Rapport de Stage recherche - M2 Informatique Fondamentale

Morphologie Mathématique, Analyse des Concepts Formels et Musicologie Computationnelle

Effectué au sein des structures :

Présenté par :
Pierre Mascarade
ENS Lyon
M2 Informatique Fondamentale

Sous la direction de :
Moreno Andreatta,
IRCAM - RepMus
Musicologie Computationnelle
Jamal Atif,
Paris Dauphine
Morphologie Mathématique - FCA
Isabelle Bloch,
Télécom ParisTech
Morphologie Mathématique - FCA

Sous le tutorat de :
Márton Karsai,
ENS Lyon
Computer Science / Complex Systems

Effectué durant la période
Février 2017 - Juin 2017

Résumé

Mots Clefs *: Analyse des Concepts Formels (FCA), Morphologie Mathématique (MorphoMath), Théorie des Treillis, Musicologie Computationnelle, Music Information Research (MIR), Descripteurs Harmoniques, Reconnaissance des Formes Harmoniques, Analyse Musicale, SageMath, Représentation des données.*

Nous discuterons dans ce document des recherches menées à l'IRCAM ayant pour objectif général d'étudier les pistes proposées par l'algèbre abstraite pour *représenter* et *comprendre* les systèmes harmoniques sous-jacent aux oeuvres musicales. Tous les moyens mis en oeuvre dans ce sens seront exposés et de nouveaux outils d'analyse symbolique de la musique seront présentés.

1 Introduction et problématique de recherche

Ce stage proposé par l'équipe *Représentation Musicale* de l'IRCAM (*Institut de recherche et coordination acoustique/musique*) fut formulé dans le but d'approfondir l'étude des relations entre la Morphologie Mathématique (**MorphoMath**), la Théorie des Treillis et l'Analyse des Concepts Formels (**FCA**) à la lumière des résultats récents obtenus dans le domaine du Music Information Research (**MIR**) au sens large [13]. Ce domaine, qui relève de la musicologie computationnelle, autrement dit de l'étude systématique des objets musicaux, pose des problèmes mathématiques difficiles, en particulier pour leur formalisation constructive et signifiante d'un point de vue musical mais aussi de leur résolution algorithmique.

Traditionnellement, la recherche en musicologie computationnelle s'oriente vers des concepts et des outils qui s'appliquent soit à l'analyse du signal audio, soit à l'analyse de structures symboliques et algébriques, mais le plus souvent de façon orthogonale. L'objectif principal de ce stage fut d'étudier des modèles mathématiques permettant éventuellement de relier ces deux approches à travers la généralisation de concepts issus de la MorphoMath (principalement utilisés en Traitement du Signal et des Images) à des structures d'*Algèbre Abstraite* représentant des objets musicaux et plus particulièrement des *Treillis de Galois*.

Afin de restreindre l'objet d'étude, dans le cadre de ce stage très *exploratoire*, je fus invité à me concentrer sur l'étude des systèmes harmoniques (pentatonique, diatonique, dodécaphponique, etc.) qui représentent le fondement des matériaux compositionels dans la musique occidentale.
Nous verrons que ces structures musicales sont représentables par des Treillis de Galois particuliers issus de la FCA ([2], [4]).

L'utilisation des formalismes conjoints de la Théorie des Treillis, de la MorphoMath et de la FCA permet d'aborder, d'un point de vue totalement nouveau car - basé sur le calcul et les représentations algébriques - des problèmes-type de la communauté MIR tels que la reconnaissance d'accords et dans une échelle plus large, la reconnaissance des progressions harmoniques. Ce sont des étapes préalables pour envisager le problème plus général, à savoir celui de l'analyse de la forme musicale et de la classification automatique du style musical.

D'un point de vue théorique, ce stage fut formulé en vue de mettre en évidence ce que pourrait apporter la MorphoMath appliquée à la FCA dans la formalisation et l'analyse des structures musicales.

2 État de l'art et pré-requis théoriques

2.1 Analyse des concepts formels

L'Analyse des Concepts Formels est une théorie algébrique basée sur la formalisation de la notion philosophique de concept. Au sein de cette théorie un concept est formellement défini par un ensemble d'exemples et un ensemble de propriétés partagées par l'ensemble de ces exemples. Ainsi tous les exemples d'un concept partagent le même ensemble de propriétés attachées au concept, de même chaque propriété attachée a un concept s'applique à tous les exemples de ce concept. Cette relation défini théoriquement un *concept formel*.

Introduisons la notation que nous utiliserons dans le reste du rapport. Soit G un ensemble d'éléments appelés *objets*, M un ensemble d'éléments appelés *attributs* et $I \subseteq G \times M$ une relation binaire entre les objets et les attributs, une paire $(g, m) \in I$ signifie *l'objet g possède l'attribut m*. Alors le triplet $\mathbb{K}(G, M, I)$ définit un *contexte formel*, on le visualise par un tableau binaire où les objets sont représentés par les lignes et les attributs

par les colonnes, pour signifier qu'un objet g possède l'attribut m une croix est placée à l'intersection de la ligne associée à g et de la colonne associée à m (cf. Fig[3]).

Nous définissons deux opérateurs de dérivation α et β qui auront un rôle central dans le reste de ce document. Soit $X \subseteq G$ un ensemble d'objets du contexte et $Y \subseteq M$ un ensemble d'attributs :

$$\alpha(X) := \{m \in M \mid \forall g \in X, (g,m) \in I\} \tag{1}$$
$$\beta(Y) := \{g \in G \mid \forall m \in Y, (g,m) \in I\} \tag{2}$$

En d'autre terme $\alpha(X)$ est l'ensemble de tous les attributs partagés par tous les objets de X, et $\beta(Y)$ est l'ensemble de tous les objets possédant les attributs de Y.

Definition 1 ***Correspondance de Galois****. Soit $(L, \preceq)$ et $(L', \preceq')$ deux ensembles partiellement ordonnés. Une paire d'opérateurs (α, β), $\alpha : L \longmapsto L'$, $\beta : L' \longmapsto L$ définissent une* Correspondance de Galois *si $\forall x \in L, \forall y \in L', y \preceq' \alpha(x) \Longleftrightarrow x \preceq \beta(y)$.*

Ainsi les deux opérateurs de dérivation α et β forment une *Correspondance de Galois* entre les *powersets* (ensemble des parties d'un ensemble) partiellement ordonnés par l'inclusion ensembliste $(\mathcal{P}(G), \subseteq)$ et $(\mathcal{P}(M), \subseteq)$. Les *concepts formels* du contexte $\mathbb{K}$ sont alors toutes les paires $(X, Y) \in \mathcal{P}(G) \times \mathcal{P}(M)$ avec $X \subseteq G$ et $Y \subseteq M$ tel que (X, Y) soit maximal vis à vis de $X \times Y \subseteq I$, c'est à dire $\alpha(X) = Y$, $\beta(Y) = X$.

Pour chaque concept formel (X, Y) - où X est l'ensemble des objets du concept et Y l'ensemble d'attributs - l'ensemble X sera nommé l'*extent* du concept, tandis que l'ensemble Y sera nommée l'*intent* du concept. Pour un concept a on dénotera son *extent* par $e(a)$ et son *intent* par $i(a)$, autrement dit $a = (e(a), i(a))$.

L'ensemble des concepts formels d'un contexte formel peut être par suite ordonné par inclusion de leur extent, c'est à dire (c-à-d) : $(X_1, Y_1) \preceq (X_2, Y_2) \Longleftrightarrow X_1 \subseteq X_2 (\Longleftrightarrow Y_2 \subseteq Y_1)$. Une telle relation d'ordre connue comme la relation *subconcept-superconcept* génère toujours un treillis complet de Galois dénoté par $(\mathfrak{B}(G, M, I); \leq)$, où $\mathfrak{B}(G, M, I)$ dénote l'ensemble de tous les concepts et $\leq$ la relation d'ordre sur le treillis (inclusion de l'extent). Un treillis de Galois est un treillis dont la construction est basée sur une correspondance de Galois. Dans le cadre de la FCA on nomme le treillis engendré par un contexte formel un ***treillis de concept*** on le dénotera par $\mathbb{C}(\mathbb{K})$, on abrégera parfois la notation à $\mathbb{C}$, autrement dit $(\mathfrak{B}(G, M, I); \leq) \equiv \mathbb{C}(G, M, I) \equiv \mathbb{C}(\mathbb{K}) \equiv \mathbb{C}$.

Theoreme 1 *([3]) Dans un treillis de concept,* infimum (**meet**) *et* supremum (**join**) *d'une famille de concepts formels $(X_t, Y_t)_{t \in T}$ sont donnés respectivement par :*

$$\bigwedge_{t \in T} (X_t, Y_t) = \left(\bigcap_{t \in T} X_t, \alpha\big(\beta(\bigcup_{t \in T} Y_t)\big) \right), \tag{3}$$
$$\bigvee_{t \in T} (X_t, Y_t) = \left(\beta\big(\alpha(\bigcup_{t \in T} X_t)\big), \bigcap_{t \in T} Y_t \right) \tag{4}$$

Ainsi chaque treillis complet peut être vu comme un treillis de concept, plus précisément un treillis complet $(L, \preceq)$ est isomorphe au treillis de concept $\mathbb{C}(L, L, \preceq)$.

Definition 2 ***Relation entre G, M et $\mathbb{C}$.*** *Soit (G, M, I) un contexte et $\mathbb{C}$ le treillis de concept associé. Soit $g \in G$ un objet et $m \in M$ un attribut. Le concept $\gamma(g) := (\beta\alpha(g), \alpha(g))$ est appelé* ***concept-objet*** *de g, c'est le concept minimal - au sens de l'extent - qui contient l'objet g dans son extent. De la même façon $\mu(m) := (\beta(m), \alpha\beta(m))$ est l'****attribut-objet*** *de m et correspond au concept minimal - au sens de l'intent - qui contient l'attribut m dans son intent. Nous définissons deux maps $p_{\mathbb{C}}$ et $q_{\mathbb{C}}$ qui projettent les objets et les attributs vers leurs concept-objets et attribut-objets respectifs, comme suit :*

$$p_{\mathbb{C}} : \left| \begin{array}{ccc} G & \longrightarrow & \mathbb{C} \\ g & \longmapsto & \gamma g \end{array} \right. \qquad q_{\mathbb{C}} : \left| \begin{array}{ccc} M & \longrightarrow & \mathbb{C} \\ m & \longmapsto & \mu m \end{array} \right. \tag{5}$$

2.2 Morphologie Mathématique et Théorie des Treillis

La MorphoMath est une théorie générale de l'analyse de formes géométriques fondée, pour sa partie déterministe, sur des objets mathématiques empruntés à la Théorie des Treillis, la Topologie, et la Géométrie Discrète entre autres, développée initialement par J.Serra dans [7]. Ses applications les plus connues relèvent du traitement des signaux et des images, elle a par la suite été étendue pour traiter des questions relatives au calcul spatial dans [17], aux raisonnement logiques (au travers de la Morpho-Logique) dans [18], ou encore pour raisonner sur des concepts émanants de la FCA dans [5].

2.2.1 Opérateurs morphologiques

Rappelons le contexte algébrique général de la MorphoMath.
Soit $(L, \preceq)$ et $(L', \preceq')$ deux treillis complets.

Definition 3 ***Dilatation*** *([7], [5], [6]). Un opérateur $\delta \colon L \to L'$ est une dilatation si il préserve le plus petit élément du treillis et si il commute avec le supremum (sup-preserving mapping) : $\forall (x_i) \in L$, $\delta(\vee_i x_i) = \vee'_i \delta(x_i)$, où $\vee$ correspond au supremum (join) associé à $\preceq$ et $\vee'$ celui associé à $\preceq'$.*

Definition 4 ***Érosion*** *([7], [5], [6]). Un opérateur $\varepsilon \colon L' \to L$ est une érosion si il préserve le plus grand élément du treillis et si il commute avec l'infimum (inf-preserving mapping) : $\forall (x_i) \in L'$, $\varepsilon(\wedge'_i x_i) = \wedge_i \varepsilon(x_i)$, où $\wedge$ correspond à l'infimum (meet) associé à $\preceq$ et $\wedge'$ à celui associé à $\preceq'$.*

Definition 5 ***Adjunction*** *([7], [5], [6]). une paire d'opérateur (ε, δ), $\delta \colon L \to L'$, $\varepsilon \colon L' \to L$, définit une* adjonction *si $\forall x \in L, \forall y \in L', \delta(x) \preceq' y \iff x \preceq \varepsilon(y)$.*

Il peut être montré que si une paire d'opérateur (ε, δ) définit une adjonction, alors :
— δ préserve le plus petit élément et ε préserve le plus grand élément
— δ est une érosion et ε une dilatation

Un premier lien qui peut être fait entre la MorphoMath et la FCA réside dans le fait que ces deux théories reposent toutes deux sur la structure des treillis complets. De plus la paire d'opérateur morphologique (ε, δ) est une *adjonction*, parfois appelée *correspondance de Galois monotone*, alors que la paire d'opérateur de dérivation (α, β) de la FCA est une *correspondance de Galois antitone*, ces deux propriétés étant équivalentes si l'on renverse l'ordre d'un des treillis sur lequel elles s'appliquent.

2.2.2 Élément structurant et exemple

Les formes de dilatation et d'érosion les plus utilisées sont souvent définies à partir de la notion d'*élément structurant*, qui dans la grande majorité des cas est une relation de voisinage mais peut également être tout autre relation binaire.

Cette notion se comprend intuitivement avec un exemple. Considérons l'espace $\mathbb{R}^2$ et prenons son *powerset* $L = \mathcal{P}(\mathbb{R}^2)$ pour construire ensemble partiellement ordonné $(L, \subseteq)$ où $\subseteq$ est l'inclusion ensembliste classique. Dénotons par B un élément structurant (sous-ensemble de $\mathbb{R}^2$), $\check{B}$ son symétrique (ensembliste) par rapport à l'origine de l'espace et B_x sa translation au point x. Les opérateurs de dilatation δ et d'érosion ε sur $(L, \subseteq)$ seront alors définis en fonction de l'élément structurant de la façon suivante :

$$\forall X \subseteq \mathbb{R}^2 \ \delta_B(X) = \{x \in \mathbb{R}^2 \mid \check{B}_x \cap X \neq \emptyset\}$$
$$\varepsilon_B(X) = \{x \in \mathbb{R}^2 \mid B_x \subseteq X\}$$

Un exemple de dilatation et d'érosion d'image binaire est donné dans la figure ci-dessous, où B est un cercle sur $\mathbb{R}^2$ ("balle" de la distance Euclidienne).

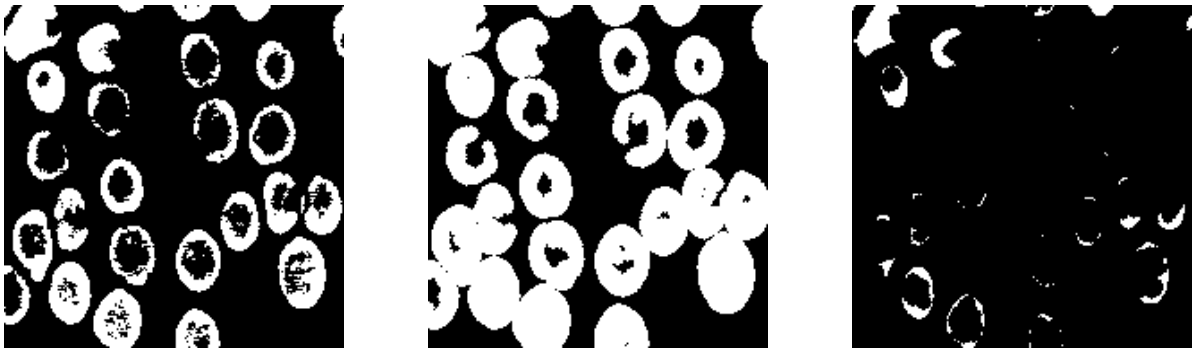


FIGURE 1 – Gauche - sous-ensemble X dans le plan Euclidien (en blanc). Centre - Sa dilatation $\delta_B(X)$ par un élément structurant B ("balle" de distance euclidienne). Droite - son érosion $\varepsilon_B(X)$ par le même élément structurant B. ([5])

Intuitivement on comprend sur ces images binaires qu'une dilatation (centre) étend les bordures de l'ensemble X (éléments blancs sur l'image binaire de gauche) d'une quantité égale au rayon de B. Tandis que l'érosion (droite) réduit les bordures de l'ensemble X de cette même quantité. En traitement des images et surtout en imagerie médicale et industrielle, l'opérateur dilatation est utilisé principalement pour connecter des composants connexes ou fermer des "trous" sur l'image liés à d'éventuels artefacts, tandis que l'érosion sert surtout à déconnecter des éléments qui qui ne devrait pas l'être par un éventuel bruit ou à supprimer des petit artefacts qui serait plus petits que l'élément structurant (au sens de l'inclusion).

2.3 Analyse des concepts formels musicaux

2.3.1 Algèbre des systèmes harmoniques

La FCA fut introduite au début des années 1980s par Rudolf Wille [8] dans le but de reformaliser et de reconstruire la Théorie des Treillis alors pleinement formalisée par Garrett Birkhoff.
Depuis sa naissance, la musique a toujours été un champs d'application privilégié pour la FCA, R.Wille lui même s'y attarde en 1985 dans [9]. Malgré l'ancienneté de cette intuition, la systématisation de l'étude des structues musicales par la FCA ne s'est pas développée dans la théorie mathématique de la musique et dans la communauté MIR avant les travaux de M.Andreatta et T.Schlemmer ([2], [4]). Ces publications furent les premières à réellement s'attarder sur l'importance des structures d'ordre (Treillis) pour analyser les systèmes harmoniques.
Afin de pouvoir étudier de tels objets musicaux mathématiquement, il nous faut une description mathématique de ces objets.
Nous allons dans la suite de ce chapitre montré comment on formalise la notion musicale de système harmonique - qui consiste en une collection de *tons musicaux* (do-ré-mi-fa...) et d'intervalles les séparant - en un objet algébrique permettant le calcul et la manipulation algorithmique.
Par intervalle on entend *intervalle harmonique*, par exemple entre *Do* et *Ré* il y a un intervalle de *seconde*, entre *Do* et *Mi* un intervalle de *tierce*, entre *Mi* et *Fa* un intervalle de seconde, etc.

Nous utiliserons dans la suite de ce document la terminolige du *Extensional Standard Language of Music* proposé par R.Wille et utilisé par T.Schlemmer et S.Schmidt dans [4] pour décrire les systèmes harmoniques à tempérament égal de n-tons. Ce langage propose toutes les briques élémentaires pour construire une théorie mathématique de la musique. Dans les paragraphes qui suivent, les notations et le lexique relatifs à l'aspect musical du problème sont présentés.

Formellement, un triplet $\mathbb{T} = (T, \delta, \mathrm{I})$ est appelé système harmonique algébrique, si T est un ensemble, $\mathrm{I} = (I, +, -, 0)$ un Groupe Abelien, et $\delta : T \times T \to I$ une application telle que pour tous $t_1, t_2, t_3 \in T$ les équations suivantes soient vraies :

$$\delta(t_1, t_2) + \delta(t_2, t_3) = \delta(t_1, t_3) \quad \text{et} \quad \delta(t_1, t_2) = 0 \text{ iff } t_1 = t_2. \tag{6}$$

Les éléments de l'ensemble T sont appelés *tons* (musicaux) et chaque sous-ensemble de T est un *accord* (musical). L'accord vide est considérés comme un silence ici. Les éléments de I sont considéré comme des intervalles (musicaux).
Dans mes recherches, j'ai considéré les systèmes harmoniques de la forme $T = (\mathbb{Z}, \delta, \mathbb{Z})$ avec $\delta(s, t) = t - s$, et $\mathbb{Z}$ le groupe additif des entiers relatifs. Aussi il nous faut, pour chaque système à étudier, fixer un entier positif $\mathcal{O} \in \mathbb{Z}_+$ qui représente l'intervalle que nous appellerons *octave* :

- mathématiquement cela correspond à la cardinalité du Groupe I,
- musicalement cela correspond au nombre de tons présents dans notre système. Par exemple si $\mathcal{O} = 5$ nous sommes dans *un* système pentatonique (par ex. do-ré-mi-sol-la), si $\mathcal{O} = 7$ dans *un* système diatonique (par ex.do-ré-mi-fa-sol-la-si), si $\mathcal{O} = 12$ dans *le* système dodécaphonique (do-do♯-ré-ré♯-mi-fa-fa♯-sol-sol♯-la-la♯-si).

Nous dénotons par $\mathbb{Z}_\mathcal{O}$ l'*anneau quotient* des entiers *modulo* $\mathcal{O}$, et dénotons par là même le système harmonique algébrique $\mathbb{T}_\mathcal{O} = (\mathbb{Z}_\mathcal{O}, \delta_\mathcal{O}, \mathbb{Z}_\mathcal{O})$, où $\delta_\mathcal{O}(x, y)$ définit la différence $y - x$ dans $\mathbb{Z}_\mathcal{O}$. Dans le langage propre à la musicologie on nomme ce genre de système : *système chromatique* et ses éléments sont appelés *chromas*. Plus précisément, dans le reste de ce rapport, pour parler de $\mathbb{T}_\mathcal{O}$ on se référera à la notion de $\mathcal{O}$-*tone equal tempered*, $\mathcal{O}$-tet en abrégé. Les $\mathcal{O}$-tets les plus utilisés en composition et en analyse sont le 7-tet et le 12-tet car respectivement liés à la *gamme diatonique* et à la *gamme dodécaphonique*, les deux gammes les plus utilisés dans la musique occidentale. Dans la figure ci-dessous on pourra observer la structure du groupe cyclique $\mathbb{Z}_{12}$ associée au 12-tet.

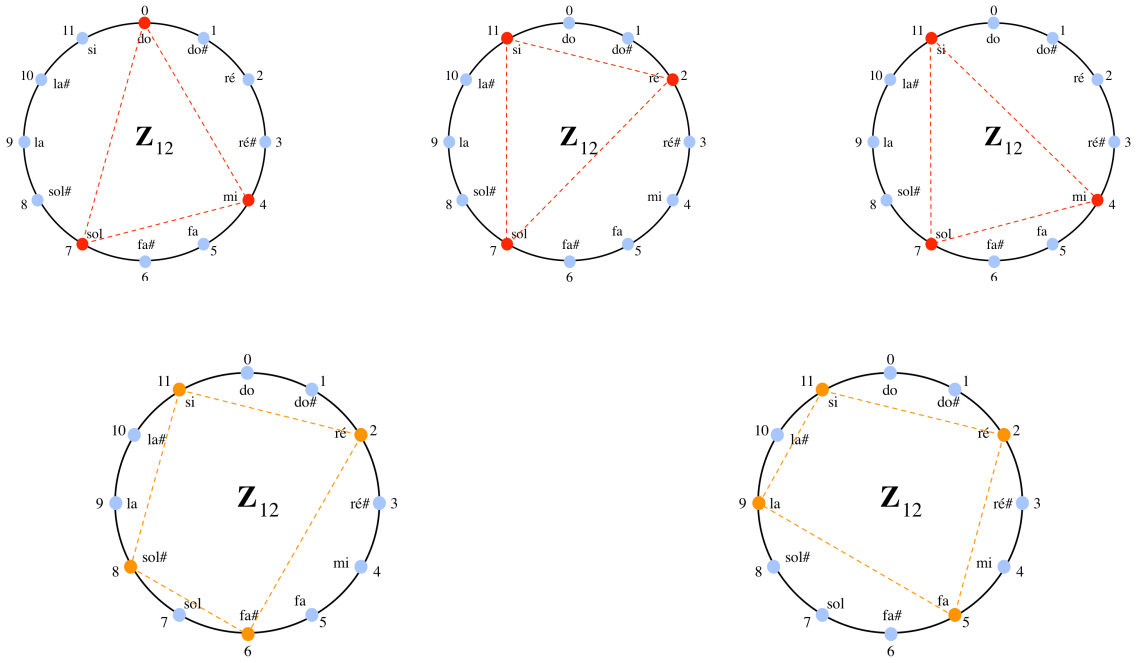
L'homomorphisme canonique de groupe $\phi_\mathcal{O} : \mathbb{Z} \longmapsto \mathbb{Z}_\mathcal{O}$ (qui projette chaque entier x vers son résidu *modulo* $\mathcal{O}$ noté $x_\mathcal{O}$) induira un morphisme de $\mathbb{T}$ vers $\mathbb{T}_\prime$. Chaque *accord* X dans $\mathbb{T}$ est projeté vers l'accord

$X_{\mathcal{O}} := \{x_{\mathcal{O}} \mid x \in X\}$ dans $\mathbb{T}_{\mathcal{O}}$ par $\phi_{\mathcal{O}}$. $X_{\mathcal{O}}$ sera nommé l'*harmonie* de X.

Afin de réduire les harmonies à leurs structures essentielles, la techniques standard dans [4] consiste à les classer en fonction des occurrences d'intervalles et de chromas qu'elles contiennent. Ainsi deux accords (ou harmonies) ont le même motif dans le cas où elles sont reliées par une transposition. Cette classification engendre des classes d'équivalence que l'on appellera *formes harmoniques*.
Formellement on pourra le modéliser comme suit : Soit une harmonie H et un chroma $t \in H$ on définit l'ensemble $I_{H,t} := \{\delta(t,t') \mid t' \in H\}$, alors $H = \{t' | t' = t + i, i \in I_{H,t}\}$. Maintenant si on transpose l'harmonie H d'un intervalle i, on peut définir $H+i := \{t'+i \mid t' \in H\}$ iff $t'+i$ existe $\forall t' \in H$. Alors on a $I_{H,t} = I_{H+i,t+i}$, autrement dit quand deux harmonies sont reliés par une transposition elles possèdent le même motif intervallique. On peut alors définir une relation d'équivalence :

$$\Psi_{\prime} := \{(H_1, H_2) \mid \exists i \ : H_1 = H_2 + i, H_1 \subseteq \mathbb{Z}_{\mathcal{O}}, H_2 \subseteq \mathbb{Z}_{\mathcal{O}}\} \tag{7}$$

Les classes d'équivalence $[H]_{\Psi}$ sont les fameuses *formes harmoniques* que nous allons considérer. Si deux harmonies appartiennent à la même classe d'équivalence, c-à-d si $H_1 \Psi H_2$, c'est qu'il existe forcément deux chromas $t_1 \in H_1$ et $t_2 \in H_2$ tel qu'elles aient le même motif intervalique $I_{H_1,t_1} = I_{H_2,t_2}$. Intuitivement la forme harmonique d'une harmonie est l'abstraction d'un ensemble d'harmonies qui permet de dénoter cet ensemble par une forme *unique* et *commune* à toutes les harmonies de l'ensemble. Si deux harmonies ont la même *forme harmonique* alors elles partagent le même nombre de notes ainsi que la structures intervalliques séparant ces notes (pas forcément les mêmes notes cependant) et donc auront des propriété acoustiques similaires. On peut donc les considérer comme équivalentes d'un point de vue musical. De cette façon tous les accords majeurs sont équivalents dans notre systèmes de classes d'équivalences, de mêmes tous les accords mineurs, tous les accords de $7^{\text{ème}}$ de Dominante, etc.



Ainsi dans les trois harmonies du haut de la figure ci-dessus (en rouge), celle de gauche (accord de *Do majeur*) et celle du milieu (accord de *Sol majeur*) possèdent la même forme harmonique $[0, 4, 7]$, où 0 est l'intervalle *nul*, 4 est la *tierce majeure* et 7 est l'intervalle de *quinte juste*. En effet elles sont reliées par une transposition de *quinte juste*. Tandis que l'harmonie représentée à droite de la figure appartient à la classe des accords mineurs qui ont tous pour forme harmonique $[0, 3, 7]$, où 3 est la *tierce mineure*. De la même façon les accords présents en bas de la figure (en orange) possèdent la même forme harmonique à savoir $[0, 2, 5, 8]$ qui est la forme harmonique des accords de $7^{ème}$ *de Dominante*.

Formellement, on dénotera *l'ensemble de toutes les formes harmoniques* d'un système harmonique $\mathbb{T}_{\mathcal{O}}$ par $\mathcal{H}(\mathbb{T}_{\mathcal{O}})$.

2.3.2 De l'algèbre des systèmes harmoniques au treillis de concept

Une fois toutes les formes harmoniques $\mathcal{H}(\mathbb{T}_{\mathcal{O}})$ d'un systèmes harmonique $\mathbb{T}_{\mathcal{O}}$ définies il existe plusieurs moyens pour équiper ce ensemble d'un *ordre* propice à le transformer en *treillis*. Les premiers treillis de concept envisagés par Wille avait une forme $\mathfrak{B}(\mathcal{H}(\mathbb{T}_\prime), \mathcal{H}(\mathbb{T}_\prime), \sqsubseteq)$ muni d'un ordre $\sqsubseteq$ relatif à l'inclusion des ensembles représentant les formes harmoniques(cf. Fig.[2]). Le problème de ces treillis réside dans la combinatoire explosive engendrée leurs concepts avec $\sqsubseteq$. En effet le 12-tet se retrouve muni de 208946771 concepts. Il devient vite impraticable d'effectuer des calculs algébriques complexes sur ce genre d'objet même algorithmiquement.

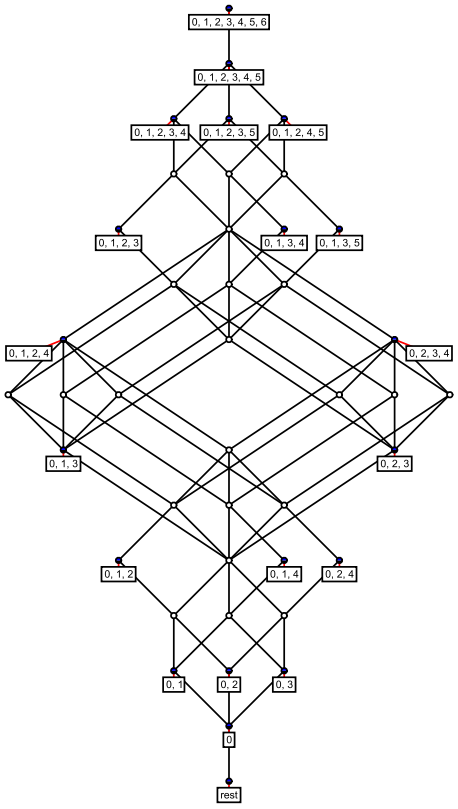


FIGURE 2 – Treillis de concept $\mathfrak{B}(\mathcal{H}(\mathbb{T}), \mathcal{H}(\mathbb{T}), \sqsubseteq)$, où l'ordre $\sqsubseteq$ correspond à l'inclusion ensembliste des formes harmoniques de $\mathcal{H}(\mathbb{T}_\prime$ du 7-tet. ([2])

Lorsqu'on souhaite générer un treillis de concept à partir d'un ensemble d'objet défini - ici l'ensemble des formes harmoniques d'un système harmonique - le choix de l'ensemble des attributs est des plus déterminants. En effet c'est lui qui va engendrer la relation d'incidence du contexte et par la même la structure du treillis. Aussi, l'ensemble des attributs doit être le plus signifiant possible vis à vis de l'*univers* des données que l'on veut *conceptualiser*.
Une des propriétés les plus signifiantes d'un point de vue musical pour déterminer le degré de consonance ou de dissonance des formes harmoniques réside dans la nature des intervalles qu'elles contiennent.
C'est pourquoi dans mes recherches j'ai principalement travaillé avec des contextes formels musicaux où les *formes harmoniques sont utilisées comme ensemble des objets*, et *l'ensemble des intervalles du système harmonique comme ensemble des attributs*, ainsi l'occurrence d'un intervalle dans une forme harmonique découle naturellement comme relation binaire d'incidence entre les deux ensembles. En plus d'amener une signification musicale des plus pertinentes pour la construction du contexte formel, cet ensemble d'attribut engendre des treillis d'une taille beaucoup plus raisonnable pour la manipulation algorithmique que les treillis $\mathfrak{B}(\mathcal{H}(\mathbb{T}_\prime), \mathcal{H}(\mathbb{T}_\prime), \sqsubseteq)$ de R.Wille.

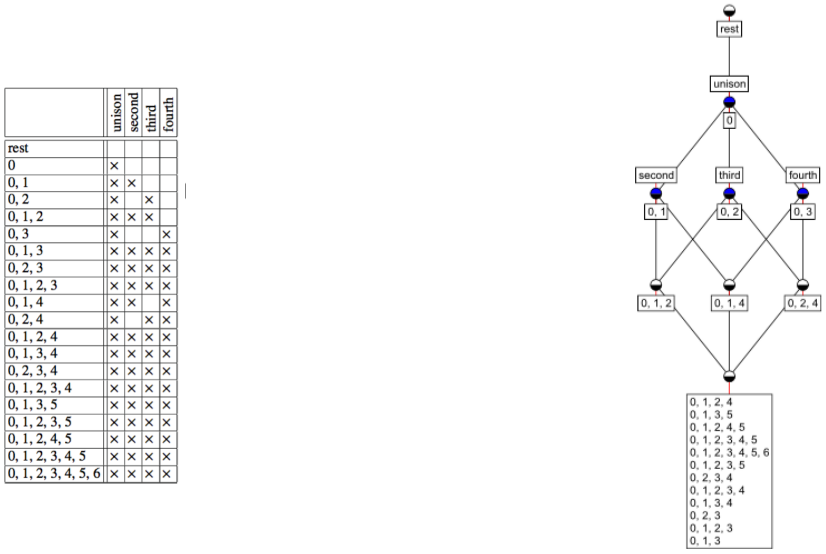
Exemple Le 7-tet $\mathbb{T}_7$. On définit le contexte formel suivant $(\mathcal{H}(\mathbb{T}_7), \mathbb{Z}_7, R)$, où R est une relation binaire telle que pour une forme harmonique $F \in \mathcal{H}(\mathbb{T}_7)$ et un intervalle $i \in \mathbb{Z}_7$, on ait $F\ R\ i$ si et seulement si $i \in I_{F,t}$ pour une harmonie et un chroma (= ton = note) $t \in F$.
, c-à-d si un couple croissant de notes de F est séparée par l'intervalle i.
Les intervalles sont nommées en fonction de le nombre ordinal du ton d'arrivé partant d'un ton fixé. Pour le 7-tet on aura donc les intervalles suivant :

— $\delta = 0$: *unisson*
— $\delta = 1$: *seconde*

— $\delta = 2$: *tierce*
— $\delta = 3$: *quarte*

Les noms données en italique correspondent aux noms des intervalles employés dans la musique.
On pourra remarquer que les intervalle $[4, 5, 6]$ du 7-tet n'apparaissent pas. En effet étant donné que les groupes $\mathbb{Z}_{\mathcal{O}}$ avec lesquels nous travaillons en musicologie computationelle sont des groupes finis, seuls $\lfloor \mathcal{O}/2 \rfloor$ intervalles différents sont nécessaires pour construire les attributs d'un contexte formel apte à discriminer des harmonies. Conformément au standard de l'algèbre des objets musicaux, on prendra les intervalles *primaires*, pour le 7-tet il s'agira de $[0, 1, 2, 3]$. Les autres intervalles ($[4, 5, 6]$) sont les complémentaires de ces intervalles primaires (éléments inverse du groupe cyclique en Théorie des Groupes), et ne sont pas nécessaires pour discriminer les formes harmoniques. Les intervalles *complémentaires* seront donc identifiés par leurs inverses étant donné que nous travaillons avec des intervalles "non-orientés".

On peut observer dans la partie gauche de la Fig.[3] le tableau binaire du contexte formel $\mathbb{K} = (\mathcal{H}(\mathbb{T}_7), \mathbb{Z}_7, R)$ engendré par cette méthode de construction de contexte pour le 7-tet. À droite, le treillis de concept $(\mathbb{C}(\mathbb{K}), \preceq)$ où l'ordre $\preceq$ correspond à l'inclusion de l'extent des concepts conformément à la FCA.



	unison	second	third	fourth
rest				
0	×			
0, 1	×	×		
0, 2	×		×	
0, 1, 2	×	×	×	
0, 3	×			×
0, 1, 3	×	×	×	×
0, 2, 3	×	×	×	×
0, 1, 2, 3	×	×	×	×
0, 1, 4	×	×		×
0, 2, 4	×		×	×
0, 1, 2, 4	×	×	×	×
0, 1, 3, 4	×	×	×	×
0, 2, 3, 4	×	×	×	×
0, 1, 2, 3, 4	×	×	×	×
0, 1, 3, 5	×	×	×	×
0, 1, 2, 3, 5	×	×	×	×
0, 1, 2, 4, 5	×	×	×	×
0, 1, 2, 3, 4, 5	×	×	×	×
0, 1, 2, 3, 4, 5, 6	×	×	×	×

FIGURE 3 – *Gauche* - Tableau binaire du contexte formel $\mathbb{K} = (\mathcal{H}(\mathbb{T}_7), \mathbb{Z}_7, R)$. *Droite* - Treillis de concept formel $(\mathbb{C}(\mathbb{K}), \preceq)$ où l'ordre $\preceq$ correspond à l'inclusion de l'extent des concepts conformément à la FCA. ([4])

Malgré l'avantage que représente la petite taille de ce treillis pour d'éventuelles manipulations algorithmiques, il se trouve que la quantité d'information sur la structure harmonique du système -tet est faible. Ceoendant elle peut etre grandement améliorée ([2], [4]) en considérant des ***contextes formels mutli-valués***. De tels contextes introduisent une notion de multiplicité dans la relation d'incidence qui lie les objets et les attributs. Dans notre cas présent il s'agira de considérer le contexte multi-valué $(\mathcal{H}(\mathbb{T}_7), \mathbb{Z}_7, \mathbb{Z}, w)$, où pour une forme harmoniques $F \in \mathcal{H}(\mathbb{T}_7)$ on définit la cardinalité intervallique suivante :

$$w(F, i) := \mid \{t \mid t \in F, i \in I_{F,t}\} \mid \tag{8}$$

Ce nouveau contexte formel renseignera sur la multiplicité des intervalles à l'intérieur d'une forme harmonique, par exemple la forme harmonique $F_a = [0, 1, 2]$ contient :
— 1 × unisson (toujours compté une seule fois)
— 1 × seconde $(0, 1)$
— 2 × seconde $(0, 1) - (1, 2)$
— 1 × tierce $(0, 2)$

Une astuce utilisée lorsqu'on travaille avec des contextes multi-valués consiste à *proportionner* le tableau binaire de telle sorte à faire apparaitre la *multiplicité d'occurrence d'un attribut* en tant qu'attribut indépendant et non pas en tant que relation valuée, on appellera cette "astuce" *scaling*. Cette dernière étape permettra de retrouver le même environnement mathématique qu'un contexte formel "classique". Ce qui nous permettra de pouvoir engendrer le treillis de concept tout en ayant ajouter une couche d'information précieuse au contexte et donc à la structure du treillis. La Fig.[4] présente la version *"scaled"* du contexte formel décrivant les multi-ensembles intervalliques des formes harmoniques du 7-tet $\mathbb{T}_7$, ainsi que le treillis de concept associé.

	unison	second	2× second	3× second	4× second	5× second	6× second	7× second	third	2× third	3× third	4× third	5× third	6× third	7× third	fourth	2× fourth	3× fourth	4× fourth	5× fourth	6× fourth	7× fourth
rest																						
0	×																					
0, 1	×	×																				
0, 2	×								×													
0, 1, 2	×	×	×						×													
0, 3	×															×						
0, 1, 3	×	×							×							×						
0, 2, 3	×	×							×							×						
0, 1, 2, 3	×	×	×	×					×	×						×						
0, 1, 4	×	×														×	×					
0, 2, 4	×								×	×						×						
0, 1, 2, 4	×	×	×						×	×						×	×					
0, 1, 3, 4	×	×	×						×							×	×	×				
0, 2, 3, 4	×	×	×						×	×						×	×					
0, 1, 2, 3, 4	×	×	×	×	×				×	×	×					×	×	×				
0, 1, 3, 5	×	×							×	×	×					×	×					
0, 1, 2, 3, 5	×	×	×	×					×	×	×	×				×	×	×				
0, 1, 2, 4, 5	×	×	×	×					×	×	×					×	×	×	×			
0, 1, 2, 3, 4, 5	×	×	×	×	×	×			×	×	×	×	×			×	×	×	×	×		
0, 1, 2, 3, 4, 5, 6	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

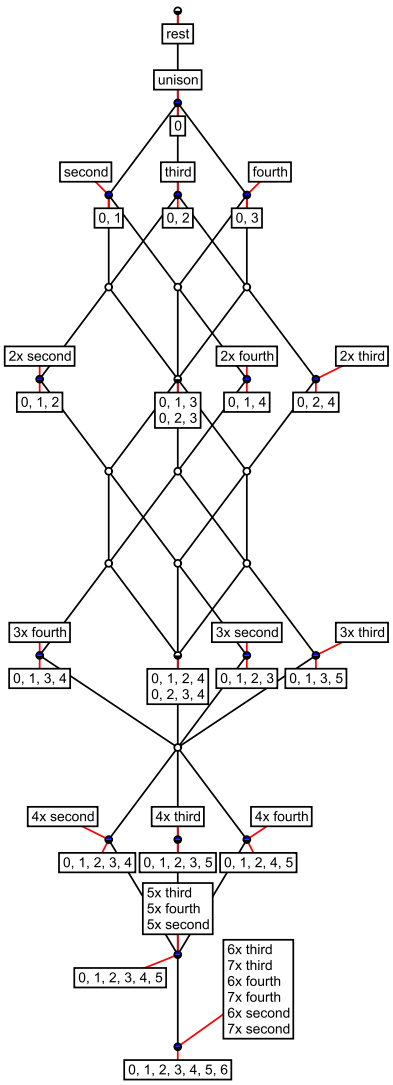


FIGURE 4 – *Gauche - "Scaled"* version du contexte formel décrivant les multi-ensembles intervalliques des formes harmoniques du 7-tet $\mathbb{T}_7$. *Droite* - Treillis de concept associé. ([4])

Cette approche peut être généralisée à tous systèmes harmoniques $\mathbb{T}_{\mathcal{O}} = (\mathbb{Z}_{\mathcal{O}}, \delta_{\mathcal{O}}, \mathbb{Z}_{\mathcal{O}})$ en engendrant le contexte formel $(\mathcal{H}(\mathbb{T}_{\mathcal{O}}), \mathbb{Z}_{\mathcal{O}}, R)$ puis $(\mathcal{H}(\mathbb{T}_{\mathcal{O}}), \mathbb{Z}_{\mathcal{O}}, \mathbb{Z}, w)$.

3 Travail méthodologique réalisé

La première partie de mon stage fut dédiée à l'étude bibliographique d'un corpus d'articles et de livres concernant les quatre sujets liés à ma problématique - qui étaient pour la plus grande partie nouveaux pour moi - à savoir :

— Théorie des Treillis
— FCA
— Morphologie Mathématique
— Théorie mathématique de la musique

Cette phase d'apprentissage et de familiarisation avec les concepts et principaux résultats de ces théories s'est étendue tout au long de la durée du stage, et je me suis efforcé de continuer à lire des articles et des ouvrages "*généraux*" en algèbre ou en géométrie discrète, parfois sans lien immédiat avec mon sujet de recherche, mais avec l'idée que c'est ainsi qu'on aiguise et qu'on affine son intuition quant aux objets qu'on manipule au quotidien.

La principale difficulté de ce genre de sujet pour le moins *exploratoire* et en même temps leur force : *la liberté d'enquête.*
Pour ne pas m'éloigner trop de la problématique j'ai cependant rapidement restreint mon espace de recherche vers l'étude d'application musicales potentiellement signifiante de la morphologie mathématique sur treillis de

concepts ; et ce en me focalisant sur les treillis de concept associé aux systèmes harmoniques 7-tet $\mathbb{T}_7$ et 12-tet $\mathbb{T}_{12}$.

3.1 Contribution au projet SAGEMATH

La teneur de mon stage relevant de la musicologie symbolique computationelle, autrement dit, de l'*algèbre appliquée*, il me fut demandé rapidement de développer des algorithmes d'analyse de treillis de concepts musicaux à l'aune d'une implémentation d'outils de la morphologie mathématique sur treillis.
Dès les premières réunions avec mon équipe pédagogique, je fus orienté vers ***SageMath***, un logiciel libre généraliste de calcul mathématique très utile pour le calcul symbolique et algébrique, compatible avec de nombreux langage dont ***Python*** que j'ai choisi pour son expressivité et son traitement des collections (*map, reduce, filter, lambda, etc.*) avec lequel je suis familier.

Étant donné la nouveauté théorique de mon domaine d'étude, je ne fus pas surpris de constater qu'aucun outil SageMath dédié à la morphologie mathématique sur treillis n'existait. Je fus en revanche bien plus étonné de constater que malgré la complétude de cet outil en Algèbre, aucune ressources algorithmiques n'étaient implémentées quant à la FCA, théorie qui a plus de trente ans et dont l'utilité sémantique en traitement des données et en Intelligence Artificielle n'est plus à démontrer.
Quelques algorithmes fort utiles pour le calcul des éléments caractéristiques et des propriétés des treillis étaient quant à eux déjà implémenté par la communauté SageMath.

Sur ce constat, et devant la nécessité d'effectuer des analyses aussi systématiques que possible dans le cadre de ce sujet expérimental, j'ai décidé de développer une bibliothèque d'algorithmes d'analyse algébrique et de visualisation de données orientés MorphoMath-FCA-Treillis.
Le projet SageMath est développé et maintenu par une communauté de mathématiciens dans une logique participative. Pour être plus précis, il existe toujours en permanence *deux versions* de SageMath, la version *released* qui est l'état stable et opérationnel du projet, et la version *develop* qui est la version en cours de construction en vue d'une prochaine *release*. Le fonctionnement est commun à beaucoup de projet partageant la philosophie libre de l'*open source*.
Lorsqu'un utilisateur veut ajouter des fonctionnalités à SageMath :

- il duplique le code source depuis le projet *GitHub* de SageMath
- il créé une nouvelle branche dans laquelle il *push* le code associé à la nouvelle fonctionnalité
- le code attend ensuite une *review* de la part d'un (souvent plusieurs) autre(s) utilisateur(s) de la communauté SageMath avant d'être validé et fusionné au code principal de la version *develop*
- le code peut ensuite être maintenu et modifié par le reste de la communauté itérativement jusqu'à obtenir une version optimale et robuste en vue d'une publication du code dans la prochaine version *release* de SageMath.

Dans cette optique j'ai décidé d'écrire - en vue d'une soumission au projet *Sage* -, une contribution algorithmique aussi complète que possible relative à l'Analyse des Concepts Formels et à la Morphologie Mathématique sur treillis, tant d'un point de vue calcul algébrique que d'un point de vue visualisation. Les premières briques algorithmiques de ces domaines dans le projet SageMath seront publiées dès que possible avec l'aide et le soutient de mes encadrant pédagogiques, notamment Isabelle Bloch et Jamal Atif. En effet le développement de ces outils algorithmiques fut soutenue par mon équipe pédagogique comme étant une *étape nécessaire* pour les avancées dans la compréhension des liens entre algèbre musicale, MorphoMath, FCA et Théorie de Treillis.
La grande majorité de mon code s'organise autour de deux classes (FormalContext, FormalConcept) et d'un fichier "boite à outils" contenant plusieurs fonction de tests de propriétés mathématiques ou de calcul algébrique. La structure du code et le découpage objet (Prog. Orientée Objet) va être complètement repensé avant la soumission du code sur la version *develop* de SageMath afin qu'il s'interface au mieux avec le code existant. L'état actuel du code est fourni dans l'*Annexe 2* de ce document.

La quantité d'algorithmes et de fonctions implémentés étant trop importante pour être exhaustivement explicité dans le présent document, je me concentrerais dans la suite de ce chapitre sur les fonctionnalités les plus signifiantes pour la problématique de recherche du stage, en essayant autant que possible d'introduire itérativement les objets mathématiques abordés en donnant à chaque fois des exemples de *computations* et leurs résultat associés.
De plus par soucis d'intelligibilité et d'unité, tous les exemples seront donnés dans le cadre du contexte formel multi-valué associé au système harmonique 7-tet $\mathbb{T}_7$, qui fut longuement explicité dans la section 2.3.2.

3.1.1 Importation d'un treillis de concept au format standard

Plusieurs outils existent pour engendrer un treillis depuis un tableau binaire de contexte formel, les plus utilisés par la communauté FCA sont : *Con'Exp* et *Galicia*. Cependant, hormis la construction et la visualisation du treillis ces outils ne permettent pas de réaliser le moindre calcul, ni d'extraire des propriétés (même rudimentaires) sur la structure du treillis.
Le point de départ du développement de cette bibliothèque SageMath fut la compatibilité avec les outils utilisés par la communauté FCA, dans le dessein d'une pérennité des données mais aussi pour permettre aux chercheurs de cette communauté de trouver du sens à l'utilisation des outils algorithmiques sur lesquels j'ai travaillé.
La structure du treillis générée par un contexte formel dans *Galicia* peut être extraite dans de nombreux formats, le plus pertinent m'a semblé être ***.lat.xml***, un formatage *xml* de la structure du treillis.
J'ai donc du implémenter en premier lieu un *parser* de fichier *.lat.xml* qui me permette de déduire le contexte formel depuis le fichier *.lat.xml* puis de reconstruire le treillis associé dans la classe *LatticePoset* de SageMath.
La figure ci dessous explicite les étapes de traduction de la structure des données jusqu'à arriver au treillis associé au contexte formel multi-valué, ici $\mathbb{T}_7$ (cf. *Annexe 2*).

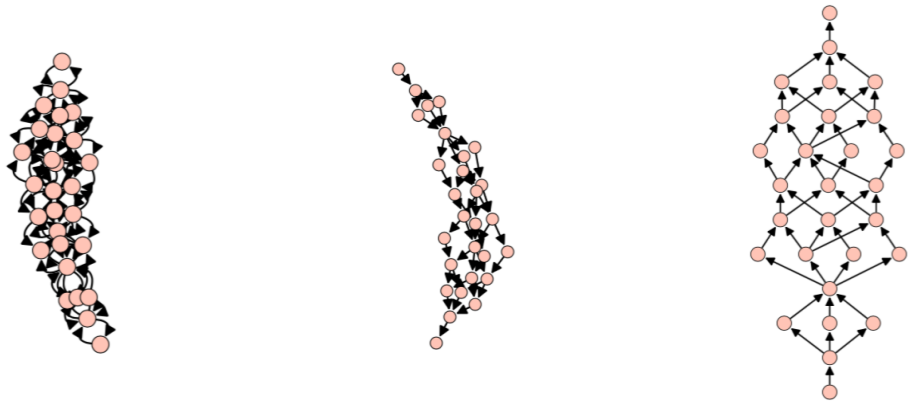


FIGURE 5 – Processus de generation du Treillis

3.1.2 Visualisation des éléments caractéristiques du treillis

Un des avantages de travailler avec des treillis finis de faible cardinalité réside dans le fait que l'on puisse les visualiser, et par là même qu'on puisse visualiser les éléments caractéristiques du treillis, ainsi les résultats de certains calculs sur l'image du treillis lui-même.
De plus, étant donné l'interdisciplinarité interne à mon sujet de recherche - impliquant des acteurs venus du milieu de la musique, compositeurs ou analystes *non mathématiciens* - l'intuition que peut apporter la visualisation des données revêt une importance certaine.

Une fois la structure du treillis générée dans le format *LatticePoset* de SageMath, les concepts du treillis sont nommés en suivant la nomenclature (standard en FCA), suivante :

- Si le concept d'intérêt a est le concept-objet d'un objet $g \in G$ (qu'il contient forcément dans son extent), c-à-d si l'objet g est projeté vers a au sens de la map $p_{\mathbb{C}}$ (**Definition 2**), alors a prend le nom de l'objet g sous la forme $\{g\}/\{\}$.
- Si a est le concept-attribut d'un attribut $m \in M$ (qu'il contient forcément dans son intent), c-à-d si l'attribut m est projeté vers a au sens de la map $q_{\mathbb{C}}$ (**Definition 2**), alors a prend le nom de l'objet m sous la forme $\{\}/\{m\}$.
- Si a est à la fois le concept-attribut d'un attribut $m \in M$ et le concept-objet d'un objet $g \in G$, alors il prend le nom $\{g\}/\{m\}$
- Si a n'est respectivement le concept-objet ou concept-attribut d'aucun objet et d'aucun attribut, alors il prend pour nom un *nombre associé à son ordre de traitement* par l'algorithme de labellisation.

Cette opération de renommage est utile pour la visualisation, car rappelons que l'ordre duquel est équipé un treillis de concept est l'inclusion de l'extent, ce qui le rend vite illisible si on souhaite afficher tous les objets/attributs contenus par chaque concepts.
Une fois le contexte renommé, on cherchera généralement à extraire les ***éléments caractéristiques*** du treillis. Définissons les avant de poursuivre, nous aurons besoin de ces objets dans le suite de ce document.

Definition 6 ***Élément join-irréductibles***. *Un élément a d'un treillis $\mathbb{C}$ est join-irréductible si (i) $a \neq 0_{\mathbb{C}}$ (où $0_{\mathbb{C}}$ dénote le plus petit élément du treillis au sens de la structure d'ordre), et (ii)* $\forall (a,b) \in \mathbb{C}^2, a = b \vee c \Rightarrow a = b$

ou $a = c$. L'ensemble de tous les éléments join-irreductibles de $\mathbb{C}$ est dénoté par $\mathcal{J}(\mathbb{C})$.

Definition 7 ***Élément meet-irréductibles***. *Un élément meet-irréductible est défini par dualité (infimum/supremum), l'ensemble de tous les éléments meet-irréductible de $\mathbb{C}$ est dénoté par $\mathcal{M}(\mathbb{C})$.*

Definition 8 ***Élément double-irréductibles***. *Un éléments est dit double-irréductible dans $\mathbb{C}$ si il est à la fois join-irréductible et meet-irréductible.*

intuitivement les éléments join-irréductible et meet-irréductibles sont en quelques sorte les "*nombres premiers du treillis*", il faut entendre par là que tous les autres éléments du treillis peuvent être exprimés comme le *join* (resp. *meet*) d'un ensemble d'éléments join-irréductible (resp. meet-irréductible). Formellement,

$$\forall a \in \mathbb{C},\ a = \bigvee \{b \in \mathcal{J}(\mathbb{C}) \mid b \preceq a\} = \bigwedge \{b \in \mathcal{M}(\mathbb{C}) \mid a \preceq b\}.$$

La décomposition en élément join-irréductibles (resp. meet-irréductible) d'un élément a dans $\mathbb{C}$ n'est pas unique en général, à moins que le treillis possède des propriétés particulière de distributivité. On entendra dans le reste du rapport la décomposition en élément join-irréductibles (resp. meet-irréductible) dans le sens d'une décomposition minimale comme suit :

Definition 9 ***Décomposition minimale en éléments irréductibles***. *Soit $A, B \subseteq \mathbb{C}$, on dira que A raffine B (écrit $A \ll B$) si pour chaque $a \in A$ il existe $b \in B$ tel que $a \preceq b$. Un ensemble $B \subseteq \mathcal{J}(\mathbb{C})$ est une décomposition en éléments join-irréductibles de a dans $\mathbb{C}$ si et seulement si $\bigvee B = a$. B est dite minimale si chaque décomposition en éléments join-irréductiblesC de a satisfont $C \ll B \implies B \subseteq C$. La décomposition minimale en éléments meet-irréductibles est définit de façon analogue.*

Les éléments join-irréductibles (resp. meet) ont une importance particulière en FCA, en effet par construction les concept-objets et attributs-objets engendrés par les maps $p_{\mathbb{C}}$ et $q_{\mathbb{C}}$ se trouvent être les éléments irréductibles (join/meet/double) du treillis.
La visualisation proposée dans ma bibliothèque de fonction SageMath permet d'un simple regard d'extraire toutes les informations discutées ci-dessus, les trois figures ci-dessous explicite cette affirmation. Les éléments colorés en jaune sur le treillis sont les double-irréductibles, les bleus sont les meet-irréductibles, tandis que les verts sont les join-irréductibles.

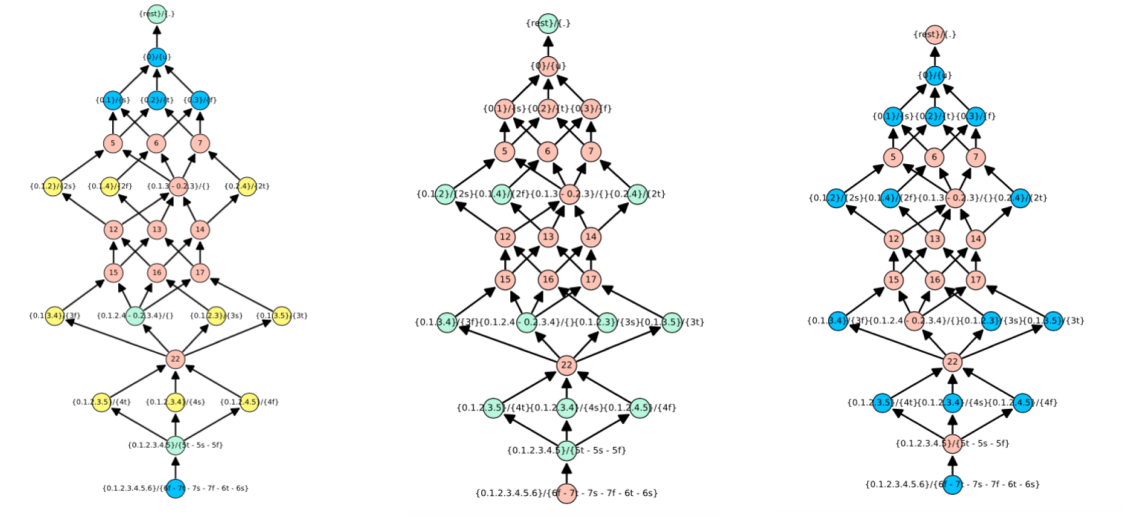


FIGURE 6 – Éléments caractéristiques du treillis de concept multi-valué du 7-tet. *Gauche* - join/meet/double-irréductibles (vert/bleu/jaune). *Centre* - join-irréductibles (vert). *Droite* -meet-irréductibles (bleu)

Deux autres ensembles d'éléments importants issus de la théorie de treillis et que nous allons utiliser par la suite sont : le *filtre* et l'*idéal* engendrés par un concept.

Definition 10 ***Filter et Idéal engendré par un concept***. *Soit un élément $a \in \mathbb{C}$, l'ensemble $F_a = \{b \in \mathbb{C} \mid a \preceq b\}$ est appelé le* filtre principal *généré par l'élément* a. *Il s'agit de de l'up-set (section commençante) non-vide clos par rapport à l'infimum* (meet) *dont le plus grand élément est* a.
L'ensemble $I_a = \{b \in \mathbb{C} \mid b \preceq a\}$ est quant à lui appelé l' idéal principal *généré par le concept a. Il s'agit de du down-set (section finissante) non-vide clos par rapport au supremum* (join) *dont le plus petit élément est* a.

Les deux figures ci dessous, explicitent ces concepts, à gauche l'idéal principal généré par le concept $\{0.3\}/\{f\}$, à droite le filtre principal généré par le concept $\{0.1.2.3.4\}/\{4s\}$.

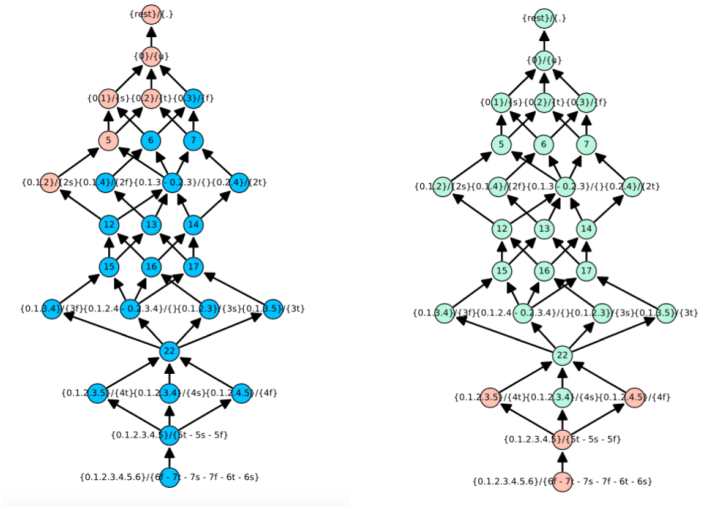


FIGURE 7 – Exemple d'ensembles caractéristiques du treillis de concept multi-valué du 7-tet. *Gauche* - Principal idéal engendré par le concept $\{0.3\}/\{f\}$ (bleu). *Droite* - Principal filtre engendré par le concept $\{0.1.2.3.4\}/\{4s\}$ (vert)

3.1.3 Opérateurs morphologiques sur treillis de concept : Dilatation et Érosion

Pour l'implémentation de ces opérateurs je suis parti des travaux mathématiques rencontrés dans mes nombreuses lectures (notemment [5], [3] et [7]) pour écrire un ensemble d'algorithmes dédiés à la déformation morphologique de treillis de concept.
Il existe plusieurs voies possibles pour définir un couple d'opérateurs morphologiques sur un treillis de concept. L'idée la plus immédiate revient à définir un élément structurant à partir d'une notion de voisinage sur le powerset d'un ensemble. La définition du couple *dilatation, érosion* à partir d'un élément structurant se rapproche de la définition *standard* de la MorphoMath. On pourra également, comme cela fut montré dans [5], définir directement les opérateurs morphologiques depuis un distance définie sur le treillis.
Dans les deux cas (élément structurant, distance), on aura besoin d'une notion de distance et donc d'une métrique sur le treillis de concept.

3.1.4 Métrique sur treillis de concept

Une métrique de treillis de concepts peut elle aussi être mathématiquement définie de plusieurs manière, toutes cohérentes avec le concept topologique de *métrique*.
Soit ω une fonction à valuation réelle sur un treillis de concept $(\mathbb{C}, \preceq)$. alors la fonction d_ω définie par :

$$\forall (a_1, a_2) \in \mathbb{C}^2,\ d_\omega(a_1, a_2) = 2\omega(a_1 \wedge a_2) - \omega(a_1) - \omega(a_2) \tag{9}$$

est une *pseudo-métrique* si et seulement si ω est une valuation supérieure décroissante.
De la même façon on peut montrer que la fonction suivante :

$$\forall (a_1, a_2) \in \mathbb{C}^2,\ d_\omega(a_1, a_2) = \omega(a_1) + \omega(a_2) - 2\omega(a_1 \vee a_2) \tag{10}$$

est une *pseudo-métrique* si et seulement si ω est une valuation inférieure décroissante. À partir de ces pseudo-métriques, on pourra obtenir des métriques sur le treillis de concept en définissant au préalable des valuations ω adapté sur $\mathbb{C}$ (respectant *l'axiome de séparabilité*).

On pourra, là encore formuler des valuations de treillis de concept par plusieurs voies mathématiques, la plus générale possède la forme :

$$\omega_f(a) = \sum_{b \preceq a} f(b) \qquad\qquad \omega^f(a) = \sum_{a \preceq b} f(b)$$

où $\omega_f(a)$ est une valuation supérieure croissante sur le treillis, et $\omega^f(a)$ une valuation inférieure décroissante. À partir d'une fonction réelle ω sur $(\mathbb{C}, \preceq)$, on obtiendra une valuation inférieure si la fonction ω possède la *supermodularité* :

$$\forall (a_1, a_2) \in \mathbb{C}^2, \omega(a_1) + \omega(a_2) \leq \omega(a_1 \wedge a_2) + \omega(a_1 \vee a_2), \tag{11}$$

et de la même façon on obtiendra une valuation supérieure si la fonction ω possède la supermodularité :

$$\forall(a_1, a_2) \in \mathbb{C}^2, \omega(a_1) + \omega(a_2) \geq \omega(a_1 \wedge a_2) + \omega(a_1 \vee a_2) \tag{12}$$

Dans la bibliothèque d'algorithme SageMath, j'ai développé toutes une série de fonctions entièrement dédiées aux métriques sur treillis complet permettant à l'utilisateur de définir rapidement des valuations de treillis de concept, puis de tester les propriétés exposées ci-dessus en vue d'en déduire les métriques associées dans le cas où la valuation remplirait les propriétés structurelles nécessaires.

Un tel outil informatique s'est montré d'une aide précieuse pour la définition algorithmique des opérateurs morphologiques sur treillis - que nous expliciterons dans la section suivante.

À partir des propriétés évoquées ci-dessus on pourra déjà définir plusieurs valuations mathématiquement cohérentes en vue d'en déduire des métriques pour équiper $(\mathbb{C}, \preceq)$. Voici une liste de valuations valides sur n'importe quel treillis de concept $(\mathbb{C}, \preceq)$:

- $\forall a \in \mathbb{C},\ \omega_G(a) = |G| - |e(a)|$ est une valuation supérieure strictement décroissante, où G est l'ensemble des objets, $e(\cdot)$ l'*extent* du concept, et où $|\cdot|$ définit la cardinalité.
- $\forall a \in \mathbb{C},\ \omega_M(a) = |i(a)|$ est une valuation inférieure strictement décroissante, où $i(\cdot)$ est l'*intent* du concept.
- $\forall a \in \mathbb{C},\ \omega_I(a) = |I(a)|$, est une valuation inférieure croissante, où $|I(a)|$ est la cardinalité de l'*idéal* engendré par le concept a.
- $\forall a \in \mathbb{C},\ \omega_F(a) = |F(a)|$, est une valuation inférieure décroissante, où $|F(a)|$ est la cardinalité du *filtre* engendré par le concept a.

Malgré la justesse et l'intérêt mathématique des métriques que l'on peut déduire de ces valuations de concept, leur principal inconvénient dans le cadre de treillis de concepts musicaux est qu'elles n'encodent pas *explicitement* d'information musicalement signifiante. Elles reposent plutôt sur des informations relatives à la structure du treillis ou à la nature du contexte formel ; ce qui dans le cadre d'un contexte formel musical possède cependant déjà un sens musical *implicite* par construction.

Le problème de la définition de valuation musicalement signifiante est une des problématiques de recherches les plus délicates en musicologie computationnelle et s'inscrit d'une manière plus générale dans la problématique de la définition de valuations adaptées au contexte formel d'intérêt en FCA. De telles pistes de recherches dépassent largement le cadre de mon stage et sont à elles seules des sujet de recherche.

Cependant j'ai cherché à définir des valuations encodant directement un sens musical en utilisant la forme générale d'une valuation de treillis : $\omega_f(a)$ ou $\omega^f(a)$.
La fonction f dans $\omega_f(a)$ ou $\omega^f(a)$ possède comme seule contrainte d'être à valuation réelle et permet ici d'injecter librement du sens musical tout en engendrant des métriques mathématiquement correctes.
On peut montrer ([4]) que la structure géométriques des treillis de concepts musicaux dépends de plusieurs facteurs musicalement interprétable. Cependant ces analyses sont locales et entièrement dépendantes du système harmonique d'étude ; si bien qu'une explication musicalement signifiante faite dans le 7-tet n'a plus lieux d'être dans le 8-tet ou le 12-tet. Cependant, certaines informations contenues dans les objets musicaux associés au treillis m'ont semblé *invariantes au système harmonique* et j'ai donc décidé de m'en servir pour définir de nouvelles valuations :

1. La divisibilité mutuelle interne des intervalles d'une forme harmonique influencera son *degré de consonance et de dissonance*. Pour un concept a on pourra donc définir dans $\omega_f(a)$ ou $\omega^f(a)$:

$$f(a) = \begin{cases} |\ i_1|i_2\ | \ \forall(i_1, i_2) \in \phi(p_{\mathbb{C}}^{-1}(a)) & \text{si } p_{\mathbb{C}}^{-1}(a) \text{ existe} \\ \min\limits_{\forall(i_1,i_2)\in\phi(g_{min})} |\ i_1|i_2\ | & \text{sinon} \end{cases} \tag{13}$$

   où i est un intervalle, $|\cdot|$ représente la cardinalité, la notation $i_1|i_2$ signifie i_1 *est un diviseur de* i_2, $p_{\mathbb{C}}^{-1}$ est la map inverse de la map objet-concept $p_{\mathbb{C}}$, $g_{min} = \min_{g\in e(a)} |\ \phi(g)\ |$ et la map $\phi(\cdot)$ projette un objet du contexte formel (c'est à dire une forme harmonique) vers un ensemble contenant tous les intervalles de cette formes harmoniques, par exemple la forme harmonique "0123" est projetée vers l'ensemble $\{0, 1, 2, 3\}$.

2. La somme de la totalité des intervalles contenus au sein d'une forme harmonique constitue une valuation musicale immédiate qui permet de discriminer grossièrement *l'étendue de l'espace harmonique* parcourue

par la forme harmonique. Pour un concept a on pourra donc définir dans $\omega_f(a)$ ou $\omega^f(a)$:

$$f(a) = \begin{cases} \sum_j i_j \ \forall i \in \phi(p_{\mathbb{C}}^{-1}(a)) & \text{si } p_{\mathbb{C}}^{-1}(a) \text{ existe} \\ \min\limits_{\forall i \in \phi(g_{min})} \sum\limits_j i_j & \text{sinon} \end{cases} \tag{14}$$

où i, $|\ \cdot\ |$, g_{min}, $p_{\mathbb{C}}^{-1}$ et $\phi(\cdot)$ sont définis comme précédemment.

3.1.5 Opérateurs morphologique sur treillis de concept

L'utilisation d'opérateurs morphologiques sur treillis de concept est un domaine de recherche très récent (2013), qui n'avait encore jamais été abordé pour les treillis de concepts musicaux. Il existe là aussi plusieurs voies pour définir des opérateurs morphologiques sur treillis de concepts, bien que la majorité des définitions connues d'opérateurs morphologiques sur treillis de concept *furent implémentées* dans la bibliothèque de fonction SageMath, seuls deux couples d'opérateurs - *qui se sont montrés les plus pertinents empiriquement* - seront discutées dans la suite de ce document.

3.1.5.1 Opérateurs morphologique sur $\mathbb{C}$ à partir d'opérateurs morphologiques sur $\mathcal{P}(G)$ et $\mathcal{P}(M)$.

On peut définir un couple d'opérateur morphologique (dilatation, érosion) sur un treillis de concept à partir de la notion morphologique d'*élément structurant*, qui - rappelons le - peut s'entendre comme une relation binaire b que l'on va définir pour les éléments de l'ensemble des objets G du contexte formel.
Pour $g \in G$ on dénotera par $b(g)$ l'ensemble des éléments de G en relation avec g au sens de la relation b. On héritera de la conception de la morphologie mathématique ou b représente un système de *voisinage* dans G, impliquant une distance sur G. Intuitivement, pour une distance d entre éléments de G, un élément structurant représentera un espace de centre g contenant tous les éléments de G à une distance d de g.
La particularité de l'opérateur que l'on va définir ci-dessous est qu'il se sert d'une distance sur $\mathbb{C}$ pour définir une distance sur G à travers la map $p_{\mathbb{C}}$ (**Definition 2**) qui associe à chaque objet son concept-objet. Formellement on a :

Definition 11 ***Métrique sur*** G. *Soit un treillis de concept* $\mathbb{C}$ *et* $d_{\mathbb{C}}$ *une métrique sur* $\mathbb{C}$ *et* $p_{\mathbb{C}}$ *la map qui a chaque objet associe son* concept-objet. *On définit une pseudo-métrique* d *sur* G *par :*

$$\forall (g_1, g_2) \in G^2,\ d(g_1, g_2) = d_{\mathbb{C}}(p_{\mathbb{C}}(g_1), p_{\mathbb{C}}(g_2)) \tag{15}$$

Une fois la métrique d définie on peut définir un *voisinage* ou *élément structurant* pour les éléments de G.

Definition 12 ***Élément structurant dans*** G. *Soit* d *une pseudo-métrique sur* G *définie depuis une distance* $d_{\mathbb{C}}$ *sur* $\mathbb{C}$. *L'élément structurant de taille* n *(*$n \in \mathbb{R}^+$*) pour un objet* $g \in G$ *est défini par :*

$$\begin{aligned} \forall g \in G,\ b(g) &= \{g_1 \in G \mid d(g, g_1) \leq n\} && (16) \\ &= \{g_1 \in G \mid d_{\mathbb{C}}(p_{\mathbb{C}}(g), p_{\mathbb{C}}(g_1)) \leq n\}. && (17) \end{aligned}$$

À partir de la notion d'élément structurant on peut définir les opérateurs de dilatation et d'érosion conformément aux définitions de la morphologie mathématique d'une dilatation/érosion par élément structurant, et qui de plus respecte la définition des opérateurs morphologique de treillis défini au chapitre.... Étant donné que la notion de voisinage définie ci-dessus est signifiante uniquement dans G, nous commencerons par définir les opérateurs *dilatation* et *érosion* sur le treillis $(\mathcal{P}(G), \subseteq)$, où $\mathcal{P}(G)$ est le powerset de l'ensemble des objets G du contexte formel et $\subseteq$ est l'ordre naturel induit par l'inclusion.

Definition 13 ***Dilatation et Érosion morphologique sur*** $(\mathcal{P}(G), \subseteq)$. *La dilatation morphologique d'un sous-ensemble* X *de* G *relativement à l'élément structurant* b *est définie par :*

$$\delta_{b^n}(X) = \{g \in G \mid b^n(g) \cap X \neq \emptyset\}, \tag{18}$$

L'érosion morphologique d'un sous-ensemble X *de* G *relativement à l'élément structurant* b *est définie par :*

$$\varepsilon_{b^n}(X) = \{g \in G \mid b^n(g) \subseteq X\}. \tag{19}$$

Les même définitions peuvent être étendue au treillis $(\mathcal{P}(M), \subseteq)$ (où $\mathcal{P}(M)$ est le powerset de l'ensemble des attributs M du contexte formel et $\subseteq$ est l'ordre naturel induit par l'inclusion ensembliste) en remplaçant simplement la map concept-objet $p_{\mathbb{C}}$ par la map attribut-objet $q_{\mathbb{C}}$.
En utilisant une relation de voisinage ces définitions découlent naturellement des définitions standard de la

morphologie mathématique et possèdent donc toutes les propriétés classiques d'une dilatation et d'une érosion.

Souvenons nous que notre but est de définir des opérateurs de dilatation/érosion sur le treillis de concept lui même et non sur les powersets des ensembles des objets/attributs du contexte. Il se trouve qu'à partir des opérateurs sur $\mathcal{P}(G)$ et $\mathcal{P}(M)$, on peut dériver des opérateurs morphologiques sur $\mathbb{C}$ - compatibles avec la structure d'un treillis de concept - de la façon suivante :

Definition 14 ***Dilatation et Érosion morphologiques sur*** $\mathbb{C}$. *Soit* $(\mathbb{C}, \preceq)$ *un treillis de concept avec le couple d'opérateur de dérivation* (α, β) *associé. Soit* δ_{b^n} *et* ε_{b^n} *une dilatation et une érosion par élément structurant* b *de voisinage* n *définit sur* $\mathcal{P}(G)$. *La dilatation et l'érosion sur* $\mathbb{C}$ *sont définis par :*

$$\forall a \in \mathbb{C},\ \delta(a) = (\beta\alpha(\delta_{b^n}(e(a))), \alpha(\delta_{b^n}(e(a)))) = (\beta\alpha(\delta_{b^n}(e(a))), \alpha(\delta_{b^n}(\beta(i(a)))))$$
$$\forall a \in \mathbb{C},\ \varepsilon(a) = (\beta\alpha(\varepsilon_{b^n}(e(a))), \alpha(\varepsilon_{b^n}(e(a)))) = (\beta\alpha(\varepsilon_{b^n}(e(a))), \alpha(\varepsilon_{b^n}(\beta(i(a)))))$$

où $e(a)$ *et* $i(a)$ *représentent respectivement l'extent et l'intent du concept* a.

L'utilisation des opérateurs de dérivation assure que δ et ε soient bien de la forme : $\delta : \mathbb{C} \to \mathbb{C}$ et $\varepsilon : \mathbb{C} \to \mathbb{C}$, c'est à dire que le résultat de la dilatation ou de l'érosion soit bien un concept du contexte formel associé au treillis de concept $\mathbb{C}$.
Sans perte de généralité, les définitions des opérateurs morphologiques définis ci-dessus peuvent être formulées à partir d'élément structurant b de voisinage n définit sur $\mathcal{P}(M)$ en remplaçant simplement la map concept-objet $p_{\mathbb{C}}$ par la map concept-attribut $q_{\mathbb{C}}$ dans la distance $d_{\mathbb{C}}$ utilisée pour engendrer l'élément structurant, et en remplaçant $e(a)$ par $i(a)$ et $i(a)$ par $e(a)$ dans les définitions de δ et ε.

3.1.5.2 Opérateurs morphologique sur $\mathbb{C}$ à partir de Distances sur $\mathbb{C}$.

On pourra également définir des opérateurs morphologiques directement sur le treillis de concept $\mathbb{C}$ en utilisant le fait que $\mathbb{C}$ est *sup-généré* par n'importe quel sous-ensemble d'élément *join-irreductibles* et *inf-généré* par n'importe quel sous-ensemble d'élément *meet-irreductibles*. Définissons d'abord formellement ces notions.

Definition 15 ***Ensemble sup-générateur et inf-générateur d'un treillis***. *Soit* L *un treillis et* $X \subset L$ *un sous-ensemble des éléments du treillis. On dira que* X *est un sup-générateur de* L *si chaque élément* $a \in L$ *est le* supremum *des éléments de* X *qu'il majore, c-à-d :*

$$\forall a \in L,\ a = \bigvee \{x \in X \mid x \leq a\}$$

On dira que X *est un inf-générateur de* L *si chaque élément* $a \in L$ *est* l'infimum *des éléments de* X *qu'il minore, c-à-d :*

$$\forall a \in L,\ a = \bigwedge \{x \in X \mid x \geq a\}$$

On peut montrer que tous sous-ensemble d'éléments *join-irreductibles* d'un treillis est un *sup-générateur* du treillis et que tous sous-ensemble d'éléments *meet-irreductibles* d'un treillis est un *inf-générateur* du treillis.

Soit un treillis de concept $\mathbb{C}$, étant donné que $\mathbb{C}$ est sup-généré par les sous-ensembles de $\mathcal{J}(\mathbb{C})$ (ensemble des éléments join-irréductibles) il est suffisant de définir une dilatation δ' sur $\mathcal{J}(\mathbb{C})$ et de dériver ensuite une dilatation pour n'importe quel concept $a \in \mathbb{C}$ en prenant le *supremum de la dilatation* δ' de la décomposition minimale en éléments join-irréductibles de a, qu'on dénotera par $\mathcal{J}(a)$. Cette définition fournit les bonnes propriétés pour une dilatation sur treillis en tant d'opérateur qui commute avec le *supremum* étant donné que $\mathbb{C}$ est sup-généré par les sous-ensembles de $\mathcal{J}(\mathbb{C})$.

Une dilatation sur $\mathbb{C}$ utilisant cet algorithme sera donc définie en deux étapes.

Definition 16 ***Dilatation d'élément join-irréductible basée sur une distance sur*** $\mathbb{C}$. *Soit* $\mathbb{C}$ *un treillis de concept et* $\mathcal{J}(\mathbb{C})$ *l'ensemble des éléments join-irréductibles de* $\mathbb{C}$. *Une dilatation d'élément join-irréductible basée sur une distance sur* $\mathbb{C}$ *se définit comme suit :*

$$\forall a \in \mathcal{J}(\mathbb{C}),\ \delta'(a) = \bigvee \{b \in \mathbb{C} \mid d(a, b) \leq n\}$$

On peut alors dériver une dilatation sur $\mathbb{C}$ en utilisant le propriété de *sup-génération* et le fait qu'une dilatation sur treillis commute avec le *supremum*.

Definition 17 ***Dilatation de concept par décomposition en éléments join-irréductibles****. Soit* $\mathbb{C}$ *un treillis de concept et* $\mathcal{J}(\mathbb{C})$ *l'ensemble des éléments join-irreductibles de* $\mathbb{C}$*. Soit* $a \in \mathbb{C}$ *un concept et* $\mathcal{J}(a)$ *sa décomposition minimale en éléments join-irréductibles. Soit* δ' *une dilatation définie sur* $\mathcal{J}(\mathbb{C})$*. On définira alors une dilatation pour n'importe quel concept* $a \in \mathbb{C}$ *par la map suivante :*

$$\delta_d : \left| \begin{array}{ccc} \mathbb{C} & \longrightarrow & \mathbb{C} \\ a & \longmapsto & \bigvee_{b \in \mathcal{J}(a)} \delta'(b) \end{array} \right. \tag{20}$$

Notons que lorsque a est *join-irréductible* $\delta_d(a) = \delta'(a)$.

Avec la même philosophie mais en utilisant le fait que *l'érosion commute avec l'infimum* et que $\mathbb{C}$ est *inf-généré* par les sous-ensemble $\mathcal{M}(\mathbb{C})$, où $\mathcal{M}(\mathbb{C})$ dénote l'ensemble des éléments *meet-irréductibles* de $\mathbb{C}$, on pourra définir l'opérateur d'érosion pour n'importe quel concept $a \in \mathbb{C}$. Définissons d'abord l'érosion d'élément meet-irréductibles.

Definition 18 ***Érosion d'élément meet-irréductible basée sur une distance sur*** $\mathbb{C}$*. Soit* $\mathbb{C}$ *un treillis de concept et* $\mathcal{M}(\mathbb{C})$ *l'ensemble des éléments meet-irréductibles de* $\mathbb{C}$*. Une érosion d'élément meet-irréductible basée sur une distance sur* $\mathbb{C}$ *se définit comme suit :*

$$\forall a \in \mathcal{M}(\mathbb{C}),\ \varepsilon'(a) = \bigwedge \{b \in \mathbb{C} \mid d(a,b) \leq n\}$$

Généralisons l'érosion à tout concept du treillis en utilisant la décomposition minimale en éléments meet-irréductible $\mathcal{M}(a)$ d'un concept $a \in \mathbb{C}$.

Definition 19 ***Érosion de concept par décomposition en éléments meet-irréductibles****. Soit* $\mathbb{C}$ *un treillis de concept et* $\mathcal{M}(\mathbb{C})$ *l'ensemble des éléments meet-irreductibles de* $\mathbb{C}$*. Soit* $a \in \mathbb{C}$ *un concept et* $\mathcal{M}(a)$ *sa décomposition minimale en éléments meet-irréductibles. Soit* ε' *une érosion définie sur* $\mathcal{M}(\mathbb{C})$*. On définira alors une érosion pour n'importe quel concept* $a \in \mathbb{C}$ *par la map suivante :*

$$\varepsilon_d : \left| \begin{array}{ccc} \mathbb{C} & \longrightarrow & \mathbb{C} \\ a & \longmapsto & \bigwedge_{b \in \mathcal{M}(a)} \varepsilon'(b) \end{array} \right. \tag{21}$$

3.1.5.3 Algorithmique et exemple de déformation morphologique sur treillis de concepts musicaux L'algorithmique liée à ces opérateurs et à toutes les briques élémentaires sur lesquelles ils se basent (*distance de treillis, distance sur les powerset* $\mathcal{P}(G)$ *et* $\mathcal{P}(M)$*, calcul d'éléments structurants, décomposition en éléments join/mett-irréductibles, opérateurs de dérivation, etc.*) font partie des algorithmes présents dans le code que j'ai développé en vue d'une contribution au projet SageMath. Ces algorithmes sont à ma connaissance et à celle de mes maîtres de stage la première implémentation informatique de ces outils combinant FCA et Morphologie Mathématique dans un logiciel de calcul algébrique.

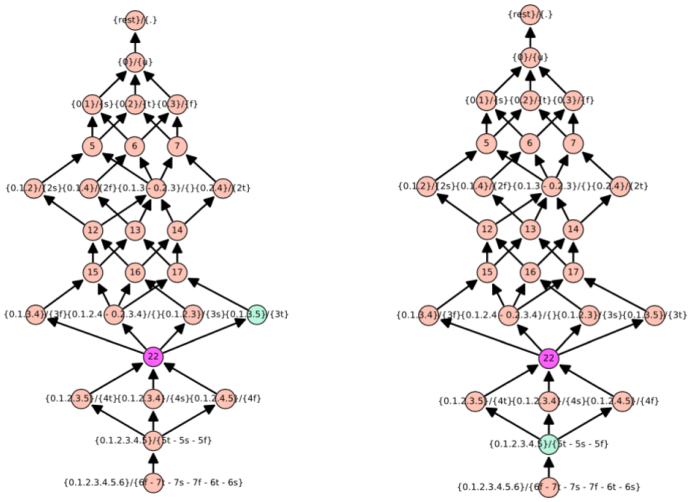


FIGURE 8 – Déformation morphologique du concept harmonique labellisé "22". *Gauche* - dilatation par décomposition en éléments join-irréductibles. *Droite* - érosion par décomposition en élément meet-irréductibles

La Figure [8] illustre respectivement la *dilatation par décomposition en éléments join-irréductibles* (gauche) et l'*érosion par décomposition en élément meet-irréductibles* (droite) du concept harmonique labellisé "22" (respectivement à la nomenclature de labellisation explicitée dans la section 3.1.2) sur le treillis de concept associé au contexte formel multi-valué du système harmonique 7-tet $\mathbb{T}_7$.

D'autres outils algébriques et morphologiques furent développés dans le dessein de disposer d'un outil d'expérimentation et d'analyse systématique des implications de la morphologie mathématique sur les treillis de concept musicaux, mais la brièveté du présent document ne me permet pas de les discuter. Le code écrit est cependant fourni en annexe si le lecteur de ce rapport souhaite s'y référer.

3.1.6 Congruence de treillis de concept et treillis quotient modulo congruence

Je me suis efforcé durant ce stage de ne pas perdre de vue les problèmes-type de la communauté de la musicologie computationnelle dans mon investigation mathématique et mes choix d'implémentation algorithmique. Les problématiques qui suscitent le plus d'engouement à l'IRCAM dans l'orientation des recherches sur l'analyse symbolique de la musique relèvent de la reconnaissance des progressions harmoniques. Il s'agit d'une étape préalable pour envisager le problème plus général de *l'analyse et de la classification automatique du style musical dans un oeuvre.*
L'utilisation des outils provenant de la Théorie des Treillis, de la Morphologie Mathématique et de la FCA étant une piste jusqu'à lors vierge pour aborder d'un point de vue algébrique les problèmes de la musicologie computationnelle, j'ai proposé une toute nouvelle méthode utilisant l'union de ces outils et permettant d'envisager la reconnaissance de motif harmonique dans une perspective d'analyse symbolique de l'écriture harmonique.

Après avoir travailler pendants plusieurs mois et affiné mon intuition en Théorie des Treillis, j'ai croisé dans mes lectures théorique la notion de ***congruence de treillis***, qui a immédiatement suscitée mon intérêt. Dans un premier temps pour des raisons purement esthétiques étant donné les liens qu'entretient le concept de *congruence* avec des théorèmes fascinants en *Algèbre Universelle*. J'ai alors décidé d'étudier le sens musical des congruences sur les treillis de concepts musicaux, riche de ma compréhension des opérateurs morphologiques sur treillis.
La congruence, est un objet algébrique, intimement lié à un des théorème les plus importants de de l'*Algèbre Universelle* : le *Théorème d'Homomorphisme* (parfois appeler premier Théorème d'Isomorphisme). Le plus souvent le concept de congruence est introduit en Théorie des Groupes, mais, appartenant à la théorie générale de l'Algèbre, ce concept se transpose sans peine à toutes les classes de structures algébrique, dont les treillis.

Définissons une congruence de treillis, en partant des homomorphismes de treillis (morphisme qui préserve la structure d'ordre du treillis) auxquels cette notion est liée.
Soit $(L, \preceq)$ et $(K, \preceq)$ deux treillis, et soit $f : L \to K$ un homomorphisme de treillis. Alors la relation d'équivalence θ définie sur L par :

$$(\forall a, b \in L)\ a \equiv b\ (mod\ \theta) \iff f(a) = f(b) \tag{22}$$

est compatible avec le *join* et le *meet*.

Definition 20 ***Relation d'équivalence sur treillis***. *Une relation d'équivalence θ sur un treillis $(L, \preceq)$ est dite* **compatible avec le join et le meet**, *si $\forall a, b, c, d \in L$:*

$$a \equiv b\ (mod\ \theta)\ et\ c \equiv d\ (mod\ \theta)$$

implique,

$$a \vee c \equiv b \vee d\ (mod\ \theta)\ et\ a \wedge c \equiv b \wedge d\ (mod\ \theta)$$

Definition 21 ***Congruence de treillis***. *Une relation d'équivalence θ sur un treillis L* compatible avec le join et le meet *est une* congruence *sur L.*

Le concept de congruence se trouve ainsi au coeur de la définition d'un morphisme de treillis qui préserve sa structure d'ordre *homomorphisme de treillis.* En effet si L et K sont des treillis et que $f : L \to K$ est un homomorphisme de treillis, alors la congruence θ sur L associée à f, est le **kernel** de l'homomorphisme f, noté $\ker f$.

Aussi, il est à noter qu'une congruence θ sur L engendre une partition du treillis L en ensembles non-vides disjoints nommés ***blocks***. Ces sous-ensemble de L sont les classes d'équivalence de θ, un *block* a la forme $[a]_\theta := \{x \in L \mid x \equiv a\ (mod\ \theta)\}$. Enfin il peut être prouvé ([3]) que si la relation d'équivalence θ est une congruence sur L, alors les bocks engendrés par θ sont des sous-treillis, c'est à dire qu'ils respectent tous les

axiomes d'un treillis, que le *meet* et me *join* sont bien définis sur eux et qu'ils sont équipés du même ordre que le treillis dont ils sont issus.

Une fois calculé, la congruence de treillis - générée de telle manière à ce qu'un ensemble de concept appartienne au même block de congruence - permet d'accéder à de nombreux objets algébriques tout à fait pertinents quant à l'essence de la structure associée à cette partition du treillis, notamment :

— Treillis-Quotient modulo congruence
— Algèbre des sous-treillis engendré par les blocks de la congruence
— Treillis des congruences du treillis

Avant de donner un exemple d'application musicale de la l'utilité des *congruence de treillis* pour la construction de descripteurs harmonico-algérbiques, il faut définir un autres objets sur lequel repose la construction des descripterus harmonico-algérbiques. Il s'agit des ***Treillis quotient*** *modulo congruence.*

Definition 22 ***Treillis Quotient (mod*** θ***).*** *Soit* θ *une relation d'équivalence sur un treillis* L*, on pourra définir les opérations* $\vee$ *et* $\wedge$ *sur l'ensemble* $L/\theta = \{[a]_\theta \mid a \in L\}$ *de blocks du treillis de la façon naturelle suivante :*

$$[a]_\theta \vee [b]_\theta := [a \vee b]_\theta \ \ et \ \ [a]_\theta \wedge [b]_\theta := [a \wedge b]_\theta$$

Il peut être montré que ces opérations sont bien définies lorsqu'elles sont indépendantes des éléments choisis pour représenter les classes d'équivalence, autrement dit $\vee$ *et* $\wedge$ *sont bien définis sur* L/θ *lorsque* θ *est une congruence. Lorsque* θ *est une congruence on appellera* $(L/\theta, \vee, \wedge)$ ***le treillis quotient de*** L ***modulo*** θ*.*

De là suit un des théorèmes les plus importants de la théorie des treillis, sur lequel repose notre méthode :

Theoreme 2 *Théorème d'Homomorphisme de Treillis. Soit* L *et* K *deux treillis, soit* f *un homomorphisme de* L *dans* K *et* $\theta = \ker f$*. Alors la map* $g : L/\theta \to K$ *donnée par* $g([a]_\theta) = f(a)$ $\forall [a]_\theta \in L/\theta$ *est bien définie, c'est à dire :*

$$(\forall a, b \in L)\ [a]_\theta = [b]_\theta \iff g([a]_\theta) = g([b]_\theta)$$

Ce qui revient à dire que g *est un* ***isomorphisme*** *entre* L/θ *et* K*. De plus si* q *dénote la map quotient alors* $\ker q = \theta$*.*

Les outils algorithmiques pour générer/visualiser la *congruence* θ générée par un ensemble de concept formel sur un treillis de concept $\mathbb{C}$, ainsi que le treillis-quotient $(\mathbb{C}/\theta, \vee, \wedge)$ ont étés implémentés et abondamment explorés.

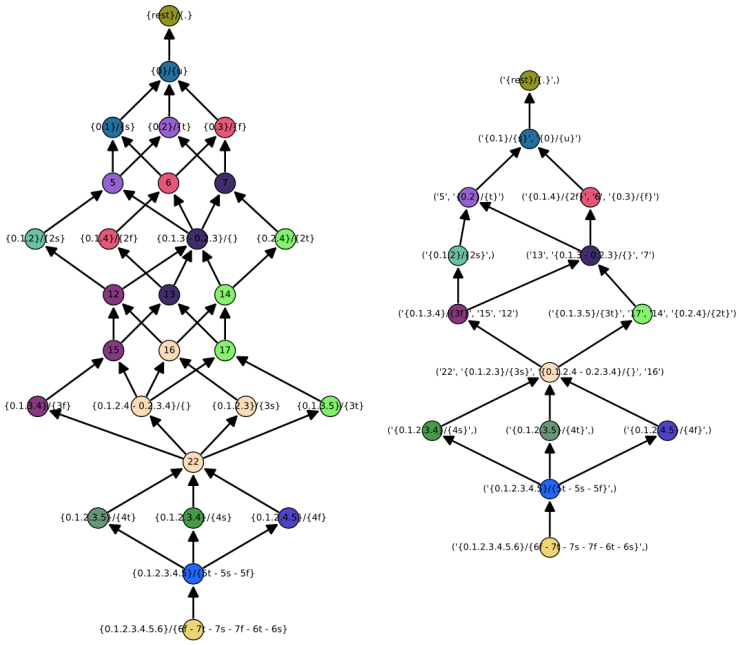


FIGURE 9 – *Gauche* - Congruence θ sur $\mathbb{C}$(7-tet). *Droite* - Treillis quotient $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$

Dans la Figure [9] on peut observer à gauche la congruence générée par le couple de formes harmoniques le plus utilisé de la musique instrumentale, à savoir $[0, 2, 4]$ et $[0, 1, 3, 5]$ représentant respectivement les formes harmoniques des accords parfaits majeurs/mineurs et des accords de $7^{\text{ème}}$ majeurs/mineurs. Intuitivement on

peut comprendre θ comme une partition du treillis de concept $\mathbb{C}$(7-tet) compatible avec sa structure d'ordre dans laquelle les deux formes harmoniques $[0, 2, 4]$ et $[0, 1, 3, 5]$ appartiennent à la même classe d'équivalence. Tous les concepts du treillis qui partagent la même couleur appartient à la même classe d'équivalence (au même block de congruence).
À droite on observe le treillis quotient $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$, on comprend ici plus intuitivement le *Théorème d'Homomorphisme de Treillis*, où la congruence θ est le *kernel* de l'homomorphisme de treillis qui projette les concepts de $\mathbb{C}$(7-tet) vers les éléments du treillis quotient $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$. L'ordre duquel est équipé $\mathbb{C}$(7-tet) est préservé par ce morphisme, au sens de la **Definition 20**. L'ensemble des concepts de chaque block de θ (ceux qui partagent la même couleur sur $\mathbb{C}$(7-tet)) sont projetés vers un unique concept les contenant dans le treillis quotient modulo θ.

On peut déjà comprendre l'intérêt de cet objet algébrique qui "*résume*" structurellement la manière dont un ensemble de formes harmoniques ***partitionne*** le système harmonique duquel elles sont issues tout en *préservant sa structure*.

À partir du treillis-quotient $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$ ainsi obtenu on peut déjà déduire - d'une manière jamais encore abordée en analyse symbolique de la musique - un certains nombres de structures harmoniques liées à la partition du système harmonique 7-tet par les concepts harmoniques générateurs de θ. En effet à partir de la structure du treillis-quotient modulo θ on pourra rechercher les *sous-treillis de* $\mathbb{C}$(7-*tet*) isomorphes à $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$. Musicalement cela correspond à tous les sous-systèmes harmoniques de $\mathbb{C}$(7-tet) ***compatibles avec la structure du système harmonique engendré par la partition structurée de*** $\mathbb{C}$(7-***tet***) ***relativement à un ensemble de formes harmoniques*** (ici les accords parfaits $[0, 2, 4]$ et les accords de $7^{\text{ème}}$ $[0, 1, 3, 5]$).
La Figure [10] montre deux *sous-treillis* de $\mathbb{C}$(7-tet) (deux sous-système harmoniques de 7-tet) isomorphes à $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$. L'utilité d'un tel objet symbolique pour la composition musicale et/ou l'analyse me parait certaine. Après plusieurs échanges avec mes encandrants pédagogiques, il est d'ailleurs probable que ces objets soient le point de départ de nouvelles recherches à l'IRCAM visant à explorer en profondeur leur conséquences harmoniques.

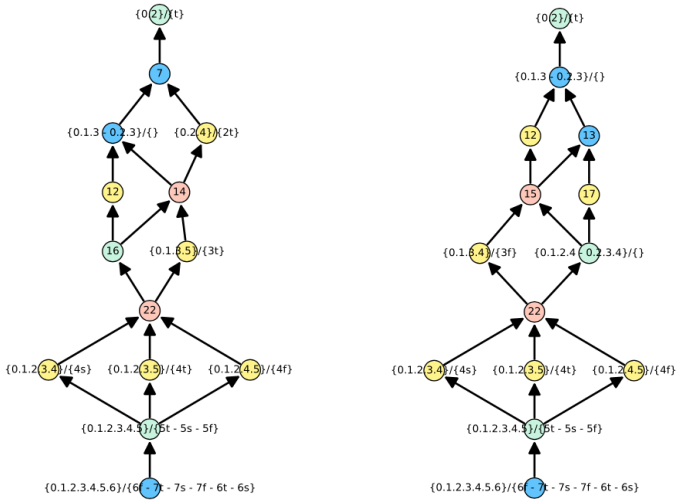


FIGURE 10 – Sous-treillis de $\mathbb{C}$(7-tet) isomorphes à $(\mathbb{C}(7\text{-tet})/\theta, \vee, \wedge)$

La procédure de coloration des concepts ainsi que leurs labels est similaire à celle explicitée dans la section 3.1.2. Ainsi les noeuds associés aux concepts double-irréductibles sont jaunes, aux join-irréductibles sont verts et aux meet-irréductibles sont bleus.

Un des *treillis-quotient modulo congruence* du 7-tet les plus intéressants s'obtient lorsqu'on cherche à quotienter le 7-tet par une congruence $\theta_\star$ dans laquelle :

- les formes harmoniques **tonales** les plus usitées (accords parfaits, accords de $7^{\text{ème}}$, accords de $9^{\text{ème}}$) vivent dans la même classe d'équivalence
- les formes harmoniques génératrices *des accords de quarte* chers à Schoenberg et à ses disciples (Berg, Webern) de la musique **atonale** viennoise, vivent dans une autres même classe d'équivalence.

La congruence $\theta_\star$ et le treillis-quotient $(\mathbb{C}(7\text{-tet})/\theta_\star, \vee, \wedge)$ sont représentés dans la Figure [11].

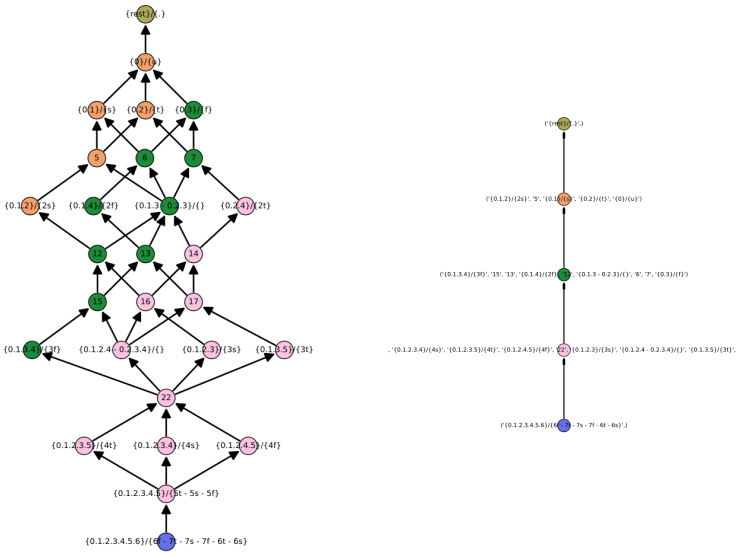


FIGURE 11 – *Gauche* - Congruence $\theta_\star$ sur le treillis de concept du 7-tet. *Droite* - Treillis quotient du treillis associé au 7-tet *modulo* $\theta_\star$

$(\mathbb{C}(\text{7-tet})/\theta_\star, \vee, \wedge)$ est une ***chaîne***, un ordre "*parfait*", la forme la plus essentielle d'un treillis.

4 Méthode de reconnaissance des formes harmoniques par Treillis quotient modulo congruence morphologique

L'algorithme mis au point ici est une méthode générale et hautement paramétrable qui permet de travailler à différente granularité temporelle pour la reconnaissance de formes harmoniques au sein d'une oeuvre, mais qui peut aussi tout à fait être utilisée pour comparer des oeuvres de compositeurs différents.

L'innovation de cette méthode d'analyse musicologique réside non seulement dans le fait qu'elle considère le matériel musical par le biais d'objets algébriques et symboliques nouveaux dans le domaine de la musicologie computationnelle (Morphologie Mathématique sur treillis de concept, congruence de treillis, treillis-quotient modulo congruence, sous-treillis isomorphes au treillis-quotient) mais surtout qu'elle propose un nouveau type de descripteurs harmoniques basés sur ces objets algébriques. Intuitivement ces descripteurs résument dans un ***objet structuré*** *l'espace harmonique parcouru par le passage de l'oeuvre à analyser.*
Il est a noter également que la compacité de ces descripteurs permet de travailler algorithmiquement avec un grand nombres d'entre eux, et de les comparer aisément pour trouver des similitudes de motifs harmoniques au sein d'une oeuvre ou entre différentes oeuvres.

4.1 Descripteurs harmonico-morphologiques

Pour un soucis de brièveté et de clarté d'exposition dans ce qui va suivre on introduit la notation suivante : On dénote par $\mathfrak{M}$ la *phrase musicale* ou *le passage de l'oeuvre* à partir duquel on souhaite générer un descripteur. Aussi on dénotera par $\mathfrak{M}$-tet ou $\mathbb{T}_{\mathfrak{M}}$ le système harmonique associé à $\mathfrak{M}$, c'est à dire le $\mathcal{O}$-tet $\mathbb{T}_{\mathcal{O}}$ dans lequel on souhaite effectuer l'analyse. On dénotera également par $\mathbb{C}(\mathfrak{M})$ le treillis de concept associé au contexte formel multi-valué issu du système harmonique $\mathfrak{M}$-tet $\mathbb{T}_{\mathfrak{M}}$. Le plus souvent il s'agira du système harmonique 7-tet ou 12-tet.
L'**Algorithm 1** expose la procédure de génération des descripteurs harmoniques pour un fragment donnée d'une oeuvre musicale.

Le fait d'opérer une dilatation et une érosion morphologique sur l'ensemble des concepts $H^{\mathfrak{M}}_{\mathbb{C}}$ permet d'obtenir un "*intervalle*" de descripteurs harmoniques $[\mathbb{C}(\mathfrak{M})/\theta_\delta, \mathbb{C}(\mathfrak{M})/\theta_\varepsilon]$ où $\mathbb{C}(\mathfrak{M})/\theta_\delta$ fait office de *descripteur sup* et $\mathbb{C}(\mathfrak{M})/\theta_\varepsilon$) de *descripteur inf.*
Les trois treillis-quotients ainsi obtenus représentent - relativement au théorème d'Homomorphisme - les codomaines d' homomorphismes de treillis dont les *kernels* sont les congruences engendrées par :
— l'ensemble des concept formel $H^{\mathfrak{M}}_{\mathbb{C}}$ associé aux formes harmoniques présentes dans $H^{\mathfrak{M}}$
— l'ensemble dilaté $\delta(H^{\mathfrak{M}}_{\mathbb{C}})$ des concepts $H^{\mathfrak{M}}_{\mathbb{C}}$
— l'ensemble érodé $\varepsilon(H^{\mathfrak{M}}_{\mathbb{C}})$ des concepts $H^{\mathfrak{M}}_{\mathbb{C}}$.

Algorithm 1 Génération d'un intervalle morphologique de descripteurs harmoniques (algébriques) par Treillis quotient modulo congruence

Require: $\mathbb{C}(\mathfrak{M})$ est un treillis de concept (contexte formel multi-valué) issu du système harmonique $\mathbb{T}_{\mathfrak{M}}$
Require: $H^{\mathfrak{M}}$ est l'ensemble des formes harmoniques présentes dans $\mathfrak{M}$
Require: ω est une valuation valide sur $\mathbb{C}(\mathfrak{M})$
Require: $\mathbb{M}_{\omega}$ est la métrique associée à ω sur $\mathbb{C}(\mathfrak{M})$
Require: (δ, ε) est un couple d'opérateurs morphologiques de treillis formant une *adjonction* sur $\mathbb{C}(\mathfrak{M})$ et utilisant la métrique $\mathbb{M}_{\omega}$
Require: n est la distance de voisinage conceptuel à considérer lors des opérations morphologiques
Ensure: $\mathbb{C}(\mathfrak{M})/\theta$, $\mathbb{C}(\mathfrak{M})/\theta_{\delta}$ et $\mathbb{C}(\mathfrak{M})/\theta_{\varepsilon}$ sont les descripteurs harmonico-morphologiques de $\mathfrak{M}$
1: **function** DESCRIPTEURS HARMONICO-MORPHOLOGIQUES($\mathbb{C}(\mathfrak{M})$, $H^{\mathfrak{M}}$, (δ, ε), n)
2: Calculer l'ensemble des concept formel $H^{\mathfrak{M}}_{\mathbb{C}}$ associé aux formes harmoniques présentes dans $H^{\mathfrak{M}}$
3: Calculer la congruence θ sur $\mathbb{C}(\mathfrak{M})$ engendrée de telle manière que tous les concepts présents dans l'ensemble $H^{\mathfrak{M}}_{\mathbb{C}}$ appartiennent au même block (classe d'équivalence) de congruence $[\cdot]_{\theta}$
4: Calculer la dilatation $\delta(H^{\mathfrak{M}}_{\mathbb{C}})$ et l'érosion $\varepsilon(H^{\mathfrak{M}}_{\mathbb{C}})$ de l'ensemble des concepts formels $H^{\mathfrak{M}}_{\mathbb{C}}$ dans un voisinage de taille n
5: Calculer la congruence θ_{δ} sur L engendrée de telle manière que tous les concepts présents dans l'ensemble $\delta(H^{\mathfrak{M}}_{\mathbb{C}})$ appartiennent au même block (classe d'équivalence) de congruence $[\cdot]_{\theta_{\delta}}$
6: Calculer la congruence θ_{ε} sur L engendrée de telle manière que tous les concepts présents dans l'ensemble $\varepsilon(H^{\mathfrak{M}}_{\mathbb{C}})$ appartiennent au même block (classe d'équivalence) de congruence $[\cdot]_{\theta_{\varepsilon}}$
7: Calculer les treillis-quotient $\mathbb{C}(\mathfrak{M})/\theta$, $\mathbb{C}(\mathfrak{M})/\theta_{\delta}$ et $\mathbb{C}(\mathfrak{M})/\theta_{\varepsilon}$ respectivement modulo θ, θ_{δ} et θ_{ε} depuis L
8: **return** $(\mathbb{C}(\mathfrak{M})/\theta, \mathbb{C}(\mathfrak{M})/\theta_{\delta}, \mathbb{C}(\mathfrak{M})/\theta_{\varepsilon})$

Musicalement $H^{\mathfrak{M}}_{\mathbb{C}}$ représente l'étendu de l'espace harmonique parcouru par $\mathfrak{M}$, $\delta(H^{\mathfrak{M}}_{\mathbb{C}})$ et $\varepsilon(H^{\mathfrak{M}}_{\mathbb{C}})$ représente respectivement une dilatation et une érosion de cet espace, qui comme on l'a dit sont à comprendre comme une *borne supérieure* et une *borne inférieure* qui seront utiles pour la comparaison avec d'autres passages de l'oeuvre.
Les congruences de treillis θ, θ_{δ} et θ_{ε} engendrées par ces deux ensembles sont à entendre comme des objets représentant la manière dont l'espace harmonique parcouru par $\mathfrak{M}$ engendre une séries de partition (inférieure, neutre et supérieure) compatible avec la structure intervallique de $\mathbb{T}_{\mathfrak{M}}$.
Quant aux treillis-quotients $[\mathbb{C}(\mathfrak{M})/\theta, \mathbb{C}(\mathfrak{M})/\theta_{\delta}, \mathbb{C}(\mathfrak{M})/\theta_{\varepsilon}]$ qui définissent nos descripteurs harmoniques, ils contiennent plusieurs informations discriminantes relatives à l'extrait d'oeuvre analysé :

— ils représentent des *motifs* de sous-système harmoniques compatibles avec la structure d'ordre intervallique de $\mathbb{T}_{\mathfrak{M}}$.

— ils forment un *résumé structurel* de la façon dont *l'espace harmonique* parcouru par $\mathfrak{M}$ partitionne $\mathbb{T}_{\mathfrak{M}}$ et ce tout en respectant son ordre conceptuel.

Ces descripteurs se trouvent être des objets beaucoup plus compacts (treillis quotienté) que les congruences de treillis, et pourtant tout aussi signifiants en tant que descripteur structurel relativement au treillis harmonique de départ.
Aussi Le fait de disposer d'un *descripteur sup.* et d'un *descripteur inf.* permettra de définir un intervalle de "*similitude*" harmonique utile lors de la comparaison des descripteurs de deux phrases musicales différentes.

La principale difficulté dans la génération de ces descripteurs réside d'une part dans le choix d'une valuation ω du treillis *pertinente musicalement*, et d'autre part dans le choix du couple d'opérateur morphologiques (δ, ε) ; qui comme on l'a vu peut être défini sur $\mathbb{C}(\mathfrak{M})$ de diverses manières. En effet du choix de ω et de (δ, ε) va dépendre l'étendu d'espace de $\mathbb{T}_{\mathfrak{M}}$ que l'on va associer à $\mathfrak{M}$ lors du calcul de ses descripteurs harmonico-morphologiques.

L'algorithmique des ces outils et méthodes de génération de descripteurs harmonico-morphologiques a été implémentée dans sa totalité dans la bibliothèque de fonction SageMath.

4.1.1 Application sur un exemple musical : descripteurs harmoniques du String Quartet II de Ligeti

Avant de donner un example complet de générations des descripteurs, il nous faut instancier tous les objets dont que la méthode utilise.

— Le passage d'oeuvre à analyser $\mathfrak{M}$ est issu du ***Quatuor à Cordes II**, mvt.I : Allegro Nervoso mm.23-39* de **Gyrogy Ligeti**. Ce passage é été choisi car c'est un exemple d'écriture typique chez Ligeti utilisant

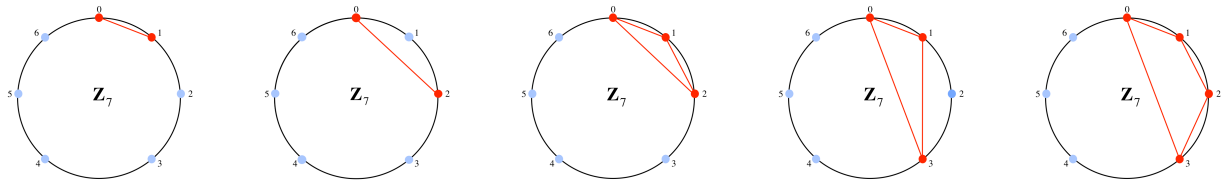


FIGURE 12 – Formes harmoniques du 7-tet les plus significatives (statistiquement et artistiquement) présentent dans $\mathfrak{M}$

la technique de "*micropolyphonie*", technique qu'il a inventé et abondamment utilisée dans le *Quatuor à Cordes II.*

— Tous les exemples ayant été donné dans le 7-tet, nous continuerons dans ce système harmonique pour un soucis d'unité, bien qu'une analyse dans le 12-tet eut été plus fine étant donné la nature extrêmement chromatique du passage à analyser, on utilisera donc le treillis de concept $\mathbb{C}$(7-tet).

— L'ensemble des formes harmoniques $H^{\mathfrak{M}}$ présentes dans $\mathfrak{M}$ est illustré dans la Figure [12]. Seules les formes harmoniques les plus significatives (statistiquement et artistiquement) du passage sont considérées.

— La valuation ω utilisée sur $\mathbb{C}$(7-tet) est la cardinalité du filtre engendré par le concept à valuer : $\forall a \in \mathbb{C},\ \omega_F(a) = |F(a)|$. N'importe quelle valuation mathématiquement valide sur un treillis (respectivement aux définitions données dans la section 3.1.4 (π) peut être utilisée. Ici une valuation beaucoup plus *explicitement musicale* devrait donner encore davantage de sens à la méthode, mais les valuations basées sur la structure du treillis (*cardinalité du filtre ou de l'idéal engendré par un concept*) fournissent déjà des résultats intéressants.

— Étant donné que ω_F est une valuation inférieure décroissante, la métrique $\mathbb{M}_{\omega_F}$ associée à ω_F sur $\mathbb{C}$(7-tet) est : $\forall (a_1, a_2) \in \mathbb{C}\text{(7-tet)}^2\ \mathbb{M}_{\omega_F(a)}(a_1, a_2) = \omega_F(a_1) + \omega_F(z_2) - 2\omega_F(a_1 \vee a_2)$.
Les opérateur morphologiques (δ, ε) sur $\mathbb{C}$(7-tet) sont respectivement une *dilatation par décomposition en éléments join-irreductibles* et une *érosion par décomposition en éléments meet-irréductibles* relativement au définitions de la section **3.1.5.2**.

— La distance de voisinage conceptuel à considérer lors des opérations morphologiques est de : 2.

Les Figure [13], [14], [15] représentent respectivement les trois étapes principales de l'**Algorithm 1**.

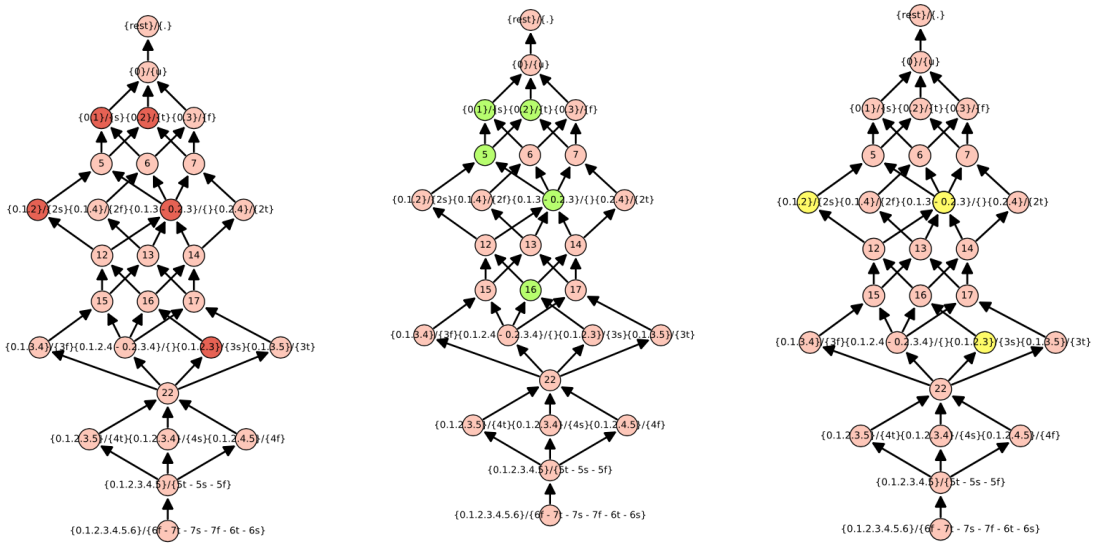


FIGURE 13 – Ensembles des concepts formels associés aux formes harmoniques présentes dans $H^{\mathfrak{M}}$. *Gauche* - neutre $H^{\mathfrak{M}}_{\mathbb{C}}$ (rouge). *Centre* - dilatation $\delta(H^{\mathfrak{M}}_{\mathbb{C}})$ (vert). *Droite* - érosion $\varepsilon(H^{\mathfrak{M}}_{\mathbb{C}})$ (jaune).

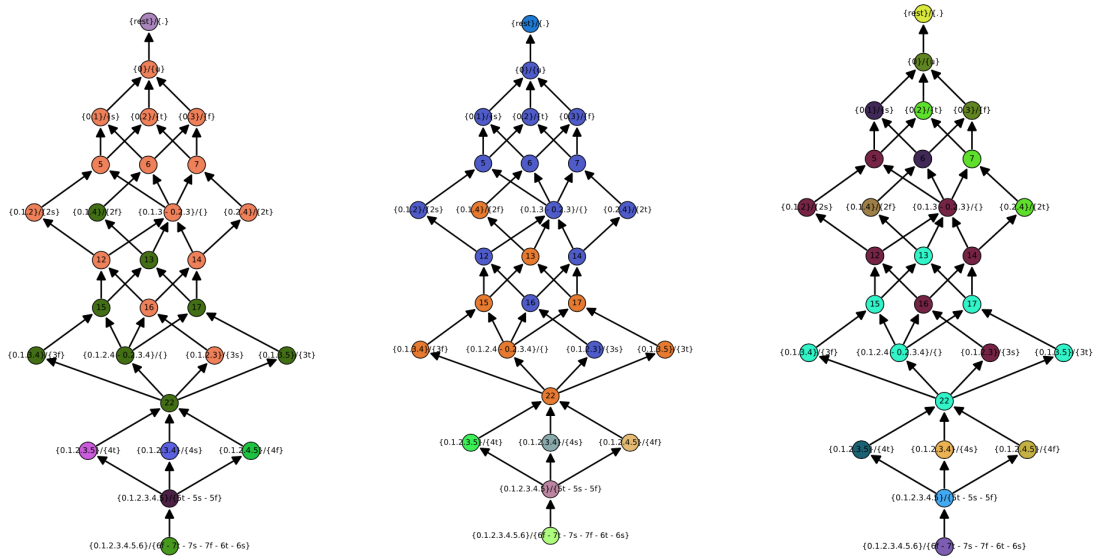


FIGURE 14 – Congruence θ, θ_δ, θ_ε sur $\mathbb{C}$(7-tet) engendrées respectivement par : *Gauche* - $H^{\mathfrak{M}}_{\mathbb{C}}$. *Centre* - $\delta(H^{\mathfrak{M}}_{\mathbb{C}})$. *Droite* - $\varepsilon(H^{\mathfrak{M}}_{\mathbb{C}})$.

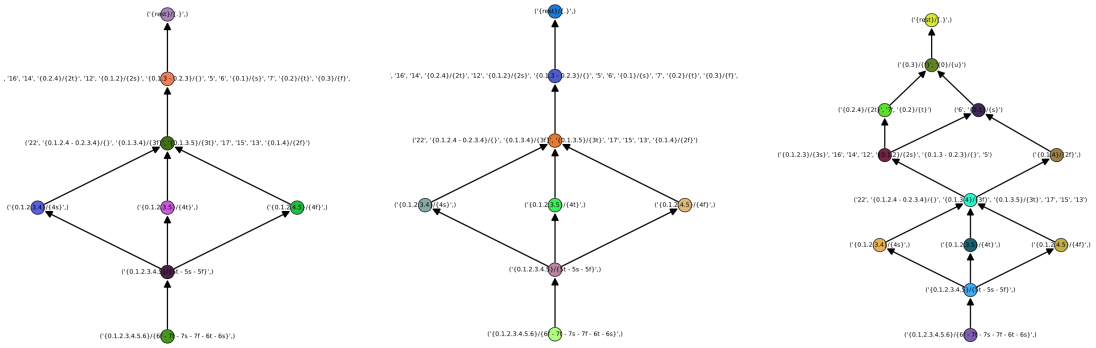


FIGURE 15 – Treillis-quotients : *Gauche* - neutre $\mathbb{C}(\mathfrak{M})/\theta$. *Centre* - dilatation $\mathbb{C}(\mathfrak{M})/\theta_\delta$. *Droite* - érosion $\mathbb{C}(\mathfrak{M})/\theta_\varepsilon$).

Le triplet de treillis-quotient modulo congruence ($\mathbb{C}(\mathfrak{M})/\theta$, $\mathbb{C}(\mathfrak{M})/\theta_\delta$, $\mathbb{C}(\mathfrak{M})/\theta_\varepsilon$) défini les descripteurs harmonico-morphologiques du passage situé entre les mesures $23-39$ du ***Quatuor à Cordes II***, *mvt.I : Allegro Nervoso* de **Gyrogy Ligeti**.
Il est à noter qu'à l'observation des congruences θ et θ_δ on pouvait déjà déduire que les treillis quotientés par ces congruences seraient isomorphes.
Une fois les descripteurs obtenus on pourra réaliser des comparaisons avec d'autres extraits de l'oeuvre en quête de passage présentant des similitudes structurelles harmoniques. La méthode de comparaison

4.2 Test de corrélation harmonico-morphologique

Pour effectuer une comparaison il s'agit de définir un objet musical de référence et un objet musical à comparer. On dénote par $\mathfrak{M}_{ref}$ la *phrase musicale* ou le passage de l'oeuvre qu'on utilisera comme référence lors de la comparaison, et on dénote par $\mathfrak{M}_{test}$ la *phrase musicale* ou le passage de l'oeuvre que l'on souhaite comparer à $\mathfrak{M}_{ref}$.
Ainsi une fois les trois descripteurs *treillis-quotient neutre*, *treillis-quotient de la dilatation* et *treillis-quotient de l'érosion* générés pour $\mathfrak{M}_{ref}$, on pourra chercher, par exemple, les $\mathfrak{M}_{test}$ qui présentent des similitudes structurelles d'un point de vue harmonique avec $\mathfrak{M}_{ref}$.

La procédure du test de corrélation harmonico-morphologique de deux phrases musicales ou de deux passages d'oeuvre s'énonce comme suit :

Algorithm 2 Test de corrélation harmonico-morphologique

Require: $\mathbb{C}(\mathfrak{M})$ est un treillis de concept associé au contexte formel multi-valué issu du système harmonique $\mathbb{T}_{\mathfrak{M}_{ref}}$
Require: $\mathfrak{M}_{ref}$ et $\mathfrak{M}_{test}$ doivent pouvoir être exprimés dans le même système harmonique, c-à-d $\mathbb{T}_{\mathfrak{M}_{test}} = \mathbb{T}_{\mathfrak{M}_{ref}}$
Require: $H^{\mathfrak{M}_{ref}}$ est l'ensemble des formes harmoniques présentes dans $\mathfrak{M}_{ref}$ et $H^{\mathfrak{M}_{test}}$ est l'ensemble des formes harmoniques présentes dans $\mathfrak{M}_{test}$
Require: ω est une valuation valide sur $\mathbb{C}(\mathfrak{M})$
Require: $\mathbb{M}_\omega$ est la métrique associé à ω sur $\mathbb{C}(\mathfrak{M})$
Require: (δ, ε) sont couple d'opérateur morphologique sur $\mathbb{C}(\mathfrak{M})$ utilisant la métrique $\mathbb{M}_\omega$
Require: n est la distance de voisinage conceptuel à considérer lors des opérations morphologiques
1: **function** TEST DE CORRÉLATION HARMONICO-MORPHOLOGIQUE($\mathbb{C}(\mathfrak{M})$, $H^{\mathfrak{M}_{ref}}$, $H^{\mathfrak{M}_{test}}$, (δ, ε), n)
2: $((\mathbb{C}(\mathfrak{M})/\theta)_{ref}$, $(\mathbb{C}(\mathfrak{M})/\theta_\delta)_{ref}$, $(\mathbb{C}(\mathfrak{M})/\theta_\varepsilon)_{ref})$ = DESCRIPTEURS HARMONICO-MORPHOLOGIQUES ($\mathbb{C}(\mathfrak{M})$, $H^{\mathfrak{M}_{ref}}$, (δ, ε), n)
3: Calculer l'ensemble des concepts formels $H_{\mathbb{C}}^{\mathfrak{M}_{test}}$ associé aux formes harmoniques présentes dans $H^{\mathfrak{M}_{test}}$
4: Calculer la Congruence θ_{test} sur $\mathbb{C}(\mathfrak{M})$ engendrée telle que tous les concepts présents dans l'ensemble $H_{\mathbb{C}}^{\mathfrak{M}_{test}}$ appartiennent au même block (classe d'équivalence) de congruence $[\cdot]_{\theta_{test}}$
5: Calculer le treillis-quotient $\mathbb{C}(\mathfrak{M})/\theta_{test}$ modulo θ_{test} depuis $\mathbb{C}(\mathfrak{M})$
6: **return** Si $\mathbb{C}(\mathfrak{M})/\theta_{test}$ est isomorphe à un des trois descripteurs harmonico-morphologiques de $\mathfrak{M}_{ref} \iff$ $\mathfrak{M}_{ref}$ et $\mathfrak{M}_{test}$ sont harmonico-morphologiquement corrélés.

Pour donner un suite cohérente avec l'exemple précédent, on pourra entendre que tout autre passage du textit**Quatuor à Cordes II** de Ligeti dont le *descripteur harmonico-morphologique neutre* serait isomorphe à un des trois descripteurs harmonico-morphologiques du passage de référence (*mvt.I* : mm.23-39, partagerait avec lui une similitude structurelle harmonique. Cette similitude serait plus ou moins grande en fonction de la distance de voisinage n choisi lors des opérations morphologiques.
Musicalement on pourra intuiter que, dans le cas d'une corrélation harmonico-morphologique, les espaces harmoniques parcourus par les deux passages de l'oeuvre sont corrélées au sens de la structure du système harmonique $\mathfrak{M}$-tet sur lequel l'oeuvre repose, ici le 7-tet.
D'ailleurs ce constat se généralise très bien pour tout autre extrait d'oeuvre dont le *descripteur harmonico-morphologique neutre* serait isomorphe à un des trois descripteurs harmonico-morphologiques du passage de référence du **Quatuor à Cordes II** de Ligeti.

On pourrait établir une variante de l'**Algorithm 2** dans laquelle on accorderait également à l'extrait d'oeuvre "test" un *intervalle* de descripteurs harmonico-morphologiques issus de la même méthode que ceux de l'extrait de référence, c-à-d un descripteur inférieur, neutre et supérieur. Les possibilités d'isomorphismes entre les descripteurs des deux extraits analysés se verraient forcément augmentées. Il faudrait avant tout déterminer dans quel cas d'analyse cette variante est plus pertinente que la version *originale*.

5 Conclusion et travaux futurs associés

Étant compositeur, et bien décidé à consacrer mon futur proche à l'écriture, ce stage ne fut pas anodin, ni dans son choix, ni dans sa suite.
Il m'a permis de faire le lien entre mes connaissance en mathématique/informatique et ma compréhension du langage musical. Je compte poursuivre dès l'année prochaine les travaux débutés dans ce stage à travers leur confrontation concrète à la composition. La phase de formalisation et de recherche théorique est très loin d'être achevée et les résultats obtenus lors de ces quelques semaines de stage sont autant de *pistes* à explorer plus profondément.

La difficulté principale que j'ai rencontrée dans ce stage fut liée à son attrait : presque toutes les théories mathématiques que j'ai abordée étaient entièrement nouvelles, j'ai, par conséquent rarement autant appris de savoirs théoriques sur des sujets mathématiques aussi divers en si peu de temps. Mes intuitions algébriques sont plus fines et j'ai enfin pu ouvrir la porte de la *Théorie des Catégories* grâce aux nombreux théorème d'Algèbre Abstraite que j'ai croisés dans mes lectures.
Ce stage fut également ma première vraie rencontre avec le monde de la recherche, avec son quotidien. L'incertitude et les résultats négatifs qui sont des aspects *inintriquables* à sa créativité m'ont dérouté plusieurs fois. Mais je sais avec plus de certitude aujourd'hui que c'est dans la recherche et dans la compréhension des liens entre *Algèbre, Informatique et Musique* que se trouve mon devenir.
En ce qui concerne les travaux futurs associés à ces recherches, il conviendra de les classer en deux catégories : court/long terme.
Mon intervention à l'IRCAM/Télécom/Dauphine ne s'achèvera qu'à la fin du mois de Juin, ce qui me laisse encore quelques jours pour continuer à avancer. J'ai plusieurs travaux en cours que je n'ai pas souhaité présenter dans ce document étant donné leur caractère parcellaire. Parmi ces questions j'aimerais arriver au moins à bout de deux :

- Depuis que je travaille avec les congruences j'ai l'intuition qu'il existe un moyen de montrer qu'une valuation d'ensemble d'élément d'un treillis de concept $\mathbb{C}$ peut être définie partir du *Treillis des congruences de* $\mathbb{C}$. Ma conjecture est la suivante : *Soit* $\mathbb{C}$ *un treillis de concept, soit* Con $\mathbb{C}$ *le treillis des congruences de* $\mathbb{C}$ *ordonné par inclusion des congruences. Soit* λ *une valuation sur* Con $\mathbb{C}$ *définie comme le supremum de la longueur des chaines qui relient le plus petit élément à l'élément à valuer. La longeur de la chaine correspond au nombre d'élément composant la chaine. Soit* $X \in \mathbb{C}$ *un ensemble de concept. Je conjecture qu'en associant la valuation par* λ *de l'élément de* Con $\mathbb{C}$ *qui correspond au* meet *(infimum) des congruences dans lesquelles les éléments de* X *appartiennent à la même classe d'équivalence, on peut valuer les éléments de* $\mathbb{C}$.
  Je sais que je ne suis plus très loin de prouver ou de réfuter cette proposition, j'aimerais le faire avant la fin du mois.

- Aussi J'ai pu constater qu'il existait des liens évidents entre les *prime form* harmoniques d'Allan Forte qui furent les prémices de l'*algébrisation* de l'harmonie musicale et la construction de treillis de concept associé à des contextes formels harmoniques multi-valués. J'aimerais éclaircir ce lien.

Pour les travaux à plus long terme qui découleraient du prélude réalisé durant ce stage, j'entrevois de nombreuses pistes :

- Mettre à l'épreuve la méthode de test de corrélation harmonico-morphologique à travers l'analyse systématique de plusieurs oeuvres du répertoire classique et contemporain.
- La plus grande frustration de mon stage est venue de mes recherches sur les valuations de treillis. J'ai essayé des dizaines de valuations différentes, la plus part du temps sans succès. Mes encadrants pédagogiques m'avaient prévenu que les question de valuation de treillis étaient frustrantes et difficiles. Il s'agirait pour l'amélioration de ces travaux de trouver des valuations de treillis de concepts harmoniques basées sur des informations musicales plus explicites et discriminantes. En effet les opérateurs morphologiques engendrant les descripteurs harmoniques étant basés sur ces valuations, si on les rend musicalement plus signifiantes, on rajoutera par là même du sens musical dans le triplet de descripteurs associé à l'extrait d'oeuvre à analyser.
- Un des points à approfondir concerne les opérateurs morphologiques eux-même. Il s'agirait d'analyser (statistiquement ? Machine Learning ?) des descripteurs harmonico-morphologiques utilisant différentes définitions d'opérateur morphologiques (δ, ε) - et ce sur un grand nombre d'exemple musicaux et de treillis harmoniques différents - pour déterminer quelle est la définition de dilation/érosion sur treillis de concept la plus apte à produire des descripteurs harmoniques.

Le projet de recherche à plus long terme que j'envisage et que j'aimerais pouvoir commencer l'année prochaine serait l'étude des implications en logique musicale que pourrait apporter la *Théorie des Topos* de Grothendick.

Références

[1] Birkhoff G., Lattice Theory, 3 rd ed, American Mathematical Society ; 1967.

[2] Schlemmer T., Andreatta M. (2013) Using Formal Concept Analysisto Represent Chroma Systems. In : Yust J., Wild J., Burgoyne J.A. (eds) Mathematics and Computation in Music. MCM 2013. Lecture Notes in Computer Science, vol 7937. Springer, Berlin, Heidelberg

[3] Davey, B., & Priestley, H. (2002). Introduction to Lattices and Order. Cambridge : Cambridge University Press. doi :10.1017/CBO9780511809088

[4] Schlemmer, T. & Schmidt, S.E. Ann Math Artif Intell (2010) 59 : 241. doi :10.1007/s10472-010-9198-6

[5] Atif, J., Bloch, I., Distel, F., & Hudelot, C., «Mathematical morphology operators over concept lattices », In Formal Concept Analysis (pp. 28- 43), Springer Berlin Heidelberg, 2013.

[6] H.J.A.M. Heijmans, C. Ronse - The algebraic basis of mathematical morphology. I. Dilations and erosions. Comput. Vision Graphics Image Process., 50 (1990), pp. 245-295

[7] Jean Serra, Image Analysis and Mathematical Morphology, vol. 1, Academic Press, Londres, 1982.

[8] Wille, R. : Restructuring lattice theory : An approach based on hierarchies of con- cepts. In Rival, I., ed. : Ordered Sets. Volume 83 of NATO Advanced Study Insti- tutes Series. Springer Netherlands (1982) 445 470

[9] Wille, R. : Musiktheorie und Mathematik. In Gootze, H., Wille, R., eds. : Musik und Mathematik : Salzburger Musikgespraach 1984 unter Vorsitz von Herbert von Karajan. Springer-Verlag, Berlin (1985) 4 31

[10] Wille, R. : Restructuring lattice theory : An approach based on hierarchies of con- cepts. In Ferre, S., Rudolph, S., eds. : Formal Concept Analysis. Volume 5548 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 314 339 Reprint.

[11] Birkhoff G., Mac Lane S., Algebra, Chelsea (ISBN 0-8218-1646-2), 1999

[12] Agon, C., Langages de programmation pour la composition musicale, HDR, UPMC, 2004

[13] Freund A., Andreatta M., Giavitto J-L.. Lattice-based and topological representations of binary relations with an application to music. Annals of Mathematics and Artificial Intelligence, Springer Verlag, 2015, 73 (3-4), pp.311-334.

[14] Bergomi, M., Dynamical and topological tools for (modern) music analysis, the ?se de doctorat en cotutelle UPMC/LIM Milan, 2015.

[15] Bergomi, M., A. Barate, B. Di Fabio, « Towards a Topological Fingerprint of Music », in in A. Bac and J.-L. Mari (eds), Computational Topology in Image Context, LNCS, pp 88-100

[16] Bigo L., D. Ghisi, A. Spicher, M. Andreatta, « Representation of Musical Structures and Processes in Simplicial Chord Spaces », Computer Music Journal, vol. 39, n° 3, p. 9-24, 2015.

[17] Bloch, I., « Modal Logics based on Mathematical Morphology for Spatial Reasoning », Journal of Applied Non Classical Logics, 12(3-4), 399-424, 2002.

[18] Bloch, I., & Lang, J., « Towards mathematical morpho- logics ». In Technologies for Constructing Intelligent Systems 2, Physica-Verlag HD, 367-380, 2002.

[19] R. Power, An analysis of Transformation procedures in Gyorgy Ligeti's String Quartet n°2, PHD at University of Illinois, Urbana-Champain, 1995.

Annexe

I - Treillis de concept de Système Harmonique

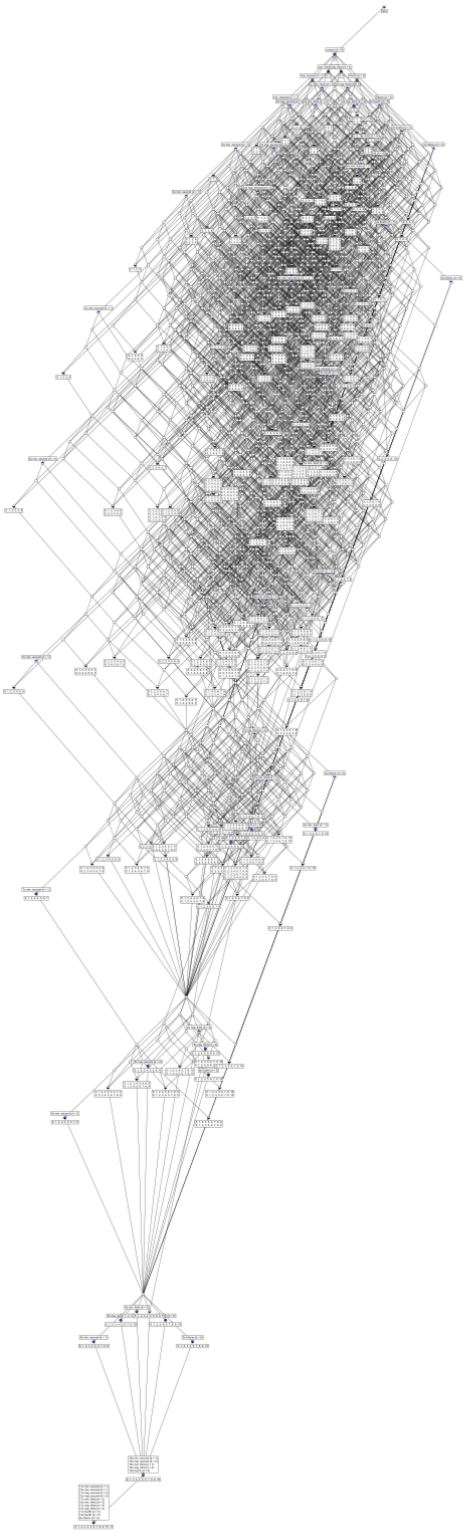


FIGURE 16 – Treillis de concept du contexte formel multi-valué du 12-tet $\mathbb{T}_{12}$. ([2])

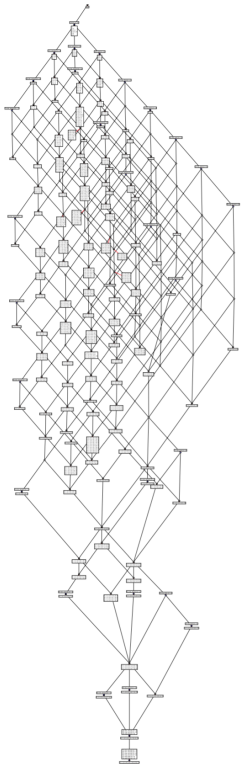


FIGURE 17 – Treillis de concept du contexte formel multi-valué du 12-tet $\mathbb{T}_{12}$ en ne prenant en compte que la multiplicité des intervalles 1 (seconde mineure), 2 (seconde majeure), 4 (tierce mineure) en tant qu'attributs du contexte. ([2])

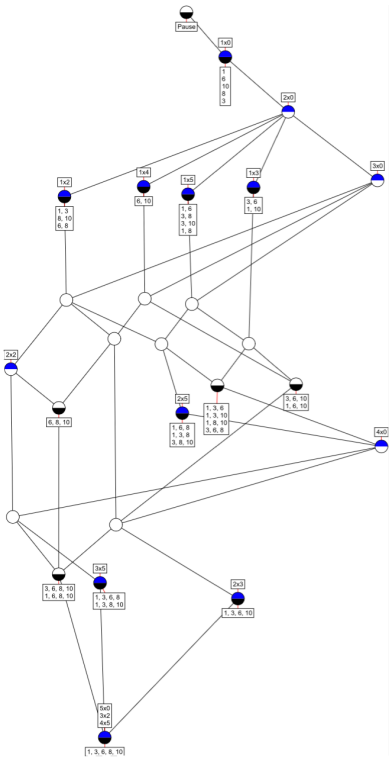


FIGURE 18 – Treillis de concept du contexte formel multi-valué du système harmonique pentatonique (5 notes) issu du 12-tet $\mathbb{T}_{12}$. ([2])

II - Esquisse de contribution à SAGEMATH: bibliothèque de calcul MorphoMath - FCA

```
import networkx
from sage.all import *
import re
import fileinput
import xml.etree.ElementTree as ET
import itertools #for iterating on pairs of element
import random
import sage.graphs.graph_plot


#all params available
plotting_options = {
    'vertex_size':200,
    'vertex_labels':False,
    'layout':None,
    'edge_style':'solid',
    'edge_color':'black',
    'edge_colors':None,
    'edge_labels':False,
    'iterations':50,
    'tree_orientation':'down',
    'heights':None,
    'graph_border':False,
    'talk':False,
    'color_by_label':False,
    'partition':None,
    'dist':.075,
    'max_dist':1.5,
    'loop_size':.075}


#=================================
#        FORMAL CONCEPT CLASS
#=================================


class FormalConcept:

    def __init__(self, extent, intent, sup_nodes):
        """ initializes a graph object """
        self.__extent = extent
        self.__intent = intent
        self.__sup_nodes = sup_nodes
        self.__name = "no-name"

    def name(self):
        """ returns the name of the concept
        """
        return self.__name

    def extent(self):
        """ returns the extent of the concept
        """
        return self.__extent

    def intent(self):
        """ returns the intent of the concept
        """
        return self.__intent

    def sup_nodes(self):
        """ returns the sup_nodes of the concept
        """
        return self.__sup_nodes

    def set_name(self, name):
        """ set the name of the concept
        """
        self.__name = name

    def set_extent(self, ext):
        """ set the object set
        """
        self.__extent = ext

    def set_intent(self, intent):
        """ set the attribute set
        """
```

```
        self.___intent = intent

    def set_sup_nodes(self, sup_nodes):
        """ set the sup nodes set
        """
        self.___sup_nodes = sup_nodes

    def rename_concept(self):
        """rename concept with meaningful name
           containing the objects and attributes of the concept
        """
        name = "{"
        i = 0
        for obj in self.extent():
            if i != 0:
                name += " * "
            name += obj
            i += 1
        name += "}/{"
        i = 0
        for att in self.intent():
            if i != 0:
                name += " * "
            name += att
            i += 1
        name += "}"
        self.___name = name

    def print_ext_int(self):
        print("Concept: " + self.name())
        print("-----------------------------")
        print("--> extent: " + str(self.extent()))
        print("--> intent: " + str(self.intent()))
        print("\n")


#=================================
#        FORMAL CONTEXT CLASS
#=================================

class FormalContext:

    def ___init___(self, objects=[], attributes=[], concepts=[]):
        """ initializes a formal context object """
        self.___objects = objects
        self.___attributes = attributes
        self.___concepts = concepts
        self.___lattice = LatticePoset() # empty sage lattice

    def objects(self):
        """ returns the objects of the context
            objects are tuples with (id, name)
        """
        return self.___objects

    def attributes(self):
        """ returns the attributes of the context
            attributes are tuples with (id, name)
        """
        return self.___attributes

    def concepts(self):
        """ returns the concepts of the context
            concepts are tuples with (id, name)
            concept[0] -> id
            concept[1] -> name
        """
        return self.___concepts

    def lattice(self):
        """ returns the concept lattice of the context
            lattice is an instance of Sage Lattice class
        """
        return self.___lattice

    def ___len___(self):
        """ define the size of a context as
            the number of concepts
        """
        return len(self.concepts())
```

```
def add_object(self, obj):
    """ add an object to the object set
    """
    if obj[0] not in map(lambda x: x[0], self.objects()):
        self.___objects.append(obj)

def add_concept(self, concept):
    """ add a concept to the concept set
    """
    if concept[0] not in map(lambda x: x[0], self.concepts()):
        self.___concepts.append(concept)

def add_attribute(self, att):
    """ add an attribute to the attribute set
    """
    if att[0] not in map(lambda x: x[0], self.attributes()):
        self.___attributes.append(att)

def set_lattice(self, lattice):
    """ set the concept lattice associated to the context
    """
    self.___lattice = lattice

def rename_concepts_wrt_att_obj_map(self):
    """ rename the concepts of the context with respect
        to object and attribute map
        - useful for large lattice -
    """
    dico_labels = self.gen_label_latt_dico()
    for cncpt in self.concepts():
        latt_elmt = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1])
        cncpt[1].set_name(dico_labels[latt_elmt])
    L = self.lattice().relabel(dico_labels)
    self.set_lattice(L)


def latticeXml_to_FormalContext(self, file_path):
    """ fill formal conctext instance with the data from a Galicia lattice XML
    """
    tree = ET.parse(file_path)
    root = tree.getroot()
    objects_xml = root[1] #<OBJS>
    attributes_xml = root[2] #<ATTS>
    concepts_xml = root[3] #<NODS>
    #object
    for obj in objects_xml:
        id = obj.get('id')
        name = obj.text
        self.add_object((id, name))
    #attribute
    for att in attributes_xml:
        id = att.get('id')
        name = att.text
        self.add_attribute((id, name))
    #concept
    for concept in concepts_xml:
        formal_concept = FormalConcept([],[],[])
        id_concept = concept.get('id')
        id_obj_xml = []
        id_att_xml = []
        id_sup_nod_xml = []
        #concept extent <EXT>
        for obj in concept[0]:
            id_obj_xml.append(obj.get('id'))
        formal_concept.set_extent(map(lambda e: e[1],filter(lambda e: e[0] in id_obj_xml, self.objects\
())))
        #concept intent <INT>
        for att in concept[1]:
            id_att_xml.append(att.get('id'))
        formal_concept.set_intent(map(lambda e: e[1], filter(lambda e: e[0] in id_att_xml, self.\
attributes())))
        #concept sup nodes <SUP_NOD>
        for sup_nod in concept[2]:
            id_sup_nod_xml.append(sup_nod.get('id'))
        formal_concept.set_sup_nodes(filter(lambda e: e[0] in id_sup_nod_xml, self.concepts()))
        formal_concept.rename_concept() #rename concept with meaningful name containing the objects and\
 attributes of the concept
        self.add_concept((id_concept, formal_concept))
```

```
    def FormalContext_to_csv(self, csv_file_path):
        """ build valid csv file from formal conctext instance
        """
        outputfile = open(csv_file_path,"w")
        for concept in self.concepts():
            name_concept = concept[1].name()
            for sup_node in concept[1].sup_nodes():
                name_sup_node = sup_node[1].name()
                outputfile.write(name_sup_node + "," + name_concept)
                outputfile.write("\n")

# - - - - - - - - - - - - - -
#      Galois Maps
# - - - - - - - - - - - - - -

    def obj_derivative(self, object_list):
        #print(object_list)
        temp_concepts = [concept[1] for concept in self.___concepts if set(object_list).issubset(concept[1].\
    extent())]
        concept_w_min_extent_length = min(map(lambda cncpt: len(cncpt.extent()), temp_concepts))
        object_list_concept = next(cncpt for cncpt in temp_concepts if len(cncpt.extent()) == \
    concept_w_min_extent_length)
        return object_list_concept.intent()


    def attribute_derivative(self, attribute_list):
        temp_concepts = [concept[1] for concept in self.___concepts if set(attribute_list).issubset(concept\
    [1].intent())]
        concept_w_min_intent_length = min(map(lambda cncpt: len(cncpt.intent()), temp_concepts))
        attribute_list_concept = next(cncpt for cncpt in temp_concepts if len(cncpt.intent()) == \
    concept_w_min_intent_length)
        return attribute_list_concept.extent()

    def object_concept_map(self, obj):
        obj_cncpt_int = self.obj_derivative(obj)
        obj_cncpt_ext = self.attribute_derivative(obj_cncpt_int)
        return next(concept[1] for concept in self.___concepts
                    if concept[1].extent() is obj_cncpt_ext and concept[1].intent() is obj_cncpt_int)

    def attribute_concept_map(self, att):
        att_cncpt_ext = self.attribute_derivative(att)
        att_cncpt_int = self.obj_derivative(att_cncpt_ext)
        return next(concept[1] for concept in self.___concepts
                    if concept[1].extent() is att_cncpt_ext and concept[1].intent() is att_cncpt_int)

    def compute_how_many_obj_map_to(self, cncpt):
            return len([obj for obj in seven_tet_context.objects() if seven_tet_context.object_concept_map\
    ([obj[1]]) == cncpt])


# - - - - - - - - - - - - - -
#   Metrics & Morpho
# - - - - - - - - - - - - - -

    def compute_dist_btwn_concepts(self, cncpt_1, cncpt_2, func_valuation):
        try:
            metric = valuation_to_metric(self, func_valuation)
        except:
            print('valuation_to_metric() function in compute_dist_btwn_concepts() failed to return a metric\
    ')
            return 0
        else:
            return metric(self, cncpt_1, cncpt_2, func_valuation)

    def compute_dist_btwn_objects(self, obj_1, obj_2, func_val):
        #print((obj_1, obj_2))
        cncpt_1 = self.object_concept_map(obj_1)
        cncpt_2 = self.object_concept_map(obj_2)
        return self.compute_dist_btwn_concepts(cncpt_1, cncpt_2, func_val)

    def compute_dist_btwn_attributes(self, att_1, att_2, func_val):
        #print((att_1, att_2))
        cncpt_1 = self.attribute_concept_map(obj1)
        cncpt_2 = self.attribute_concept_map(obj2)
        return self.compute_dist_btwn_concepts(cncpt_1, cncpt_2, func_val)

    def compute_struct_elmt_from_obj(self, center_obj, func_val, neighborhood_dist):
            return [obj[1] for obj in self.objects()
                    if self.compute_dist_btwn_objects(center_obj, [obj[1]], func_val) <= neighborhood_dist]
```

```
def compute_struct_elmt_from_att(self, center_att, func_val, neighborhood_dist):
        return [att[1] for att in self.attributes()
                if self.compute_dist_btwn_attributes(center_att, [att[1]], func_val) <= \
 neighborhood_dist]

def dilate_obj_set_wrt_struct_elmt(self, obj_set, func_val, neighborhood_dist):
    return [obj[1] for obj in self.objects()
            if set(self.compute_struct_elmt_from_obj([obj[1]], func_val, neighborhood_dist)).\
 intersection(set(obj_set)) != set()]

def erode_obj_set_wrt_struct_elmt(self, obj_set, func_val, neighborhood_dist):
    return [obj[1] for obj in self.objects()
            if set(self.compute_struct_elmt_from_obj([obj[1]], func_val, neighborhood_dist)).issubset(\
 set(obj_set))]

def dilate_att_set_wrt_struct_elmt(self, att_set, func_val, neighborhood_dist):
    return [att[1] for att in self.attributes()
            if set(self.compute_struct_elmt_from_att([att[1]], func_val, neighborhood_dist)).\
 intersection(set(att_set)) is not set()]

def erode_att_set_wrt_struct_elmt(self, att_set, func_val, neighborhood_dist):
    return [att[1] for att in self.attributes()
            if set(self.compute_struct_elmt_from_att([att[1]], func_val, neighborhood_dist)).issubset(\
 set(att_set))]

def dilate_cncpt_to_att_obj_power_sets(self, cncpt, func_val, neighborhood_dist):
    return (self.dilate_obj_set_wrt_struct_elmt(cncpt.extent(), func_val, neighborhood_dist),
            self.erode_att_set_wrt_struct_elmt(cncpt.intent(), func_val, neighborhood_dist))

def erode_cncpt_to_att_obj_power_sets(self, cncpt, func_val, neighborhood_dist):
    return (self.erode_obj_set_wrt_struct_elmt(cncpt.extent(), func_val, neighborhood_dist),
            self.dilate_att_set_wrt_struct_elmt(cncpt.intent(), func_val, neighborhood_dist))

def dilate_cncpt_from_obj_dilation_and_deriv_op(self, cncpt, func_val, neighborhood_dist):
    dilate_ext = self.dilate_obj_set_wrt_struct_elmt(cncpt.extent(), func_val, neighborhood_dist)
    dilated_cncpt_int = self.obj_derivative(dilate_ext)
    dilated_cncpt_ext = self.attribute_derivative(dilated_cncpt_int)
    return next(concept[1] for concept in self.concepts()
                if concept[1].extent() is dilated_cncpt_ext and concept[1].intent() is \
 dilated_cncpt_int)

def erode_cncpt_from_obj_erosion_and_deriv_op(self, cncpt, func_val, neighborhood_dist):
    erode_extent = self.erode_obj_set_wrt_struct_elmt(cncpt.extent(), func_val, neighborhood_dist)
    eroded_cncpt_int = self.obj_derivative(erode_extent)
    eroded_cncpt_ext = self.attribute_derivative(eroded_cncpt_int)
    return next(concept[1] for concept in self.concepts()
                if concept[1].extent() is eroded_cncpt_ext and concept[1].intent() is eroded_cncpt_int)

def dilate_join_irreduc_cncpt(self, join_irreduc_cncpt, func_val, neighborhood_dist):
    if func_cncpt_elmnt_to_lattice_elmnt(self, join_irreduc_cncpt) in self.lattice().join_irreducibles\
 ():
        cncpts_at_dist = [cncpt[1] for cncpt in self.concepts()
                if self.compute_dist_btwn_concepts(cncpt[1], join_irreduc_cncpt, func_val) <= \
 neighborhood_dist]
        lattice_elmts_at_dist = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt), \
 cncpts_at_dist)
        join_latt_elmt_at_dist = self.lattice().join(lattice_elmts_at_dist)
        return func_lattice_elmnt_to_cncpt_elmnt(self, join_latt_elmt_at_dist)
    else:
        print('The concept given to the function \'dilate_join_irreduc_cncpt\' is not join irreductible\
')

def erode_meet_irreduc_cncpt(self, meet_irreduc_cncpt, func_val, neighborhood_dist):
    if func_cncpt_elmnt_to_lattice_elmnt(self, meet_irreduc_cncpt) in self.lattice().meet_irreducibles\
 ():
        cncpts_at_dist = [cncpt[1] for cncpt in self.concepts()
                if self.compute_dist_btwn_concepts(cncpt[1], meet_irreduc_cncpt, func_val) <= \
 neighborhood_dist]
        lattice_elmts_at_dist = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt), \
 cncpts_at_dist)
        meet_latt_elmt_at_dist = self.lattice().meet(lattice_elmts_at_dist)
        return func_lattice_elmnt_to_cncpt_elmnt(self, meet_latt_elmt_at_dist)
    else:
        print('The concept given to the function \'erode_meet_irreduc_cncpt\' is not meet irreductible'\
)

def minimal_join_decompose_cncpt(self, cncpt):
    #lattice_elmt_to_decompose = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt)
    #joinands_latt_elmt = self.lattice().canonical_joinands(lattice_elmt_to_decompose)
```

```
    #return map(lambda e: func_lattice_elmnt_to_cncpt_elmnt(self, e), joinands_latt_elmt)
    L = self.lattice()
    lattice_elmt_to_decompose = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt)
    join_irreduct_elmts = L.join_irreducibles()
    full_join_decompo = [join_irr_elmt for join_irr_elmt in join_irreduct_elmts if L.is_lequal(\
join_irr_elmt, lattice_elmt_to_decompose)]
    join_power_set = func_generate_power_set(full_join_decompo)
    join_decompo_sets_latt_elmt = filter(lambda set_in: L.join(set_in) is  lattice_elmt_to_decompose, \
join_power_set)
    join_decompo_sets_cncpt_elmt = map(lambda set_latt_elmt: [func_lattice_elmnt_to_cncpt_elmnt(self, \
latt_elmt) for latt_elmt in set_latt_elmt], join_decompo_sets_latt_elmt)
   #return the set that have the minimum added intent length which ensure that all other join-irreduc \
decomposition refined it
    return min(join_decompo_sets_cncpt_elmt, key= lambda cncpt_set: sum(len(cncpt.intent()) for cncpt \
in cncpt_set))

def minimal_meet_decompose_cncpt(self, cncpt):
    #lattice_elmt_to_decompose = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt)
    #meetands_latt_elmt = self.lattice().canonical_meetands(lattice_elmt_to_decompose)
    #return map(lambda e: func_lattice_elmnt_to_cncpt_elmnt(self, e), meetands_latt_elmt)
    L = self.lattice()
    lattice_elmt_to_decompose = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt)
    meet_irreduct_elmts = L.meet_irreducibles()
    full_meet_decompo = [meet_irr_elmt for meet_irr_elmt in meet_irreduct_elmts if L.is_lequal(\
lattice_elmt_to_decompose, meet_irr_elmt)]
    meet_power_set = func_generate_power_set(full_meet_decompo)
    meet_decompo_sets_latt_elmt = filter(lambda set_in: L.meet(set_in) is  lattice_elmt_to_decompose, \
meet_power_set)
    meet_decompo_sets_cncpt_elmt = map(lambda set_latt_elmt: [func_lattice_elmnt_to_cncpt_elmnt(self, \
latt_elmt) for latt_elmt in set_latt_elmt], meet_decompo_sets_latt_elmt)
   #return the set that have the minimum added intent length which ensure that all other join-irreduc \
decomposition refined it
    return min(meet_decompo_sets_cncpt_elmt, key= lambda cncpt_set: sum(len(cncpt.extent()) for cncpt \
in cncpt_set))

def dilate_cncpt_from_join_irreduc_dilation(self, cncpt, func_val, neighborhood_dist):
    cncpt_join_decompo = self.minimal_join_decompose_cncpt(cncpt)
    dilation_set_cncpt_join_decompo = map(lambda join_irr_cncpt: self.dilate_join_irreduc_cncpt(\
join_irr_cncpt, func_val, neighborhood_dist), cncpt_join_decompo)
    dilation_set_latt_elmt = map(lambda dilated_cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, \
dilated_cncpt), dilation_set_cncpt_join_decompo)
    join_dilation_set = self.lattice().join(dilation_set_latt_elmt)
    return func_lattice_elmnt_to_cncpt_elmnt(self, join_dilation_set)

def erode_cncpt_from_meet_irreduc_erosion(self, cncpt, func_val, neighborhood_dist):
    cncpt_meet_decompo = self.minimal_meet_decompose_cncpt(cncpt)
    erosion_set_cncpt_meet_decompo = map(lambda meet_irr_cncpt: self.erode_meet_irreduc_cncpt(\
meet_irr_cncpt, func_val, neighborhood_dist), cncpt_meet_decompo)
    erosion_set_latt_elmt = map(lambda eroded_cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, \
eroded_cncpt), erosion_set_cncpt_meet_decompo)
    meet_erosion_set = self.lattice().meet(erosion_set_latt_elmt)
    return func_lattice_elmnt_to_cncpt_elmnt(self, meet_erosion_set)

def is_dilation_over_lattice(self, dilation_func, val, neigborhood_dist):
    latt_elmt_list = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1]), self.concepts\
())
   #test that dilattion preserve least element
    bottom_concept = func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().bottom())
    join_of_dilation_bot = func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().join(
                           [func_cncpt_elmnt_to_lattice_elmnt(self, dilation_func(bottom_concept, val, \
neigborhood_dist))]))
    dilation_of_join_bot = dilation_func(func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().join(
                           [func_cncpt_elmnt_to_lattice_elmnt(self, bottom_concept)])), val, \
neigborhood_dist)
    if join_of_dilation_bot != dilation_of_join_bot:
        return False
    else:
        #test that dilation commutes with supremum
        for cncpt in self.concepts():
            join_of_dilation = func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().join(
                               [func_cncpt_elmnt_to_lattice_elmnt(self, dilation_func(cncpt[1], val, \
neigborhood_dist))]))
            dilation_of_join = dilation_func(func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().\
join(
                               [func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1])])), val, \
neigborhood_dist)
            if not join_of_dilation == dilation_of_join:
                return False
        return True
```

```
    def is_erosion_over_lattice(self, erosion_func, val, neigborhood_dist):
        latt_elmt_list = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1]), self.concepts\
    ())
        #test that erosion preserve greatest element
        top_concept = func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().top())
        meet_of_erosion_top = func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().meet(
                                [func_cncpt_elmnt_to_lattice_elmnt(self, erosion_func(top_concept, val, \
    neigborhood_dist))]))
        erosion_of_meet_top = erosion_func(func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().meet(
                                [func_cncpt_elmnt_to_lattice_elmnt(self, top_concept)])), val, \
    neigborhood_dist)
        if meet_of_erosion_top != erosion_of_meet_top:
            return False
        else:
            #test that erosion commutes with infimum
            for cncpt in self.concepts():
                meet_of_erosion = func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().meet(
                                    [func_cncpt_elmnt_to_lattice_elmnt(self, erosion_func(cncpt[1], val, \
    neigborhood_dist))]))
                erosion_of_meet = erosion_func(func_lattice_elmnt_to_cncpt_elmnt(self, self.lattice().meet(
                                    [func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1])])), val, \
    neigborhood_dist)
                if meet_of_erosion != erosion_of_meet:
                    return False
            return True


    def is_sup_generator(self, family_cncpts):
        latt_elmt_list = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1]), self.concepts\
    ())
        family_latt_elmt_list = map(lambda family_cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, \
    family_cncpt), family_cncpts)
        for latt_elmt in latt_elmt_list:
            if latt_elmt != self.lattice().join([family_latt_elmt for family_latt_elmt in \
    family_latt_elmt_list if self.lattice().is_lequal(family_latt_elmt, latt_elmt)]):
                print(latt_elmt, family_latt_elmt)
                print(self.lattice().join([family_latt_elmt for family_latt_elmt in family_latt_elmt_list \
    if self.lattice().is_lequal(family_latt_elmt, latt_elmt)]))
                return False
        return True

    def is_inf_generator(self, family_cncpts):
        latt_elmt_list = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt[1]), self.concepts\
    ())
        family_latt_elmt_list = map(lambda family_cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, \
    family_cncpt), family_cncpts)
        for latt_elmt in latt_elmt_list:
            if latt_elmt != self.lattice().meet([family_latt_elmt for family_latt_elmt in \
    family_latt_elmt_list if self.lattice().is_gequal(family_latt_elmt, latt_elmt)]):
                return False
        return True

# - - - - - - - - - - - - -
#        Print
# - - - - - - - - - - - - -

    def compute_congruence_from_cncpts(self, cncpt_list):
        latt_elemt_list = [func_cncpt_elmnt_to_lattice_elmnt(self, cncpt) for cncpt in cncpt_list]
        return self.lattice().congruence([latt_elemt_list])

    def compute_height_of_meet_cong_in_cong_latt_containing_cncpts(self, cncpt_list):
        latt_elmt_list = [func_cncpt_elmnt_to_lattice_elmnt(self, x) for x in cncpt_list]
        cong_latt = self.lattice().congruences_lattice()
        cong_sub_latt = [sub_cong for sub_cong in cong_latt if congruence_equiv_class_contains(sub_cong, \
    latt_elmt_list)]
        meet_cong = cong_latt.meet(cong_sub_latt)
        cong_latt_chains = list(cong_latt.chains())
        sup_length_chain_from_bot_to_meet_cong = max([chain for chain in cong_latt_chains if cong_latt.\
    bottom() in chain and meet_cong in chain], key=lambda e: len(e))
        return len(sup_length_chain_from_bot_to_meet_cong)


# - - - - - - - - - - - - -
#        Print
# - - - - - - - - - - - - -

    def gen_label_latt_dico(self):
        dico = {}
        i = 0
        for cncpt in self.concepts():
```

```
        key = cncpt[1].name()
        object_that_map_to_this_concept = [obj[1] for obj in self.objects() if self.object_concept_map\
([obj[1]]) == cncpt[1]]
        attrib_that_map_to_this_concept = [att[1] for att in self.attributes() if self.\
attribute_concept_map([att[1]]) == cncpt[1]]
        if len(object_that_map_to_this_concept) == 1 and len(attrib_that_map_to_this_concept) == 1:
            dico[key] = '{' + str(object_that_map_to_this_concept[0]) + "}/{" + str(\
attrib_that_map_to_this_concept[0]) + '}'
        elif len(object_that_map_to_this_concept) > 1 or len(attrib_that_map_to_this_concept) > 1:
            j = 1
            dico_value = '{'
            for obj in object_that_map_to_this_concept:
                dico_value += str(obj)
                if j != len(object_that_map_to_this_concept):
                    dico_value += ' - '
                j = j + 1
            j = 1
            dico_value += '}/{'
            for att in attrib_that_map_to_this_concept:
                dico_value += str(att)
                if j != len(attrib_that_map_to_this_concept):
                    dico_value += ' - '
                j = j + 1
            dico_value += '}'
            dico[key] = dico_value
        elif len(object_that_map_to_this_concept) == 1:
            dico[key] = '{' + str(object_that_map_to_this_concept[0]) + '}/{.}'
        elif len(attrib_that_map_to_this_concept) == 1:
            dico[key] = '{' + str(attrib_that_map_to_this_concept[0]) + '}/{.}'
        else:
            dico[key] = str(i) #because Sage do not currently accept non-injective relabelling (label \
can't be the same?)
        i = i + 1
    return dico


def print_lattice(self, v_size, fig_size):
    #dico_elmt_labels = self.gen_label_latt_dico()
    dico_color = {}
    double_irr = self.lattice().double_irreducibles()
    dico_color['#FFF585'] = double_irr
    join_irr = self.lattice().join_irreducibles()
    dico_color['#C7F5DF'] = [elmt for elmt in join_irr if elmt not in double_irr]
    meet_irr = self.lattice().meet_irreducibles()
    dico_color['#5CC3FF'] = [elmt for elmt in meet_irr if elmt not in double_irr]
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = v_size,
                        figsize = fig_size)

def print_join_irreduct_elmt(self):
    #the commented line were useful when the lattice was not renamed with obj/att map label
    #dico_elmt_labels = self.gen_label_latt_dico()
    join_ir = self.lattice().join_irreducibles()
    #join_ir_relabelled = map(lambda elmt: dico_elmt_labels[elmt], join_ir)
    dico_color = {'#C7F5DF':join_ir}
    print("Join-irreducibles elements:")
    #self.lattice().show(element_labels=dico_elmt_labels,vertex_colors=dico_color,vertex_size = 500,\
figsize = [12,10])
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = 500,
                        figsize = [8,8])


def print_meet_irreduct_elmt(self):
    meet_ir = self.lattice().meet_irreducibles()
    dico_color = {'#5CC3FF':meet_ir}
    print("Meet-irreducibles elements:")
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = 500,
                        figsize = [8,8])


def print_principal_filter(self, cncpt_list):
    try:
        cncpt_list[0]
    except AttributeError:
        #goes here if a single concept (not in a list) is given to the function
        lattice_elmt_list = [func_cncpt_elmnt_to_lattice_elmnt(self, cncpt_list)]
        cncpt_list_name = [cncpt_list.name()]
    else:
```

```
        lattice_elmt_list = []
        cncpt_list_name = []
        for cncpt in cncpt_list:
            lattice_elmt_list.append(func_cncpt_elmnt_to_lattice_elmnt(self, cncpt))
            cncpt_list_name.append(cncpt.name())
    filter_elmts = self.lattice().order_filter(lattice_elmt_list)
    dico_color = {'#C7F5DF':filter_elmts}
    print("Principal filter generated by element(s): " + str(cncpt_list_name))
    print('----------------------------------------')
    print("\n")
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = 500,
                        figsize = [8,8])


def print_principal_ideal(self, cncpt_list):
    try:
        cncpt_list[0]
    except AttributeError:
        #goes here if a single concept (not in a list) is given to the function
        lattice_elmt_list = [func_cncpt_elmnt_to_lattice_elmnt(self, cncpt_list)]
        cncpt_list_name = [cncpt_list.name()]
    else:
        lattice_elmt_list = []
        cncpt_list_name = []
        for cncpt in cncpt_list:
            lattice_elmt_list.append(func_cncpt_elmnt_to_lattice_elmnt(self, cncpt))
            cncpt_list_name.append(cncpt.name())
    ideal_elmts = self.lattice().order_ideal(lattice_elmt_list)
    dico_color = {'#5CC3FF':ideal_elmts}
    print("Principal ideal generated by element(s): " + str(cncpt_list_name))
    print('----------------------------------------')
    print("\n")
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = 500,
                        figsize = [8,8])


def print_dilation_cncpt(self, cncpt, dilation_function, func_val, neighborhood_dist):
    dilation_cncpts = dilation_function(cncpt, func_val, neighborhood_dist)
    try:
        dilation_cncpts[0]
    except:
        dilation_latt_elmts = [func_cncpt_elmnt_to_lattice_elmnt(self, dilation_cncpts)]
    else:
        dilation_latt_elmts = map(lambda dilated_cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, \
 dilated_cncpt),dilation_cncpts)
    src_latt_elmt = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt)
    dico_color = {'#C7F5DF': [src_latt_elmt], '#FC7EFF': dilation_latt_elmts}
    print("Dilation of concept: " + cncpt.name() + ':')
    print("\n")
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = 500,
                        figsize = [8,8])


def print_erosion_cncpt(self, cncpt, erosion_function, func_val, neighborhood_dist):
    erosion_cncpts = erosion_function(cncpt, func_val, neighborhood_dist)
    try:
        erosion_cncpts[0]
    except:
        erosion_latt_elmts = [func_cncpt_elmnt_to_lattice_elmnt(self, erosion_cncpts)]
    else:
        erosion_latt_elmts = map(lambda eroded_cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, \
 eroded_cncpt), erosion_cncpts)
    src_latt_elmt = func_cncpt_elmnt_to_lattice_elmnt(self, cncpt)
    dico_color = {'#C7F5DF': [src_latt_elmt], '#FC7EFF': erosion_latt_elmts}
    print("Erosion of concept: " + cncpt.name() + ':')
    print("\n")
    self.lattice().show(vertex_colors=dico_color,
                        vertex_size = 500,
                        figsize = [8,8])

def print_congruence_and_quotient_latt(self, cncpt_list):
    #replace all with L.show(partition=congruence)
    cncpt_list_name = map(lambda cncpt: cncpt.name(), cncpt_list)
    latt_elmt_list = [map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt), cncpt_list)]
    congruence = self.lattice().congruence(latt_elmt_list)
    quotient_latt = self.lattice().quotient(congruence, labels='tuple')
    dico_color_cong = {}
```

```
        dico_color_quotient_latt = {}
        for block in congruence:
            dico_key = '#' + gen_hex_colour_code()
            dico_color_cong[dico_key] = block
        for sub_latt in quotient_latt:
            dico_key_q_latt = next(key for key in dico_color_cong.keys() if set(sub_latt) == set(\
    dico_color_cong[key]))
            dico_color_quotient_latt[dico_key_q_latt] = {sub_latt}
        print("Congruence generated by concepts: " + str(cncpt_list_name))
        print('----------------------------------------')
        print("\n")
        self.lattice().show(vertex_size = 500,
                                      figsize = [12,10],
                                      vertex_color = dico_color_cong)
        print("Quotient Lattice modulo congruence (above): ")
        print('----------------------------------------')
        print("\n")
        quotient_latt.show(vertex_size = 500,
                           figsize = [8,8],
                           vertex_color = dico_color_quotient_latt)

    def print_congruence(self, congruence):
        self.lattice().show(vertex_size = 400,figsize = [10,10], partition=congruence)


    def print_sub_latt_isomorphic_to_quotient_latt(self, cncpt_list, nmbr_of_sub_latt_to_print):
        iso_sub_latt = self.compute_sub_latt_isomorphic_to_quotient_latt(cncpt_list)
        len_iso_sub_latt = len(iso_sub_latt)
        i = 0
        if nmbr_of_sub_latt_to_print > len_iso_sub_latt: nmbr_of_sub_latt_to_print = len_iso_sub_latt
        while i != nmbr_of_sub_latt_to_print:
            rand_int = random.randint(1, len_iso_sub_latt)
            #dico_elmt_labels = self.gen_label_latt_dico()
            dico_color = {}
            double_irr = iso_sub_latt[rand_int].double_irreducibles()
            dico_color['#FFF585'] = double_irr
            join_irr = iso_sub_latt[rand_int].join_irreducibles()
            dico_color['#C7F5DF'] = [elmt for elmt in join_irr if elmt not in double_irr]
            meet_irr = iso_sub_latt[rand_int].meet_irreducibles()
            dico_color['#5CC3FF'] = [elmt for elmt in meet_irr if elmt not in double_irr]
            iso_sub_latt[rand_int].show(vertex_colors=dico_color,
                                        vertex_size = 400,
                                        figsize = [7,7])
            i = i + 1

#     def print_struct_elmt(self, center_obj, val):
#         struct_elmts = self.compute_struct_elmt_from_obj(center_obj, val)
#         ideal_elmts = self.lattice().order_ideal(lattice_elmt_list)
#         d = {'5CC3FF':struct_elmnt}
#         print("Structuring element generated by object: " + str(center_obj))
#         print("Principal ideal generated by element(s): " + str(cncpt_list_name))
#         print('----------------------------------------')
#         print("\n")
#         self.lattice().plot(vertex_colors=d).show()
#         print("\n")


# - - - - - - - - - - - - - -
#        Lattice
# - - - - - - - - - - - - - -

    def compute_sub_latt_isomorphic_to_quotient_latt(self, cncpt_list):
        #cncpt_list_name = map(lambda cncpt: cncpt.name(), cncpt_list)
        latt_elmt_list = [map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(self, cncpt), cncpt_list)]
        congruence = self.lattice().congruence(latt_elmt_list)
        quotient_latt = self.lattice().quotient(congruence, labels='tuple')
        return list(self.lattice().isomorphic_sublattices_iterator(quotient_latt))




#============================
#        Function space
#============================



#----------------------------
#         TOOL BOX
```

```
#-----------------------------

def replace_words_in_text_file(input_file_path, output_file_path, pair_list):
    """
    This function replace words by other in a text file, according to pairs of words.
    """
    file = open(input_file_path,"r")
    filedata = file.read()
    file.close()
    for pair in pair_list:
        filedata = filedata.replace(pair[0], pair[1])
    output_file = open(output_file_path,"w")
    output_file.write(filedata)

def gen_hex_colour_code():
    return ''.join([random.choice('0123456789ABCDEF') for x in range(6)])


#-----------------------------
#       HARMONIC TOOL BOX
#-----------------------------

def harmonic_formal_concept_to_set_of_interval(cncpt):
    if isinstance(cncpt, FormalConcept):
        if cncpt.intent() == []:
            return [0]
        else:
            try:
                char_extent = map(lambda obj: obj.split('.'), cncpt.extent())
            except:
                print("Concept name is not formated correctly, must use '\.\' between tone.")
            else:
                min_len_obj_in_extent = min(map(lambda obj: len(obj), char_extent))
                set_interval_set_char = [obj for obj in char_extent if len(obj) == min_len_obj_in_extent]
                set_interval_set_int = map(lambda interval_set_char: map(lambda interval: int(interval), \
     interval_set_char), set_interval_set_char)
                set_interval = min(set_interval_set_int, key = lambda interval_set_int: len([(a,b) for a in\
      interval_set_int
                                                                                            for b in \
     interval_set_int
                                                                                            if b != 0 and\
      a != b and a % b == 0]))
                return set_interval
    else:
        print("Parameter error: the function argument must be a FormalConcept instance.")

def lee_distance_btwn_elmt_cyclic_grp(group_order, elmt_1, elmt_2):
    r = Integers(group_order)
    diff =[r(elmt_1)-r(elmt_2), r(elmt_2)-r(elmt_1)]
    return min(diff)


def lee_dist_list_btw_interval(interval_set, group_order):
    lee_dist_list = map(lambda pair: lee_distance_btwn_elmt_cyclic_grp(group_order, pair[0], pair[1]),
                        [pair for pair in itertools.combinations(interval_set, 2)])
    return lee_dist_list

def freq_interval_cardinality(interval_list):
    card = 0
    interval_set = list(set(interval_list))
    for interval in interval_set:
        card += interval_list.count(interval)
    return card

#-----------------------------
# TRANSLATE FORMAT OBJECTS
#-----------------------------

def dot_to_valid_csv(dot_file_path, csv_file_path):
    """
    This function transform a .dot file generated from FCASTONE into a Sage understandable .csv file.
    """
    file = open(dot_file_path,"r")
    text = file.readlines()
    file.close()
    outputfile = open(csv_file_path,"w")
    keyword1 = re.compile(r"\[|\{|\}")
    keyword2 = re.compile(r" -> ")
    for line in text:
        if not keyword1.search (line):
```

```
            outputfile.write(keyword2.sub(",",line))

#-----------------------------

def csv_to_valid_digraph(csv_file_path):
    """
    This function transform a .csv file into a Sage Lattice.
    """
    #1) convert the .csv file into a Sage DiGraph
    graph = networkx.read_edgelist(csv_file_path,delimiter=",")
    stupid_directed_graph = DiGraph(graph)
    print("Stupid Digraph: ")
    print('----------------------------------------')
    print("\n")
    stupid_directed_graph.show(vertex_labels=False, figsize = [5,5])
    #2) convert the Sage DiGraph into a digraph that carry the lattice struct
    file = open(csv_file_path,"r")
    text = file.readlines()
    file.close()
    for line in text:
        vertex = [vertex.strip() for vertex in line.split(',')]
        valid_digraph = stupid_directed_graph
        if valid_digraph.has_edge(vertex[0],vertex[1]):
            #print(vertex[0],vertex[1])
            valid_digraph.delete_edge(vertex[0],vertex[1])
    print("Valid Digraph: ")
    print('----------------------------------------')
    print("\n")
    valid_digraph.show(vertex_labels=False, figsize = [5,5])
    return valid_digraph

#-----------------------------

def digraph_to_lattice(valid_digraph):
    """
    This function transform a Sage DiGraph into a Sage Lattice.
    """
    poset = Poset((valid_digraph.networkx_graph().nodes(),valid_digraph.networkx_graph().edges()),\
     cover_relations=True)

    if poset.top() and  poset.bottom():
        galois_lattice = LatticePoset(valid_digraph)
        print("Galois Lattice: ")
        print('----------------------------------------')
        print("\n")
        galois_lattice.show(label_elements=False, vertex_size=400, figsize=[8,8])
    return galois_lattice


#-----------------------------
# VALUATIONS TEST
#-----------------------------

func_supermodular_property = lambda FC_or_CL, a, b, a_meet_b, a_join_b, f: f(FC_or_CL, a) + f(FC_or_CL, b) \
    <= f(FC_or_CL, a_meet_b) + f(FC_or_CL, a_join_b)
func_submodular_property = lambda FC_or_CL, a, b, a_meet_b, a_join_b, f: f(FC_or_CL, a) + f(FC_or_CL, b) >=\
     f(FC_or_CL, a_meet_b) + f(FC_or_CL, a_join_b)
func_lattice_pair_to_concept_pair = lambda F_ctxt, lattice_pair: (next(x[1] for x in F_ctxt.concepts() if x\
    [1].name() == lattice_pair[0]),
                                                                  next(y[1] for y in F_ctxt.concepts() if y[1].\
    name() == lattice_pair[1]))
func_concept_pair_to_lattice_pair = lambda F_ctxt, concept_pair: (next(x for x in F_ctxt.lattice().list() \
    if x == concept_pair[0].name()),
                                                                  next(y for y in F_ctxt.lattice().list() if \
    y == concept_pair[1].name()))
func_lattice_elmnt_to_cncpt_elmnt = lambda F_ctxt, lattice_elmnt: next(x[1] for x in F_ctxt.concepts() if x\
    [1].name() == lattice_elmnt)
func_cncpt_elmnt_to_lattice_elmnt = lambda F_ctxt, concept_elmt: next(x for x in F_ctxt.lattice().list() if\
     x == concept_elmt.name())
func_generate_power_set = lambda my_set: reduce(lambda z, x: z + [y + [x] for y in z], my_set, [[]])

#-----------------------------
def is_lower_valuation(F_context, val):
    #val is a real-valued function on a concept lattice
    if not isinstance(F_context, FormalContext):
        print('Parameter type error: the first parameter of this function must be an instance of \
    FormalContext class.')
    else:
        try:
            cncpt_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
```

```
    valuation
            valuation_test = val(F_context, cncpt_test)
        except AttributeError:
            #lambda valuation is based on SAGE lattice elements
            L = F_context.lattice()
            for lattice_elmt_pair in itertools.product(L.list(), repeat=2): #compute too much useless pairs\
     -> itertools.combinations()
                elmts_pair_meet = L.meet(lattice_elmt_pair[0],lattice_elmt_pair[1])
                elmts_pair_join = L.join(lattice_elmt_pair[0],lattice_elmt_pair[1])
                if not func_supermodular_property(L,
                                                  lattice_elmt_pair[0],
                                                  lattice_elmt_pair[1],
                                                  elmts_pair_meet,
                                                  elmts_pair_join,
                                                  val):
                    #print(lattice_elmt_pair)
                    return 0
            return 1
        else:
            #lambda valuation is based on FormalConcept instances
            for concept_pair in map(lambda e: (e[0][1], e[1][1]), itertools.product(F_context.concepts(), \
    repeat=2)): #compute too much useless pairs
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, concept_pair)
                lattice_pair_meet = F_context.lattice().meet(lattice_pair[0],lattice_pair[1])
                lattice_pair_join = F_context.lattice().join(lattice_pair[0],lattice_pair[1])
                concept_pair_meet_and_join = func_lattice_pair_to_concept_pair(F_context, (\
    lattice_pair_meet, lattice_pair_join))
                if not func_supermodular_property(F_context,
                                                  concept_pair[0],
                                                  concept_pair[1],
                                                  concept_pair_meet_and_join[0],
                                                  concept_pair_meet_and_join[1],
                                                  val):
                    #print(map(lambda c: c.name(), concept_pair))
                    return 0
            return 1


#-----------------------------
def is_upper_valuation(F_context, val):
    #val is a real-valued function on a concept lattice
    if not isinstance(F_context, FormalContext):
        print('Parameter type error: the first parameter of this function must be an instance of \
     FormalContext class.')
    else:
        try:
            cncpt_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
     valuation
            valuation_test = val(F_context, cncpt_test)
        except AttributeError:
            #lambda valuation is based on SAGE lattice elements
            L = F_context.lattice()
            for lattice_elmt_pair in itertools.product(L.list(), repeat=2): #compute too much useless pairs
                elmts_pair_meet = L.meet(lattice_elmt_pair[0],lattice_elmt_pair[1])
                elmts_pair_join = L.join(lattice_elmt_pair[0],lattice_elmt_pair[1])
                if not func_submodular_property(L, lattice_elmt_pair[0],
                                                  lattice_elmt_pair[1],
                                                  elmts_pair_meet,
                                                  elmts_pair_join,
                                                  val):
                    #print(lattice_elmt_pair)
                    return 0
            return 1
        else:
            #lambda valuation is based on FormalConcept instances
            for concept_pair in map(lambda e: (e[0][1], e[1][1]), itertools.product(F_context.concepts(), \
    repeat=2)): #compute too much useless pairs
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, concept_pair)
                lattice_pair_meet = F_context.lattice().meet(lattice_pair[0],lattice_pair[1])
                lattice_pair_join = F_context.lattice().join(lattice_pair[0],lattice_pair[1])
                concept_pair_meet_and_join = func_lattice_pair_to_concept_pair(F_context, (\
    lattice_pair_meet, lattice_pair_join))
                if not func_submodular_property(F_context,
                                                  concept_pair[0],
                                                  concept_pair[1],
                                                  concept_pair_meet_and_join[0],
                                                  concept_pair_meet_and_join[1],
                                                  val):
                    #print(map(lambda c: c.name(), concept_pair))
                    return 0
```

```
            return 1


#-----------------------------
def is_isotone_fnct(F_context, f):
    #a real-valued fonction is isotone (increasing) if for all pairs of lattice concept a1<=a2 we have f(a1\
     )<=f(a2)
    if not isinstance(F_context, FormalContext):
        print('Parameter type error: the first parameter of this function must be an instance of \
     FormalContext class.')
    else:
        try:
            cncpt_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
     valuation
            valuation_test = f(F_context, cncpt_test)
        except AttributeError:
            #lambda valuation is based on SAGE lattice elements
            L = F_context.lattice()
            #consider only the pairs a1<=a2 in the lattice L
            for lattice_pair in filter(lambda e: L.is_lequal(e[0], e[1]), itertools.product(L.list(), \
     repeat=2)):
                if not (f(L, lattice_pair[0]) <= f(L, lattice_pair[1])):
                    return 0
            return 1
        else:
            #lambda valuation is based on FormalConcept instances
            a_inf_b_lattice_pairs = filter(lambda lattice_elmt_pair: F_context.lattice().is_lequal(\
     lattice_elmt_pair[0], lattice_elmt_pair[1]),
                                           itertools.product(F_context.lattice().list(), repeat=2))
            for concept_pair in map(lambda pair: func_lattice_pair_to_concept_pair(F_context, pair), \
     a_inf_b_lattice_pairs):
                if not (f(F_context, concept_pair[0]) <= f(F_context, concept_pair[1])):
                    return 0
            return 1


#-----------------------------
def is_antitone_fnct(F_context, f):
    #a real-valued fonction is antitone (decreasing) if for all pairs of lattice concepts a1<=a2 we have f(\
     a1)>=f(a2)
    if not isinstance(F_context, FormalContext):
        print('Parameter type error: the first parameter of this function must be an instance of \
     FormalContext class.')
    else:
        try:
            cncpt_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
     valuation
            valuation_test = f(F_context, cncpt_test)
        except AttributeError:
            #lambda valuation is based on SAGE lattice elements
            L = F_context.lattice()
            #consider only the pairs a1<=a2 in the lattice L
            for lattice_pair in filter(lambda e: L.is_lequal(e[0], e[1]), itertools.product(L.list(), \
     repeat=2)):
                if not (f(L, lattice_pair[0]) >= f(L, lattice_pair[1])):
                    return 0
            return 1
        else:
            #lambda valuation is based on FormalConcept instances
            a_inf_b_lattice_pairs = filter(lambda lattice_elmt_pair: F_context.lattice().is_lequal(\
     lattice_elmt_pair[0], lattice_elmt_pair[1]),
                                           itertools.product(F_context.lattice().list(), repeat=2))
            for concept_pair in map(lambda pair: func_lattice_pair_to_concept_pair(F_context, pair), \
     a_inf_b_lattice_pairs):
                if not (f(F_context, concept_pair[0]) >= f(F_context, concept_pair[1])):
                    return 0
            return 1


#-----------------------------
def is_suitable_valuation(F_context, val):
    if is_antitone_fnct(F_context, val) and is_upper_valuation(F_context, val):
        print('This valuation is a decreasing upper valuation')
    elif is_antitone_fnct(F_context, val) and is_lower_valuation(F_context, val):
        print('This valuation is a decreasing lower valuation')
    elif is_isotone_fnct(F_context, val) and is_lower_valuation(F_context, val):
        print('This valuation is an increasing lower valuation')
    elif is_isotone_fnct(F_context, val) and is_upper_valuation(F_context, val):
        print('This valuation is an increasing upper valuation')
    else:
```

```
        print('This valuation is not a suitable valuation')
    return


#-----------------------------
def valuation_to_metric(F_context, val):
    #antitone & upper val
    if is_antitone_fnct(F_context, val) and is_upper_valuation(F_context, val):
        def metric(F_context, cncpt1, cncpt2, val):
            try:
                concpet_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
    valuation
                valuation_test = val(F_context, concpet_test)
            except AttributeError:
                #lambda valuation is based on SAGE lattice elements
                L = F_context.lattice()
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_meet = L.meet(lattice_pair[0],lattice_pair[1])
                return 2*val(L, lattice_pair_meet) - val(L, lattice_pair[0]) - val(L, lattice_pair[1])
            else:
                #lambda valuation is based on FormalConcept instances
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_meet = F_context.lattice().meet(lattice_pair[0],lattice_pair[1])
                concept_pair_meet = func_lattice_elmnt_to_cncpt_elmnt(F_context, lattice_pair_meet)
                return 2*val(F_context, concept_pair_meet) - val(F_context, cncpt1) - val(F_context, cncpt2\
    )
    #antitone & lower val
    elif is_antitone_fnct(F_context, val) and is_lower_valuation(F_context, val):
        def metric(F_context, cncpt1, cncpt2, val):
            try:
                concpet_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
    valuation
                valuation_test = val(F_context, concpet_test)
            except AttributeError:
                #lambda valuation is based on SAGE lattice elements
                L = F_context.lattice()
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_join = L.join(lattice_pair[0],lattice_pair[1])
                return val(L, lattice_pair[0]) + val(L, lattice_pair[1]) - 2*val(L, lattice_pair_join)
            else:
                #lambda valuation is based on FormalConcept instances
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_join = F_context.lattice().join(lattice_pair[0],lattice_pair[1])
                concept_pair_join = func_lattice_elmnt_to_cncpt_elmnt(F_context, lattice_pair_join)
                return val(F_context, cncpt1) + val(F_context, cncpt2) - 2*val(F_context, concept_pair_join\
    )
    #isotone & lower val
    elif is_isotone_fnct(F_context, val) and is_lower_valuation(F_context, val):
        def metric(F_context, cncpt1, cncpt2, val):
            try:
                concpet_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
    valuation
                valuation_test = val(F_context, concpet_test)
            except AttributeError:
                #lambda valuation is based on SAGE lattice elements
                L = F_context.lattice()
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_meet = L.meet(lattice_pair[0],lattice_pair[1])
                return val(L, lattice_pair[0]) + val(L, lattice_pair[1]) - 2*val(L, lattice_pair_meet)
            else:
                #lambda valuation is based on FormalConcept instances
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_meet = F_context.lattice().meet(lattice_pair[0],lattice_pair[1])
                concept_pair_meet = func_lattice_elmnt_to_cncpt_elmnt(F_context, lattice_pair_meet)
                return val(F_context, cncpt1) + val(F_context, cncpt2) - 2*val(F_context, concept_pair_meet\
    )
    #isotone & upper val
    elif is_isotone_fnct(F_context, val) and is_upper_valuation(F_context, val):
        def metric(F_context, cncpt1, cncpt2, val):
            try:
                concpet_test = F_context.concepts()[0][1] #pick the first concept for testing the lambda \
    valuation
                valuation_test = val(F_context, concpet_test)
            except AttributeError:
                #lambda valuation is based on SAGE lattice elements
                L = F_context.lattice()
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_join = L.join(lattice_pair[0],lattice_pair[1])
                return 2*val(L, lattice_pair_join) - val(L, lattice_pair[0]) - val(L, lattice_pair_join[1])\
    #verify
```

```
            else:
                #lambda valuation is based on FormalConcept instances
                lattice_pair = func_concept_pair_to_lattice_pair(F_context, (cncpt1,cncpt2))
                lattice_pair_join = F_context.lattice().join(lattice_pair[0],lattice_pair[1])
                concept_pair_join = func_lattice_elmnt_to_cncpt_elmnt(F_context, lattice_pair_join)
                return 2*val(F_context, concept_pair_join) - val(F_context, cncpt1) - val(F_context, cncpt2\
    ) #verify
    #return the function def defined above (technique known as currying)
    return metric


#-----------------------------



#-----------------------------
# Morphological Operator TEST
#-----------------------------

def is_dilation_over_lattice(F_context, dilation_func, val, neigborhood_dist):
    latt_elmt_list = map(lambda cncpt: func_cncpt_elmnt_to_lattice_elmnt(cncpt), F_context)
    #test that dilattion preserve least element
    bottom_concept = func_lattice_elmnt_to_cncpt_elmnt(F_context, F_context.lattice().bottom())
    join_of_dilation_bot = func_lattice_elmnt_to_cncpt_elmnt(F_context, F_context.lattice().join(
                           [func_cncpt_elmnt_to_lattice_elmnt(F_context, dilation_func(bottom_concept, val,\
     neigborhood_dist))]))
    dilation_of_join_bot = dilation_func(func_lattice_elmnt_to_cncpt_elmnt(F_context, F_context.lattice().\
    join(
                           [func_cncpt_elmnt_to_lattice_elmnt(F_context, bottom_concept)])), val, \
    neigborhood_dist)
    if join_of_dilation_bot == dilation_of_join_bot:
        return False
    else:
        return True
        #test that dilation commutes with supremum
        for cncpt in F_context.concepts():
            join_of_dilation = func_lattice_elmnt_to_cncpt_elmnt(F_context, F_context.lattice().join(
                               [func_cncpt_elmnt_to_lattice_elmnt(F_context, dilation_func(cncpt[1], val, \
    neigborhood_dist))]))
            dilation_of_join = dilation_func(func_lattice_elmnt_to_cncpt_elmnt(F_context, F_context.lattice\
    ().join(
                               [func_cncpt_elmnt_to_lattice_elmnt(F_context, cncpt[1])])), val, \
    neigborhood_dist)
            if not join_of_dilation == dilation_of_join:
                return False
        return True


#-----------------------------
#   Lattice
#-----------------------------

def lattice_properties(lattice):
    test1 = seven_tet_context.lattice().is_subdirectly_reducible(certificate=True)
    test2 = seven_tet_context.lattice().is_uniform()
    test3 = seven_tet_context.lattice().is_regular()
    test4 = seven_tet_context.lattice().is_supersolvable()
    test5 = seven_tet_context.lattice().is_simple(certificate=False)
    test6 = seven_tet_context.lattice().is_dismantlable()
    test7 = seven_tet_context.lattice().is_modular()
    test8 = seven_tet_context.lattice().is_distributive()
    test9 = seven_tet_context.lattice().is_vertically_decomposable()
    test10 = seven_tet_context.lattice().is_isoform(certificate=False)
    test11 = seven_tet_context.lattice().breadth()

    print("The lattice is: \n")
    print("- subdirectly reducible : " + str(test1) + "\n")
    print("- uniform               : " + str(test2) + "\n")
    print("- regular               : " + str(test3) + "\n")
    print("- supersolvable         : " + str(test4) + "\n")
    print("- simple                : " + str(test5) + "\n")
    print("- dismantlable          : " + str(test6) + "\n")
    print("- modular              : " + str(test7) + "\n")
    print("- distributive          : " + str(test8) + "\n")
    print("- vertically decmpsable : " + str(test9) + "\n")
    print("- isoform               : " + str(test10) + "\n")
    print("- breadth               : " + str(test11) + "\n")
```

```
def congruence_equiv_class_contains_cncpts(congruence, cncpt_list):
    latt_elemt_list = [func_cncpt_elmnt_to_lattice_elmnt(seven_tet_context, cncpt) for cncpt in cncpt_list]
    for equiv_class in congruence:
        if all(latt_elmt in equiv_class for latt_elmt in latt_elemt_list):
            return True
    return False
```