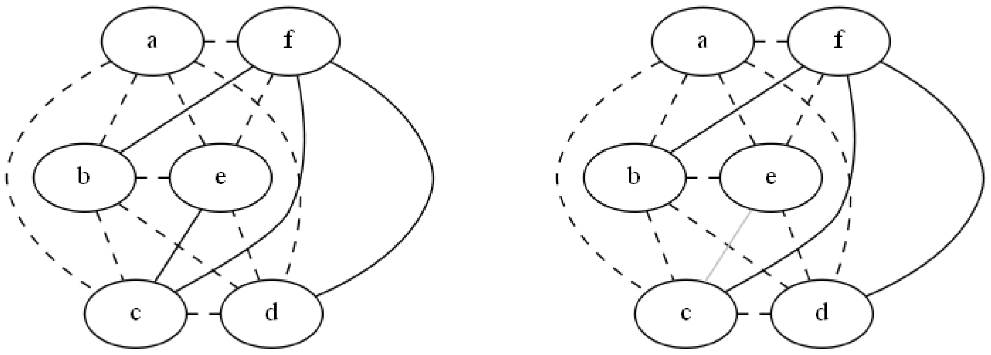


Figure 1.11: Graph of Example 1.22: Threshold variations for labeled and linear Gaifman graphs.

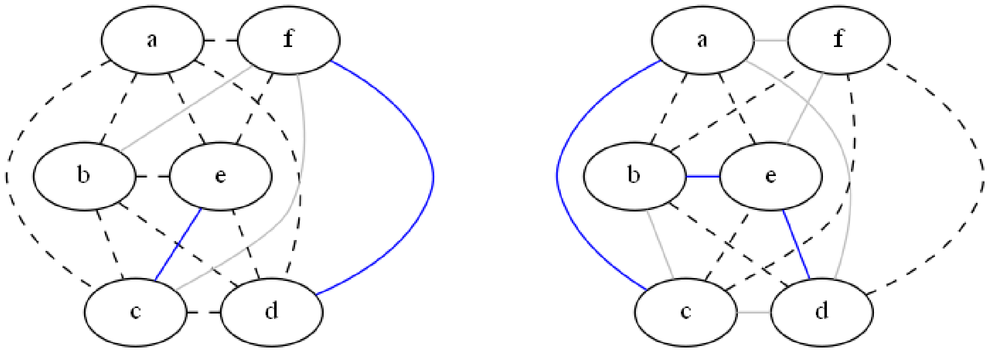


Figure 1.12: Graph of Example 1.22: Threshold variations for exponential and shortest path Gaifman graphs.

we will give one as lower threshold and 4 as upper threshold, keeping just those vertices with 2 and 3 as path size, right of Figure 1.12.

As another discretization method, we have the K-means [40] clustering, it is a method commonly used in data mining, whose objective is to partition a set of n values in k clusters. In this case, the quantity of cluster will be the quantity of equivalence classes $\mathcal{E}_i$ and each $c_{x,y}$ will belong to the cluster i whose centroid is closer. The resulting Gaifman graph is a 2-structure with k equivalence classes.

There are efficient heuristics that are commonly used and quickly converge to a local optimum following an iterative refinement approach to solve the K-means problem that, in the general case, is computationally difficult (NP-hard). However, in the specific case of our application to work with just one dimension is enough, that is not NP-hard.

To implement this method we use the *CkMeans.1d.dp* [40] approach adapted to python [5]. The library solves the K-means problem using the Wang and Song's algorithm for one dimension.

Different to linear and exponential Gaifman graph variants, when we combine the K-means Gaifman graph variant with the thresholded variant the equivalence classes defined on the edges could change since the centroids may get different values, examples are shown in Section 5.6.

[5] `https://gist.github.com/drewda/1299198`

Chapter 2

Clan Decomposition versus Closure Spaces

2.1 Introduction

In this chapter we connect modular decomposition and its generalization, clan decomposition, to more standard tools in Data Analysis, such as closure and implication mining. We provide a fundamental study of the process behind the application of graph decomposition, and we explain how the modular decompositions, and the more general clan decompositions, fit a specific partial view of a closure space that corresponds to so-called *modular implications* and *clan implications* respectively.

In order to provide an explanation in a constructive way, and following the structure of this thesis, we will work with modular decomposition at first and then with clan decomposition. We based the general structure of our proposal on the scheme of Figure 2.1.

Thus, in the Section 2.2 we show the procedure that we propose to get the set of implications that describes a graph, *modular implications*. It is represented in the general scheme by the arrow from Gaifman graph to Implications set. We develop the theory behind the connection of modules and closures, and behind the connection of strong modules and strong closures, we prove that they are the same sets, these are represented by the double arrows in the general scheme. We also argue the relations between the modular decomposition and the closure space lattice. For the arrow from Implications set to Gaifman graph, we give an algorithm to reconstruct the graph from

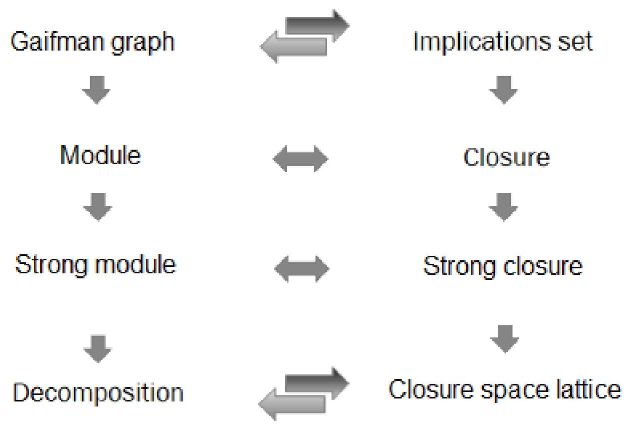


Figure 2.1: General scheme of module-closure approach.

the set of modular implications in Section 2.2.1 and also, we give some constraints that must be satisfied to ensure that given implications set describes a graph.

In Section 2.3 we extend some of the results to clans and we suggest why some other results are not extended.

Finally, in Section 2.4, we argue that it is possible to go from the modular decomposition to the closure space lattice, and viceversa. It is important to point out that it is not enough to work only with the strong closure space, even though the strong closures and the strong modules are the same sets we need all the lattice to know the type of each strong module, and strong clan, as we prove in Section 2.2 and Section 2.3. These analyses are represented in the general scheme by the arrows in the bottom of Figure 2.1.

2.2 Closures from modules

Given a graph, it is possible to describe the conditions for a subgraph being a module in the form of a set of implications. As we already indicate, in Proposition 1.12, if x and y are elements of any module $\mathcal{M}$ and z sees them in a different way, then $z \in \mathcal{M}$. Taking vertices as propositional variables and subgraphs as models, this is equivalent to: $\mathcal{M} \models xy \Rightarrow z$. As we said in the preliminaries, $\mathcal{M} \models xy \Rightarrow z$, if either $xy \not\subseteq \mathcal{M}$ or $xyz \subseteq \mathcal{M}$, since by the premise $x \in \mathcal{M}$ and $y \in \mathcal{M}$, $xy \subseteq \mathcal{M}$, thus $xyz \subseteq \mathcal{M}$ being $z \in \mathcal{M}$.

In this way, we have the set subgraph module described by a set of implications $\mathcal{M}$, thus the closure $\overline{xy}$ of the set xy is the largest set Z such that $\mathcal{M} \models xy \Rightarrow Z$.

Following this idea, given a graph, we generate a set of implications from it. We call this set of implications the *modular implications* of the graph. In each implication, the antecedent will be conformed by a pair of vertices in the graph, and the consequent will be conformed by those vertices that see the vertices of the antecedent in different ways.

Given a graph $\mathcal{G}$ and a vertex x in it, to denote the immediate neighbors of x in $\mathcal{G}$ we use $\mathcal{N}_{\mathcal{G}}(x)$ and we define the distinguishing set of a pair of vertices x, y as:

$$\mathcal{D}_{\mathcal{G}}(x,y) = (\mathcal{N}_{\mathcal{G}}(x) \setminus \mathcal{N}_{\mathcal{G}}(y)) \cup (\mathcal{N}_{\mathcal{G}}(y) \setminus \mathcal{N}_{\mathcal{G}}(x)) \setminus \{x,y\}.$$

That is, the set $\mathcal{D}_{\mathcal{G}}$ gathers together all the vertices that see one given pair of vertices, x and y, in different ways. It is neccesary to remove explicitly x and y from the distinguish set since, in the case they are connected, the absence of self-edges would make each of them qualify to distinguish themselves from the other, whereas the definition of module only searches for distinguishing vertices outside the module.

We construct the set of modular implications as follows:

Definition 2.1. *Let $\mathcal{G}$ be a graph and let x and y be two different vertices in it, the corresponding modular implication is $xy \Rightarrow \mathcal{D}_{\mathcal{G}}(x,y)$. The set of modular implications for $\mathcal{G}$ is formed by the modular implication corresponding to every pair of different vertices, x, y for which the right-hand side is nonempty, that is $\mathcal{D}_{\mathcal{G}}(x,y) \neq 0$.*

We can state the following:

Proposition 2.2. *Let $\mathcal{G}$ be a graph and let x and y be two different vertices in it, in the corresponding modular implication $xy \Rightarrow Z$ we have $z \in Z$ if and only if $z \notin \{x,y\}$ and z is connected to exactly one of x, y. Therefore, $\{x,y\}$ is a module (of size 2) if and only if $Z = \emptyset$.*

The proof follows directly from the definitions.

Example 2.3. *Let us get the set of modular implications from the graph in left of Figure 2.2:*

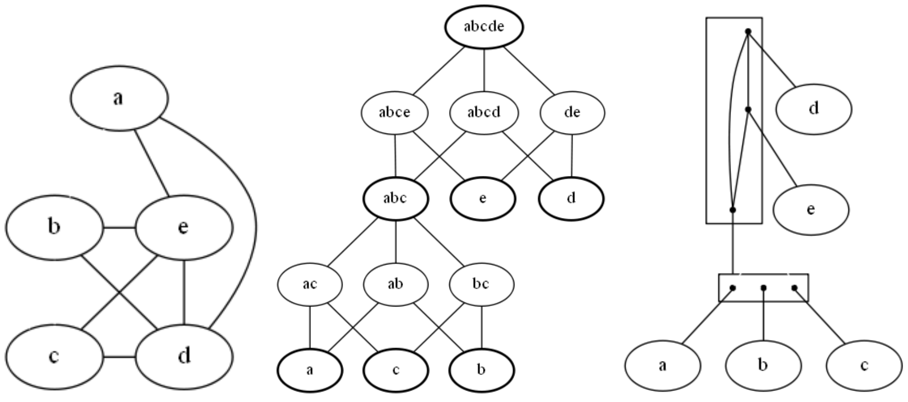


Figure 2.2: Graph of Example 2.3: A graph, its closure lattice and its modular decomposition.

$ad \Rightarrow bc$	$bd \Rightarrow ac$	$cd \Rightarrow ab$
$ae \Rightarrow bc$	$be \Rightarrow ac$	$ce \Rightarrow ab$

The pairs $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{d, e\}$, are size-2 modules and would lead to empty right-hand sides in their modular implications; accordingly, these implications are discarded. Thus, ab, ac, bc and de are closed sets.

In the center of the Figure 2.2 we find the closure space lattice described by the modular implications obtained from the graph, we mark in bold those closed sets that do not overlap any other closed set (strong closures, as appear in Definition 1.4). We show the decomposition of the graph in the right of Figure 2.2.

The main result of this section is:

Theorem 2.4. *The modules of a graph and the closures defined by its set of modular implications are the same sets.*

Proof. (Closures are modules.) Let X be a closure, and suppose that there is some y that can distinguish two arbitrary different $x_1, x_2 \in X$, with $y \notin \{x_1, x_2\}$; one of the modular implications will be $x_1x_2 \Rightarrow Y$ with $y \in Y \neq \emptyset$. As X is a closure, it must satisfy all the modular implications, and both antecedents are in X, thus $y \in X$. Hence, no y outside X may distinguish two elements inside X, which is the definition of module.

(Modules are closures.) Let X be a module. It suffices to show that it satisfies all the modular implications: let $x_1x_2 \Rightarrow Y$ be one of them. If either

$x_1 \notin X$ or $x_2 \notin X$, then X satisfies the implication by failing the antecedent. If $x_1, x_2 \in X$ then, since X is a module, no item outside X can distinguish them; but, according to Proposition 2.2, Y is the set of items that distinguish them, hence $Y \subseteq X$ and X satisfies the modular implication. $\square$

Corollary 2.5. *Strong modules and strong closures are the same sets.*

In the following, and until explicitly indicated otherwise, whenever we talk about closures, we are referring to the closures described by the set of modular implications.

By Theorem 2.4 and Corollary 2.5 we have that modules and closures, strong or not, are the same sets. As we have seen, the type of the module is also important in our visualization of co-occurrence patters. Thus, we would like to be able to know the type of each module, primitive or complete, by using just the information in the closure lattice.

Following this idea, the next theorem shows us that the type of a strong module is given by the immediate closure subsets of its corresponding strong closure into the lattice. The immediate closure subsets of a set X are those Y below X in the closure lattice such that no intermediate set Z, $Y \subset Z \subset X$, is closed.

Theorem 2.6. *The type of the coarsest quotient graph of a module is primitive if and only if the immediate closure subsets of its corresponding closure in the closure lattice are strong closures and they are more than three.*

Proof. Let $\mathcal{M}$ be a module. By Corollary 2.5, its corresponding closure in the closure lattice is also $\mathcal{M}$. Let us suppose its coarsest quotient graph collapses the maximal strong modules $\mathcal{M}_i$ ($i \in I$ for some index set I).

($\Rightarrow$) Let $\mathcal{M}$ be a primitive module, that is its corresponding coarsest quotient graph is primitive; we have to prove that there is not any union of $\mathcal{M}_i$'s closure subset $\mathcal{F}$ such that $\mathcal{M}_i \subset \mathcal{F} \subset \mathcal{M}$. $\mathcal{F}$ must be a union of $\mathcal{M}_i$ since, by the premise, $\mathcal{M}_i$ are strong closures so $\mathcal{F}$ does not overlap any of them. But if $\mathcal{F}$ includes at least two $\mathcal{M}_i$, by definition of primitive there must be at least one other $\mathcal{M}_i$ distinguishing them; thus $\mathcal{F}$ could not be a closure.

($\Leftarrow$) We have to prove that if the immediate closures $\mathcal{M}_i$ of a strong closure $\mathcal{M}$ are strong closures and more than three, then $\mathcal{M}$ is a primitive module. Let $J \subset I$ be the index set of a proper subset of the modules $\mathcal{M}_i$, consisting of at least two of them (we can guarantee this possibility by the

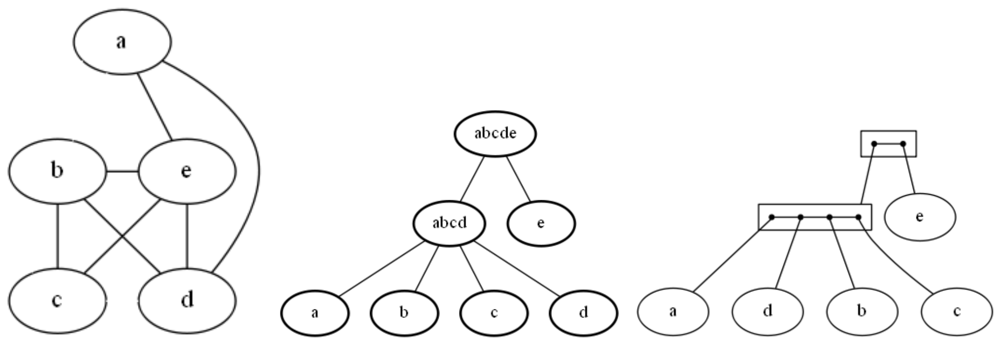


Figure 2.3: Graph of Example 2.7: A graph with a primitive module into its tree decomposition.

premise). Suppose that all the remaining $\mathcal{M}_i$'s, $i \notin J$, cannot distinguish between them, so $\bigcup_{j \in J} \mathcal{M}_j$ must be a closure; this would contradict the fact that all the $\mathcal{M}_i$'s are immediate closed subsets of $\mathcal{M}$. Thus, for arbitrary $J \subset I$, there must be at least one $\mathcal{M}_i$, $i \notin J$, that may distinguish between some of them so that no such $\bigcup_{j \in J} \mathcal{M}_j$ is a module. Subsets that are not in the form $\bigcup_{j \in J} \mathcal{M}_j$ are not modules either because each M_i being a strong module means that no module overlaps any of them. Thus, $\mathcal{M}$ is a primitive module. $\square$

Let us see some examples to illustrate both cases:

Example 2.7. *In the left of Figure 2.3 is a graph (known as the "gem graph") with a primitive quotient graph into its decomposition. By Proposition 1.17, a primitive quotient graph of 4 vertices must be indeed P_4. We get from it the following set of modular implications:*

$$\begin{array}{llll}
ab \Rightarrow c & ae \Rightarrow bc & be \Rightarrow a & de \Rightarrow c \\
ac \Rightarrow bd & bc \Rightarrow d & cd \Rightarrow a & \\
ad \Rightarrow b & bd \Rightarrow ac & ce \Rightarrow ad &
\end{array}$$

This implication set generates the closure lattice in the center of Figure 2.3. The node abcde has two children, by Theorem 2.6 its respective module in the decomposition tree is complete, we may see this in the right of Figure 2.3. The node abcd has four strong closures as children, thus by Theorem 2.6, its equivalent strong module into the decomposition is primitive as we may see in the decomposition tree.

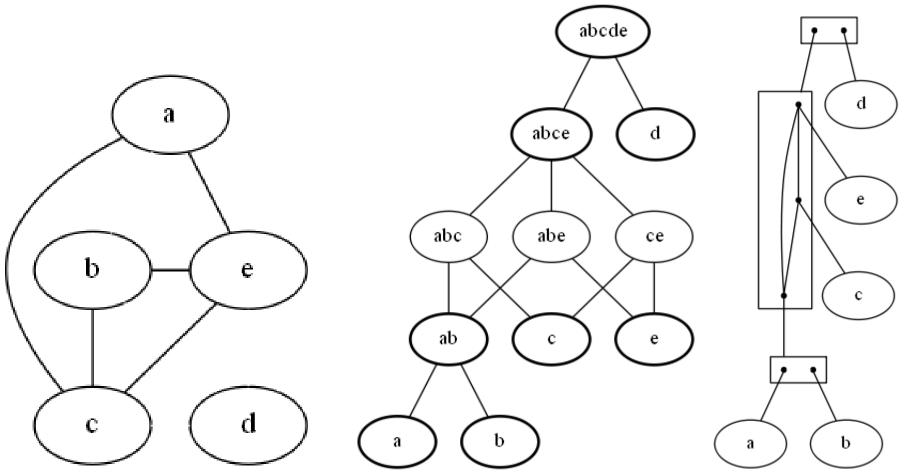


Figure 2.4: Graph of Example 2.8: A graph with complete modules into its tree decomposition.

Example 2.8. *In Figure 2.4 there is an example of a graph with only complete quotient graphs into its decomposition. From the left of Figure 2.4 we get the set of modular implications:*

$ac \Rightarrow b$	$ae \Rightarrow b$	$bd \Rightarrow ce$	$cd \Rightarrow abe$
$ad \Rightarrow ce$	$bc \Rightarrow a$	$be \Rightarrow a$	$de \Rightarrow abc$

With ab and ce closed sets. At the center of Figure 2.4 we have the closure space lattice from this set of implications. As we can see, the closed set abcde has two children so its respective quotient graph in the decomposition is a complete module. The closed set abce has three children overlapping, by the Theorem 2.6 we may deduce its respective coarsest quotient graph in the decomposition is complete as we can see in the right of Figure 2.4. The last closed set in the closure lattice is ab, it has two children so its respective quotient graph is a complete module as we may see in the decomposition tree.

As we can see in the previous example, Example 2.8, there is an alternation on the equivalence classes in the complete modules, we may generalized it in the following proposition.

Proposition 2.9. *For a complete module $\mathcal{M}$, if one of the elements $\mathcal{M}_i$ in its coarsest quotient is a complete module too, the edges in the coarsest* *quotient graph of one of them are a clique while the edges in the other coarsest quotient graph are disconnected.*

Proof. Assume that it is not true, that is, both the edges of the coarsest quotient graph of $\mathcal{M}$ and the edges of thee coarsest quotient graph of $\mathcal{M}_i$ are the same. Thus, for arbitrary $\mathcal{M}'$ in the coarsest quotient of $\mathcal{M}_i$, it may not distinguish the remaining elements in the coarsest quotient of $\mathcal{M}_i$ from the remaining elements in the coarsest quotient of $\mathcal{M}$, making $\mathcal{M}\backslash\mathcal{M}_i\cup\mathcal{M}_i\backslash\mathcal{M}'$ a module but it is not possible since the coarsest quotient contains the maximal strong modules. □

2.2.1 Graph reconstruction from module implications

In this section we show that from the set of modular implications of a graph $\mathcal{G}$ defined on the vertices $\mathcal{U}= \{v_1, \ldots, v_n\}$ it is possible to construct a graph $\mathcal{G}_a$ with exactly the same decomposition taking just those implications which left part is in the form $v_1v_2, v_2v_3, \ldots, v_{n-1}v_n$.

As Proposition 1.11 states, permuting absence and presence of all edges does not change the ways vertices distinguish each other, and therefore, leaves the same set of modules, thus we give the same interpretation to a graph and to its complement. Following our algorithm we may reconstruct the original graph or its complement, for our purpose it is irrelevant since we are interested on its decomposition and this will be the same in both cases.

Before to explain our algorithm to construct the graph we have the following propositions.

Proposition 2.10. *Let $Z = \mathcal{D}_{\mathcal{G}}(x, y)$, if we have the module implications $xy \Rightarrow Z$ we know $(x, z) \neq (y, z)$ for all $z \in Z$. Also we know, $(x, w) = (y, w)$ for all $w \in \mathcal{U}\backslash Z$*

The idea to reconstruct the graph is to take from the set of modular implications a reduced number of implications, we show that those implications give us enough information about the relations between the all edges of the graph, and based on this information we reconstruct the graph.

Thus, using the Algorithm 1 we construct a connex graph $\mathcal{G}'$ from $\mathcal{G}$, whose vertices are all possible edges from the graph $\mathcal{G}$ and the edges determine similarities between the edges on the linked vertices, in this way once a type of edge is assigned on the graph $\mathcal{G}_a$, we may deduce the type for the remaining edges. The order of the edges is related to the first two levels of the SE-tree [36].

Algorithm 1 To construct a connex graph $\mathcal{G}'$ on the set of vertices determined by the edges of a graph $\mathcal{G}$

```
Input: Set of modular implications of $\mathcal{G}$.
Output: $\mathcal{G}' = (\mathcal{U}_{\mathcal{G}'},\mathcal{E}_{\mathcal{G}'})$, with binary labeling on its edges.
Let $\mathcal{U}_{\mathcal{G}'}$ be an empty set.
Let $\mathcal{E}_{\mathcal{G}'}$ be an empty set.
for all $j = \overline{2,n}$ do
  Add $v_1v_j$ to $\mathcal{U}_{\mathcal{G}'}$
  if $2 < j$ then
    Add $(v_1v_{j-1}, v_1v_j)$ to $\mathcal{E}_{\mathcal{G}'}$ in the following way
    if $v_1 \in Z$, such that $v_{j-1}v_j \Rightarrow Z$ then
      Label the edge $(v_1v_{j-1}, v_1v_j)$ with 0
    else
      Label the edge $(v_1v_{j-1}, v_1v_j)$ with 1
    end if
    for all $i = \overline{2, j-1}$ do
      Add $v_iv_j$ to $\mathcal{U}_{\mathcal{G}'}$
      Add $(v_{i-1}v_j, v_iv_j)$ to $\mathcal{E}_{\mathcal{G}'}$ in the following way
      if $v_j \in Z$, such that $v_{i-1}v_i \Rightarrow Z$ then
        Label the edge $(v_{i-1}v_j, v_iv_j)$ with 0
      else
        Label the edge $(v_{i-1}v_j, v_iv_j)$ with 1
      end if
    end for
  end if
end for
```

For the Algorithm 1, it is possible to determine when two edges are different following the Proposition 2.10. In this way, two given edges, v_iv_k and v_jv_k of $\mathcal{G}$, are in different equivalence classes if for the implication $v_iv_j \Rightarrow Z$, $v_k \in Z$.

The Algorithm 1 constructs a connex graph since given two vertices on it there is a path that connects them, the Algorithm 2 gives us this path. In fact, the length of the path that connects the edge v_lv_k to v_mv_p is $(l-1) + (p-k) + (m-1)$ where $p > k$, while if $p = k$, the length of the path is $|l-m|$.

Algorithm 2 To give the path that connects two vertices $v_l v_k$ and $v_m v_p$ in the graph $\mathcal{G}' = (\mathcal{U}_{\mathcal{G}'}, \mathcal{E}_{\mathcal{G}'})$

```
Input: Two vertices $v_l v_k \in \mathcal{U}_{\mathcal{G}'}$ and $v_m v_p \in \mathcal{U}_{\mathcal{G}'}$, where $k \leq p$, without losing generality.
Output: The set of vertices that conform the path that connects the vertices $v_l v_k$ and $v_m v_p$
Let $Path$ be an empty set
if $k == p$ then
  if $l < m$ then
    for all $i = l$ to $m$ do
      Add $v_i v_p$ to $Path$
    end for
  else
    for all $i = m$ to $l$ do
      Add $v_i v_p$ to $Path$
    end for
  end if
else
  for all $i = l$ to 1 do
    Add $v_i v_k$ to $Path$
  end for
  for all $i = k + 1$ to $p$ do
    Add $v_1 v_i$ to $Path$
  end for
  for all $i = 2$ to $m$ do
    Add $v_i v_p$ to $Path$
  end for
end if
Return $Path$
```

Example 2.11. *Let us apply the Algorithm 1 on the following set of implications gotten from the graph of Figure 5:*

$ab \Rightarrow cd$	$ae \Rightarrow bd$	$be \Rightarrow ac$	$de \Rightarrow a$
$ac \Rightarrow be$	$bc \Rightarrow de$	$cd \Rightarrow e$	
$ad \Rightarrow b$	$bd \Rightarrow c$	$ce \Rightarrow ad$	

Following the Algorithm 1 we get the connex graph $\mathcal{G}_E = (\mathcal{U}_{\mathcal{G}E}, \mathcal{E}_{\mathcal{G}E})$,

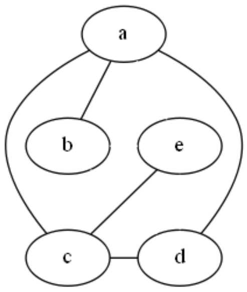


Figure 2.5: Graph of Example 2.11: Initial graph.

where $\mathcal{U}_{\mathcal{G}E} = \{ab, ac, bc, ad, bd, cd, ae, be, ce, de\}$*. Let gray lines be the edges for type 1 and let dotted lines be the edges for type 0.*

The possible reconstructed graph are shown in Figure 2.7. As $\mathcal{G}_a$ *is a graph, we just have connected and disconnected vertices, to reconstruct the graph let us assign a type of edge to the initial vertex ab.*

Thus, assume the edge (a, b) *is a solid line, that is a and b are connected. As we may see in Figure 2.6* (a, c) *and* (a, d) *are the same type than* (a, b)*, while* (b, c) *is not, thus b and c are disconnected, and a is connected with c and d. We also know that* (a, d) *is different than* (a, e) *and* (b, d)*, thus a and e are disconnected, and also, b and d are disconnected. We know* (b, d) *and* (c, d) *are different, thus, c and d are connected. While,* (a, e) *is the same type than* (b, e) *thus, b and e are disconnected. Also, we have that* (b, e) *is different from* (c, e) *being c and e connected. At the same time* (c, e) *is different from* (d, e)*, thus d and e are disconnected. The resulting graph is shown in the left of Figure 2.7.*

If we had started by assuming that the edge (a, b) *does not exist, following the constraints described in Figure 2.6, we get the graph in the right of Figure 2.7.*

Now, assume we have some modular implications set, to determine if it describes a graph $\mathcal{G}$, by making an argument similar to the previously one, we need to be able to construct a connex graph $\mathcal{G}'$. Intuitively, this will result in having a sequence from one edge to another, that is, we will have a sequence in the left side of the implications. Following this idea is probably that these sequence of edges describe a spanning tree $\mathcal{G}_S$ defined on the set of vertices of $\mathcal{G}$.

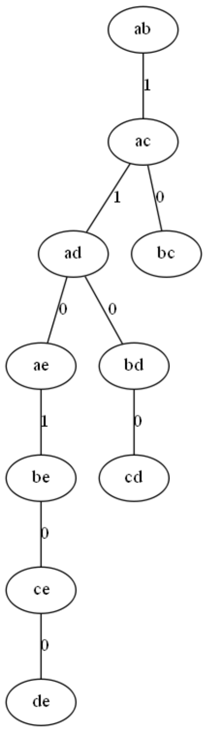


Figure 2.6: Graph of Example 2.11: Resulting using Algorithm 1 on the graph of Figure 2.5.

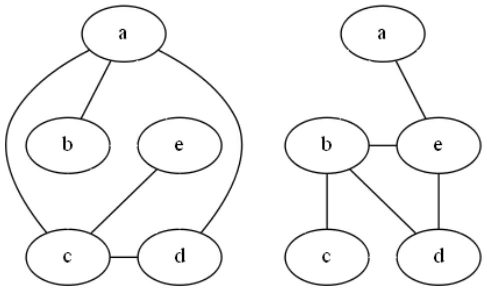


Figure 2.7: Graph of Example 2.11: Reconstructed graph.

2.3 Closures from clans

Given a 2-structure (Definition 1.18) is possible to obtain the implications set that describes it. In this section we describe the procedure that we propose to obtain them and some found results from these implications.

According to clan definition, let x, y be elements of any clan $\mathcal{C}$, if there is z such that sees in a different way x and y, that is the edges (x, z) and (x, y) are in different equivalence classes, so $z \in \mathcal{C}$; equivalent to $\mathcal{C} \models xy \Rightarrow z$.

Based on it we generate a set of implications from a 2-structure that we call *clan implications*, whose left side will be a pair of vertices and their right side will be conform by those vertices that see them (the left side) in a different way. If there is not any vertex that may distinguish them, no implications are generated and they will conform a closed set.

Once we obtained the set of implications it is possible to know the associated closure space lattice, finding those sets that involve the elements that are in the implications.

Formally, given a 2-structure $\mathcal{G}$ and a vertex x in it, we denote by $\mathcal{E}_{i,\mathcal{G}}(x)$ the set of vertices connected with x by edges belonging to the equivalence class $\mathcal{E}_i$ in $\mathcal{G}$, and define the "distinguishing set" of a pair of vertices x, y, as follows:

$$\mathcal{D}_{\mathcal{G}}(x, y) = \bigcup_i ((\mathcal{E}_{i,\mathcal{G}}(x) \setminus \mathcal{E}_{i,\mathcal{G}}(y)) \cup (\mathcal{E}_{i,\mathcal{G}}(y) \setminus \mathcal{E}_{i,\mathcal{G}}(x))) \setminus \{x, y\}$$

This set collects together all the vertices that see one given pair in different ways. For example, for different i, j, assume the edge that connects z with x is in the equivalence class $\mathcal{E}_i$ and the edge that connects z with y is in the equivalence class $\mathcal{E}_j$. Thus, z will be in $\mathcal{E}_{i,\mathcal{G}}(x) \setminus \mathcal{E}_{i,\mathcal{G}}(y)$ and also in $\mathcal{E}_{j,\mathcal{G}}(y) \setminus \mathcal{E}_{j,\mathcal{G}}(x)$.

We construct the set of clan implications as follows:

Definition 2.12. *Let $\mathcal{G}$ be a 2-structure and let x and y be different vertices in it, the corresponding clan implication is $xy \Rightarrow \mathcal{D}_{\mathcal{G}}(x, y)$. The set of clan implications for $\mathcal{G}$ is conformed by the clan implications corresponding to every pair of different vertices, for which the right-hand side is nonempty: $\mathcal{D}_{\mathcal{G}}(x, y) \neq \emptyset$.*

From the definition of clan we can state the following:

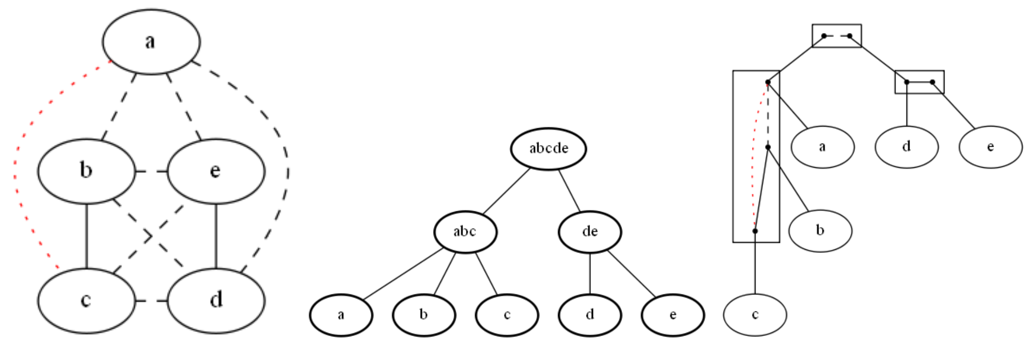


Figure 2.8: Graph of Example 2.14: Initial 2-structure, its closure lattice and its clan decomposition.

Proposition 2.13. *Let $\mathcal{G}$ be a 2-structure and let x and y be different vertices in it, in the corresponding clan implication $xy \Rightarrow Z$ we have $z \in Z$ if and only if $z \notin \{x, y\}$ and z is connected to each of x, y by inequivalent edges. Therefore, $\{x, y\}$ is a clan (of size 2) if and only if $Z = \emptyset$.*

Let us see an example of getting the set of clan implications from a graph and the closure space described by it. In an illustrative way we colored differently the different equivalence classes.

Example 2.14. *From the left of Figure 2.8 we obtain the following set of clan implications:*

$ab \Rightarrow c$	$ae \Rightarrow cd$	$be \Rightarrow cd$
$ac \Rightarrow b$	$bc \Rightarrow a$	$cd \Rightarrow abe$
$ad \Rightarrow ce$	$bd \Rightarrow ce$	$ce \Rightarrow abd$

The pair $\{d, e\}$ is a size-2 clan and lead to empty right-hand side in its clan implication. Thus, de is a closed set. In the center of Figure 2.8 we find the closure space lattice described by the clan implications set and the clan decomposition tree is in the right oof Figure 2.8.

We may extend Theorem 2.4 and Theorem 2.6 to clans in a natural way:

Theorem 2.15. *1. Clans and closures from the graph implications are the same sets. Implying as well that strong clans and strong closures are the same sets.*

2. *The type of a clan is primitive if and only if its immediate closures subsets of its corresponding strong closure in the closure lattice are strong and they are more than two.*

Proof. 1. *(Closures are clans.)* Let X be a closure, and suppose that there is some y that can distinguish two arbitrary $x_1, x_2 \in X$; one of the clan implications will be $x_1x_2 \Rightarrow Y$ with $y \in Y \neq \emptyset$. As X is a closure, it must satisfy all the clan implications, and both antecedents are in X, thus $y \in X$. Hence, no y outside X may distinguish two elements inside X, which is the definition of clan.

1. *(Clans are closures.)* Let X be a clan. It suffices to show that it satisfies all the clan implications: let $x_1x_2 \Rightarrow Y$ be one of them. If either $x_1 \notin X$ or $x_2 \notin X$, then X satisfies the implication by failing the antecedent. If $x_1, x_2 \in X$ then, since X is a clan, no item outside X can distinguish them; but, according to Proposition 2.13, Y is the set of items that distinguish them, hence $Y \subseteq X$ and X satisfies the clan implication.

2.($\Rightarrow$) Let $\mathcal{C}$ be a clan by the first part of this theorem, we may state that its corresponding closure in the closure lattice is also $\mathcal{C}$, let us suppose its coarsest quotient graph collapses the maximal strong clans $\mathcal{C}_i$. Let $\mathcal{C}$ be a primitive clan, that is its corresponding coarsest quotient graph is primitive; we have to prove that there is not any closed subset $\mathcal{F}$ such that $\mathcal{C}_i \subset \mathcal{F} \subset \mathcal{C}$. Assume there is such and $\mathcal{F}$.

$\mathcal{F}$ must be a union of $\mathcal{C}_i$ since, by the premise, $\mathcal{C}_i$ are strong closures so $\mathcal{F}$ does not overlap any of them. Also, $\mathcal{F}$ includes at least two of them. By definition of primitive, there must be at least one other $\mathcal{C}_i$ distinguishing them; we can pick $y \notin \mathcal{F}$ in the distinguishing clan and x_1 and x_2 in the distinguished ones inside $\mathcal{F}$: then $\mathcal{F}$ fails the clan implication $x_1x_2 \Rightarrow Y$ and is not a closure.

2.($\Leftarrow$) We have to prove that if the immediate closures $\mathcal{C}_i$, with $i \in I$ for some index set I, of a strong closure $\mathcal{C}$ are strong closures and more than two, then $\mathcal{C}$ is a primitive clan. Let $J \subset I$ be the index set of a proper subset of the clans $\mathcal{C}_i$, consisting of at least two of them (we can guarantee this possibility by the premise).

Suppose that all the remaining $\mathcal{C}_i$'s, $i \notin J$, cannot distinguish between them, so $\bigcup_{j\in J} \mathcal{C}_j$ must be a closure; this would contradict the fact that all the $\mathcal{C}_i$'s are immediate closed subsets of $\mathcal{C}$. Thus, for arbitrary $J \subset I$, there must be at least one $\mathcal{C}_i$, $i \notin J$, that may distinguish between some of them so that no such $\bigcup_{j\in J} \mathcal{C}_j$ is a clan. Subsets that are not in the form $\bigcup_{j\in J} \mathcal{C}_j$

are not clans either because each C_i being a strong clan means that no clan overlaps any of them. Thus, $\mathcal{M}$ is a primitive clan.

$\square$

The difference between Theorem 2.6 and Theorem 2.15 is the number of strong nodes that are required. When we work with more than two equivalence classes is possible to have a primitive node with just three internal nodes as we can see in the primitive clan of Figure 2.8, conformed by $\{a, b, c\}$: the possible unions of two of their internal items are not closed sets because we will have the clan implication that involves the third one in the right side.

Also the Proposition 2.9 is extended to clans in a natural way:

Proposition 2.16. *For a complete clan $\mathcal{C}$, if one of the elements $\mathcal{C}_n$ in its coarsest quotient is a complete module too, the edges in the coarsest quotient graph of one of them are in the equivalence class $\mathcal{E}_i$ while the edges in the other coarsest quotient graph are in the equivalence class $\mathcal{E}_j$, being $\mathcal{E}_i \neq \mathcal{E}_j$.*

Proof. Assume that it is not true, that is, both the edges of coarsest quotient graph of $\mathcal{C}$ and the edges of coarsest quotient graph of $\mathcal{C}_n$ are in the same equivalence class. Thus, for arbitrary $\mathcal{C}'$ in the coarsest quotient of $\mathcal{C}_n$, $\mathcal{C}'$ may not distinguish the remaining elements in the coarsest quotient of $\mathcal{C}_n$ from the remaining elements in the coarsest quotient of $\mathcal{C}$, making $\mathcal{C} \setminus \mathcal{C}_n \cup \mathcal{C}_n \setminus \mathcal{C}'$ a clan but it is not possible since the coarsest quotient contains the maximal strong clans. $\square$

2.4 Decomposition tree and closure lattice

It is possible to construct the clan decomposition tree directly from the closure space defined by the implications obtained from the initial 2-structure, and viceversa.

If we take from the closure space just the strong closures, they would be the same sets than the strong clans by Theorem 2.15, and the types of the clans are also obtained using Theorem 2.15. To give the internal 2-structures of each clan is a more complicated task since we do not have information that allows us to determine the equivalence class for every edge, unless we were working just with two equivalence classes. Yet, it is possible to determine the type of each strong clan.

Thus, the Algorithm 3 describes the procedure to get the clan decomposition tree from the closure lattice.

Algorithm 3 Algorithm to obtain the clan decomposition tree from a closure space lattice.

```
 1: Input: Closure space lattice $S$ of the closure implications described by
    the 2-structure $\mathcal{G}$.
 2: Output: Strong clan decomposition tree of the 2-structure $\mathcal{G}$.
 3: Let $Child(X)$ be the set of immediate closures subsets of $X$.
 4: for all $X$ closure in $S$ do
 5:   if $X$ is a strong closure then
 6:     $X$ is a strong clan in the decomposition tree.
 7:     if $Child(X)$ are strong closures and more than two then
 8:       The clan $X$ is primitive.
 9:       The coarsest quotient of $X$ is $Child(X)$.
10:     else
11:       The clan $X$ is complete.
12:       The coarsest quotient of $X$ is $Child(Child(X))$.
13:     end if
14:   end if
15: end for
```

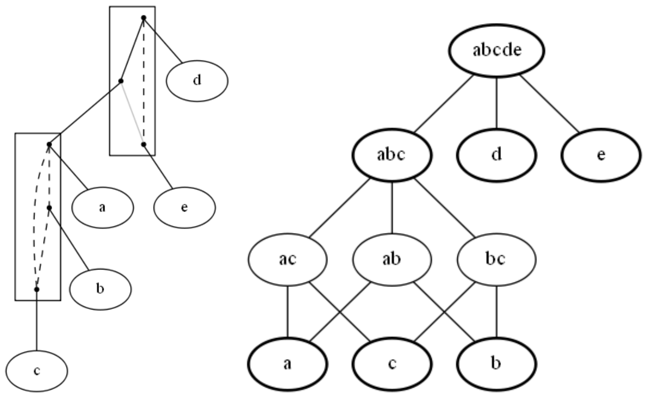


Figure 2.9: Graph of Examples 2.17 and 2.18: Clan decomposition of a 2-structure and its closure space lattice.

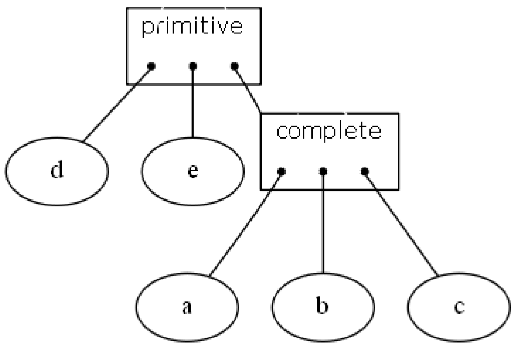


Figure 2.10: Graph of Example 2.17: Clan decomposition of a 2-structure from the closure space lattice of Figure 2.9.

Example 2.17. *Let us apply the Algorithm 3 to the closure space lattice in right of Figure 2.9. We have abcde as strong closure, thus abcde is a strong clan in the decomposition tree. As* $Child(abcde) = \{abd, d, e\}$ *and are all of them strong closures and more than two, abcde is a primitive clan and its coarsest quotient is conform by* $\{abd, d, e\}$*. Then, we have abd as strong closure, thus abc is a strong clan in the decomposition tree. As* $Child(abc) = \{ac, ab, bc\}$ *are not strong closures, abc is a complete clan and its coarsest quotient is conform by a,c,b (*$Child(Child(abc))$*), which in turn are strong closure thus strong clans too. The result is shown in Figure 2.10.*

Again by the Theorem 2.15 if we take the strong clans in the decomposition, they would be the same sets than the strong closures in the lattice. And based on the type of each strong clan and its maximal strong clans in its coarsest quotient we may get all the closures in the lattice. The Algorithm 4 describes this procedure.

Example 2.18. *Let us apply the Algorithm 4 to the clan decomposition tree in left of Figure 2.9. We have the strong clan abcde, thus it is a strong closure in the closure lattice, since abcde is a primitive clan its children in the closure lattice are the elements in the coarsest quotient of abcde, those are abc, d and e. Then, we have abc as strong clan, being abc a strong closure in the closure lattice. As abc is a complete clan, the children of abc in the closure lattice will be the possible pair combinations of the elements in the coarsest quotient of abc (line 8 on the Algorithm 4). The coarsest quotient*

Algorithm 4 Algorithm to obtain the closure space lattice from a clan decomposition tree.

```
1: Input: Strong clan decomposition tree of the 2-structure $\mathcal{G}$.
2: Output: Closure space lattice $S$ of the closure implications set described
   by the 2-structure $\mathcal{G}$ .
3: for all $X$ strong clan in the decomposition tree do
4:    $X$ is a strong closure in the closure space lattice.
5:    if $X$ is primitive or its coarsest quotient has two elements then
6:       The child of $X$ in the closure space lattice are those elements in the
         coarsest quotient of $X$.
7:    else
8:       Add the power set of the coarsest quotient of $X$ to the children of
         $X$ in the closure space lattice.
9:       for all $X_i$ in the coarsest quotient of $X$ do
10:         Add $X_i$ to the children of its corresponding sets in the power set.
11:      end for
12:   end if
13: end for
```

of abc is $\{a, b, c\}$, thus the children of abc in the closure lattice will be ab, ac and bc. Also (following the line 11) we add a as child of ab and ac, we add b as child of ab and bc and we add c as child of ac and bc. Getting as a result the closure lattice of the right in Figure 2.9.

Chapter 3

Algorithms and implementations

3.1 Introduction

This chapter is focused on discussing our decomposition algorithm that we explain in Section 3.2. It involves the implementation of complementary algorithms like *pack* and *split* functions, and the use of the Union-Find data structure[1]; we also give the theorems that support our algorithms and, in order to illustrate each step of the main algorithm, we show a detailed example.

Of course, our algorithm is not the first decomposition algorithm developed, in Section 3.3 we comment on a couple of related algorithms, we use some examples to see in a general way the differences with our algorithm.

Finally, in Section 3.4, we will describe the implementation of a data analysis tool based on our approach, that is divided into three phases: the construction of the graph from the dataset, the application of the 2-structure decomposition method and the visualization of the strong clan decomposition tree.

[1] `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`

3.2 Decomposition algorithm

Over the years several algorithms have been developed to perform modular decomposition and clan decomposition methods. Our algorithm is an incremental algorithm, that means the vertices of the 2-structure are added one by one, and every time a vertex is added we get a strong clan decomposition tree. Therefore, we describe our algorithm by focusing on the effect of adding one more vertex to an existing clan decomposition tree, starting at the root.

By comparing the root clan with the new vertex, we may classify the nodes in the coarsest quotient of the root clan into three lists: the list of nodes "visible with the *color of the clan*[2]", the list of "other visible nodes" and the list of the so-called "nonvisible nodes". More precisely:

- The list of nodes visible as the color of the clan will contain those nodes whose edges to the new vertex are in the same equivalence class than the internal edges of the clan; it can be nonempty only when the root clan is complete.

- The list of visible nodes will contain those nodes in the coarsest quotient of the clan for which an edge can be determined from the new vertex, that is: let Y_i be one of the strong clans in the coarsest quotient of the current clan and let x be the new vertex, if all the edges from x to the elements in the coarsest quotient of Y_i are in the same equivalence class then we can define an edge from x to Y_i. To determine the existence of this edge we use the Union-Find structure, it is explained in detail in the Subsection 3.2.1.

- Conversely, the list of nonvisible nodes will contain all those nodes in the coarsest quotient of the clan whose internal elements are seen in different ways by the new vertex, thus we may not assign a single edge color from the new vertex to these nodes; when we are in this case, our algorithm uses the *split* function from the Algorithm 6, it is explained in detail in the Subsection 3.2.3.

[2]We call *color of the clan* the equivalence class of the edges in a complete clan.

3.2.1 Union-Find structure

We use the Union-Find data structure[3] to get the defined edges between vertices, between clans and between vertices and clans. In fact, to implement these functions we rely directly on the pseudocode that appears in [10].

The edges between vertices are initialized according to the original 2-structure. Let x and y be two vertices in the graph, the function *Find* returns the equivalence class for the edge that connects them. Thus edges in the same equivalence class will be return the same value for the function *Find*.

In this way, when a new clan is obtained, the edges to it from the vertices not involved in the clan (if they exist) are generated using the *MakeSet* function and using the *Union* function those edges are added to their corresponding equivalence class determined by the *Find* function.

So, to know if an edge is defined it is enough to call *Find* function with the edge as parameter, which is really useful.

Complexity:

We implement the Union-Find structure using the union-by-rank heuristic with a running time of $O(m \log n)$, where m is the number of *MakeSet*, *Union* and *Find* operations and n is the number of *MakeSet* operations [10].

3.2.2 Pack function

The pack function is executed in order to determine the edges to a clan. It is applied once a clan is gotten or a new element is added to an existing clan. The purpose of this is to help us to obtain in a linear process the lists of "visible with the color of the clan nodes", the list of "other visible nodes" and the list of the so-called "nonvisible nodes". We use the Algorithm 5 for this function.

Complexity:

Let n be the total number of vertices of the 2-structure $\mathcal{G}$, let k be the total number of vertices in the clan C and let $|CQ_C|$ be the total number of elements in the coarsest quotient of C, the algorithm takes time $(n-k) \times$

[3] `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`

$|CQ_C|$. That is because the step **for** in line 5 takes $|V_{\mathcal{G}-C}|$ times, as V_C is the list of vertices in $\mathcal{G}$ not in C its length is $n-k$ thus the **for** is executed $n-k$ times, and the step **while** in line 10 takes at most $|CQ_C|$ times.

The worst case for the coarsest quotient is to have just single items, that is $|CQ_C| = k$, in this way, the algorithm takes $(n-k) \times k = nk - k^2$ times, where $k \leq n$.

Algorithm 5 Algorithm to pack a clan of a 2-structure

```
1: Input: A 2-structure $\mathcal{G}$ and a clan $C$ on it.
2: Output: The edges from/to the clan $C$ using the Union-Find data structure.
3: Let $CQ_C$ be the list of elements into the coarsest quotient of $C$.
4: Let $V_{\mathcal{G}-C}$ be the list of vertices in $\mathcal{G}$ not in $C$.
5: for all $v$ in $V_{\mathcal{G}-C}$ do
6:    InitialEdge = $v, CQ_C[0]$
7:    InitialClass = Find(InitialEdge)
8:    SameClass = True
9:    j = 1
10:   while SameClass and $j < len(CQ_C)$ do
11:      if InitialClass == Find($v, CQ_C[j]$) then
12:         j+=1
13:      else
14:         SameClass = False
15:      end if
16:   end while
17:   if SameClass then
18:      NewEdgeTo = $v, C$
19:      NewEdgeFrom = $C, v$
20:      MakeSet(NewEdgeTo)
21:      MakeSet(NewEdgeFrom)
22:      Union(InitialEdge,NewEdgeTo)
23:      Union(InitialEdge,NewEdgeFrom)
24:   end if
25: end for
```

3.2.3 Split function

Assume the vertex x sees in a different way some of the internal nodes in the coarsest quotient of some clan Y, thus we have to split Y. Let $Y_0, Y_1, \ldots, Y_n$ be the elements in the coarsest quotient of Y, the result to split Y with respect to x are those Y_i that have an edge defined from x. If there is Y_i not seen by x, Y_i is split with respect to x and the result is added to the split of Y. This process is applied until we have only defined edges. The algorithm to split a node is the Algorithm 6, it is based on the Theorem 3.4.

Complexity:

The split function makes a recursive call when any of the elements in the corresponding coarsest quotient is not distinguishable, does not matter the clan type (lines 9 and 18 of Algorithm 6). Thus the worst case is to do the split of all clans in the decomposition tree, unless all those elements are leaves. The clan decomposition tree with most clans is that with just complete clans, whose coarsest quotients have two elements: assume, one of the elements in all the coarsest quotient is a leaf, in this way we get the decomposition tree as deeply as possible, having a total of $n-1$ clans for a 2-structure with n vertices, as the last clan have only leaves, the maximum clans where we can apply the split function is $n-2$; while, if both of the elements in all the coarsest quotient are clans, for a 2-structure with n vertices we will apply the split function in a total of $\sum_{i=1}^{k-1} 2^i + (n-2k)$ times, where $k = \lfloor \log_2 n \rfloor$.

Assume $n - 2 > \sum_{i=1}^{k-1} 2^i + (n-2k)$ where $k = \lfloor \log_2 n \rfloor$:

$$n > 2 + \sum_{i=1}^{k-1} 2^i + (n-2k)$$

$$0 > 2 + \sum_{i=1}^{k-1} 2^i - 2k$$

$$0 > 1 + \sum_{i=0}^{k-1} 2^i - 2k$$

$$0 > 1 + 2^k - 1 - 2k$$

$$0 > 2^k - 2k,$$

a contradiction.

Thus, for a 2-structure with n vertices the split function takes at most $\sum_{i=1}^{k-1} 2^i + (n-2k)$ where $k = \lfloor \log_2 n \rfloor$, thus:

$$\sum_{i=1}^{k-1} 2^i + (n-2k) < 1 + \sum_{i=1}^{k-1} 2^i + (n-2k)$$

$$\sum_{i=1}^{k-1} 2^i + (n-2k) < \sum_{i=0}^{k-1} 2^i + (n-2k)$$

$$\sum_{i=1}^{k-1} 2^i + (n-2k) < 2^k - 1 + n - 2k$$

$$\sum_{i=1}^{k-1} 2^i + (n-2k) < n - 1 + n - 2k$$

$$\sum_{i=1}^{k-1} 2^i + (n-2k) < 2n - 2k - 1$$

Thus, the split function runs in time $O(n)$.

Algorithm 6 Algorithm to split a clan from a node

```
1: **Input:** A clan $C$ and a node $n$.
2: **Output:** $C_v$, the set of maximal strong clans below $C$ visible from $n$.
3: Let $C_v$ be an empty set.
4: if $C$ is primitive then
5:    for all $\mathcal{M}$ in the coarsest quotient of $C$ do
6:       if $\mathcal{M}$ is visible from $n$ then
7:          Add $\mathcal{M}$ to the maximal strong clans in $C_v$.
8:       else
9:          Split $\mathcal{M}$ from $n$ (Algorithm 6 with $\mathcal{M}$ and $n$ as parameters), add
            each maximal strong clan in the result to the set of maximal strong
            clans in $C_v$.
10:      end if
11:   end for
12: else
13:   Let $C_i$ be an empty set for each different equivalence class $i$ (color).
14:   for all $\mathcal{M}$ in the coarsest quotient of $C$ do
15:      if $\mathcal{M}$ is visible from $n$ then
16:         Add $\mathcal{M}$ to $C_i$, being $i$ the way in which $n$ sees $\mathcal{M}$.
17:      else
18:         Split $\mathcal{M}$ from $n$ (Algorithm 6 with $\mathcal{M}$ and $n$ as parameters), add
            each maximal strong clan in the result to the set of maximal strong
            clans in $C_v$.
19:      end if
20:   end for
21:   for all $C_i$ do
22:      if $C_i$ has more than one maximal strong clan then
23:         Add $C_i$ (as a single clan) to the set of maximal strong clans in $C_v$.
24:      else
25:         Add the unique $\mathcal{M}$ in $C_i$ to the set of maximal strong clans in $C_v$.
26:      end if
27:   end for
28: end if
29: Return $C_v$.
```

Let us see a couple of examples of the application of the Algorithm 6.

Example 3.1. *Assume we add the vertex n to the decomposition shown in*

Figure 3.1, having n connected by solid lines with b, c, h, j and l, and by broken lines by the remaining vertices into the decomposition. Thus, the maximal strong clans $\{b, c, d, e\}$ and $\{f, g, h, \{i, j, l, k\}\}$ into the coarsest quotient of the root clan are non visibles from n.

As the root clan is complete (Line 12, Algorithm 6) the elements in its coarsest quotient graph seen in the same way by n will conform a new maximal strong clan and those elements that are not seen by n are split. In this case a and b will conform a new maximal strong clan since they are connected to n by broken lines, while the maximal strong clans $\{b, c, d, e\}$ and $\{f, g, h, \{i, j, k, l\}\}$ are split. As the clan $\{b, c, d, e\}$ is a complete clan all the elements in its coarsest quotient seen in the same way by n will conform new strong clans, as result we get $\{b, c\}$ and $\{d, e\}$ maximal strong clans that are added to the coarsest quotient of the root clan. For the clan $\{f, g, h, \{i, j, k, l\}\}$, again a complete clan, we get $\{f, g\}$ and $\{h\}$ as maximal strong clans; while the clan $\{i, j, k, l\}$ is non visible. As $\{i, j, k, l\}$ is a complete clan we get as new maximal strong clans $\{i, k\}$ and $\{j, l\}$ as their elements are seen in the same way by n.

In this way, as a result of adding n we get as maximal strong clans $\{\{a, m\}, \{b, c\}, \{d, e\}, \{f, g\}, h, \{i, k\}, \{j, l\}\}$, the final decomposition is shown in Figure 3.2.

3.2.4 Clan decomposition algorithm

As an initial case to our algorithm, we have two cases. One of them is to add the new vertex to an empty tree, in this case the vertex is added to a complete clan with just the vertex in its coarsest quotient. The other case is to add the new vertex to a tree whose root has just one element x, in this case the new vertex is added to a complete clan whose coarsest quotient consists of the new vertex and x.

We say that X is a subclan of Y, or alternatively Y is a superior clan of X, if X is a maximal strong clan and thus belongs to the coarsest quotient of Y.

Assume we have a strong clan tree with at least two vertices, and a new node to add to it. We have to look for its position into the tree, starting from the root, going over the strong clan tree as deep as necessary until find its place: a clan is represented by its coarsest quotient graph, so each node there corresponds to a subtree decomposing the corresponding maximal strong clan; then we may walk towards the leaves recursively. We will refer

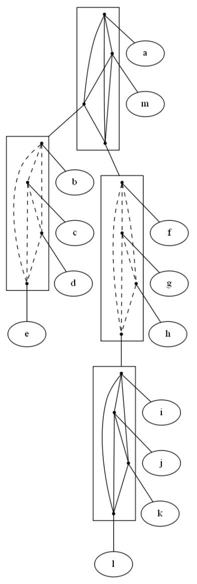


Figure 3.1: Graph of Example 3.1: Initial decomposition.

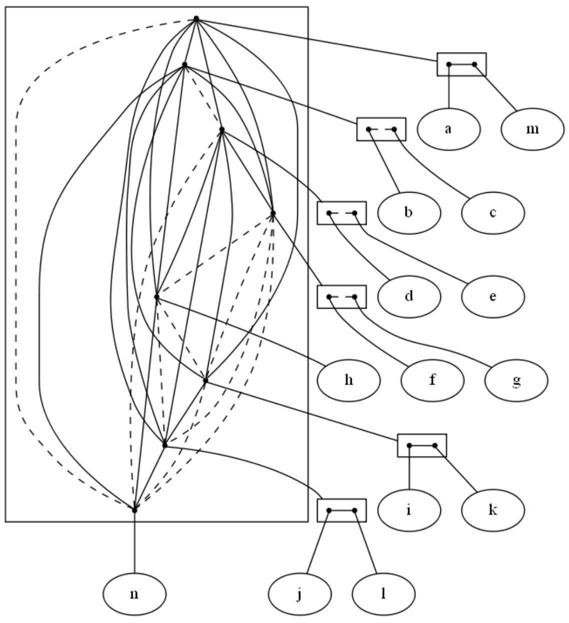


Figure 3.2: Graph of Example 3.1: Resulting decomposition.

to the clan in which we are as current clan.

If we are not in any of the initial cases we follow the next steps:

1. If the current clan is a complete clan, the new vertex can:
   - (a) Become one member of the clan, preserving the type of the clan: The new node sees all the internal nodes of the current clan with the same color as the color of the clan, the new node is added as a maximal strong trivial clan into the coarsest quotient of the current clan and the type of clan continues being complete. No recursive call is made. (Lines 10-11 in the Algorithm 7)
   - (b) Generate a subclan: The new node sees at least one but not all the internal nodes with the same color as the color of the clan, those elements that are not seen by the color of the clan are removed from the coarsest quotient of the current clan and moved to the coarsest quotient of a new complete clan, sibling to the nodes seen by the color of the clan. Thus, this new complete clan and the nodes seen by the color of the clan will be in the coarsest quotient of the current clan. The new vertex is recursively added to the new clan. (Lines 12-20 in the Algorithm 7)
   - (c) Generate a superior clan: The internal nodes are in the list of visible nodes and are seen by the same color but different to the color of the clan. Then, the current clan and the new vertex will be maximal strong clans into the coarsest quotient of a superior complete clan. No recursive call is made. (Lines 21-23 in the Algorithm 7)
   - (d) Become one member of the clan, changing the type of the clan to primitive: When not all the nodes in the clan are seen by the same color and none of them are seen by the color of the clan, we add the new vertex to the coarsest quotient of the current clan and:
     - (i) The nodes in the list of visible nodes are grouped into clans according how they are seen form the new vertex and,
     - (ii) The nodes in the list of nonvisible are split. This may redefine the maximal strong clans: if there are maximal strong complete clans in the coarsest quotient of the current clan whose coarsest quotient elements are seen in the same way from the new vertex, since they are not distinguishable from any node.

(Lines 24-30 in the Algorithm 7)

2. If the current clan is a primitive clan, the new vertex can:

   (a) Enter to the clan of one of the nodes inside of the coarsest quotient of the current clan: If the new vertex and one of the internal nodes see the remaining nodes in the same way. (Lines 32-33 in the Algorithm 7)

   (b) Generate a superior clan: All of the internal nodes are in the list of visible nodes and are seen by the same color. Then, the current clan and the new vertex will be in the coarsest quotient of a superior complete clan. (Lines 34-37 in the Algorithm 7)

   (c) Become one member of the coarsest quotient of the current clan, preserving the type of the clan: When not all the nodes in the clan are seen by the same color. Thus, the new vertex is added to the coarsest quotient of the current clan and the nodes in the list of nonvisible are split; the elements of the split node may not generate any new clan since there exists nodes that already distinguish them.(Lines 38-44 in the Algorithm 7)

The general algorithm to process a graph is Algorithm 8. It uses the Algorithm 7 each time a new vertex is added.

In turn the Algorithm 7 uses the Algorithm 6 to split a node, the Algorithm 5 to determine the existing edges from the new clan to the remaining vertices and the Algorithm 5 to pack the resulting clan and when a new internal clan is obtained.

Complexity of Algorithm 7:

Let $|CQ_C|$ be the total number of elements in the coarsest quotient of C and let $|L_v|$ be the length of the list L_v, the **for** on line 20 takes $|CQ_C| - |L_v|$ times. The **for** involves for the copy of the coarsest quotient elements on lines 28 and 41 takes $|CQ_C|$ times each one. While the **for** in lines 32 and 45 takes $|L_n|$ plus the time of the split function, which is linear on the number of vertices involve.

Being all of them independent cases, thus the worst case is to be in the **for** that involves the split function and to it the worst cases is that any of the element in the coarsest quotient of the current clan can be distinguished,

Algorithm 7 Algorithm to get the strong clan decomposition tree when a vertex is added to a specific clan.

```
1: Input: A vertex $v$ to be added, a clan $C$ and the 2-structure induced by the vertices on $C$ plus $v$.
2: Output: Strong clan decomposition tree.
3: Let $L_n$ be a list of nonvisible maximal strong clans in the coarsest quotient of $C$ from $v$,
4: Let $L_v$ be a list of visible maximal strong clans in the coarsest quotient of $C$ from $v$,
5: Let $L_c$ be a list of visible, as the color of the clan, maximal strong clans in the coarsest quotient of
   $C$ from $v$.
6: if $C$ is empty or its coarsest quotient has just one element then
7:     Add $v$ to the coarsest quotient of $C$.
8:     Assign the type of $C$ as complete.
9: else if the type of $C$ is complete then
10:     if $L_c$ is equal to the maximal strong clans in the coarsest quotient of $C$ then
11:         Add $v$ to the coarsest quotient of $C$.
12:     else if $|L_c| \geq 1$ then
13:         Let $C_{aux}$ be an auxiliary complete clan.
14:         for all maximal strong clans $\mathcal{M}$ not in $L_c$ do
15:             Remove $\mathcal{M}$ from the coarsest quotient of $C$.
16:             Add $\mathcal{M}$ to the coarsest quotient of $C_{aux}$.
17:         end for
18:         Add $C_{aux}$ to the coarsest quotient of $C$.
19:         Add $v$ to the clan $C_{aux}$. (Algorithm 7 with $v$, $C_{aux}$ and the respective 2-structure as param-
            eters).
20:         Pack the coarsest quotient of $C_{aux}$. (Algorithm 5)
21:     else if $L_v$ is equal to the maximal strong clans in the coarsest quotient of $C$ and all of them are
        seen in the same way from $v$ then
22:         Copy the elements to the coarsest quotient of $C$ to the auxiliary complete clan $C_{aux}$, and
            remove them from $C$.
23:         Add $C_{aux}$ and $v$ to the coarsest quotient of $C$.
24:     else
25:         Change the type of the clan $C$ to primitive.
26:         for all maximal strong clans $\mathcal{M}$ in $L_n$ do
27:             Remove the $\mathcal{M}$ from the coarsest quotient of $C$.
28:             Split $\mathcal{M}$ from $v$ (Algorithm 6 with $\mathcal{M}$ and $v$ as parameters), add each maximal strong clan
                in the result to the coarsest quotient of $C$.
29:         end for
30:     end if
31: else
32:     if there is one maximal strong clan $\mathcal{M}$ in the coarsest quotient of $C$ that sees the remaining
        maximal strong clans in the same way than $v$ does then
33:         Add $v$ to the clan $\mathcal{M}$ (Algorithm 7 with $v$, $\mathcal{M}$ and the respective 2-structure as parameters).
34:     else if $L_v$ is equal to the maximal strong clans in the coarsest quotient of $C$ and all of them are
        seen in the same way from $v$ then
35:         Copy the elements to the coarsest quotient of $C$ to the auxiliary primitive clan $C_{aux}$, and
            remove them from $C$.
36:         Change the type of the clan $C$ to complete.
37:         Add $C_{aux}$ and $v$ to the coarsest quotient of $C$.
38:     else
39:         Add $v$ to the coarsest quotient of $C$
40:         for all maximal strong clans $\mathcal{M}$ in $L_n$ do
41:             Remove the $\mathcal{M}$ from the coarsest quotient of $C$.
42:             Split $\mathcal{M}$ from $v$ (Algorithm 6 with $\mathcal{M}$ and $v$ as parameters), add each maximal strong clan
                in the result to the coarsest quotient of $C$.
43:         end for
44:     end if
45: end if
46: Pack $C$ (Algorithm 5 with $\mathcal{G}$ and $C$ as parameters).
47: Return $C$
```

in that case $|L_n| = |CQ_C|$. The maximum size of this coarsest quotient is $|CQ_C| = \frac{n}{2}$ since each clan in it must have at least two vertices. As each element into the coarsest quotient has two vertices, the split function will have 2 as running time, having all the process n as running time.

In this way the complexity of the Algorithm 7 is linear on the number of vertices.

Algorithm 8 Algorithm to obtain a strong clan decomposition tree from a 2-structure

```
1: Input: A 2-structure.
2: Output: Strong clan decomposition tree.
3: Let current strong clan decomposition tree be an empty clan.
4: Let $\mathcal{U}$ be the set of vertices in the 2-structure.
5: for all $v \in \mathcal{U}$ do
6:     Current strong clan = root clan of the current strong clan decomposition tree.
7:     Current strong clan decomposition tree = Apply the Algorithm 7 with $v$, the current strong clan and the respective 2-structure induced by them as parameters.
8: end for
```

Complexity of Algorithm 8:

Let n be the total number of vertices of a 2-structure, the run time of the Algorithm 8 is $\sum_{i=1}^{n} i$ since the Algorithm 7 is linear. Thus, the Algorithm 8 runs in time $O(n^2)$.

3.2.5 Algorithms 6, 7 and 8: correctness and examples

Let $\mathcal{C} \subseteq \mathcal{U}$ be a clan to which we attempt at adding $x \in \mathcal{U}$. We assume $|\mathcal{C}| \geq 2$ as the base cases are directly correct by definition. Let $\mathcal{M}$ be the coarsest quotient of $\mathcal{C}$, which can be either primitive or complete; it consists of maximal strong clans $\mathcal{M} = \{\mathcal{C}_i | i \in I\}$ for some index set I, with $\mathcal{C}_i \subseteq \mathcal{U}$, each $\mathcal{C}_i$ is handled through its own coarsest quotient $\mathcal{M}_i$ like $\mathcal{C}$ is handled through $\mathcal{M}$. We allow for the slight abuse of notation $(\mathcal{C}_i, \mathcal{C}_j) \in \mathcal{E}_p$ to refer to all edges with one endpoint within $\mathcal{C}_i$ and another within $\mathcal{C}_j$ being in $\mathcal{E}_p$. Since they are clans, such p is well-defined.

If $\mathcal{M}$ is a complete clan, then $p_{\mathcal{M}}$ is the "color of the clan", that is, the index of the equivalence class of its edges, $\mathcal{E}_{p_{\mathcal{M}}}$; we often omit the subindex $\mathcal{M}$ when it is clear from the context. We denote $J \subseteq I$ the (indices of) maximal strong clans in it that are seen from x with color $p_{\mathcal{M}}$: $J = \{i \in I | \forall z \in \mathcal{C}_i (x, z) \in \mathcal{E}_p\}$. This is the formalization of the "list of nodes visible with the color of the clan".

Likewise, $\mathcal{C}_i$ being visible from x is formalized as $\exists q \forall z \in \mathcal{C}_i((x, z) \in \mathcal{E}_q)$ and its negation, which says that adding x requires to split $\mathcal{C}_i$, is $\forall q \exists z \in \mathcal{M}_i((x, z) \notin \mathcal{E}_q)$.

Since maximal strong clans in $\mathcal{M}$ are proper subclans of $\mathcal{C}$, $|I| \geq 2$. Additionally, it is known that if $\mathcal{M}$ is primitive then every proper subclan $\mathcal{C}' \subset \mathcal{C}$ is included in one of the maximal strong subclans: $\exists i(\mathcal{C}' \subseteq \mathcal{C}_i)$.

Definition 3.2. *Given a clan $\mathcal{C} \subset \mathcal{U}$, a proper subclan of $\mathcal{C}' \subset \mathcal{C}$, and a vertex $x \notin \mathcal{C}$, we say that x is like $\mathcal{C}'$ in $\mathcal{C}$ if x sees the rest of $\mathcal{C}$ in the same way as $\mathcal{C}'$: for all $y \in \mathcal{C}'$, and for all $z \in (\mathcal{C} \setminus \mathcal{C}')$, the edges (x, z) and (y, z) are equivalent.*

The following theorems are focused on identifying the components of $\mathcal{M}'$, the desired coarsest quotient of $\mathcal{C} \cup \{x\}$, for the cases (1)(b), (1)(d), (2)(c) and (2)(a) of the Algorithm 7. The correctness of the rest of the cases is easier to argument.

For the case (1)(b) above, we have the following theorem.

Theorem 3.3. *If $\mathcal{M}$ is complete and $\emptyset \neq J \neq I$, then $\mathcal{M}' = \{\mathcal{C}_j | j \in J\} \cup \{\{x\} \cup \bigcup_{j \notin J} \mathcal{C}_j\}$.*

Proof. Items inside each $\mathcal{C}_j$ with $j \in J$, which is a clan of $\mathcal{C}$, are not distinguishable from either other $\mathcal{C}_j$'s or x; thus, they are clans of $\mathcal{C} \cup \{x\}$. By the same token, so is $\{\{x\} \cup \bigcup_{j \notin J} \mathcal{C}_j\}$. Assume they are not strong clans. Overlaps of other clans within $\mathcal{C}$ with the $\mathcal{C}_i$'s cannot exist because $\mathcal{M}$ is the coarsest quotient of $\mathcal{C}$. Hence, a hypothetical overlapping clan must include x and some $y \in \mathcal{C}_j$ for some $j \in J$, but not all $z \in \bigcup_{j \notin J} \mathcal{C}_j$. But, then, $(y, z) \in \mathcal{E}_p$ and $(x, z) \notin \mathcal{E}_p$ so it is not a clan. Thus, they are strong clans. Larger clans cannot exist either as, then, ignoring x would lead to larger clans already in $\mathcal{M}$. $\square$

In the next theorem, the complete case corresponds to (1)(d) above and the primitive case is (2)(c). The sequence of maximal strong clans corresponds to the successive recursive calls of Algorithm 6.

Theorem 3.4. *If $\mathcal{M}$ is primitive and x is not like any $\mathcal{C}_i$, or if $\mathcal{M}$ is complete with $J = \emptyset$, and besides, in either case, $\forall q \exists z \in \mathcal{C}(x,z) \notin \mathcal{E}_q$, then $\mathcal{C}' \in \mathcal{M}'$ if and only if $\mathcal{C}' = \{x\}$ or there is a sequence $\mathcal{C} = \mathcal{C}'_0 \supset \mathcal{C}'_1 \supset \cdots \supset \mathcal{C}'_n \supset \mathcal{C}'$, where:*

1. *each $\mathcal{C}'_{i+1}$ is a maximal strong clan in the coarsest quotient of $\mathcal{C}'_i$ for $i < n$ (holds vacuously if $n = 0$),*
2. *x does not split $\mathcal{C}'$,*
3. *x splits each $\mathcal{C}'_i$ for $i \leq n$, and either*
   (a) *the coarsest quotient of $\mathcal{C}'_n$ is primitive and $\mathcal{C}'$ is one of its components or*
   (b) *the coarsest quotient of $\mathcal{C}'_n$ is complete and there is q such that $\mathcal{C}'$ is the union of all those components of the coarsest quotient of $\mathcal{C}'_n$ that are connected to x with edges of class $\mathcal{E}_q$.*

Note that, for some such $\mathcal{C}'$, n can turn out to be 0.

Proof. We study first the case of $\{x\}$. Let $\mathcal{C}'$ be the maximal strong clan $\mathcal{C}' \in \mathcal{M}'$ that has $x \in \mathcal{C}'$. Assume $|\mathcal{C}'| \geq 2$.

In case $\mathcal{M}$ is complete, say with color p, and with $J = \emptyset$, use $|\mathcal{C}'| \geq 2$ to fix $z \in \mathcal{C}'$ such that $z \neq x$, and let $\mathcal{C}_{i_z} \in \mathcal{M}$ such that $z \in \mathcal{C}_{i_z}$. Since $|\mathcal{M}| \geq 2$, there is at least some other $\mathcal{C}_{i_y} \in \mathcal{M}$, $i_y \neq i_z$; and some $y \in \mathcal{C}_{i_y}$ must have $(x,y) \notin \mathcal{E}_p$ as, otherwise, $i_y \in J$. Then, y sees differently x and z because $(y,z) \in \mathcal{E}_p$, the color of $\mathcal{M}$, which contradicts the fact that $\mathcal{C}'$ is a clan.

Now, in case $\mathcal{M}$ is primitive, since $|\mathcal{C}'| \geq 2$, the set $\mathcal{C}' \setminus \{x\}$ is nonempty and, hence, also a clan in $\mathcal{C}$: there is i such that $\mathcal{C}' \setminus \{x\} \subseteq \mathcal{C}_i$, and the maximality of $\mathcal{C}'$ implies equality: $\mathcal{C}' \setminus \{x\} = \mathcal{C}_i$. Then, the other maximal strong clans cannot distinguish $\mathcal{C}_i$ from x as, otherwise, $\mathcal{C}'$ is not a clan, and therefore x is like $\mathcal{C}_i$, in contradiction with the hypothesis.

Thus, in both cases, $|\mathcal{C}'| = 1$ and necessarily $\mathcal{C}' = \{x\}$. That is, the single clan in $\mathcal{M}'$ containing x is the singleton $\{x\}$.

We move on to the case where $x \notin \mathcal{C}'$, which implies $\mathcal{C}' \subseteq \mathcal{C}$. It has to be a proper subset, $\mathcal{C}' \subset \mathcal{C}$, because $\mathcal{C}$ in full is not a clan in $\mathcal{C} \cup \{x\}$: otherwise we would contradict the hypothesis that $\forall q \exists z \in \mathcal{C}(x,z) \notin \mathcal{E}_q$ (equivalently, x splits $\mathcal{C}$).

Assume first $\mathcal{C}' \in \mathcal{M}'$. Together with $\{x\} \in \mathcal{M}'$ that we have already argued, this implies that there is q such that $\forall z \in \mathcal{C}'((x,z) \in \mathcal{E}_q)$. Fix that q.

Also, $\mathcal{C}'$ must be a clan in $\mathcal{C}$ as, otherwise, it cannot be a clan in the larger set $\mathcal{C} \cup \{x\}$ and, by the same reason, $\mathcal{C}'$ is not split by x.

We construct the claimed sequence inductively. The basis is, of course, $\mathcal{C}'_0 = \mathcal{C}$. Note that we just argued that x splits it.

Suppose the construction has proceeded up to some $\mathcal{C}'_i \supset \mathcal{C}'$ where x splits $\mathcal{C}'_i$, the inclusion being proper because x does not split $\mathcal{C}'$. The coarsest quotient of $\mathcal{C}'_i$ can be either primitive or complete.

If the coarsest quotient of $\mathcal{C}'_i$ is primitive, then there is $\mathcal{C}'_{i+1}$ in the coarsest quotient of $\mathcal{C}'_i$ such that $\mathcal{C}'_{i+1} \supseteq \mathcal{C}'$. We check whether all the edges from x to $\mathcal{C}'_{i+1}$ are equivalent. If so, then $\mathcal{C}'_{i+1}$ is a strong clan in $\mathcal{C} \cup \{x\}$ and, by maximality of $\mathcal{C}'$, it must be $\mathcal{C}'_{i+1} = \mathcal{C}'$, so that $\mathcal{C}'$ is one of the components of the coarsest quotient of $\mathcal{C}_i$ and the construction stops with $n = i$. Otherwise, x splits $\mathcal{C}'_{i+1}$ and the construction has proceeded one further step.

Alternatively, if the coarsest quotient of $\mathcal{C}'_i$ is complete, we consider its decomposition $\{\mathcal{C}''_j | j \in I'\}$ for some index set I'. By the properties of complete clans, $\mathcal{C}'$ being a clan in $\mathcal{C}$ is either a proper subset of some $\mathcal{C}''_j$, or a union of one or more of them (but not all since $\mathcal{C}' \subset \mathcal{C}'_i$ properly). In the latter case, we have the claimed statement by finishing the construction with $n = i$ and letting q be the color of the edges from $\mathcal{C}'$ to x; maximality of $\mathcal{C}'$ implies that all the components having that edge color in the connection to x are included in $\mathcal{C}'$.

Finally, in the former case, $\mathcal{C}' \subset \mathcal{C}''_j$ for $j \in I'$, note that if $\mathcal{C}''_j$ were not split by x, $\mathcal{C}''_j$ being a clan of $\mathcal{C}$, then it would be a clan in $\mathcal{C} \cup \{x\}$, contradicting the maximality of $\mathcal{C}'$. Thus, $\mathcal{C}''_j$ must be split by x and the construction goes on, taking this $\mathcal{C}''_j$ as $\mathcal{C}'_{i+1}$.

The construction cannot run forever because $\mathcal{C}$ is finite.

Conversely, we must prove that every set $\mathcal{C}'$ for which a chain as given exists is in $\mathcal{M}'$: we have that $\mathcal{C} = \mathcal{C}'_0 \supset \mathcal{C}'_1 \supset \cdots \supset \mathcal{C}'_n \supset \mathcal{C}'$ (each $\mathcal{C}'_i$ a maximal strong clan in the previous one, each split by x, but not $\mathcal{C}'$) and that $\mathcal{C}'$ is either one of the components of a primitive coarsest quotient of $\mathcal{C}_n$, or a union of one or more components of a complete coarsest quotient of $\mathcal{C}_n$, namely, those that see x with color q.

We need to see that $\mathcal{C}'$ is a clan (in $\mathcal{C} \cup \{x\}$), strong, and maximal. When the coarsest quotient of $\mathcal{C}_n$ is primitive with $\mathcal{C}'$ one of its components, the chain of maximal strong clans keeps $\mathcal{C}'$ a strong clan all the way up; besides, these are the only strong clans that can contain $\mathcal{C}'$. Additionally, $\mathcal{C}$ is also maximal because x splits all the other clans along the way, so no superset of $\mathcal{C}$ is a strong clan of $\mathcal{C} \cup \{x\}$.

When the coarsest quotient of $\mathcal{C}_n$ is complete, $\mathcal{C}'$ is a union of components in that coarsest quotient; therefore, it is a clan in $\mathcal{C}$. Also, x does not split $\mathcal{C}'$ either because all those components are connected to x by edges of color q, so $\mathcal{C}'$ is a clan in $\mathcal{C} \cup \{x\}$.

To see that it is a strong clan, observe that a hypothetical clan including part of $\mathcal{C}'$ plus some proper part of another component of $\mathcal{C}_n$ would overlap that component, which cannot be the case because all these components are strong; and cannot include any other component as a whole, because all those connected by color q were already taken in for $\mathcal{C}'$ so any other one is distinguished from $\mathcal{C}'$ by x. Additionally, if a hypothetical clan overlaps $\mathcal{C}$ by including vertices from outside $\mathcal{C}_n$, then it would overlap $\mathcal{C}_n$, which cannot be the case either as $\mathcal{C}_n$ is a strong clan.

The argument for maximality is similar: now, the hypothetical clan includes all of $\mathcal{C}'$ instead of a part of it, plus additional vertices, and we argue like before: if these vertices come from $\mathcal{C}_n$, either they are a proper subset of another component, and then that component would not be strong, or they include totally other components, and then x distinguishes them and is not a clan; and if they come from outside $\mathcal{C}_n$, then they would overlap $\mathcal{C}_n$, which cannot be the case; that is because not all the components of $\mathcal{C}_n$ are taken into $\mathcal{C}'$ since we know $\mathcal{C}' \subset \mathcal{C}'_n$ properly and the components left outside $\mathcal{C}'$ are distinguished from $\mathcal{C}'$ by x. □

Regarding case (2)(a) above, the fact that needs specific analysis is the reason why one subclan to recurse on is unambiguously identified and will lead to a correct outcome.

Theorem 3.5. *Let $\mathcal{M}$ be a primitive coarsest quotient of a clan $\mathcal{C}$ and let $x \notin \mathcal{C}$. Then there exists at most one maximal strong subclan $\mathcal{C}_i \in \mathcal{M}$ such that x is like $\mathcal{C}_i$ in $\mathcal{C}$. If there exists one such $\mathcal{C}_i$, then $\mathcal{C}_i \cup \{x\}$ is a maximal strong subclan of $\mathcal{C} \cup \{x\}$.*

Proof. Consider $\mathcal{C}_i \in \mathcal{M}$ and $\mathcal{C}_j \in \mathcal{M}$ with $i \neq j$, and suppose x is like $\mathcal{C}_i$. Since $\mathcal{M}$ is primitive, there is a third clan $\mathcal{C}_k \in \mathcal{M}$ that distinguishes $\mathcal{C}_i$ from $\mathcal{C}_j$. As x is like $\mathcal{C}_i$, the edges connecting x to $\mathcal{C}_k$ are equivalent to those connecting $\mathcal{C}_i$ to $\mathcal{C}_k$, hence they are not equivalent to those connecting $\mathcal{C}_j$ to $\mathcal{C}_k$; this implies that x is not like $\mathcal{C}_j$.

If such $\mathcal{C}_i$ is available, let $\mathcal{C}' = \mathcal{C}_i \cup \{x\}$: by the choice of i, it is a clan; it must be a strong clan because, as just argued, x cannot form a clan with any other $\mathcal{C}_j$ and no clan of $\mathcal{C}$ can overlap $\mathcal{C}_i$, which was already strong. Likewise, any proper superset of $\mathcal{C}'$ becomes a proper superset of $\mathcal{C}_i$ once x is removed, and cannot be a clan due to the maximality of $\mathcal{C}_i$. $\square$

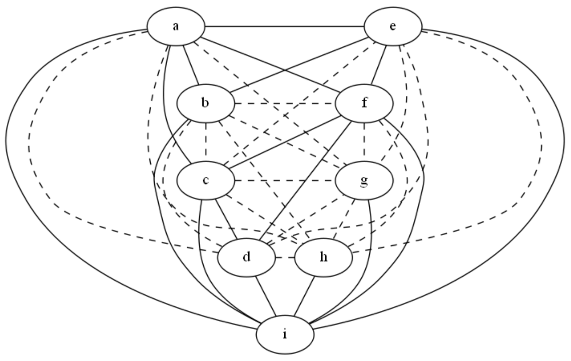


Figure 3.3: Example: Graph to apply the decomposition Algorithm 7.

As an example of how the algorithms work, we apply them on the graph of Figure 3.3.

At the begin we add the vertex a to an empty clan decomposition tree, getting a tree with just one singleton clan. In the next step we add the vertex b to it, having as a result a complete clan with two maximal strong clans in its coarsest quotient. Both steps are initial cases, they are shown in Figure 3.4.

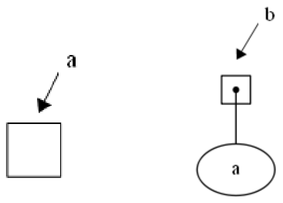


Figure 3.4: Initial cases from Algorithm 7

When we add c to this clan, we are in the case 1.(b) since a is seen from c by the color of the clan but b is not. Thus, the maximal strong clans that are not seen from the color of the clan are removed to another clan and c is added to it, in this case we add c to a complete clan with just b in its coarsest quotient (the initial case). All the process is shown in Figure 3.5.

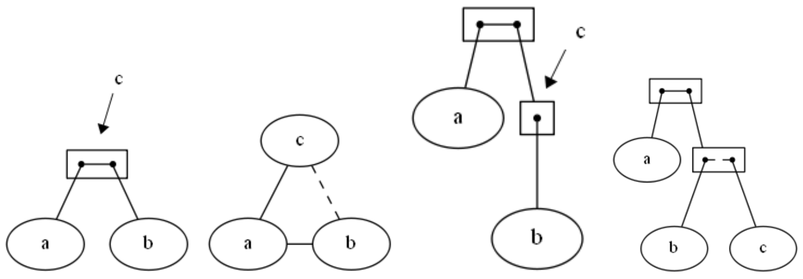


Figure 3.5: Applying 1.(b) from Algorithm 7

Now, we have in the root a complete clan and the vertex d is added to it. We find d may not see one of the maximal strong clans in the coarsest quotient of the root, the case 1.(d), center of Figure 3.6. Thus, the type of the clan is changed to primitive and the maximal strong clans that are seen from d in the same way will conform a new coarsest quotient while those nodes that are not seen from d are split; in this case we do not have any new coarsest quotient and the maximal strong clan conform by b and c is split, getting as result the decomposition tree shown in the right of Figure 3.6.

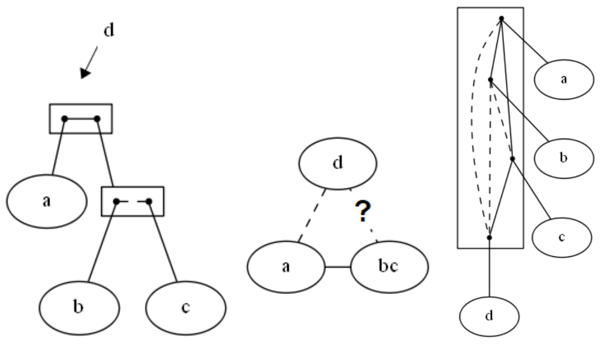


Figure 3.6: Applying 1.(d) from Algorithm 7

In the next step we add the vertex e to a primitive clan root, left of Figure 3.7. We find that there is one maximal strong clan b in its coarsest quotient that sees all the remaining maximal strong clans in the same way than the vertex to be added, the case 2.(a), we can see it in the center of Figure 3.7. Thus, the new vertex is added to it, the resulting clan decomposition tree is shown in the right of Figure 3.7.

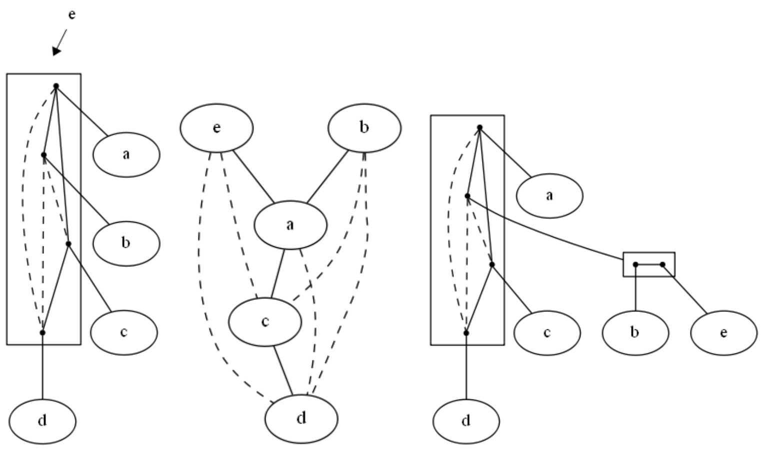


Figure 3.7: Applying 2.(a) from Algorithm 7

When we add f to the primitive clan on the root of the current decomposition tree, left of Figure 3.8, we find f can not see one of the maximal strong clans, the clan conform by b and e, as we have a primitive clan we are in case 2.(c), and the elements of the clan are split, right of Figure 3.8.

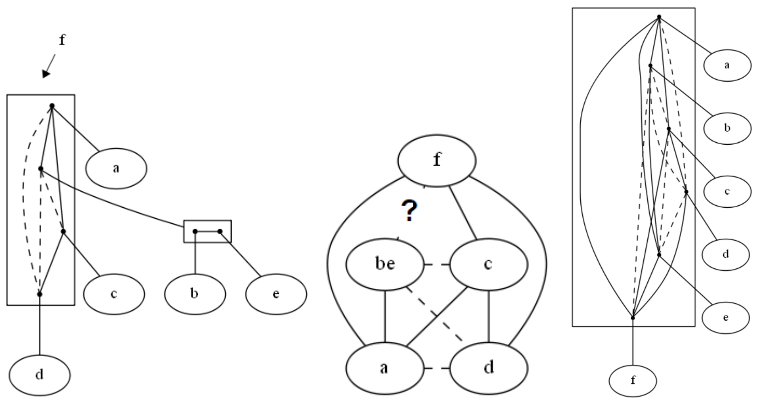


Figure 3.8: Applying 2.(c) from Algorithm 7

In the next step, we add g to the primitive root of this decomposition tree, and we find that g sees all of the maximal strong clans in the coarsest quotient of the root clan in the same way, case 2.(b), center of Figure 3.9. Thus, this root clan and the new vertex will be in the coarsest quotient of a new complete clan, right of Figure 3.9.

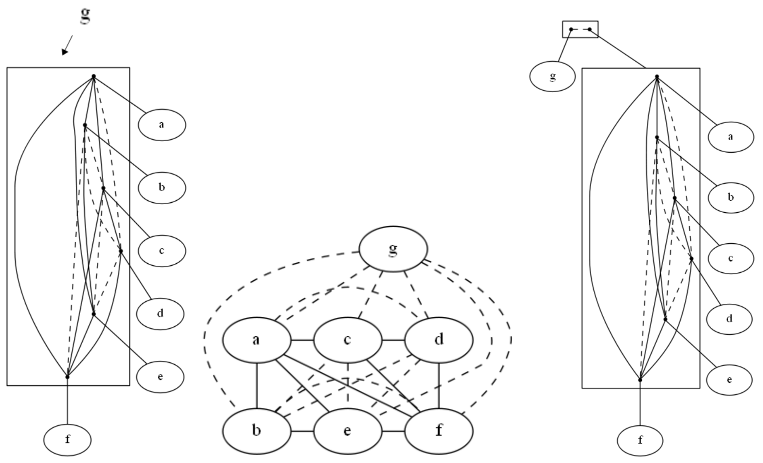


Figure 3.9: Applying 2.(b) from Algorithm 7

When we add h to the complete clan in the root of the current decomposition tree, left of Figure 3.10, we find that h sees in the same way than the color of the clan to all the maximal strong clans in the coarsest quotient of the root clan, the case 1.(a), center of Figure 3.10. Thus, h is added as another maximal strong clan to the coarsest quotient of the complete root clan, right of Figure 3.10.

Finally, to add i produces a superior clan, since i sees in the same way all the maximal strong clans of the current complete root clan but different than the color of the clan, the case 1.(c), center of Figure 3.11. Thus, this root clan and the new vertex will be in the coarsest quotient of a new complete clan, the resulting clan decomposition tree is shown in the right of Figure 3.11.

As an example to detail the use of the split function, we have the following case.

Example 3.6. *Assume we add n to the decomposition shown in the left of Figure 3.12, such that n is connected by solid lines just with b, c, h, j and l.*

Following the algorithm we will give the maximal strong clans in the coarsest quotient of the current vertices plus the new vertex n.

Since the coarsest quotient of the root clan is complete those nodes seen in the same way by n will conform a maximal strong clan, in this case we get the clan $\{a, m\}$. And those nodes that are not visible from n are split, thus we split the clan $\{b, c, d, e\}$ and the clan $\{f, g, h, \{i, j, k, l\}\}$. When we split the clan $\{b, c, d, e\}$ we have again it is a complete clan, thus following

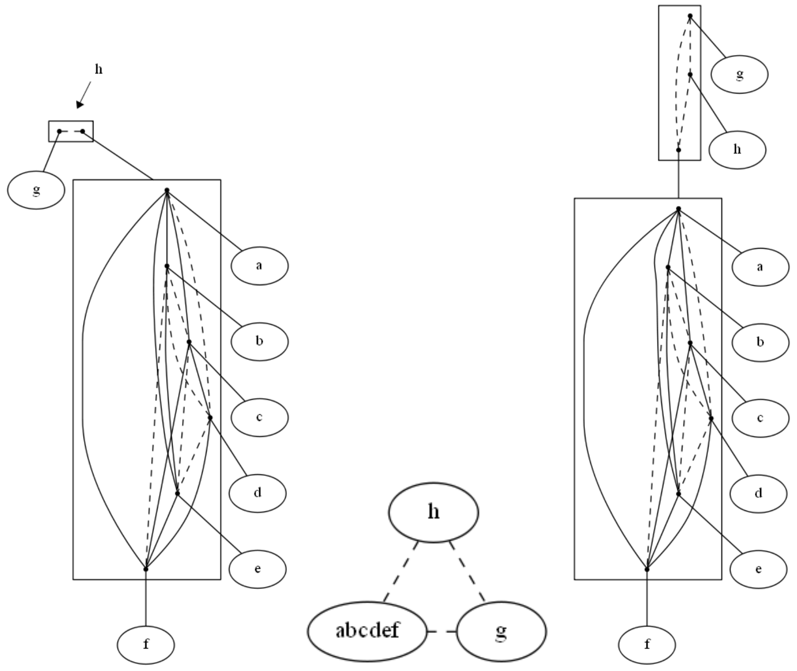


Figure 3.10: Applying 1.(a) from Algorithm 7

the previous constrains we get $\{b, c\}$ *and* $\{d, e\}$ *as maximal strong clans.*

As the clan $\{f, g, h, \{i, j, k, l\}\}$ *is a primitive clan, the clans into its coarsest quotient that are seen from* n *continue being maximal strong clans in the new decomposition, while those clans that are not seen by* n *are split. In this case we continue having* f, g *and* h *as maximal strong clans, and we split the clan* $\{i, j, k, l\}$ *since it is not possible to determine an edge to it from* n. *Splitting the clan* $\{i, j, k, l\}$, *as it is a complete clan, we get* $\{j, l\}$ *and* $\{i, k\}$ *as maximal strong clans.*

The resulting decomposition is shown in the right of Figure 3.12.

3.3 A comparison with related algorithms

There are proposed algorithms to get the modular decomposition and the 2-structure clan decomposition, a detailed survey already exists in [19]. To do an extensive comparison between them and our algorithm would take an extensive work out of the line of this research but we explain in detail the two

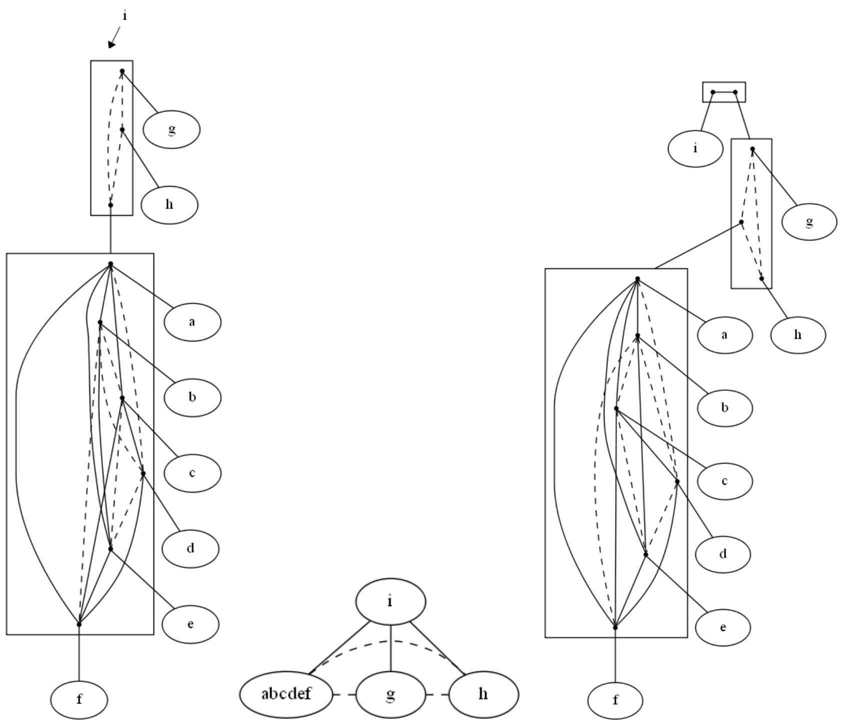


Figure 3.11: Applying 1.(c) from Algorithm 7

most relevant works: [28] and [29]. The order in which we mention them is not related to how closer they are to our algorithm. Those earlier algorithms differ among them and with ours in their quite diverse terminology for the main notions. Although the type of modules are called prime and degenerate in these works, we will follow our nomenclature of primitive and complete, respectively.

Besides that, the main differences between the algorithm in [29] and our algorithm, are the algorithm in [29] is applied only on graphs instead of 2-structures and the use of a parallel structure to the decomposition tree. This parallel structure is a tree where refinements are made until get the decomposition tree. The literal application of the algorithm exactly as it is described there would not work on 2-structures. As an overview, the algorithm in [29] begins with a tree that is modified until reaching the strong decomposition tree, that is in each step we have a tree but, in contrast to our algorithm, this includes all the nodes and the final decomposition tree is not reached until the end of the algorithm.

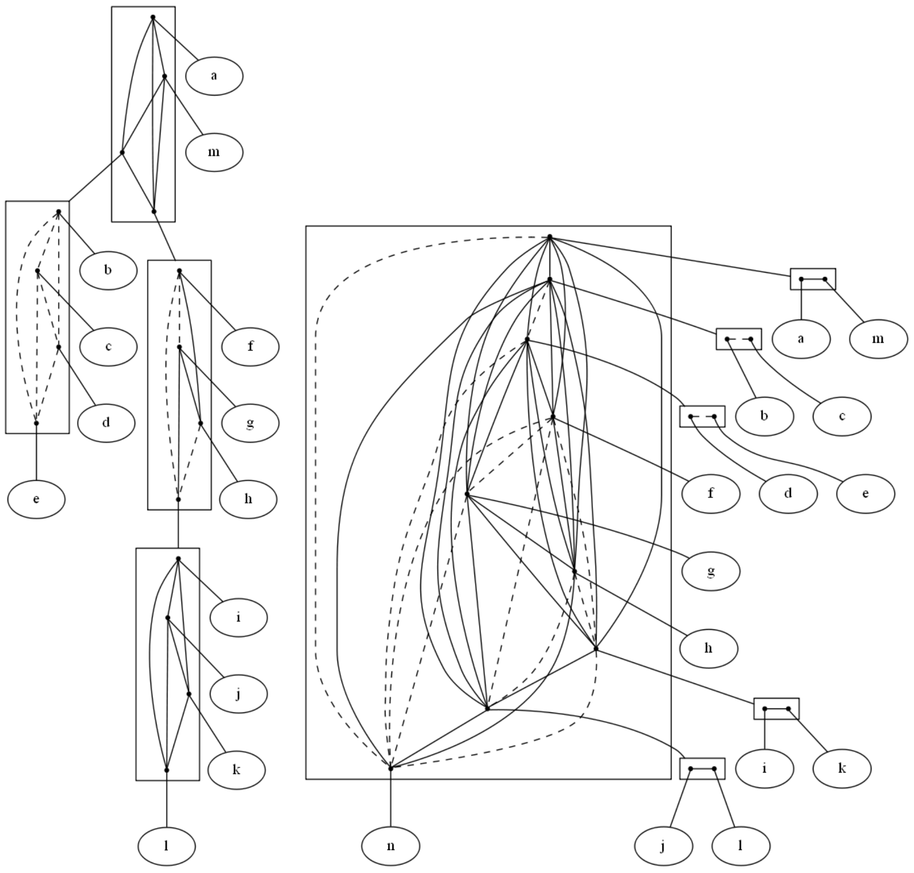


Figure 3.12: Example 3.6 to illustrate the use of Theorem 3.4

The other algorithm to be considered is [28], it is an incremental algorithm as our algorithm. It works not only with complete and primitive clans as our algorithm does, but also with linear clans. The linear clans appear when decomposing 2-structures that are not necessarily symmetric; these are akin to directed graphs. As we already stated, we do not take them into account because in our data analysis all our 2-structures are symmetric, since they rely on undirected Gaifman graphs. As a main procedural difference, this algorithm builds a new structure $M(\mathcal{G}', z)$, where z is the vertex added to $\mathcal{G}$ and $\mathcal{G}' = \mathcal{G} + z$, this new structure does the labeling process each time a new node is added. In our algorithm, we use the Union-Find structure (Subsection 3.2.1) each time a new vertex is added.

The algorithm in [29] is applied on undirected graphs. According to the module definition, they are working with two equivalence classes, and the time of execution is $O(n+m)$ where n and m are the number of vertices and edges of the graph respectively.

The algorithm begins with a tree T that is modified until reaching the strong decomposition tree $MD(\mathcal{G})$. There are three classes of M that characterize tree T in the different refinement phases:

$M1$: The internal nodes are labeled primitive or complete. For each complete node, there is a system of representative from its children that induces a complete subgraph in $\mathcal{G}$.

$M2$: Same as $M1$, but the children of each complete node M_D are modules in the restricted graph M_D and its coarsest quotient.

$M3$: Same as $M2$, but every node of T is a module in $\mathcal{G}$.

The algorithm starts with the Theorem in [11] that says that in every primitive undirected graph $\mathcal{G}$ there is an induced P_4. There is a linear algorithm that computes a P_4 tree which is the entry $M1$ tree, let us see how the refinement of the tree is obtained.

Getting $M1$ tree As we already say, the initial $M1$ tree is a P_4 tree T. If the graph does not induce a P_4, the initial $M1$ tree will be the cotree of $\mathcal{G}$. In this case we only have a tree in which all the nodes are complete.

If a P_4 is gotten, the *CreateTree* algorithm in [37] is run to obtain the P_4 tree from the graph, this tree tell us when the nodes are complete or primitive.

From $M1$ to $M2$ Once we get a primitive node, it is largely ignored, because only complete nodes could violate the conditions needed for the tree to be an $M2$ tree. There are three main functions used here:

- Forcing graphs: Find the elements of the modules inside a complete module.
- SCC: Label each complete module with the forcing acyclic graph on its children.
- PQ: It is a refining of SCC, remove nodes either not modules or nodes that have outgoing edges.

So the procedure to obtain an $M2$ from a $M1$ is to apply PQ to the T tree and then apply SCC to the resulting graph; SCC runs the Forcing graph algorithm.

It is an $M2$ tree since if a child A of a complete node of $M1$ disagrees on a vertex of $\mathcal{G}$ that is contained in one of its siblings, B, thus B also disagrees on a vertex of $\mathcal{G}$ that is contained in A. When a forced graph is computed on any refinement of $M1$, their connected components are strongly connected.

From $M2$ to $M3$ There is apply the *Decomp* function on the original undirected graph $\mathcal{G}$ and on $M2$ tree T corresponding to $\mathcal{G}$. This function find the members of T that are modules in $\mathcal{G}$, since they can not overlap with any other module of $\mathcal{G}$ by definition of $M2$. Any other member of $MD(\mathcal{G})$ is a non trivial children of a complete node of T. Inside of a complete node, find the elements that conform modules using a list of its children strong neighbors (SN). Finally, for each member in $MD(\mathcal{G})$ that is not a node in T, insert it in T. Purge T of those members that are not modules and return T as result.

Thus, in the first kind of tree the internal nodes are labeled primitive or complete by recognizing the P_4 induced. If the original graph does not induce any P_4, the initial graph will have just complete internal nodes. To get the second kind of tree, only the complete nodes of the previous kind of tree will be analyzed in order to find modules within the complete nodes. To reach the third kind of tree, modules that are not strong are removed.

Example 3.7. *From the Figure 3.13 we get $\{f, h, g, abcde\}$ as induced P_4, while the graph conformed by $\{a, b, c, d, e\}$ does not induce P_4. In this way,*

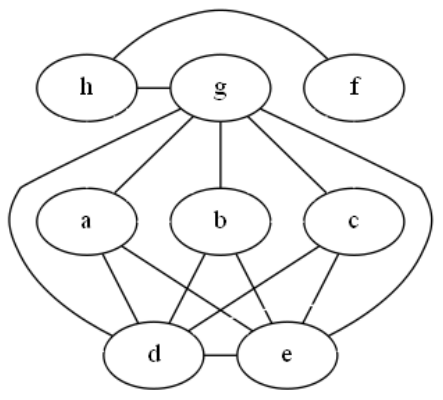


Figure 3.13: Graph of Example 3.7: Initial graph.

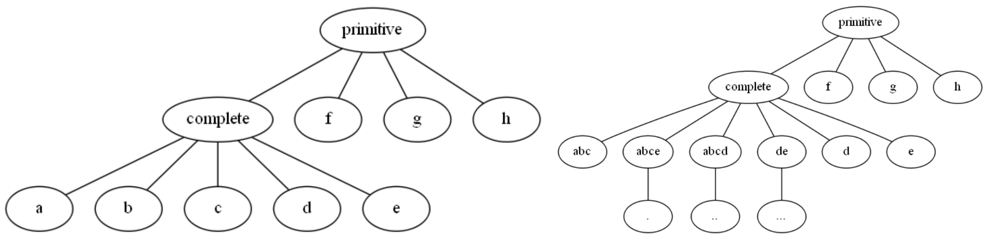


Figure 3.14: Graph of Example 3.7: Intermediate decomposition phases.

the first kind of tree is a tree with two internal nodes, one labeled as primitive and the other one labeled as complete, left of Figure 3.14.

The next step is to analyze just the complete modules looking for more modules (strong or not), those only could be complete since all primitive ones were obtained in the previous step. We get, among other modules, $\{abc\}$ *as complete strong module and in turn* $\{a, b, c\}$ *are also modules of it, the process to get the second kind of tree is shown in right of Figure 3.14.*

The last step is to prune the previous tree, removing those modules that are not strong modules. The final decomposition is shown in Figure 3.15. Thus we illustrate the different modus operandi with respect to our algorithm.

The algorithm in [28] generalizes the elements of the Muller and Spinrad algorithm for decomposition of undirected graphs [32]. The structure used in that algorithm is replaced by a scheme that labels each edge with at most one node. Having an incremental algorithm, that is, from the clan decomposition tree of a $\mathcal{G}$ graph, $MD(\mathcal{G})$, is generated the clan decomposition

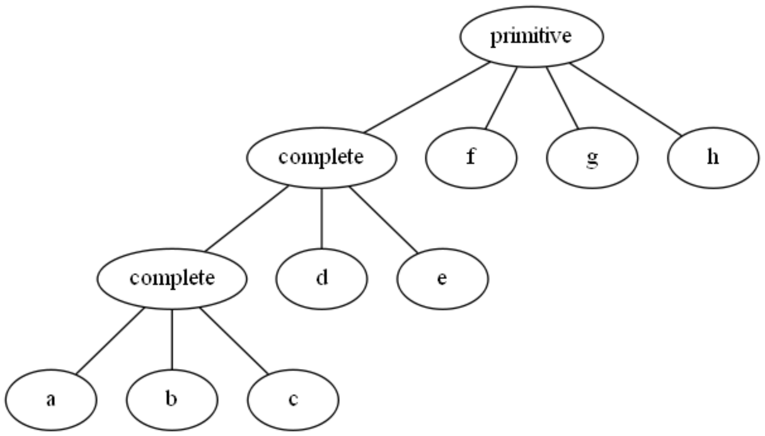


Figure 3.15: Graph of Example 3.7: Modular decomposition tree.

tree of $\mathcal{G}' = \mathcal{G} + z$ where z is the new node. Since our algorithm works only with complete and primitive clans, we will avoid those parts of the algorithm referent to linear clans. Once this was said, the algorithm runs as follows:

- Label the clans. The clans of the $MD(\mathcal{G})$ are labeled uniform, if z may not distinguish its elements (they are seeing in the same way), or non uniform in other way.
- Find Z_0. It is selected as Z_0 the maximum clan labeled as non uniform.
- Obtain $DL(Z_0, \mathcal{G}', z)$. According to the type of the clan Z_0 the elements of $DL(Z_0, \mathcal{G}', z)$ are determined. If Z_0 is primitive, $DL = Z_0$. Else, assume the equivalence class (color) of the clan is a, $DL = z \cup x$ where $x \in Children(\mathcal{G})$ such that x is not in the same equivalence class than the clan according with z, that is z does not see x by a.
- These elements will conform a new clan child of Z_0. Removing of the children of Z_0 such elements that are in DL. If $|DL| = 2$, then the clan type of DL will be complete (primitive if it is not).
- Construct $M(\mathcal{G}', z)$. There is a parallel structure called $M(\mathcal{G}', z)$. For each clan in $MD(\mathcal{G})$, if a clan is uniform with z, that is, z is connected to the member of the clan by the same kind of edges, it will be in $M(\mathcal{G}', z)$. Else, if the module is primitive, its children will be in $M(\mathcal{G}', z)$ and if

the clan type is complete, its maximum children seen in the same way will be join and added to $M(\mathcal{G}', z)$ (union siblings).

- Assign the split labels as appropriate.

Split label algorithm The main purpose of the split label is to find in Z_0 which node is clan with z.

The algorithm to update the labels is: For each complete clan M in $MD(\mathcal{G})$ and for each pair $\{S, T\}$ children of M that are contained in $DL(Z_0, \mathcal{G}', z)$ and members of different classes of M partition. The partition of a clan is determined by the equivalence classes of its children according to z.

- If S and T are uniform with respect to z, then z is the split node of all edges connecting S and T.
- If at least one of them is non uniform with respect to z, let us say T:
  - If T is complete, let A be any partition class of the T's children, so $T \setminus A$ is not empty. Any node in A is a split node of the edges connecting S and $T \setminus A$, while any node in $T \setminus A$ is a split node for the edges connecting $S \setminus A$.
  - If T is primitive, take $s \in S$, if $T \cup \{s\}$ is a clan in $\mathcal{G}'$, there exists U in children of T such that $U \cup \{s\}$ is a clan of $\mathcal{G}'$. A node v of $T \setminus U$ distinguishes s and U, assigning v as the split label of edges connecting S and U.

The algorithm to update the split labels to edges incident to z: If there is a split label for (u, v), $u \in Z_0$ and $v \in dom(\mathcal{G}) \setminus Z$, copy it to (z, v) and (v, z). If $DL(Z_0, \mathcal{G}', z)$ is primitive:

- If Z_0 is primitive, for each U in the children of Z_0, v that distinguish z from U will be a label to them.
- If Z_0 is complete, select U such that $U \in children_{\mathcal{G}}(Z_0)$ and $U \in DL(Z_0, \mathcal{G}', z)$. Let $X = \cup children_{\mathcal{G}'}(DL(Z_0, \mathcal{G}', z)) \setminus \{z\}$ be such that $U \cap X = \emptyset$. $u \in U$ will be the split label of z to X. Let $V \in children_{\mathcal{G}}(Z_0)$ be such that $V \in X$, $v \in V$ that distinguish z and u the split label of all edges connecting z and $(DL(Z_0, \mathcal{G}', z) \setminus \{z\}) \setminus X$.

Thus, the algorithm works with two trees. One of them keeps the clan decomposition of the graph, each internal node of the tree represents a clan but its internal 2-structure is not shown. The other tree stores node information and information between the nodes by split labels, we will talk a little more about them later.

Example 3.8. *Suppose we have a complete clan $\{a, b, c, d, F, G\}$ whose elements are connected by black color edges, and a new node e is added, Figure 3.16. This new node does not see some of the clan elements, F and G, and the rest of the elements are seen by not black color, say red to a, b and blue to c, d, Figure 3.17. Suppose G is a primitive clan conform by g_1, g_2, g_3 and F is a complete clan conform by f_1, f_2, Figure 3.18; and their elements, g_1, g_2, g_3, f_1, f_2, are uniform with respect to e, Figure 3.19.*

The modular decomposition before to add e is shown in Figure 3.20. The nodes are filled by white if there is an unambiguous color to connect the node with e, and gray otherwise ("not seen"). The algorithm ask for the type of the node that is not seen by e. If the node is primitive its children are the same plus the new vertex. Else (the node is complete), its children will be those nodes that are not connected with e by the same kind of edges than the edges of the node. The other tree stores each visible clan of the previously described tree. If the node is complete its children are grouped according to how they are seen from the new vertex (e, in this example). The edges are labeled by a split label: w is a split label for the edges X,Y if $(X, w) \neq (Y, w)$. In this way, this tree works complementing the information about the children of the nodes in the decomposition tree. The final tree decomposition is shown in Figure 3.21.

The main difference with Algorithm 7 is that, instead of building this tree and doing the labeling process, our algorithm uses the Union-Find structure: each time a new clan is generated, its edges to the other clans (sigleton or not) are added according to their equivalence classes if they exist.

3.4 Implementation

This section is focused in the details of our implementation, `https://github.com/MelyPic/Gaifman-graphs_Mod`, not only about the decomposition algorithm but also about the construction of the standard Gaifman graph and its

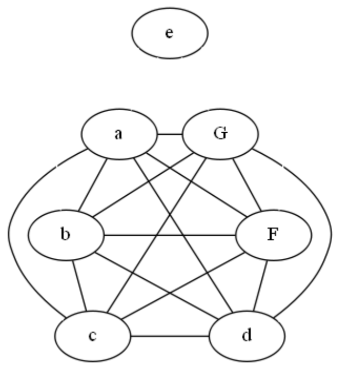


Figure 3.16: Graph of Example 3.8: 2-Structure to add node e.

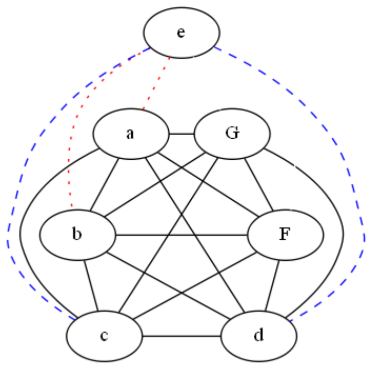


Figure 3.17: Graph of Example 3.8: How e is connected with the other nodes.

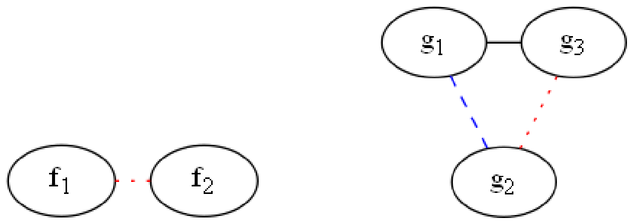


Figure 3.18: Graph of Example 3.8: F and G internal 2-Structure.

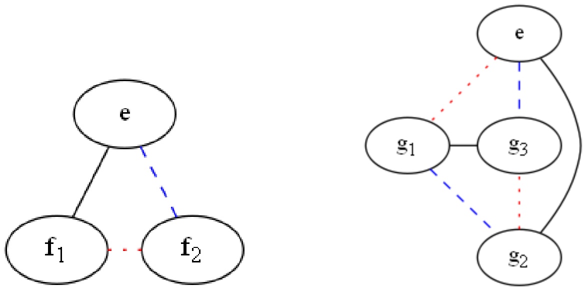


Figure 3.19: Graph of Example 3.8: How e sees F and G internal 2-structure.

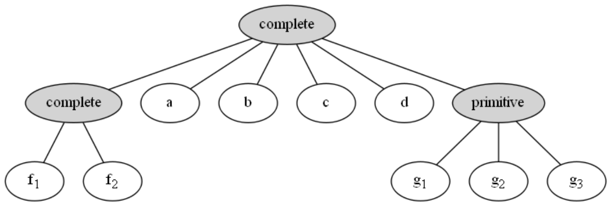


Figure 3.20: Initial tree decomposition and tree decomposition according e sees its nodes

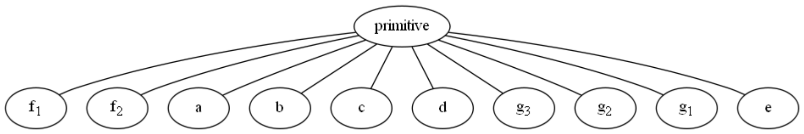


Figure 3.21: Final tree decomposition

variants, and about the creation of the file with the resulting decomposition. Each of these processes are explained separately.

One thing to consider is, like the co-occurrences of the attribute values into the dataset are record by these Gaifman graphs, in practice, as we said in Section 1.8, we get as many equivalence classes as different multiplicities leading us to many singleton clans on the decomposition since most vertices distinguish most others. Thus, to think on the discretization process to implement is important and needs not be always the same, it depends on the behavior of data. That is, we have to observe how different are the multiplicities on the co-occurrences.

In our implementation we use the NetworkX to get a known graph and GraphViz to draw the resulting decomposition.

3.4.1 Construction of the Gaifman graph from a dataset

The main purpose in this sections is to explain how we obtain the matrix that represents the desired Gaifman graph from the data in our implementation.

As first step, we select the source of the data. The data can be obtained from one table (one file), from a set of tables (several files) or from a NetworkX graph.

For any of the two first options, we ask for the type of the files since the presentation of the format is different according to the file type. Our project works with three types of files: *.csv*, *.arff* and *.txt*. But the reading file process must be treated with care since there are different configurations in the data. For example, there are cases where the attribute value `a` is taken as the same does not matter value of which attribute is, but there are other cases where it is important. Think in the same string `yes` as a value of columns named `homeowner` and `haschildren`, the same `yes` represents different information, by different values of the universe, the strings `haschildren:yes` and `homeowner:yes`.

Processing these data, in any of the possible cases, we generate the matrix M of the Gaifman graph that represents the information. In the initial version of the matrix, the value of the $M_{i,j}$, the position i, j into the matrix M, contains the quantity of times the attribute values that i and j represent appear together into the dataset, that is the number of co-occurrences for i and j. This matrix represents what we call *Current graph*, a 2-structure with as many equivalence classes as different co-occurrence values.

Thus, the user may select one of the following options to apply the decomposition algorithm on:

- The current Gaifman graph: the graph previously described.
- The standard Gaifman graph: A graph with just two equivalence classes, in this case the matrix has 1 in the position $M_{i,j}$ if $M_{i,j} \neq 0$ in the initial graph, representing i and j are connected, and 0 otherwise, representing they are disconnected.
- The thresholded Gaifman graph: We ask to the user for a lower threshold and upper threshold values, we will preserve from the original graph those co-occurrences values between the lower and the upper thresholds, the other values will be consider as disconnected. That is, assume a and b are the lower and upper threshold respectively, if $a \leq M_{i,j} \leq b$ then i and j are consider as connected items, that is, $M_{i,j} = 1$ for the matrix that represent the thresholded Gaifman graph. If the threshold values are not satisfied, i and j are consider as disconnected items, that is, $M_{i,j} = 0$.
- The linear Gaifman graph: This graph could have many equivalence classes defined on its edges. In order to specify the equivalence classes the user gives the interval size to divide the co-occurrences. You may find the formal definition in Section 1.7. Additionally, the user may give a lower threshold (all the values less than or equal to it would be disconnected) and an upper threshold (all the values grater than or equal to it would be disconnected too)
- The exponential Gaifman graph: It is a graph where the equivalence classes are determined by the logarithmic function of the matrix values, the formal definition is in Section 1.7. Also, it is possible for the user, to give a lower threshold value and an upper threshold value that works in the same way than in the linear Gaifman graph case.
- The shortest path Gaifman graph: It is a graph where the value of the element $M_{i,j}$ is the length of the shortest path that connects the node i with j in the standard Gaifman graph. Additionally, a threshold on the length of the path could be assigned, the paths out of the thresholds will be disconnected.

- The K-mean Gaifman graph: In this graph the number of equivalence classes is determined by the user by giving the number of clusters, the formal definition is in Section 1.7. The user also may give a lower threshold value and an upper threshold value, in this case, their value may change the position of the clusters.

Once the graph to work with is selected, we may apply on it the clan decomposition method.

3.4.2 Applying the clan decomposition method on a 2-structure

In this section we explain the main functions, classes and their use in the implementation of the decomposition method, the general algorithm was explained in Section 3.2.4.

In this part of the program there are defined two main classes: *Edge* and $MyClan$.

The objects of the class *Edge* represent the connection between two vertices, we may obtain the starting vertex, $EdgeFrom$, and the ending vertex, *EdgeTo*, these attributes are specified when a object *Edge* is initialized. Since we are working with undirected graphs for the edge (x, y) we have that x is an $EdgeFrom$ y and also x is an *EdgeTo* y.

To handle the class of equivalence to which each edge belongs we use the Union-Find data structure approach[4] ($Find$, $Union$ and $MakeSet$).

$EdgesNodes$ has *Edge* objects, on which $MakeSet$ operates directly, and thus $Find$ and $Union$ too. So, the edges in the same equivalence class will have the same father, that is the function $Find$ will return the same value.

Our decomposition is divided into clans represented by $MyClan$ objects, those clans could be singleton elements or clans by themselves. The internal members of a clan are named as its children, that is, we name as children of a clan those elements in its coarsest quotient.

At the end of the program we obtain a main clan object, the root of the tree, where all the internal nodes are clans and the leaves are singleton elements.

We may divide the class functions according to their tasks, some of them are focused in adding elements (*add_nodes_from*, *add_clan*, *add_node*), removing elements (*remove_nodes_from*, *remove_clan*, *remove_node*) and giving the

[4] `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`

elements in the coarsest quotient of a clan (*nodes*) or return a clan with a specific coarsest quotient if it exists (*getclanwithnodes*). Also we may access to the type of the clan by *clantype*.

Once a clan is generated or a new vertex is added to it we use the function *Pack* to get all the edges to it from the remaining vertices if they are and if it is possible to determine the edge. To pack the members of a clan we also use the Union-Find data structure, the algorithm is explained in Section 3.2.2.

Another fundamental function is call *SplitClan*. To implement this function we follow the algorithm in Section 3.2.3.

The main function of our algorithm is the function $AddNode$, as we may deduce by the name, is the function that add a node to a clan. This function is the implementation of the Algorithm 7.

Where the first step is to ask how the elements of the clan are seen from the new node, to do it we use the function $HowClansAreSeen$.

The function $HowClansAreSeen$ returns the list of nodes visibles with the color of the clan, the list of other visible nodes and the list of the non-visible nodes as they are defined in Section 3.2.4. To know what to do with this information, the type of the clan is required, we analyzed all the possible cases in Section 3.2.4.

Since the clans are packed as they are generated, to determine if a clan is seen by the new node or not is easy, we just need to search it in the edges and know its equivalence class by the function $Find$.

3.4.3 Visualizing the strong clan decomposition tree

The purpose of this section is the visualization of the main clan obtained previously, being this clan the decomposition tree. We employ the tool Graphviz to do this.

The main task is deal with how decide when the elements will be visualized. The quantity of elements could be so large, drawing all of them may result in an not understandable tree. So, as a first approach we decided to create a node (*Others*) to group them.

In the case of singletons elements, when all the elements that we grouped in this node are singletons, it is sufficient to fix a number of elements that we want to see to obtain an understandable and informative image. It is different if there are involved elements that are clans because we may lose important information inside of them, it is not easy to determine how many of the elements we want to see. The process to decide the optimal number of items to be shown depends on many factors and it is not easy to specify.

The *Others* node is also used to represent isolated vertices in the corresponding Gaifman graph. Those are vertices that have no connections, neither among them, nor with other vertices. So, they are grouped in a single vertex labeled as *Others*.

Leaving that aside, the following are some important functions for the visualization process:

- *DecomposeClan* has a clan as input and the output are two lists, one with all single nodes in the clan and the other one with the clans in its coarsest quotient.

- *FindClans* also has a clan as input and returns a list of all its descendant clans in a clan, doing this in a depth-first search. Not only the clans in its coarsest quotient but also the clans in the coarsest quotient of each respective clan, and so on.

- *GiveName* recibes a list of the elements in the clan and returns a string with the attribute names corresponding to all clan members.

- *MakeCluster* generates an element cluster in the *.dot* file. Use the function *GiveName* to assign the name of cluster and the function *DecomposeClan* to know its members. In this section we decide how many singleton and clans elements will be consider as too many to draw them one by one, that is, here we determine the use of the *Others* node.

The program goes through the internal clans and draws for each one a subgraph cluster according to the specific constrains, calling the previously described functions as they are necessary. Finally, the program returns the *.png* of the *.dot* generated.

As we have seen, the decomposition results will depend on the chosen Gaifman graph variant, along with the assigned parameters and thresholds, which will make the edges belong to an equivalence class or another. Therefore, in the final visualization of the decomposition, we will be able to see the equivalence classes of the edges, but not the amount of co-occurrences between one item and another. However, we will have access to this information through the matrix of the initial Gaifman graph.

Chapter 4

Exploring data through graph decompositions

4.1 Introduction

In this chapter and the next, we show some applications of the decomposition methods and their interpretation.

Before to go on it is important to point out, as we referred previously, that to work with many attribute values may cause to have clans, into the decompositions, with many single items, leading to large primitive clans, whose mathematical study gets pretty complicated [15], and thence to diagrams that one can not understand, like in Figure 4.1. To handle it, we implement several measures like the *frequency thresholds*; being the same attitude followed in frequent set mining [25] [38].

Nonetheless, there are cases where this problem persists thus, in our diagrams clans containing more than handful of nontrivial clans are not drawn in detail. Besides, if there are few nontrivial clans, but many trivial ones, then the trivial clans are grouped in a single node labeled as *Others*, sort of a merge of them all, like in Figure 4.8.

As we move on, one case turns out to be common in our experiments: whereas Gaifman graphs do not have isolated vertices, in our generalizations this is no longer true. Many datasets will lead to isolated vertices in the corresponding Gaifman graph. The set of those isolated vertices forms a quite large clan that clutters the diagram but contributes nothing to the analysis beyond all these vertices are actually isolated. We use again the label *Others*

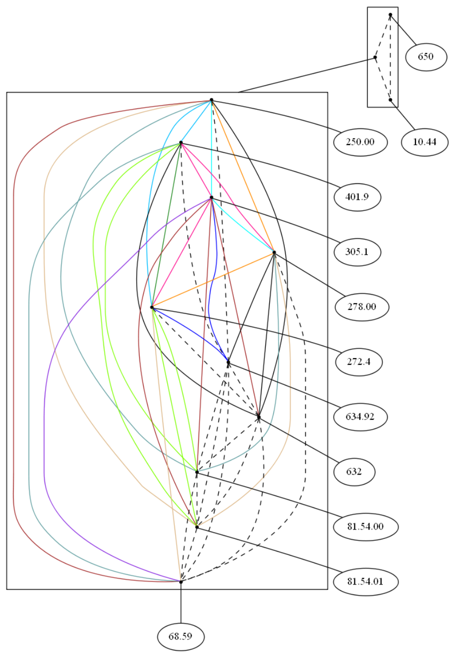


Figure 4.1: Example: “not understandable” decomposition.

to represent these items, all alike from the decomposition perspective, as a single vertex. One example of this measure is the Gaifman graph shown in the left of Figure 4.5.

Once all these details has been mentioned, let us go to the chapter structure. At first, in Section 4.2 we talk about some characteristics on the data information to be consider on the Gaifman graph construction from a dataset.

In Section 4.3 we work with standard Gaifman graph versions and with the thresholded variant of the Gaifman graphs, in those cases we apply modular decomposition since we only have two equivalence classes: connected and disconnected items, either because the items do not co-occur together or because their multiplicity is below a determined co-occurrence threshold. We also show the use of the frequency threshold, previously mentioned, whose objective is to reduce the number of vertices in the Gaifman graph.

We extend the capabilities of our approach by generalizing the notion of Gaifman graph. Hence, we develop our work using the more general decomposition method called the 2-structures decomposition method. In Section 4.4 we work with linear, exponential and shortest path Gaifman graph variants applying them on different datasets with different threshold values.

4.2 From Datasets to Gaifman graphs

Often, in practice, each column of a table has its own semantics and, even if a data value seems to superficially coincide in occurrences at different columns, it might mean different things. There are times that we must to consider them as different values, if domain information was available advising data pieces to be considered different. Besides, there are other times where it is better to take these values as one and the same, that is, regardless of the attribute that they represent. As we said in Section 3.4.1, we assume that a preprocessing has been run looking for the optimal option.

4.3 Modular decomposition

We move on to explain some examples of our analysis strategy based on modular decomposition.

One of the datasets used to apply our decomposition method is the Zoo dataset, a dataset from the UC Irvine repository [12]; it contains 17 attributes, with 39 possible attribute values, of 100 animal species. We have preprocessed it slightly so that the semantics of each item is clearly identifiable (e. g. `predator_False` or `toothed_True`).

The decomposition of the Zoo dataset standard Gaifman graph is shown in Figure 4.2. The topmost node of this decomposition is, the trivial clan with the whole universe; in this case, it turns out to decompose as a primitive clan whose coarsest quotient graph collapses many trivial clans, that we choose not to draw complete; however, one nontrivial clan also appears: `mammal` and `milk_True` are indistinguishable from the perspective of all the other elements in the dataset. That is, for every other piece of information, either it goes together with each in some tuples (one such item could be `hair_True`), or it does not go together with any of them ever (for instance: `feathers_True`). In our diagrams, as we already said, clans containing more than a handful of nontrivial clans are not drawn in detail: just the clan type label (primitive or complete) is shown.

We describe next some outcome of analysis on the very well-known dataset Mushroom, also known as "Agaricus-Lepiota". In this dataset a number of

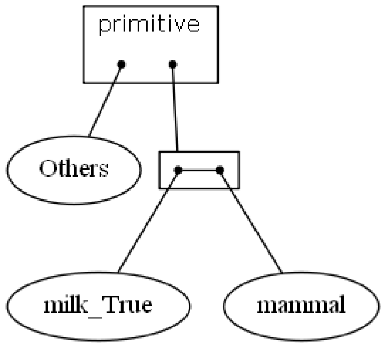


Figure 4.2: Modular decomposition: Zoo dataset graph.

purported attributes of potential mushrooms of these families are expected to be useful to predict whether each of the 8124 observations would correspond to an edible or a poisonous mushrooms [14].

In this case, we restrict ourselves to items appearing at least 2000 times, that is we give 2000 as *frequency threshold*. Even then, we do not display the complete decomposition tree, we just show a module that deserves to be discussed. First, of course a number of *false twins*[1] appear such as `grass-living` versus `wood-living` mushrooms. Again a P_4 case is shown in Figure 4.3, if a mushroom has foul odor, then it is a poisonous mushroom, as foul odor never appears in the same transaction with edible.

Figure 4.4 is a little bit more difficult to interpret but we find there, for example, that bruised mushrooms have smooth stalk surfaces, both below and above, but never exhibit silky surfaces either below or above, and not all smooth stalk surfaces, either below or above, are bruised, since both are also connected to the no-bruises vertex. The induced P_4 predicted by the theory is indeed there as well.

We also impose a threshold on the co-occurrence of items. It is important to point out the difference between *co-occurrence threshold* and *frequency threshold*: in the first one we work with all the possible attribute values but those edges labeled below than a determined co-occurrence threshold are consider as disconnected since in the second case we construct the Gaifman graph just with those attribute values that appear more than a determined frequency threshold. We will see more examples later on.

[1]Vertices with the same set of neighbors are commonly called "twins" in graph theory: "true twins" if they are connected and "false twins" otherwise

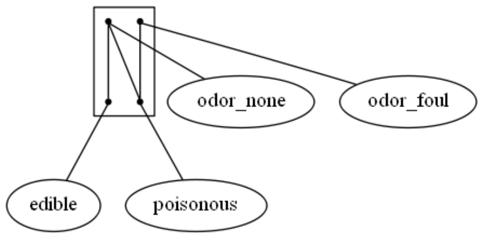
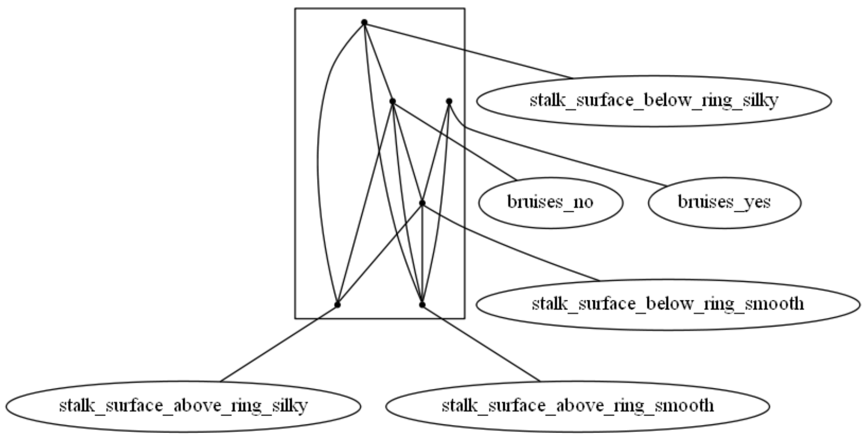


Figure 4.3: Modular decomposition: A non-trivial module in the Mushroom dataset graph.



stalk_surface_below_ring_silky
bruises_no
bruises_yes
stalk_surface_below_ring_smooth
stalk_surface_above_ring_silky
stalk_surface_above_ring_smooth

Figure 4.4: Modular decomposition: A second non-trivial module in the Mushroom dataset graph.

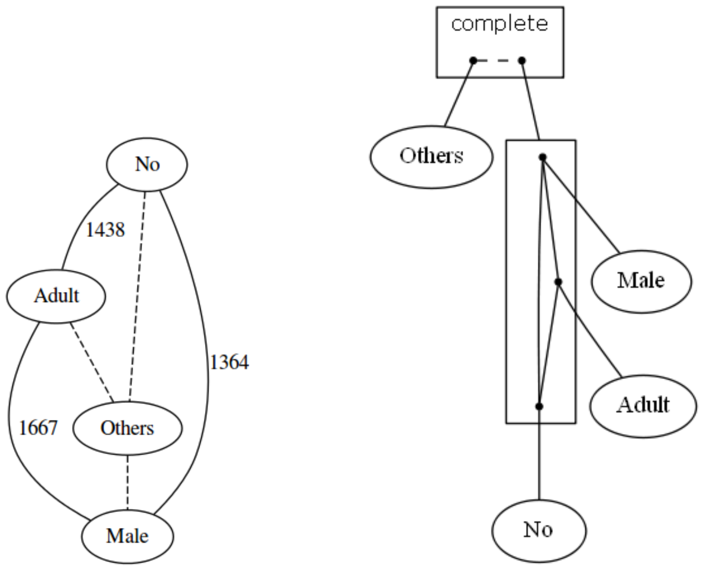


Figure 4.5: Modular decomposition: Titanic with 1000 as co-occurrence threshold graph.

If we restrict the Titanic Gaifman graph of Section 4.3 with 1000 as *co-occurrence threshold*, we obtain the left of Figure 4.5 where the item *Others* involves all the remaining vertices. Its modular decomposition is shown in the right of Figure 4.5. The resulting graph supports the saying "women and children first" since show us than most of the non-survivors were men.

Thus, to obtain these results is sufficient to work with the decomposition method, but as we said, there are cases where this method is not enough. In the following sections we will see more results where we apply the 2-structure decomposing method properly.

4.4 Clan decomposition

As we saw earlier, part of our contribution is working with variations of Gaifman graphs which are the result of applying, on the multiplicity of their edges, discretization methods, resulting in a 2-structure. These discretizations are explained in detail in Section 1.8.

In the following sections we will show and explain some examples of the decomposition of discretized Gaifman graph.

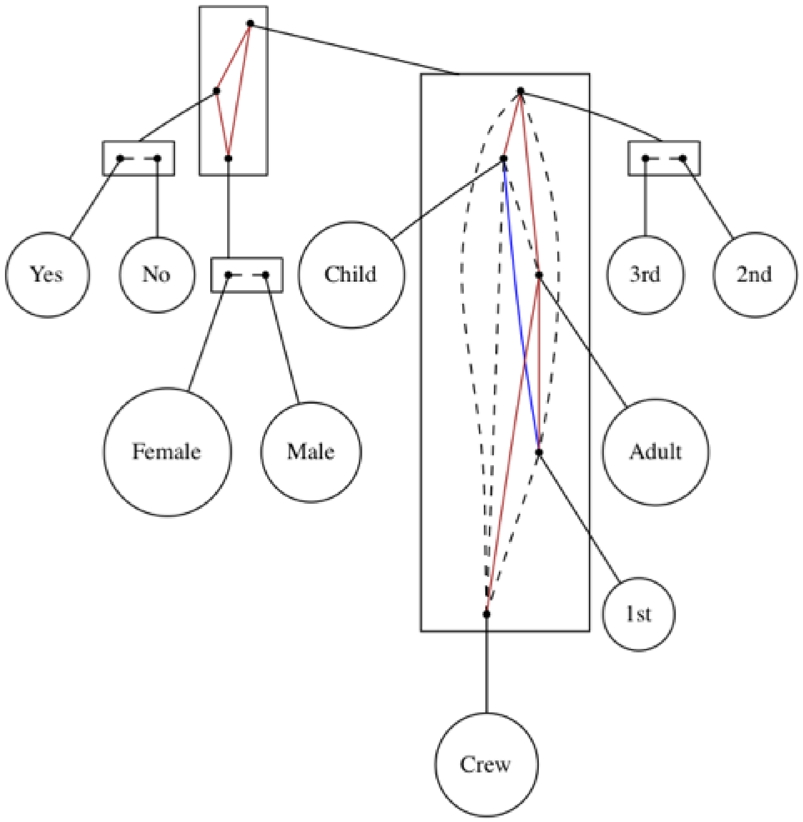


Figure 4.6: Clan decomposition: Titanic linear threshold Gaifman graph.

4.4.1 Decomposition of linear Gaifman graph

In Figure 4.6 we have the result of applying the clan decomposition method on the linear threshold Gaifman graph for the Titanic dataset. We differentiate a few multiplicity values with different colors plus an upper threshold. The upper co-occurrence threshold given was 20, all those values greater than or equal to this threshold are in the same equivalence class. We may deduce, by the resulting graph, that there were few children in the first class, there were found six co-occurrences of these values. In this case the linear interval size is not really important since we just have one edge whose multiplicity is between 0 and the assigned upper threshold value.

In Figure 4.7 we find the result of applying the clan decomposition on the Zoo dataset of the linear Gaifman graph with 10 as interval size. As we

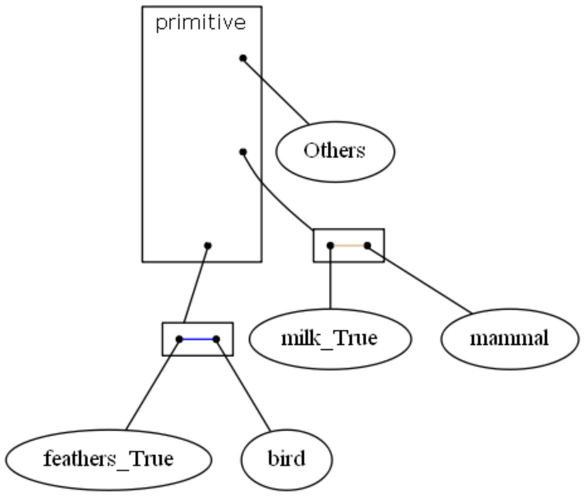


Figure 4.7: Clan decomposition: Zoo linear Gaifman graph.

already said, the dataset has 17 attributes and a total of 42 attribute values of about 100 animal species. In the decomposition we find two clans which tell us that mammals drink milk and birds have feathers. The remaining items are all involve in the *Others* node.

The Figure 4.8 shows the clan decomposition of the linear Gaifman graph from the Zoo dataset, where the interval size is 25. Since the maximum multiplicity of the edges in the Gaifman graph of Zoo is 81, we obtain four equivalence classes, but we do not see all of them: this is due simply to our visualization limits. Here we have three interesting clans: the fact that an animal is mammal (`milk_true`) has a strong co-occurrence with that it does not lay eggs and has hair. Following this, those animals that do not lay eggs are not mammals. And also there is a strong co-occurrence between those animals that are mammals and have four legs.

We also work with the decomposition of the Votes dataset [12] 2-structure, that contains information about the votes for each of the U.S. House if Representatives Congress-men on the 16 key votes. Figure 4.9 is the result of applying the decomposition method on the linear Gaifman graph with 100 as interval size of those values that appear more than 100 times, that is the frequency threshold is 100. We find a clan conformed by republicans and the negative of adoption of the budget resolution, since they are around 141 republican of 168 votes against adoption of budget resolution.

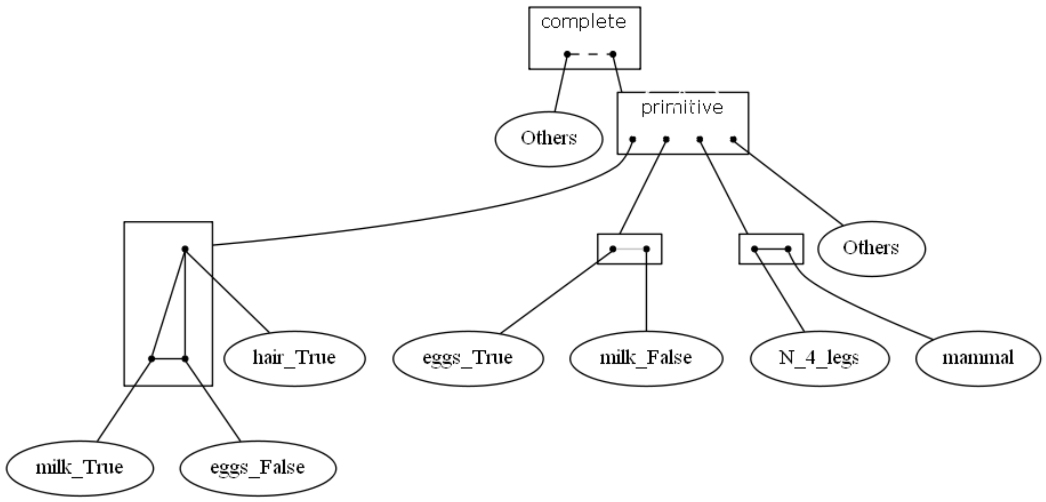


Figure 4.8: Clan decomposition: Zoo linear Gaifman graph.

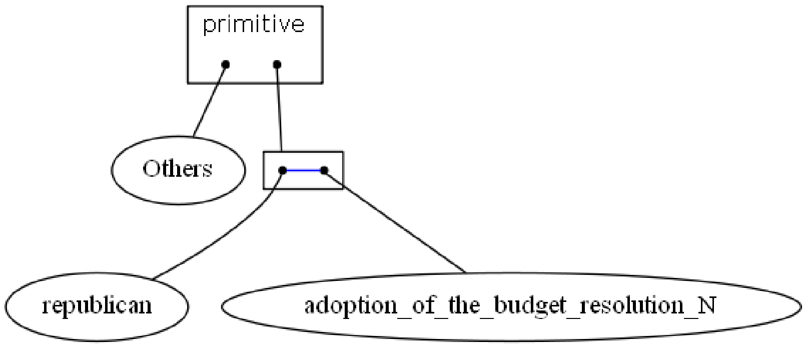


Figure 4.9: Clan decomposition: Votes linear Gaifman graph.

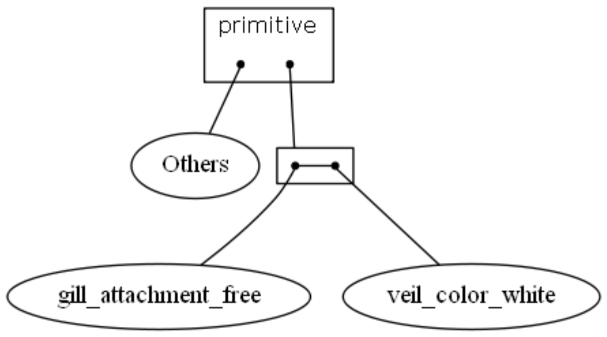


Figure 4.10: Clan decomposition: Mushroom exponential Gaifman graph.

4.4.2 Decomposition of exponential Gaifman graph

Comparing the exponential variant with the linear variant of the Gaifman graph we have on one hand that, the exponential variant frees the user from having to bet on a specific interval width; on the other, there could be cases where some information may be lost when we work with large intervals.

In Figure 4.10 we find the result of applying the clan decomposition method on the 2-structures described by the exponential Gaifman graph of the Mushroom dataset with 2000 as frequency threshold, that is taking just those attributes appearing more than 2000 times. As we can see most of the attribute values co-occur with each other in different ways excepting the items `gill_attachement_free` and `veil_color_white` that have around the same quantity of co-occurrences with the other items, and also, we find that they co-occur very often, around 7900 times. Considering there are a total of 8000 rows, we may say that they co-occur almost always, thus most on the mushrooms into the dataset do not have gills and have a white veil.

In order to illustrate a case working with multiple tables we work with UW-CSE dataset from the relational dataset repository `http://relational.fit.cvut.cz`. Figure 4.11 was obtained applying a co-occurrence threshold value of 30: multiplicities below this value remain as non-edges (broken lines of the completion), and isolated vertices under this condition are removed and jointly replaced by the *Others* vertex. Indeed, the line in the topmost rectangle is a broken line which means that the items grouped in the node *Others* co-occur among them less than 30 times. In the decomposition, Figure 4.11, we find the amount of students that are not professors is almost same than the amount of persons not registered in any phase and who have

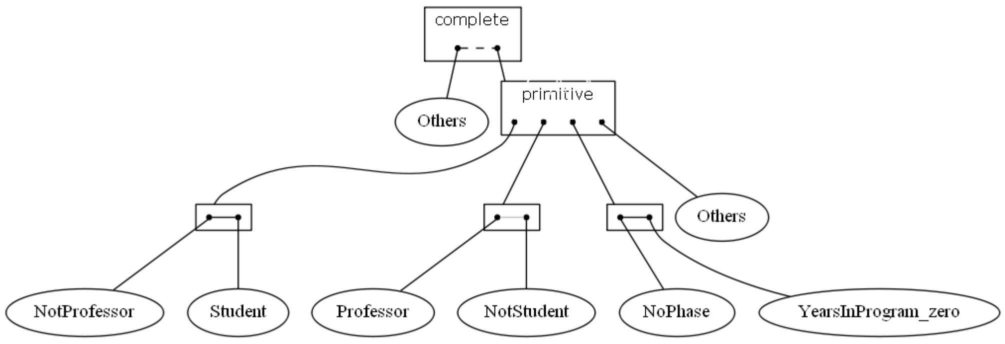


Figure 4.11: Clan decomposition: UW-CSE exponential Gaifman graph.

zero years in the program, while the amount of professors, of course not students, is in a different rank.

4.4.3 Decomposition of shortest path Gaifman graph

We propose to explore the shortest path variant of the Gaifman graph. The decomposition of this version provides us with a complementary information about the data behavior. For example, the Figure 4.12 shows the decomposition of the shortest path variant for the linear Gaifman graph Zoo dataset whose decomposition is shown in Figure 4.7. Despite generating the same clans, we find different equivalence classes: in the linear variant decomposition the edge that connects `feathers_True` with `bird` is in a different equivalence class than the edge that connects `milk_True` with `mammal`, while in the shortest path variant decomposition both are in the same equivalence class since they are directly related.

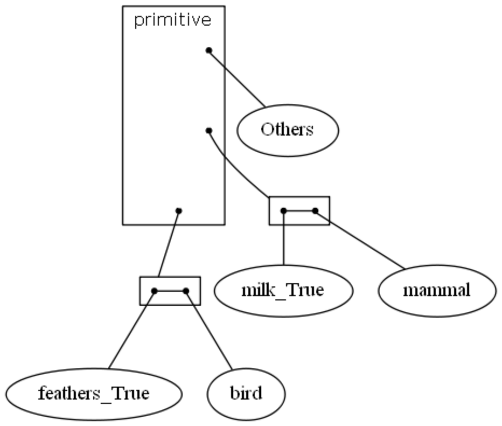


Figure 4.12: Clan decomposition: Zoo shortest path Gaifman graph.

Chapter 5

Analyzing medical diagnostic data

5.1 Introduction

In the next analysis we work with a medical dataset. This dataset was provided to us by the Hospital de la Santa Creu i Sant Pau, under a collaboration agreement: between that institution and UPC. This is a public hospital located in Barcelona, with about 430K visits and 40K admissions per year. The medical dataset contains information on all urgent hospitalizations for the years 2015-2016. It is in a transactional format that consists of a sequence of (Excel-like) rows corresponding to patients and, in each, there are, organized in columns, diagnostics, treatments and patient conditions encoded in ICD-9-CM[1]. Additional information as gender, provenance, etc., is also in the data but we do not take this information into account, we only work with diagnostics, treatments and patient conditions, working with them in different phases.

The dataset contains around 80000 rows with information about 80000 anonymous patients. The total number of potential diagnostics is 5637, growing to 7741 if treatments are also considered and to 8250 if patient conditions are considered too. Thus, considering the set as relational would result in a huge dimensionality, with vast amounts of zeros. Hence, we chose to see our dataset as transactional, each row consisting of a set of diagnostics, treatments, and/or conditions; we will describe all the particularities for each

[1] `https://www.cdc.gov/nchs/icd/icd9cm.htm`

application. Most of these results were presented and published in [6].

5.2 Modular decomposition

As first example of this dataset analysis we will work only with diagnostic values, we follow the same attitude as in previous examples: we set a minimum frequency threshold, so that diagnostics appearing less often than the threshold are not taken into account for that visualization.

Thus, we impose 100 as frequency threshold getting the follow diagnostics as vertices of our initial graph:

- 650 `Normal delivery`,
- 632 `Missed abortion`,
- 305.1 `Tabacco use disorder`,
- 401.9 `Unspecified essential hypertension`,
- 272.4 `Other and unspecified hyperlipidemia`.

The decomposition obtained is shown in Figure 5.1. At the top of the figure we find that the item 650 `Normal delivery` does not appear together with any of the other diagnostics, since 650 is connected by a broken line with all the remain items. Inside the large box, we find co-occurrences between all the remaining items excepting 272.4 and 632. To have a better understanding of this dataset in the next section we apply the clan decomposition on the linear Gaifman graph variant of this graph, and we find interesting results.

5.3 Linear Gaifman graph decomposition

On the hospitalization dataset we applied our clan decomposition method on different linear Gaifman graphs.

At first we work just with the diagnostics using a frequency threshold of 100 as in the example of Figure 5.1, thus we work with the same items.

Additionally, we use a co-occurrence threshold of 8, and 1000 as interval size to get the linear Gaifman graph variant. The result of applying the clan decomposition method on the described 2-structure is shown in Figure 5.2. At the top of the figure we find that the item 650 `Normal delivery`

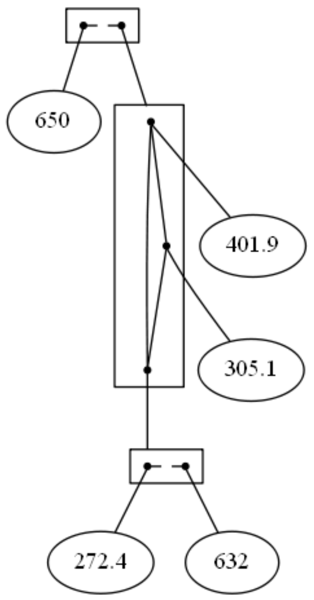


Figure 5.1: Modular decomposition: Diagnostics appearing more than 100 times (frequency threshold).

does not appear together with any of the other diagnostics. The large box represents together all the remaining items, we see no co-occurrences between 632 `Missed abortion` and the clan conform by the hypertension items and few co-occurrences with 305.1 `Tabacco use disorder`, around 12 co-occurrences, while the item 305.1 `Tabacco use disorder` co-occurs around 2000 times with the members of the hypertension clan. In the hypertension clan, we find the diagnostics 272.4 `Other and unspecified hyperlipidemia` and 401.9 `Unspecified essential hypertension` linked by a high-frequency co-occurrences, checking that out, we find over 11000 cases of co-occurrences.

In the next example we continue working with diagnostics, in this case we take those diagnostics that appear more than 80 times, having as new nodes:

- 250.00 `Diabetes mellitus without of complication, type II or unspecified type, not stated as uncontrolled`,
- 278.00 `Obesity unspecified`,

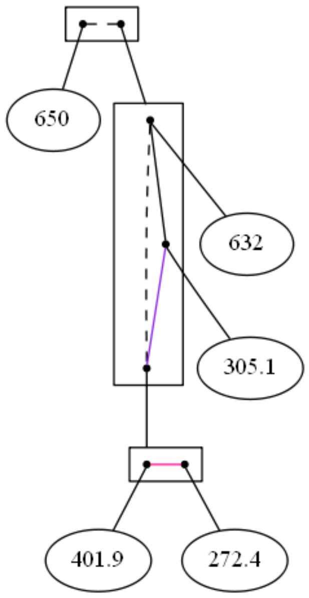


Figure 5.2: Clan decomposition: Diagnostics, linear Gaifman graph.

- 634.92 `Spontaneous abortion complete without complication`.

In Figure 5.3 we have at the top the item 650 `Normal delivery` disconnected with all of the other items, we find a clan conformed by the items 305.1 `Tobacco use disorder` and 278.00 `Obesity unspecified`. Both of them co-occur around 500 times. The item 250 `Diabetes mellitus` and the members of this clan co-occur more than 1000 times but less than 2000 times, while 401.9 `Unspecified essential hypertension` and 272.4 `Other and unspecified hyperlipidemia` co-occur with the same clan more than 2000 times; and the clan co-occurs just few times with the items 634.92 `Spontaneous abortion complete without complication` and 632 `Missed abortion`.

In the next example we show the decomposition on the linear Gaifman graph of those diagnostics, treatments, patients conditions that appear more than 250 times, with an interval size of 1000, and 500 as co-occurrence threshold value. The items involved are:

- 632 `Missed abortion`,
- 272.4 `Other and unspecified hyperlipidemia`,

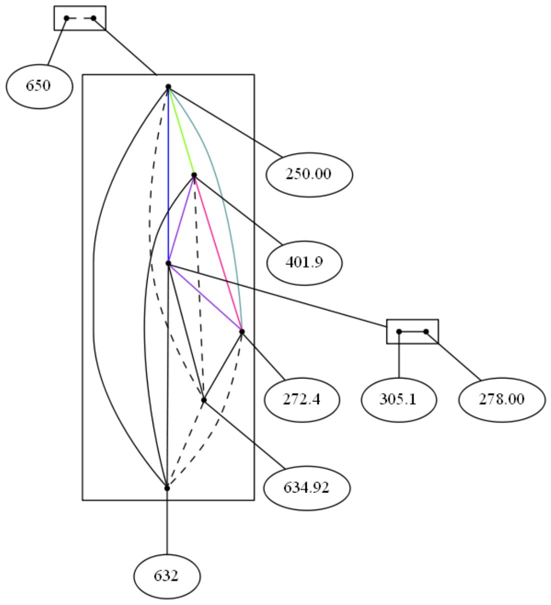


Figure 5.3: Clan decomposition: Diagnostics, linear Gaifman graph.

- V15.82 `Personal history of tobacco use`,
- V58.66 `Long-term (current) use of aspirin`,
- V58.61 `Long-term (current) use of anticoagulants`.

We have Figure 5.4 as the resulting figure. Here we find that 632 `Missed abortion` has not co-occurrences with any of remaining nodes. And we find a clan, conformed by V58.66 `Long-term (current) use of aspirin` and V58.61 `Long-term (current) use of anticoagulants`, they are connected by a broken line since they co-occur around 300 times, that is they co-occur less than the co-occurrence threshold (500), not often enough according to it.

We also see that this clan, the anticoagulant clan items, co-occurs more frequently with 272.4 `Other and unspecified hyperlipidemia` than with V15.82 `Personal history of tobacco use`. And the items that co-occur more frequently than any others are 272.4 and V15.82. That is, the number of people who have hyperlipidemia and take anticoagulants is greater than people who have a history of smoking ad take anticoagulants; while, it is more frequent to have cases with history of tobacco use and having hyperlipidemia.

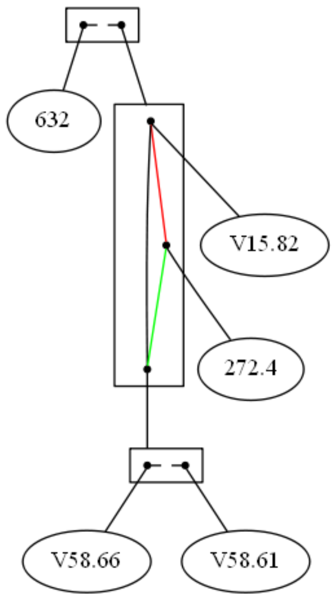


Figure 5.4: Clan decomposition: Diagnostics, treatments and patient conditions, linear Gaifman graph.

5.4 Exponential Gaifman graph decomposition

Let us analyze those diagnostics with a frequency threshold of 100 as in the first examples for the previous sections. In Figure 5.5 we find the decomposition of the diagnostic exponential Gaifman graph. In the top node we see that 650 `Normal delivery` has not co-occurrences with any of the other nodes. Also we find a clan conformed by the item 272.4 `Other and unspecified hyperlipidemia` and the item 401.9 `Unspecified essential hypertension` diagnostics, that is, those diagnostics have the same behavior with the rest of the diagnostics: they do not have co-occurrences with 632 `Missed abortion` diagnostic and they have around two thousand co-occurrences with 305.1 `Tobacco use disorder`. On the other hand, `Tobacco use disorder` co-occurs with `Missed abortion` around ten times that compared with the 80000 possible cases is too few.

Considering diagnostics and treatments we have 7741 different values thus we give a frequency threshold to reduce them. The Figure 5.6 shows the decomposition of the exponential Gaifman graph of those items that appear

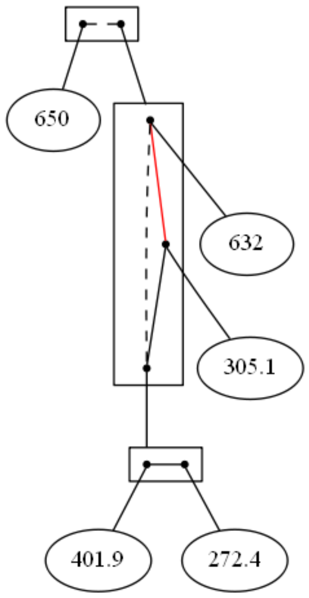


Figure 5.5: Clan decomposition: Diagnostics, exponential Gaifman graph.

more than 100 times. We see two clans, one of them conformed by the diagnostics items: 272.4 `Other and unspecified hyperlipidemia` and 401.9 `Unspecified essential hypertension` linked by a high-frequency joint occurrence (checking that out, we find over 11000 cases), we will call it hypertension diagnostic clan, we get this clan also in the previous example. And the other one, a completely new clan, in the bottom with two similar codes, 81.54. . . , with a broken line connecting them that indicates that they are incompatible: they are both Total knee replacement, each referring to one laterality option, and, of course, it is hardly ever the case that both knees are replaced at once, we will refer to it as knee replacement procedure.

The larger box represents together all the remaining items except one. At the top of the figure we read that 650 `Normal delivery` does not appear together with any of the other items represented in the figure, because 650 is connected with a broken line to the box corresponding to the remaining diagnostics and treatments; whereas, inside the larger box, we see no joint occurrences (that is, broken lines) from 632 `Missed abortion` to the hypertension diagnostic clan and to the knee replacement procedure clan but it is connected with diagnostic 305.1 `Tobacco use disorder` showing up that they co-occur a not too significant number of times (about a dozen

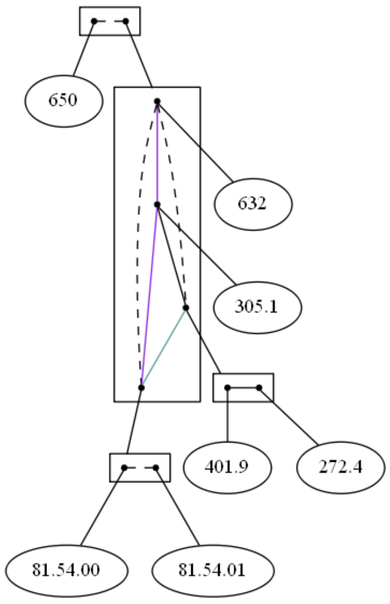


Figure 5.6: Clan decomposition: Diagnostics and treatments, exponential Gaifman graph.

times each), same as 305.1 `Tobacco use disorder` co-occurs with the knee replacement procedure clan. While, 305.1 `Tobacco use disorder` is significantly connected (in the range of 2000 occurrences) with both nodes of the hypertension diagnostic clan.

And, finally, the knee replacement procedure clan and the hypertension diagnostic clan appear jointly around 100 times.

5.5 Shortest path Gaifman graph decomposition

In the next examples we get the decomposition of the shortest path version of some clan decomposition already calculated, all of them are applied on the medical dataset. We presented those results in [34]. As a general result, we find coincidentally three equivalence classes in the 2-structures drawing by broken lines (to link items whose shortest path is equal to zero), solid lines (to link items whose shortest path is equal to one) and dotted lines (to link items whose shortest path is equal to two). First, in Figure 5.7, we have the decomposition of the shortest path version of those diagnostics that appear more than 80 times, the decomposition of its linear variant is in Figure 5.3.

In this current decomposition, we find again the item 650 `Normal delivery` disconnected to the remaining items. The items 305.1 `Tobacco use disorder` and 278.00 `Obesity unspecified` connected to a clan with all the remaining items conform a clan, since they co-occur at least one time. While inside of the clan that contains the remaining items, we find different equivalence classes that is because some of the items are not directly related. Also, inside of this box we find a clan conform by the items 401.9 `Unspecified essential hypertension` and 250.00 `Diabetes mellitus`, they are in the same clan because both of them co-occur or do not co-occur with the same items; and also we may see that they appear together since they are directly connected. We may think that the items 272.4 `Other and unspecified hyperlipidemia` and 401.9 `Unspecified essential hypertension` could be in a clan but they are not because there is a case where the items 401.9 `Unspecified essential hypertension` and 632 `Missed abortion` co-occur but missed abortion does not have any co-occurrence with 272.4 `Other...hyperlipidemia`.

The Figure 5.8 shows the decomposition of the shortest path version of the linear Gaifman graph decomposed in Figure 5.4. Therefore, we work with the diagnostics, treatments and patient condition with 250 as frequency threshold and 500 as co-occurrence threshold. As we can see the resulting figure is quite similar to Figure 5.4, excepting for the type of the edges, that is, excepting for the equivalence relation to which they belong. At the top of the figure we reaffirm that the item 632 `Missed abortion` is not connected with any other item, will the items into the large box are directly connected, they used co-occur together. Finally, the items V58.66 `Long-term (current) use of aspirin` and V58.61 `Long-term (current) use of anticoagulants` are connected by a dotted line because the shortest path to go from one to another has two as length.

The application of the shortest path version of the decomposition on the diagnostics and treatments Gaifman graph that appear more than 100 times, allows us a better understanding of the behavior of the items within the larger box in the Figure 5.6. The resulting figure of this analysis is show in Figure 5.9, we can verify that the node 650 `Normal delivery` is not connected with any other node. The node 305.1 `Tobacco use disorder` is directly related, with different co-occurrences values, within all the remaining nodes.

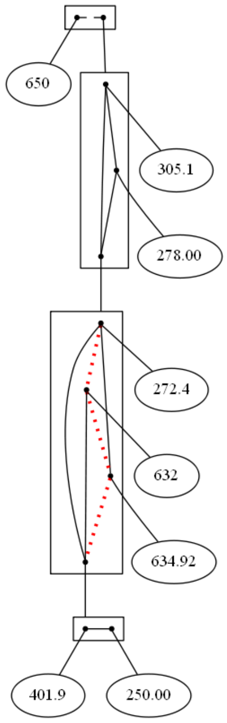


Figure 5.7: Clan decomposition: Diagnostics appearing at least 80 times, shortest path Gaifman graph.

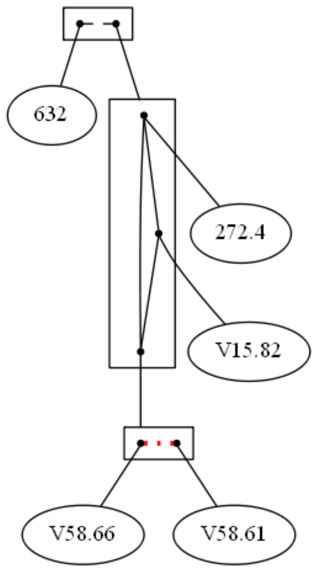


Figure 5.8: Clan decomposition: Diagnostics, treatments and patient conditions appearing at least 250 times, shortest path Gaifman graph.

While 632 `Missed abortion` is not directly related with hypertension diagnostic clan items and knee replacement clan, there could be cases where the patient medical conditions consist on 632 `Missed abortion`, 305.1 `Tobacco use disorder` and some of the items into the large box clan. Again, with different multiplicities of co-occurrences, the members of the larger box clan are directly correlated. This shows how the hypertension diagnostic clan items are connected to each other and with both cases of knee replacement. While, to have both knees replaced, does not happen.

5.6 K-means Gaifman graph decomposition

In this section we analyse the decomposition of the K-means Gaifman graph variant. The Figure 5.10 shows the result of applying the clan decomposition method on the Gaifman graph of those diagnostics, treatments and patient conditions with 250 as frequency threshold applying the K-means method with 2 clusters, while Figure 5.11 shows decomposition of the same Gaifman graph but applying the K-means method with 3 clusters.

In both cases, we have 632 `Missed abortion` as isolated item since it has

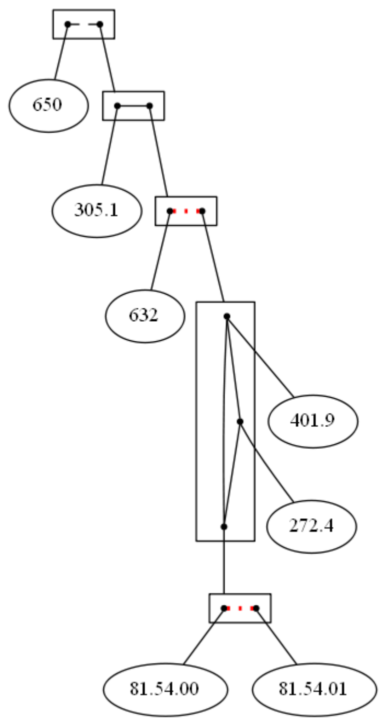


Figure 5.9: Clan decomposition: Diagnostics and treatments appearing at least 100 times, shortest path Gaifman graph.

not co-occurrences with any of remaining items.

Despite the fact that in these cases we do not use co-occurrence threshold values, we can observe similar behaviors in the data since the items V58.66 `Long-term (current) use of aspirin` and V58.61 `Long-term (current) use of anticoagulants` also are in the same clan.

In fact, when we work with three clusters they conform a clan, bottom of Figure 5.11, in this decomposition we may see in a more clear way the differences on the co-occurrences. As we described previously, the item 272.4 `Other and unspecified hyperlipidemia` co-occurs more frequently with V15.82 `Personal history of tobacco use` and with the anticoagulant clan than V15.82 `Personal history of tobacco use` co-occurs with the anticoagulant clan.

As we said in Section 1.8, adding thresholds to a K-means Gaifman graph can cause edges to belong to different equivalence classes compare with the not threshold K-means Gaifman graph version. For example, if we give 1500 as lower threshold value to the K-means Gaifman graph variant with 3 clusters, the edges that connect V15.82 and 272.4 with the anticoagulant clan are modified getting the decomposition shown in Figure 5.12.

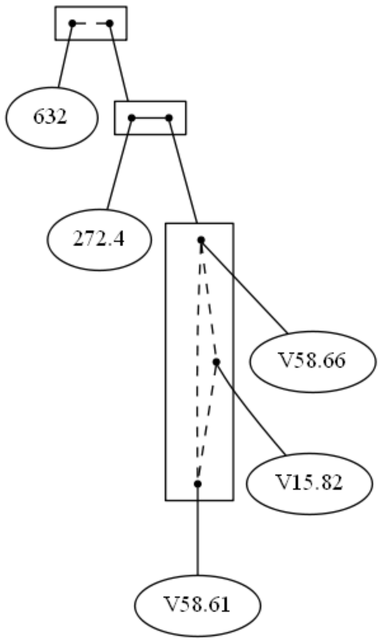


Figure 5.10: Clan decomposition: Diagnostics, treatment and patient condition appearing at least 250 times, K-means Gaifman graph, 2 clusters.

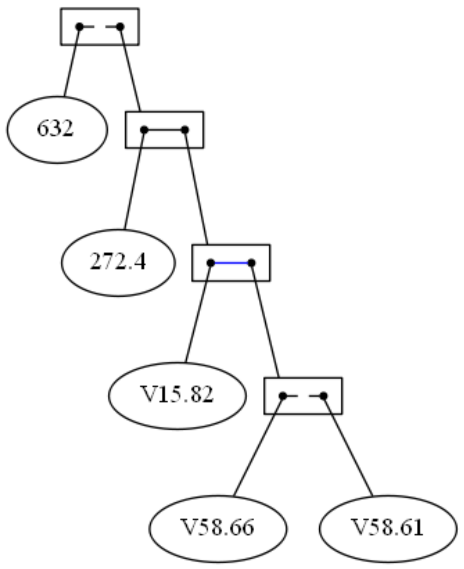


Figure 5.11: Clan decomposition: Diagnostic, treatment and patient condition appearing at least 250 times, K-means Gaifman graph, 3 clusters.

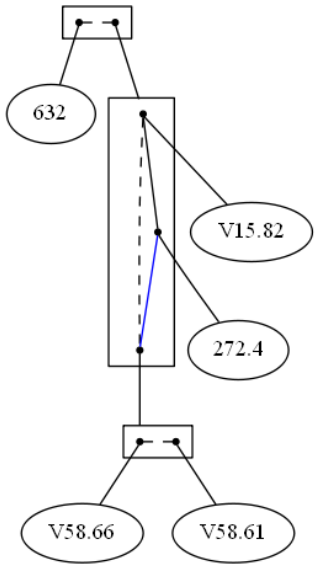


Figure 5.12: Clan decomposition: Diagnostic, treatment and patient condition appearing at least 250 times K-means Gaifman graph, 3 clusters with 1500 as lower threshold.

Chapter 6

Conclusions

6.1 Discussion and perspectives

In this dissertation, the major topic is the study of how to provide, in an unsupervised way, hierarchical visualizations of the co-occurrence patterns on data. It could be applied to work as an additional or helping tool to other data analysis tools, since to have a general view of the data behavior is so useful in many areas.

First, along this thesis, we have seen that the known notion of modular decomposition of a graph can be understood, in a quite natural way from a perspective of data analysis, as a variant of closure space visualization; then, that this process can be applied to datasets via a known logical construction, namely the Gaifman graph; and, also, that both the theoretical connection and the practical applicability of the decomposition process can be generalized to quantitatively enabled variants of Gaifman graphs through a known generalization of modular decompositions, namely the clan decomposition of 2-structures. In the case of (natural completions of) standard Gaifman graphs, it is well-known that the given decomposition via 2-structures coincides with the classical modular decomposition of undirected graphs [30]. Indeed, in this case, the modules of the graph are exactly the clans of its natural completion, seen as a 2-structure with just two equivalence classes, strong clans are exactly so-called strong modules, and the resulting decomposition tree is essentially the same. The other variations of Gaifman graph proposed here were thresholded, linear, exponential, shortest path and K-means-based Gaifman graphs.

It is important to point-out that the use of 2-structure clan decomposition on data analysis is novel, and of course, it is inspired on theory developed many years ago.

We have included a number of developments that relate the taxonomy of modules to the corresponding local structure of the closure space, since it was necessary to develop a precise understanding of exactly what was the set of co-occurrence patterns we were displaying. Thus, we developed the underlying theory, oriented both towards better understanding and also towards the algorithmic point of view. We identified a closure space related to so-called "modular implications" in the particular case of standard Gaifman graphs and to "clan implications" for the more general cases, and characterized modules and clans in terms of subfamilies of the corresponding closed sets. We also developed algorithms for related tasks, including, mainly, the ones we use in our current open source implementation, and discuss how these algorithms relate to existing ones for modular decomposition.

We have described an incremental algorithm to compute clan decompositions. There is ample room to study improvements to this algorithm and, as mentioned above, the possibilities of using our results to obtain further algorithms. But, as we have seen here, it already allows us to argue the potential value of this approach for data analysis.

Data analysis and data mining relies, most often, on statistical studies. One may be surprised that our approach does not follow that major line: we are interested in exploring complementarity with the existing processes. We believe that our visualizations can act usefully not at all replacing, but complementing the statistical approaches, by pointing out specific pairs of items, possibly conditioned to other items, whose correlation studies might be candidates to priority analysis, for instance, or in many other ways.

Since we work with the co-occurrence of data, one might think that this approach could be related to frequent set mining [1] [25] and rare pattern mining [22], let us discuss about it:

Our approach did not provide yet any interesting results when we directly applied it in combination with standard quantitative pattern mining concepts like support or confidence thresholds. Frequent set mining finds itemsets that appear in a dataset with a frequency no less than a determined threshold, thus, as a substantial difference with respect to our proposal: we do not take into account the frequency of the itemsets, we take into account the co-occurrences of each pair of items and despite the fact that in some cases we use frequency thresholds, these are only applied on single items or on pairs of items and we can not only know if they are frequent or not, that is if they satisfy the threshold or not, but we can also observe the different frequencies (by the equivalence classes) and their co-occurrence patterns with the other items. The rare pattern mining, unlike frequent pattern mining, finds itemsets that whose frequency of appearance in a dataset is below some defined threshold. We may compare this with our approach since we may give lower and upper thresholds values as low as we require, but again those thresholds are only applied on single items or on pairs of items.

Another existing topic related to our study is the notion of blockmodeling in network analysis [33]. The blockmodeling goal is to reduce an incomprehensible network to another more understandable and easy to interpret, similar to the proposed use of our graph decomposition method. Thus, comparing blockmodeling to the construction of the Gaifman graph, taking into account the partial implications, the network will be determined by the Gaifman graph and the relations between the items will be determined by the confidence of the connected items, in this way we will have a directed graph allowed by the blockmodeling theory but not by our current graph decomposition method.

6.2 Results and limitations

The paper that we published in IDA 2018 [5] was the first to explain the interest of these notions in exploratory data analysis. We also put forward there a number of variations of Gaifman graphs that allowed for finer analysis. In the paper published in CBMS 2019 [6] we illustrate the process and some of the possible results applying the data analysis approach based on the decomposition of Gaifman graphs variations on a medical dataset. Throughout this thesis we also provide proofs of our theorems and many additional examples of the usage of these tools for exploratory data analysis.

We have resorted to a relatively simple implementation in Python and relying on the standard graph module NetworkX and on the graphical capabilities of the pydot interface to the Dot engine of Graphviz[1], in fact, all the graph layouts in the figures in this paper were automatically computed by the Dot engine. Finding out how to configure it to produce acceptable output was a nontrivial task, since certain incompatibilities could have occurred with the Dot code.

[1] `https://en.wikipedia.org/wiki/Graphviz`

We must discuss a clear limitation. Like in so many other exploratory data analysis frameworks, for a given dataset we may not be lucky: it may happen that a given selection of Gaifman graph, once decomposed, has no nontrivial clans, or decomposes into just a few quite large primitive substructures that provide little or no insight about the data. After all, parameter tuning is a black art in many data mining approaches.

All that provokes to have some visual limitations, the obtained decomposition graphs are somewhat too large or complex to provide intuition through visualization. We even have chosen to omit the edges inside a clan whenever it is composed of too many subclans, the other consideration was to give a frequency threshold to reduce the quantity of vertices in the Gaifman graph. To decide the frequency threshold value we must to observe carefully the general behavior of data. All in all, for the time being, the human brain is a must during the exploration of interesting parameter settings. Useful advice to choose properly the thresholds and the sorts of Gaifman graphs among available options remains to be found.

6.3 Future work

There is ample room to study improvements and the possibilities of using our results to obtain further algorithms. For instance, the multiplicity-based generalizations we have proposed are quite basic; more sophisticate approaches to define the equivalence relation between edges might be advantageous. We believe that other than K-means unsupervised discretization methods could be applied. In addition, many other tunings can be applied to the Gaifman graph before applying the decomposition procedure. Similar to the shortest path version, we could work with the vertices connectivity, that is, the number of disjoint paths (that one can relate to Menger's theorem).

As seen throughout the work, closure spaces play an important role in our development. It is well-known that, in data analysis tasks, categorical concepts benefit from a relaxation allowing for exceptions, whether they come from varied inputs or even from material errors in coding or transmission. Likewise, we could relax through allowing exceptions the notions of module and clan. The concept we would end up with seems to us very close to the notion of blockmodeling employed in social network analysis: if we take into account the confidence of the items to determining their corresponding edges in the Gaifman graph, as we proposed in Section 6.1, we will have the present of a third type of 2-structure, the linear 2-structure. The analysis of the decomposition of such a graph is left out of this investigation, but it would be interesting to be analyzed.

In regards to compressing the data, two existing works could be implemented to our approach, [35] and [39]. In the first work, they study the mining of top-K frequent closed itemsets in order of decreasing support, whereas in our case we work with standard sets and even there is a possibility that its adaptation to our application would allow to have a faster algorithm. This would be a long task as it would require more than one additional chapter to this thesis. In the second paper, they address the problem of pattern mining by using the MDL principle that establishes that the best set of patterns were the set which best compresses the data. As a result, they developed the heuristic Krimp algorithm that works with all closed sets evaluating how much they allow to compress. The application of both theories to our work would lead to additional research which would of course be interesting.

A line that we do not analyze in this work is the presence of foreign keys. We would like to focus on investigating how foreign keys influence in the construction of the initial graph. For example in Section 4.4, in Figure 4.11, we have the clan decomposition of the dataset UW-CSE which is a multirelational dataset, we wonder whether the decomposition will change taking into account the foreign keys.

As we know, the primary key is an attribute, or set of attributes, whose values identify unequivocally each transaction in the father table. When a primary key migrates to another table, child table, it becomes in a foreign key. In this way, foreign key determines the relation between two tables. Assuming integrity on data, the values of the foreign keys in the child tables must link up with just one transaction in the father table, the correct use of foreign keys allows referential integrity as well. The referential integrity is that a transaction in the child table cannot have a foreign key value that is not a primary key value in the father table. There could be cases where the attributes of the primary keys have different names than the attributes of the foreign keys, to our work this do not imply a real problem since based on our current proposal we construct the desired graph taking as vertices the set of attribute values.

Thus, to construct the Gaifman graph taking into account the presence of foreign keys we propose to do the denormalization of the data by the primary keys. Denormalization is a database technique in which we add redundant data to one or more tables. This can help us to avoid costly joins in a relational database. We have to be careful not to count the co-occurrences of the foreign keys more times than necessary, because although they are happening, they will not be telling us anything really representative because they only serve as indexes or references in the data and can appear so many times making the rest of the co-occurrences lose in the hierarchy when it should not be.

Related to visualization, we have proposed a few simple strategies to encompass complex substructures upon visualization (the *Others* node) but a more systematic study of the ways in which visualizations can become helpful is necessary; we believe that the answers will come from some notion of interactive data analysis process. Here we may separate the cases of how the *Others* node is used: when it represents isolated vertices could be not necessary to know all the isolated ones; but in the case the *Others* node represents too large clans we may implement a zoom function to visualize the internal 2-structure of it. We would like to offer self-describing, more informative, perhaps even animated visualizations.

Bibliography

[1] Charu C. Aggarwal, Mansurul Bhuiyan, and Mohammad Al Hasan. Frequent pattern mining algorithms: A survey. In Charu C. Aggarwal and Jiawei Han, editors, *Frequent Pattern Mining*, pages 19–64. Springer, 2014.

[2] Albert Atserias, José L. Balcázar, and Marie Ely Piceno. Relative entailment among probabilistic implications. *Logical Methods in Computer Science*, 15(1), 2019.

[3] Eric Badouel and Philippe Darondeau. Theory of regions. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 529–586. Springer, 1996.

[4] José Luis Balcázar and Marie Ely Piceno. Hierarchical visualization of co-occurrence patterns as clans in generalized gaifman graphs. Submitted to a journal, 2020.

[5] José Luis Balcázar, Marie Ely Piceno, and Laura Rodríguez-Navas. Decomposition of quantitative gaifman graphs as a data analysis tool. In Wouter Duivesteijn, Arno Siebes, and Antti Ukkonen, editors, *Advances in Intelligent Data Analysis XVII - 17th International Symposium, IDA 2018, 's-Hertogenbosch, The Netherlands, October 24-26, 2018, Proceedings*, volume 11191 of *Lecture Notes in Computer Science*, pages 238–250. Springer, 2018.

[6] José Luis Balcázar, Marie Ely Piceno, and Laura Rodríguez-Navas. Hierarchical visualization of co-occurrence patterns on diagnostic data. In

32nd IEEE International Symposium on Computer-Based Medical Systems, CBMS 2019, Cordoba, Spain, June 5-7, 2019, pages 168–173. IEEE, 2019.

[7] Nikos Bikakis. Big data visualization tools. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies.* Springer, 2019.

[8] Aaron Ceglar and John F. Roddick. Association mining. *ACM Comput. Surv.*, 38(2), 2006.

[9] Vasek Chvátal and Chính T. Hoàng. On the P_4-structure of perfect graphs i. even decompositions. *J. Comb. Theory, Ser. B*, 39(3):209–219, 1985.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[11] Derek G. Corneil, H. Lerchs, and L. Stewart Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163–174, 1981.

[12] Dheeru D. and Karra Taniskidou E. Uci machine learning repository, 2017.

[13] Rina Dechter and Judea Pearl. Structure identification in relational data. *Artif. Intell.*, 58(1-3):237–270, 1992.

[14] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[15] A. Ehrenfeucht and G. Rozenberg. Primitivity is hereditary for 2-structures. *Theor. Comput. Sci.*, 70(3):343–358, 1990.

[16] Andrzej Ehrenfeucht, Tero Harju, and Grzegorz Rozenberg. *The Theory of 2-Structures - A Framework for Decomposition and Transformation of Graphs.* World Scientific, 1999.

[17] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54, 1996.

[18] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 18(1):25–66, 1967.

[19] Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.

[20] Michael Hahsler, Sudheer Chelluboina, Kurt Hornik, and Christian Buchta. The arules r-package ecosystem: Analyzing interesting patterns from large transaction data sets. *Journal of Machine Learning Research*, 12:2021–2025, 2011.

[21] R. Khardon and D. Roth. Reasoning with models. *Artificial Intelligence*, 87(1-2):187 – 213, 1996.

[22] Yun Sing Koh and Sri Devi Ravana. Unsupervised rare pattern mining: A survey. *TKDD*, 10(4):45:1–45:29, 2016.

[23] Carson K. Leung, Vadim V. Kononov, Adam G. M. Pazdor, and Fan Jiang. Pyramidviz: Visual analytics and big data visualization for frequent patterns. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2016, Auckland, New Zealand, August 8-12, 2016*, pages 913–916. IEEE Computer Society, 2016.

[24] L. Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series, Springer, first edition, 2004.

[25] José María Luna, Philippe Fournier-Viger, and Sebastián Ventura. Frequent itemset mining: A 25 years review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, 9(6), 2019.

[26] Frédéric Maffray and Myriam Preissman. *A Translation of Gallai's Paper: 'Transitive Orientierbare Graphen'*, pages 25–66.

[27] José Luis Balcázar Marie Ely Piceno, Laura Rodríguez-Navas. Cooccurrence patterns in diagnostic data. *Computational Intelligence*, 2020.

[28] Ross M. McConnell. An o(n^2) incremental algorithm for modular decomposition of graphs and 2-structures. *Algorithmica*, 14(3):229–248, 1995.

[29] Ross M. McConnell and Jeremy P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[30] Ross M. McConnell and Jeremy P. Spinrad. Modular decomposition and transitive orientation, 1999.

[31] R. H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. *Rival (Ed.) Graphs and Orders, D.Reidel, Boston*, pages 41–101, 1985.

[32] John H. Muller and Jeremy P. Spinrad. Incremental modular decomposition. *J. ACM*, 36(1):1–19, 1989.

[33] Mark E. J. Newman. *Networks: An Introduction.* Oxford University Press, 2010.

[34] Marie Ely Piceno and Laura Rodríguez-Navas. A graphical tool for the interpretation of medical data. In *ACM Celebration of Women in Computing: womENcourage*, 2019.

[35] Andrea Pietracaprina and Fabio Vandin. Efficient incremental mining of top-k frequent closed itemsets. In Vincent Corruble, Masayuki Takeda, and Einoshin Suzuki, editors, *Discovery Science, 10th International Conference, DS 2007, Sendai, Japan, October 1-4, 2007, Proceedings*, volume 4755 of *Lecture Notes in Computer Science*, pages 275–280. Springer, 2007.

[36] Ron Rymon. Search through systematic set enumeration. In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, USA, October 25-29, 1992*, pages 539–550, Cambridge, MA, USA, 1992. Morgan Kaufmann.

[37] Jeremy P. Spinrad. P_4-trees and substitution decomposition. *Discrete Applied Mathematics*, 39(3):263–291, 1992.

[38] Pang-Ning Tan, Michael S. Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison-Wesley, 2005.

[39] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. Krimp: mining itemsets that compress. *Data Min. Knowl. Discov.*, 23(1):169–214, 2011.

[40] Haizhou Wang and Mingzhou Song. Ckmeans.1d.dp: Optimal k-means Clustering in One Dimension by Dynamic Programming. *The R Journal*, 3(2):29–33, 2011.

[41] M. Wild. A theory of finite closure spaces based on implications. *Advances in Mathematics*, 108:118–139, 1994.