Travail d'étude et de recherche

Graphe des coupes sur images de labels

Romain Perrin

21 mai 2018

mots-clefs : morphologie mathématique, traitement d'image, images multivaluées, arbre des coupes, graphe des coupes

Encadrement : Benoît Naegel, équipe IMAGeS, ICube bureau C230 - b.naegel@unistra.fr

Je voudrais remercier Benoît Naegel *pour son encadrement, ses suggestions et ses conseils qui m'ont beaucoup aidé tout au long de ce TER et sans qui ce projet serait probablement moins abouti.*

Table des matières

1 Contexte

La morphologie mathématique a d'abord été définie sur les images binaires, puis sur les images en niveaux de gris. L'extension de cette définition aux images multivaluées est motivée par de multiples applications dans différents domaines. Dans ce cadre, la notion d'opérateur connexe réuni de puissants outils de traitement d'image qui reposent sur des représentations hiérarchiques des images : arbre des coupes, arbre de partitionnement binaire... En particulier, la notion d'arbre des coupes a fait l'objet de nombreux travaux et il existe plusieurs algorithmes efficaces permettant de le calculer en temps linéaire [1].

La principale difficulté de l'extension de cette théorie aux images multivaluées est justement causée par la nature de cet ensemble de couleurs de l'image. En effet, cet ensemble de couleurs n'est pas nécessairement muni d'une relation d'ordre totale contrairement aux ensembles binaires et niveaux de gris. Il existe deux principales familles d'approches permettant d'étendre la définition des arbres des coupes aux images multivaluées. La première famille est composée de méthodes visant à préserver la structure d'arbre de l'arbre des coupes. En revanche, la seconde famille est composée de méthodes usant de structures de données plus complexes n'étant plus des arbres telle que la structure de graphe orienté acyclique dans le cas qui nous intéresse ici. Bien que plus complexes, ces approches offrent néanmoins une plus grande richesse en tirant partie de la totalité de l'information structurelle présente dans l'ensemble de couleurs et permet de fait de créer des outils de traitement d'image précis.

Dans le cadre de ce TER, on se concentre sur cette deuxième famille de méthodes définissant la notion de graphe des coupes comme un graphe orienté acyclique par extension de l'arbre des coupes aux images multivaluées en se basant sur les travaux de B. Naegel et N. Passat décrit dans [2], [3] et [4].

2 Notations

Pour comprendre la notion de graphe des coupes, il est nécessaire de commencer par introduire la notion d'arbre des coupes qui lui est antérieure.

Soit Ω un ensemble fini et non vide.
Soit V un ensemble fini, non vide et muni d'une relation d'ordre $\leqslant$.
Une image est une fonction $I : \Omega \to V$.
Les ensembles Ω et V sont notés respectivement l'ensemble support et l'espace des valeurs de I.
Pour tout $x \in \Omega$, $I(x) \in V$ est la valeur de I en x.
Si $(V, \leqslant)$ est totalement (respectivement partiellement) ordonné, on dit que I est une image en niveaux de gris (respectivement multivaluée).
Un diagramme de Hasse d'une ensemble ordonné $(X, \leqslant)$ est un couple $(X, \prec)$ où $\prec$ est la relation de couverture associée à $\leqslant$.

Soit $x \in \Omega$ et $v \in V$.
On définit une fonction de seuillage λ pour un valeur $v \in V$ de I comme

$$\lambda_v : \begin{array}{l} V^\Omega \to 2^\Omega \\ I \mapsto \{x \in \Omega \mid I(x) \geqslant v\} \end{array}$$

On définit également la fonction cylindrique $C_{(X,v)}$ comme

$$C_{(X,v)} : \begin{array}{l} \Omega \to V \\ x \mapsto \begin{cases} v & \text{si } x \in X \\ \bot & \text{sinon} \end{cases} \end{array}$$

Si $(X, \leqslant)$ est un ensemble ordonné et que $x \in X$, on note $x^\uparrow = \{y \in X \mid y \geqslant x\}$ l'ensemble des éléments plus grands que x et $x^\downarrow = \{y \in X \mid y \leqslant x\}$ l'ensemble des éléments plus petits que x.
On note $C[X]$ l'ensemble des composantes connexes de X.
Si $Y \subseteq X$ le maximum de Y est noté $\overset{\leqslant}{\curlyvee} Y$ et le minimum est noté $\overset{\leqslant}{\curlywedge} Y$.

Une image $I : \Omega \to V$ peut être décomposée en fonctions cylindriques induites par opérations de seuillage et symétriquement, I peut être reconstruite par composition de ces fonctions cylindriques.

$$I = \overset{\leqslant}{\bigvee_{v \in V}} \overset{\leqslant}{\bigvee_{X \in C[\lambda_v(I)]}} C_{(X,v)}$$

Un diagramme de Hasse d'un ensemble ordonné $(X, \leqslant)$ est un couple $(X, \prec)$ où $\prec$ est la relation de couverture associée à $\leqslant$, définie pour tout $x, y \in X$ par $x \prec y$ si et seulement si $x < y$ et s'il n'existe aucun $z \in X$ tel que $x < y < z$.
On note Ψ l'ensemble des composantes connexes obtenues par seuillages successifs de I.

$$\Psi : \bigcup_{v \in V} C[\lambda_v(I)]$$

3 Arbre des coupes

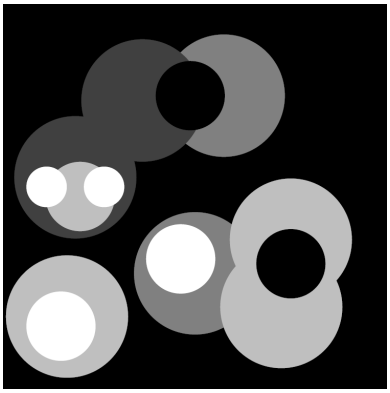
On se donne les propriétés suivantes :

— (P1) si $X \subseteq \Omega$ alors l'ensemble $C[x]$ est une partition de X.

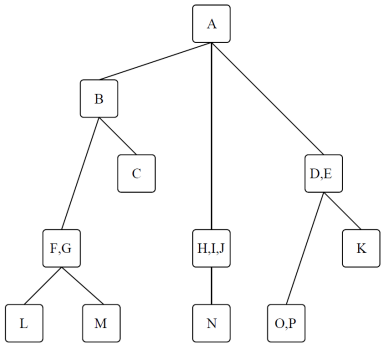
— (P2) si $X \subseteq Y \subseteq \Omega$, alors pour tout $A \in C[X]$, il existe un unique $B \in C[Y]$ tel que $A \subseteq B$.

— (P3) si $X \subseteq Y \subseteq \Omega$, $A \in C[Y]$ et $A \subseteq X$, alors $A \in C[X]$.

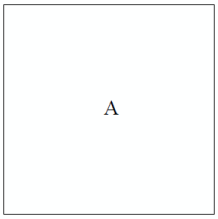
On suppose que $(X, \leqslant)$ est un ensemble totalement ordonné.
On montre des propriétés (P1) et (P2) ainsi que de la totalité de la relation $\leqslant$ que pour tout $(x^{\uparrow}, \subseteq)$ est un ensemble totalement ordonné.
Le diagramme de Hasse de l'ensemble $(\Psi, \subseteq)$ a une structure d'arbre de racine Ω.
Ce diagramme de Hasse est alors appelé arbre des coupes (component-tree).



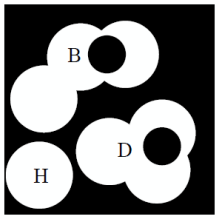
(a) I



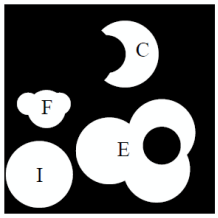
(b) $\mathfrak{T}$



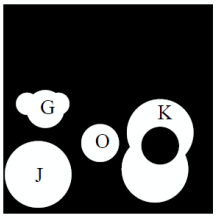
(c) $\lambda_0(I)$



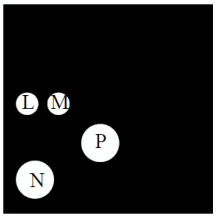
(d) $\lambda_1(I)$



(e) $\lambda_2(I)$



(f) $\lambda_3(I)$



(g) $\lambda_4(I)$

FIGURE 1 – (a) une image en niveaux de gris $I : \Omega \to V$ avec $\Omega \subset \mathbb{R}^2$ et $V = \{0, 1, 2, 3, 4\} \subset \mathbb{Z}$ (de 0 en noir à 4 en blanc) muni de la relation d'ordre classique $\leqslant$ sur $\mathbb{Z}$. (c-g) Les images seuillées $\lambda_v(I) \subseteq \Omega$ pour v variant de 0 à 4. (b) L'arbre des coupes $\mathfrak{T}$ de I. Les lettres (A-P) dans les nœuds correspondent aux composantes connexes de (c-g).

4 Graphe des coupes

Dans la section précédente, nous considérions $(V, \leqslant)$ muni d'une relation d'ordre total.
Dans la suite, nous relâchons la contrainte de totalité sur $\leqslant$.
Afin de définir la notion de graphe des coupes, nous enrichissons l'ensemble Ψ en assignant à chaque composante connexe la valeur $v \in V$ correspondant à la valeur de seuillage tel que $\Theta = \bigcup_{v \in V} C[\lambda_v(I)]x\{v\}$.
On étend la relation d'inclusion sur Θ en considérant ces valeurs. On obtient la relation d'ordre suivante $\trianglelefteq$ définie par $(X_1, v_1) \trianglelefteq (X_2, v_2) \iff (X_1 \subset X_2) \vee ((X_1 = X_2) \wedge (v_2 \leqslant v_1))$.
On peut désormais définir la notion de graphe des coupes de la sorte : le graphe des coupes G de I est le diagramme de Hasse $(\Theta, \blacktriangleleft)$ de l'ensemble partiellement ordonné $(\Theta, \trianglelefteq)$.

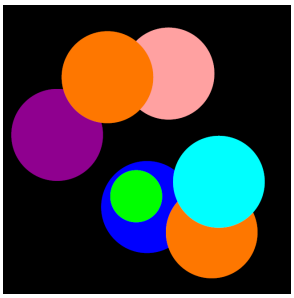
Le graphe des coupes peut être décliné en trois variantes notées $\mathfrak{G}$, $\dot{\mathfrak{G}}$ et $\ddot{\mathfrak{G}}$ de taille, complexité et richesse décroissante.

$\dot{\mathfrak{G}}$ est obtenu en considérant $\dot{\Theta} = \{(X, v) \in \Theta \mid \forall (X, v') \in \Theta, v \not< v'\}$.
$\ddot{\mathfrak{G}}$ est obtenu en considérant $\ddot{\Theta} = \{(X, v) \in \Theta \mid \exists x \in X, v = I(x)\}$.

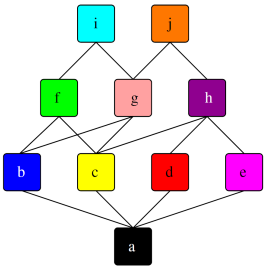
Les graphes des coupes héritent de la formule de (dé)composition classiquement associée aux arbres des coupes c'est-à-dire $I = \bigvee^{\leqslant}_{K \in \Theta} C_K$ où $\bigvee$ est l'opérateur de borne supérieure, $\trianglelefteq$ est l'ordre ponctuel sur les fonctions induites par $\leqslant$, et pour tout $(X, v) \in \Theta$, $C_{(X,v)}$ est la fonction cylindrique définie par

$$C_{(X,v)} : \begin{array}{c} \Omega \to V \\ x \mapsto \begin{cases} v & \text{si } x \in X \\ \bot & \text{sinon} \end{cases} \end{array}$$

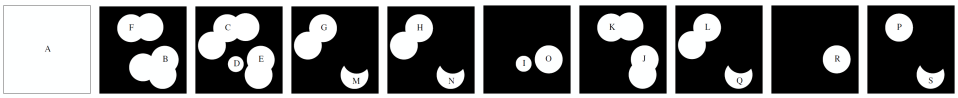
.



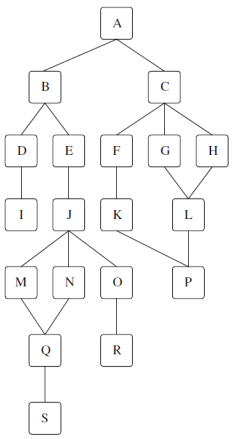
(a) I



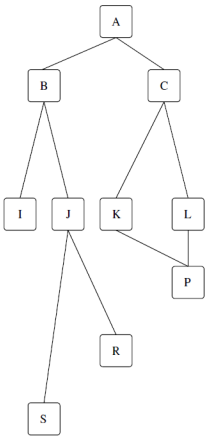
(b) $\mathfrak{T}$



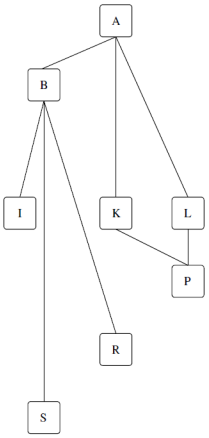
(c) $\lambda_a(I)$ (d) $\lambda_b(I)$ (e) $\lambda_c(I)$ (f) $\lambda_d(I)$ (g) $\lambda_e(I)$ (h) $\lambda_f(I)$ (i) $\lambda_g(I)$ (j) $\lambda_h(I)$ (k) $\lambda_i(I)$ (l) $\lambda_j(I)$



(m) $\mathfrak{G}$



(n) $\dot{\mathfrak{G}}$



(o) $\ddot{\mathfrak{G}}$

FIGURE 2 – (a) une image $I : \Omega \to V$ avec $V = \{a, ..., j\}$ (b) le diagramme de Hasse de la relation d'ordre $(X, \leqslant)$ (c-l) les images seuillées $\lambda_v(I)$ pour $v \in V$ (m-o) les trois variantes du graphe des coupes $\mathfrak{G}$, $\dot{\mathfrak{G}}$ et $\ddot{\mathfrak{G}}$ dont les lettres (A-S) correspondent aux composantes connexes de (c-l).

5 Objectifs

Dans la mesure où l'on ne dispose pas encore d'algorithme efficace pour calculer les variantes $\mathfrak{G}$ et $\dot{\mathfrak{G}}$, on se concentre sur le calcul de la variante $\ddot{\mathfrak{G}}$ au moyen de l'algorithme décrit dans [4] adapté de [3]. On dispose actuellement d'un code C++ prenant une image au format RVB en argument et produisant la variante $\ddot{\mathfrak{G}}$ en utilisant l'ordre marginal sur l'espace RVB en tant que fonction de comparaison.

$$\begin{aligned} &[\![0,255]\!]^3 \times [\![0,255]\!]^3 \to \{vrai, faux\} \\ \leqslant_{(\{R_1,V_1,B_1\},\{R_2,V_2,B_1\})} : &\; x \mapsto \begin{cases} vrai & \text{si } R_1 \leqslant R_2 \bigwedge V_1 \leqslant V_2 \bigwedge B_1 \leqslant B_2 \\ faux & \text{sinon} \end{cases} \end{aligned}$$

L'objectif de ce TER est d'enrichir ce programme afin d'utiliser des images de labels dont la relation d'ordre sur les labels sera matérialisée par un diagramme de Hasse fournit en entrée en plus de l'image au format RVB. Un index sera également ajouté en paramètre. Celui-ci permettra d'établir la correspondance entre les labels arbitraires définis dans le diagramme de Hasse et les valeurs de l'espace RVB au moyen d'une fonction de correspondance C.

$$C_l : \begin{aligned} &L \to [\![0,255]\!]^3 \\ &x \mapsto \{R_l, V_l, B_l\} \end{aligned}$$

avec $l \in L$ un label et L l'ensemble de définition des labels de I.

6 Travail réalisé

6.1 Étude documentaire

La partie préliminaire du travail consistait à se documenter sur la théorie des arbres des coupes, pré-requis nécessaire dans la mesure où les graphes des coupes sont une extension des arbres des coupes aux images multivaluées avec absence d'ordre total. La lecture s'est notamment portée sur l'étude et la comparaison des algorithmes capables de produire l'arbre des coupes d'une image en niveaux de gris [1]. Dans un second temps, il était nécessaire d'approfondir ces connaissances par l'étude de la théorie des graphes des coupes dans [2] et [3]. Enfin il était de mise de prendre connaissance des algorithmes permettant le calcul du graphe des coupes d'une image à travers l'étude de [4]. Cette étude documentaire a permis de fixer le contexte du projet ainsi que de mettre en perspective le travail à réaliser.

La seconde partie du travail documentaire a été d'étudier le projet ainsi que le code existant dans le dépôt Github [6]. Cette étude de code combinée à l'étude des papiers de recherche a notamment permis de mettre en évidence le fait que les variantes $\mathfrak{G}$ et $\dot{\mathfrak{G}}$ ne disposent pas d'un algorithme suffisamment robuste à ce jour et de fait ne seraient pas traitées. Une seconde étude de code sur une version nouvellement développée (sur le Gitlab ICube cette fois-ci) et bénéficiant grandement de la généricité était nécessaire afin de fixer le cadre du projet et de planifier l'intégration du code qui serait développé.

6.2 Implémentations

Cette section décrit les différentes classes qui ont été développées ainsi que les choix d'implémentation qui ont été fait afin de produire le graphe des coupes. Chaque sous-section se concentre sur la description d'une classe particulière décrivant les structures de données utilisées, les méthodes développées ainsi que les interactions entre lesdites classes.

6.2.1 Classe Label

La classe ***Label*** a pour objectif de représenter les labels arbitraires définis dans le diagramme de Hasse. En effet, via le diagramme de Hasse, on travaille sur un relation d'ordre partielle définie sur des labels arbitraires et non sur les pixels de l'image, d'où la nécessité de disposer d'une structure de données permettant de les représenter. Les labels seront lus dans un graphe dit diagramme de Hasse qui est représenté dans le format DOT. Le format DOT [5] définit les identifiants de nœuds qui contiendront les labels comme étant des chaînes alphanumériques ne commençant pas par un nombre, des nombres, des chaînes doublement cotées ("...") ou bien des chaînes HTML (<...>).

Le choix s'est porté sur l'utilisation de chaînes de caractères de la STL, capables de contenir toutes les possibilités offertes par le langage DOT. De plus, elles permettent la facilité d'utilisation dans les containers et les algorithmes offerts par la STL.

6.2.2 Classe Index

Comme dit précédemment, les images de sont pas à proprement parler des images de labels. Ce niveau d'abstraction est obtenu en augmentant l'image avec l'addition d'un index. Cet index est l'implémentation de la fonction de correspondance C de la section "Objectifs".

La classe ***Index*** permet d'effectuer le passage de l'environnement concret représenté par l'espace des valeurs de l'image (RVB) à l'environnement abstrait représenté par l'ensemble des labels. Étant donné que les labels ne sont rien de plus que des chaînes de caractères, il serait inefficace du point de vue de la mémoire de créer une image paramétrée par lesdits labels. En revanche, il existe une astuce consistant à utiliser un double appairage, c'est-à-dire de créer deux niveaux d'abstraction. Le premier niveau est le passage de l'espace des valeurs de l'image V à l'ensemble des labels abstraits L. Le second niveau consiste à créer un appairage entre les labels abstraits L et un ensemble d'entiers N. On assure la propriété de transitivité des deux indirections. Ainsi à chaque valeur de l'espace des valeurs de l'image $v \in V$, on associe un et un seul label abstrait $l \in L$ et de même un et un seul entier $n \in N$.

Le premier appairage est représenté au moyen d'une table de hachage (map de la STL). Il y a cependant deux choix d'indexation possibles. De deux choses l'une :

— Soit l'on choisi d'indexer la table de hachage sur le type de pixel de l'image c'est-à-dire que l'ensemble des valeurs de l'image V est l'ensemble des clefs de la table. Cela revient à implémenter la fonction de correspondance inverse

$$C^{-1} : \begin{array}{l} [\![0,255]\!]^3 \rightarrow L \\ x \mapsto l \in L \end{array}$$

De fait l'obtention d'un label à partir d'une valeur de l'image revient à accéder à un élément d'une table de hachage. La complexité de cette opération est logarithmique en fonction de la taille de la table [7]. Ce choix est intéressant lorsque l'on a besoin d'effectuer la conversion $v \in V \rightarrow l \in L$.

— Soit l'on choisi d'indexer la tache de hachage sur les labels abstraits c'est-à-dire que l'ensemble des labels L est l'ensemble des clefs de la table. Cela revient à implémenter la fonction de correspondance

$$C : \begin{array}{l} L \rightarrow [\![0,255]\!]^3 \\ x \mapsto \{R_l, V_l, B_l\} \end{array}$$

De fait l'obtention d'un label à partir d'une valeur de l'image revient à effectuer une recherche dans la table sur les valeurs et non les chefs. La complexité de cette opération est linéaire en fonction de la taille de la table. Ce choix est intéressant lorsque l'on a besoin d'effectuer la conversion $l \in L \rightarrow v \in V$.

Le premier choix a été retenu pour ce projet étant donné que lors de la construction du graphe des coupes, on dispose d'une image dont l'information est une valeur de l'image $v \in V$ (ici V = RVB) et on a besoin d'obtenir le label abstrait $l \in L$ correspondant. L'indexation sur V est donc plus efficace en temps du point de vue de la production du graphe des coupes.

La classe Index prodigue tout de même deux méthodes permettant d'obtenir un sens ou l'autre du premier niveau d'abstraction entre V et L.

— ***Label getLabelFromPixelType(RVB pixelType)*** qui est une implémentation de la fonction de correspondance C^{-1}.

— ***RVB getPixelTypeFromLabel(Label label)*** qui est une implémentation de la fonction de correspondance C.

Afin de spécifier cet index, un format de fichier simple a été mis en place. L'appairage $T \rightarrow L$ est chargé au moyen d'une fonction ***loadMappingFromFile*** depuis un fichier dont la structure est la suivante :

— Chaque ligne comporte une association $L \leftrightarrow T$.
— Dans le cas où L = Label (la classe Label) le premier argument est une chaîne de caractères.
— Dans le cas où T = RVB (définit en tant que triplet d'entiers) il y a 3 arguments pour T.
— Tous les arguments peu importe leur nombre sont séparés par un espace.

Le second appairage est également représenté au moyen d'une table de hachage (map de la STL). Pour les même raisons que pour le premier appairage, cette table est indexée sur les labels en usant de L comme ensemble des clefs. L'astuce consistant à générer une image contenant les entiers représentant les labels et non les lebels eux-mêmes est effectuée au moyen de la méthode ***void convertImageToInt(Image<RVB>)***. De fait il est nécessaire de définir les opérateurs de comparaisons sur les entiers plutôt que sur les labels.

6.2.3 Classe HasseDiagram

Tel que décrit dans la section "Classe Label", les labels sont inscrits non directement dans l'image mais dans un graphe dit diagramme de Hasse. Ce graphe contient les relations de l'ordre partiel sur lesdits labels. Ce diagramme de Hasse est spécifié dans le format DOT [5]. La classe ***HasseDiagram*** ne prodigue pas de méthode particulières en dehors d'une méthode de chargement d'un graphe et d'une méthode permettant de rechercher des chemins dans le graphe (cette méthode sera décrite dans la section suivante).

La méthode ***loadFromFile*** doit permettre d'analyser un fichier au format DOT et de remplir un structure de graphe orienté acyclique. Ledit analyseur est assez ardu à réaliser "à la main" dû à sa complétude. Il existe cependant plusieurs bibliothèques et outils pour réaliser cet analyseur.

— La bibliothèque ***Graphviz*** [8] permet de lire un fichier au format et de remplir une structure de la bibliothèque ***cgraph***. Ceci dit, ***cgraph*** est une bibliothèque de spécification de graphes écrite en C non en C++. De plus, les bibliothèques ***Graphviz*** et ***cgraph*** seront des dépendances supplémentaires du projet.

— La bibliothèque ***pydot*** [9] est une interface écrite en Python pour le langage de spécification de graphe DOT. Elle possède notamment un analyseur pour les fichier écrits dans le langage DOT. Cette solution est pertinente à moyen et long terme dans la mesure où ce projet est

destiné à être transposé en Python. Dans le cas présent, le projet est écrit en C++ et faire exécuter du code en Python par des méthodes en C++ nécessite des ajustements. L'usage de l'API ***Python/C*** [10] est nécessaire afin d'effectuer cette adaptation.

— La bibliothèque ***Boost*** [11] dispose d'une méthode ***read_graphviz*** [12] permettant de lire des fichier écrits dans langage DOT. Il faut cependant noter que cette implémentation n'est pas basée sur la bibliothèque ***Graphviz*** mais sur la description qu'en fait la documentation de Graphviz et sur des tests empiriques. De fait certains résultats peuvent être légèrement différents. Cette solution nécessite d'ajouter la bibliothèque ***Boost*** ou du moins au minimum ses modules ***Graph*** et ***Regex*** ainsi que d'y ajouter des bibliothèques statiques et/ou dynamiques. Ceci brise l'esprit du projet dont la politique est d'être développé en "header-only" par souci de simplicité d'utilisation et de compilation.

— La bibliothèque ***Boost::Spirit*** [13] contient un module ***Boost::Spirit::qi*** [14] permettant de générer un analyseur à partir des règles de la grammaire. Cela introduit également une dépendance à la bibliothèque ***Boost***.

— Pour rester dans le domaine des outils générant des analyseurs, il existe également ***Flex (Lex)*** [15] permettant de générer un analyseur lexical et ***Bison (Yacc)*** [16] permettant de réaliser un analyseur syntaxique à partir de la grammaire donnée par [5]. Ces outils sont anciens et donc éprouvés mais plutôt adaptés aux projets écrits en C. Néanmoins, il possèdent un mode C++ qui pourrait être intéressant. Tout comme les solutions précédents, cette solution ajoute deux dépendances pour ***Flex*** et pour ***Bison***.

Une première version d'un analyseur pour une grammaire réduite par rapport à celle donnée par [5] a été réalisée avec les outils Flex et Bison en C. Cependant, par manque de temps et de connaissances vis à vis du mode C++ offert par ces outils, une version définitive en C++ n'a pu être intégrée au projet final. En tant que solution temporaire, la méthode ***loadFromFile*** effectue une analyse "faite maison" qui comporte certains inconvénients. Cette méthode ne prend en compte que les identifiants de nœuds et ne supporte pas les options telles que le renommage d'un label via l'instruction [label = "chaîne"]. Néanmoins, la méthode est capable d'analyser le fichier DOT et de remplir la structure de diagramme de Hasse utilisée pour les tests avec succès.

6.2.4 Classe LabelOrdering

Le diagramme de Hasse contenu dans la classe ***HasseDiagram*** contient la relation d'ordre partiel sur les labels. La classe ***LabelOrdering*** est une classe dérivée de la classe ***Ordering***, spécialisée par des entiers représentant les labels abstraits (labelOrdering = Ordering<int>). la classe LabelOrdering contient deux informations distinctes : les relations entre tous les labels définis dans le diagramme de Hasse et la priorité associée à chaque label.

Afin de rendre le calcul du graphe des coupes le plus rapide possible, les comparaisons qui y sont effectuées entre les différents labels se doivent d'être le moins gourmand en temps. Le choix a donc été fait de pré-calculer et stocker toutes les comparaisons possibles entre labels afin de garantir un temps de réponse constant. La classe LabelOrdering contient de ce fait un tableaux bidimensionnel de taille $card_L^2$ avec L l'ensemble des labels définis dans le diagramme de Hasse.

Il faux cependant noter qu'il n'y a pas deux mais trois possibilités de réponse lors de la comparaison de labels :

— Soit les deux labels sont comparables et la condition (par exemple inférieur ou égal) est vraie
— Soit les deux labels sont comparables et la condition est fausse
— Soit les deux labels sont incomparables

La méthode ***computeRelations*** permet de calculer les relations entre labels et de remplir le tableau correspondant. Cette table contient trois critères représentés par un triplet de booléens (A,(B,C)). Le premier booléen A indique si les labels sont comparables c'est-à-dire s'il existe un chemin dans le diagramme de Hasse entre les deux labels. Le couple (B,C) contient les critères d'infériorité (au sens large) et d'égalité entre labels. La méthode ***computeRelations*** fait appel à la méthode récursive ***existsPathForLessEqual*** de la classe HasseDiagram qui effectue une recherche de chemin d'un label A vers un label B. Cette méthode retourne un couple de booléens dont le premier contient le critère de comparabilité (comparable/non comparable) et le second le résultat de la condition d'infériorité au sens large :

— (vrai,vrai) si et seulement si il existe un chemin de A vers B (traduisant la relation A $<$ B).
— (vrai,faux) si et seulement si il existe un chemin de B vers A (traduisant la relation A $>$ B)
— (faux,N/A) si et seulement si il n'existe pas de chemin de A vers B ni de B vers A. Techniquement la valeur de second booléen n'importe pas.

La méthode ***assignPriorities*** permet d'attribuer à chaque label une priorité. Cette priorité est utilisée lors du calcul du graphe des coupes. La priorité dépend également du diagramme de Hasse. Elle est telle que chaque label a une priorité plus grande que ses parents directs et plus faible que ses enfants directs. Dans le cas de deux labels incomparables, les priorités n'ont pas d'influence (elles peuvent être les mêmes). De fait la méthode ***assignPriorities*** utilise une méthode récursive ***spreadPriority*** qui attribue à chaque fils du nœud couramment traité la priorité du nœud courant - 1 (l'élément le plus prioritaire est celui ayant la plus faible valeur). Ces priorités sont stockées dans un tableau unidimensionnel de taille $card_L$ aù L est l'ensemble des labels. Il n'y a pas besoin de clef dans la mesure où l'indice de la case du tableau est en réalité l'entier représentant le label lui étant associé.

En tant que classe fille de Ordering<T>, la classe ***LabelOrdering*** se doit de fournir une implémentation des méthodes de comparaisons entre labels ***islessequal*** et ***isequal*** ainsi que de la méthode d'obtention de priorité ***getPriority***. Ces méthodes sont définies ainsi :

— Étant donné que le tableau contient toutes les informations nécessaires, on obtient le résultat de ***islessequal*** en accédant à la case du tableau sur la i^e ligne et la j^e colonne avec i et j les deux entiers lus dans l'image d'entiers représentant les deux labels à travers le second appairage. La case de ce tableau contient trois booléens sous forme d'un triplet (A,(B,C)). A indique si les labels sont comparables, B indique si les labels sont inférieurs au sens large et C indique si les labels sont égaux. Afin d'obtenir le critère "inférieur au égal" on retourne la conjonction de A et B. La complexité de cette méthode revient à accéder à un case d'un tableau bidimensionnel et à effectuer une conjonction booléenne ce qui peut être considéré

de complexité $O(1)$.

— Similairement le tableau contient toutes les informations nécessaires à l'obtention du résultat de la méthode ***isequal***. Ledit résultat est obtenu par une conjonction booléenne entre le premier booléen A contenant le critère de comparabilité et le troisième booléen contenant le critère d'égalité. La complexité est identique à celle de la méthode ***islessequal*** soit $O(1)$.

— La méthode ***getpriority*** retourne simplement la valeur de priorité présente dans le tableau correspondant. Sa complexité est en $O(1)$.

6.2.5 Classe CGraph

La classe ***CGraph*** permet de produire le graphe des coupes d'une image. Pour être plus précis, il produit la variante Ö dudit graphe. Cette classe était déjà présente dans le projet au commencement de ce TER. De fait très peu d'adaptations ont été nécessaires afin de la rendre fonctionnelle dans le cadre présent.

La classe CGraph produit le graphe des coupes à partir des objets suivants :

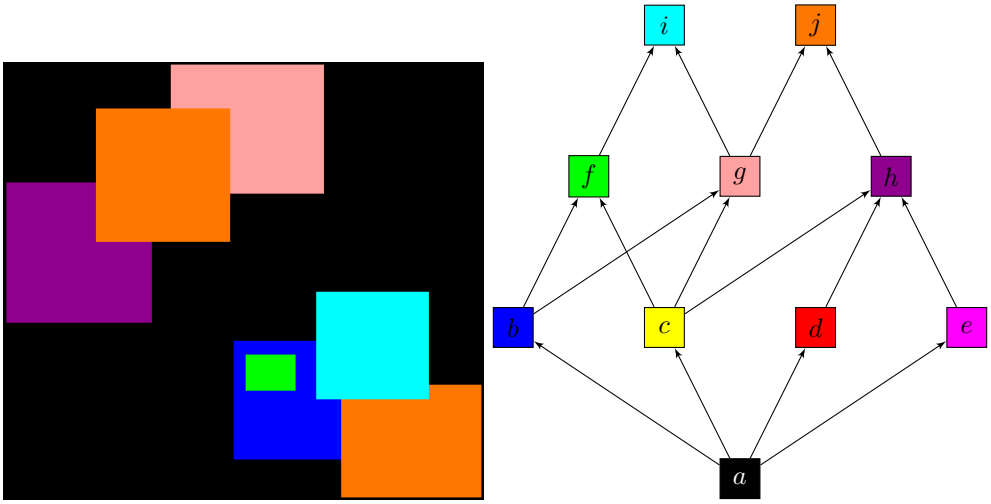
— Une image I sur dont les pixels sont de type T (Image<T>)
— Un graphe de régions d'adjacence ou RAG calculé sur l'image I c'est-à-dire contenant des nœuds de valeur T (Rag<T>)
— Une relation d'ordre (partielle) sur le type de pixel T de l'image I (Ordering<T>)
La classe ***CGraph*** produit alors un graphe des coupes pour le type de pixel T (CGraph<T>).

Dans la mesure où un second niveau d'abstraction entre labels abstraits et identifiants (entiers) a été créé afin de rendre les calculs plus efficaces et que par conséquent l'image est une image d'entiers (Image<int>), le graphe de régions d'adjacence est également définit sur des entiers (Rag<int>). L'ordre partiel sur les labels a été définit sur les identifiants représentant les labels plutôt que les labels eux-mêmes définissant par cela un ordre sur les identifiants (Ordering<int>). De par la concordance des types, le graphe des coupes résultant est définit sur les mêmes identifiants de labels (CGraph<int>). Une seule adaptation a été nécessaire : spécialiser la méthode ***writeDot*** de la classe CGraph avec le paramètre int.

Une amélioration possible serait de convertir le graphe des coupes définit sur les identifiants de labels en graphe de coupes définit sur les labels eux-mêmes. Cela est purement cosmétique et permettrait de lire l'information du graphe directement sans avoir à effectuer la conversion explicitement.

7 Résultats

Les tests ont été effectués sur l'image (a) qui possède des propriétés particulières permettant de tester en profondeur les résultats obtenus par le programme. Le diagramme de Hasse (b) ainsi que l'image (a) sont similaires à ceux utilisés dans [2], [3] et [4]. L'image ne comporte pas tous les labels définis dans le diagramme de Hasse (b). De plus on peut noter que le label J est représenté par deux régions distinctes de l'image sans que celles-ci ne soient adjacentes. On est donc capable de tester si le programme prend bien en compte les critères de comparabilité (ou d'incomparabilité) entre labels, applique correctement la relation d'ordre partiel définit par (b) et construit correctement la variante $\ddot{\mathfrak{G}}$ du graphe des coupes donnée dans [4].



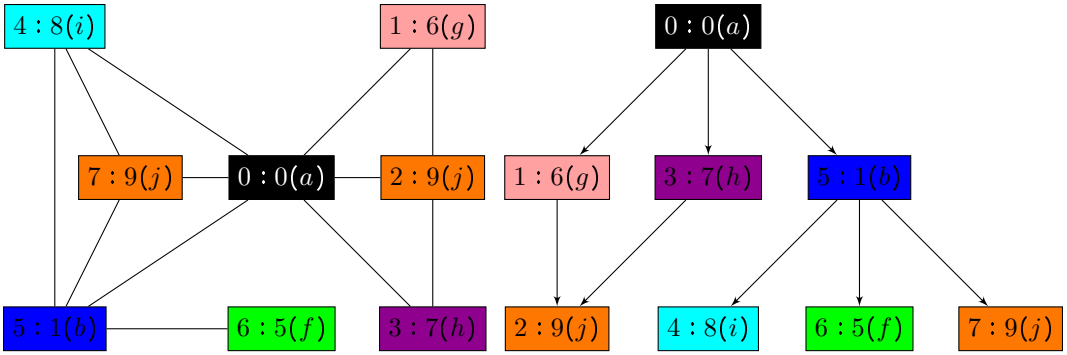
(a) I (b) $\mathfrak{T}$

Label	Rouge	Vert	Bleu	ID	Label	Rouge	Vert	Bleu	ID
A	0	0	0	0	F	0	255	0	5
B	0	0	255	1	G	255	161	161	6
C	255	255	0	2	H	143	0	143	7
D	255	0	0	3	I	0	255	255	8
E	255	0	255	4	J	254	119	0	9

(c) Appairage RVB $\leftrightarrow$ Labels $\leftrightarrow$ identifiants

FIGURE 3 – (a) Une image $I : \Omega \to V$ avec $V = [\![0, 255]\!]^3$. (b) Un diagramme de Hasse $\mathfrak{T}$ offrant une relation d'ordre partielle sur V. (c) Un appairage $[\![0, 255]\!]^3 \leftrightarrow \{a, ..., j\}$ sur lequel on a rajouté un second appairage $\{a, ..., j\} \leftrightarrow [\![0, 9]\!]$.

On constate dans le RAG que les deux composantes de label **j** sont distinctes car non adjacentes l'une de l'autre. On remarquera aussi que la partie droite du RAG correspond aux trois rectangles disposés dans le coin supérieur gauche de l'image (labels g, h et j) et que la partie gauche correspond aux quatre rectangles disposés dans le coin inférieur droit de l'image (labels b, g, i et j). On montre



(a) Graphe de régions d'adjacence ou RAG

(b) Graphe des coupes $\ddot{\mathfrak{G}}$

FIGURE 4 – (a) Graphe de régions d'adjacence ou RAG avec les lettres représentant les labels définis dans le diagramme de Hasse $\mathfrak{T}$. (b) Variante $\ddot{\mathfrak{G}}$ du graphe des coupes de l'image I obtenue par le programme. Pour (a) et (b), le premier nombre est le numéro du nœud dans le graphe de régions d'adjacence (respectivement dans le graphe des coupes), le second est l'identifiant du label contenu dans le nœud et la chaîne entre parenthèses est le nom du label correspondant à l'identifiant.

que le graphe des coupes obtenu est identique à celui décrit dans [2], [3] et [4] aux noms de nœuds près. La mise en couleur des nœuds du graphe des coupes ainsi que l'affichage des labels qui leur sont associés permet de mettre en lumière l'organisation des couleurs (ainsi que des labels associés) suivant l'ordre définit pas le diagramme de Hasse $\mathfrak{T}$ au sein de l'image I. Les cas pathologiques sont correctement traités notamment le fait que les labels **i** et **j** ne sont pas comparables ce qui mène à deux branches distinctes (nœuds 4 et 7) au niveau du nœud **b** (étiqueté 5). On retrouve un configuration similaire de l'autre côté de graphe où les labels **g** et **h** (nœuds 1 et 3) ne sont pas comparables et donc mènent à deux branches distinctes au niveau du nœud **a** (étiqueté 0). Ce graphe des coupes est correct pour ce diagramme de Hasse particulier et cette image particulière.

8 Conclusion

Auparavant, le programme était capable seulement de construire la variante $\ddot{\mathfrak{G}}$ du graphe des coupes d'une image RVB. Celle-ci ne comportait pas de labels arbitraires et utilisait l'ordre marginal sur RVB en tant que relation d'ordre. Il s'agissait donc d'un ordre total, non paramétrable. Le développement d'une classe ***Label*** a permis de représenter les labels arbitraires qui désormais feront office de type de base sur lequel la relation d'ordre sera définie. La classe Label est bien entendu compatible avec les identifiants de nœuds de graphe du langage DOT. Le développement de la classe ***HasseDiagram*** ajoute une grande flexibilité au programme. En effet, celle-ci permettant de charger un graphe écrit dans le langage de spécification DOT, offre désormais la possibilité de paramétrer la relation d'ordre qui sera utilisée. Il suffit simplement de définir un diagramme de Hasse suivant les règles du langage DOT. Il est de fait possible de définir des ordres totaux (comme par exemple l'ordre marginal anciennement utilisé) comme des ordres partiels ce qui n'était pas possible de manière simple. Avec le développement de la classe ***Index*** ainsi que la mise en place d'un format de fichier simple d'utilisation, la correspondance entre le monde "concret" des valeurs de l'image et le monde "abstrait" des labels du diagramme de Hasse est assurée. Enfin, le développement de la classe ***LabelOrdering***, véritable plaque tournante du programme, fournit désormais le moyen de comparer les labels arbitraires définis dans le diagramme de Hasse au moment de générer le graphe des coupes. L'astuce consistant à créer un second niveau d'abstraction par dessus les labels assure que les comparaisons entre labels (en pratique les identifiants de labels) ont une complexité constante en $O(1)$ rendant la génération du graphe des coupes rapide du point de vue temporel et efficace du point de vue de la mémoire.

L'objectif principal, à savoir la génération de la variante $\ddot{\mathfrak{G}}$ du graphe des coupes d'une image de labels paramétrés munie d'une relation d'ordre partielle, a été rempli. Cependant, la fonction de chargement du diagramme de Hasse au format DOT n'est pour l'instant pas satisfaisante. Faute de temps et d'expérience avec les outils utilisés, un solution temporaire a été implémentée. Celle-ci n'est en aucun cas suffisamment robuste et complète pour permettre de charger n'importe quel graphe décrit dans le langage DOT. Elle est suffisante pour charger le diagramme de Hasse qui a été utilisé pour les tests mais ne saurait être une solution pérenne.

Un certain nombre d'améliorations sont envisageables afin de rendre le programme plus flexible. En particulier, la classe ***Index*** ainsi que le format de fichier décrivant l'appairage entre le monde concret des valeurs de l'image et le monde abstrait des labels du diagramme de Hasse permettent de ne prendre en compte que le format de représentation de couleurs RVB. Rendre cette classe générique en la paramétrant par le type de valeurs de l'image T rendrait le programme apte a utiliser d'autres formats de représentation des couleurs. Il serait également intéressant d'implémenter une fonction de conversion du graphe des coupes obtenu qui est défini sur les identifiants de labels (des entiers donc) afin de faire apparaître lesdits labels plutôt que les identifiants. Cela faciliterait la lecture et l'analyse du graphe résultant, ne nécessitant plus que les personnes analysant le graphe ne soient forcées de faire explicitement la conversion.

9 Bibliographie

[1] E. Carlinet, T Géraud "A fair comparison of many max-tree computation algorithms", Computing Research Repository, http://arxiv.org/abs/1212.1819

[2] N. Passat, B. Naegel "Component-trees and multivalued images : Structural properties", Journal of Mathematical Imaging and Vision, Vol. 49 Num 1, p. 37-50, 2014

[3] B. Naegel, N. Passat "Towards connected filtering based on component-graphs", ISMM, ser. Lect Notes Comput Sc, vol. 7883, 2013, pp. 350-361

[4] B. Naegel, N. Passat "Colour image filtering with component-graphs", International Conference on Pattern Recognition (ICPR), p. 1621-1626, IEEE, Stockholm, Sweden, 2014

[5] https://www.graphviz.org/doc/info/lang.html

[6] https://github.com/bnaegel/component-graph

[7] http://www.cplusplus.com/reference/map/map/operator[]/

[8] https://www.graphviz.org/pdf/libguide.pdf

[9] https://pypi.org/project/pydot/

[10] https://docs.python.org/3/c-api/index.html

[11] https://www.boost.org

[12] https://www.boost.org/doc/libs/1_67_0/libs/graph/doc/read_graphviz.html

[13] https://www.boost.org/doc/libs/1_67_0/libs/spirit/doc/html/index.html

[14] https://www.boost.org/doc/libs/1_67_0/libs/spirit/doc/html/spirit/qi.html

[15] ftp://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html

[16] https://www.gnu.org/software/bison/manual/