Définition et preuve d'un algorithme fonctionnel de segmentation d'image basé sur les hypercartes

Jean-François Dufourd[1]

1: LSIIT : Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection, UMR CNRS-ULP 7005, Pôle Technologique, Boulevard S. Brant, BP10413, F67412, Illkirch
dufourd@lsiit.u-strasbg.fr

Résumé

Nous présentons la conception d'un nouvel algorithme fonctionnel de segmentation d'image 2D par fusion de cellules d'une subdivision, la preuve formelle de sa correction totale et la dérivation d'un programme impératif optimal. Les subdivisions de plans colorées sont modélisées par des hypercartes. Les spécifications formelles sont exprimées en Calcul des constructions inductives. Les preuves sont assistées par le système Coq. Le coeur du travail est une induction simple sur des termes décrivant les hypercartes. Le programme final est écrit en C.

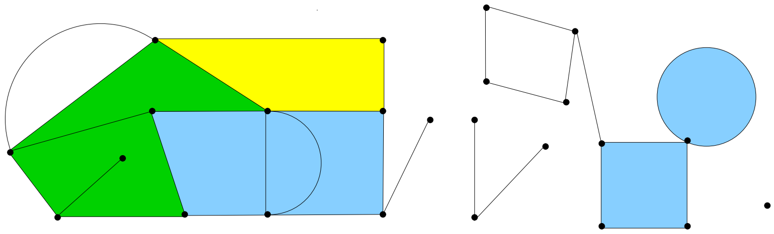
1. Introduction

Nous présentons la conception d'un nouvel algorithme fonctionnel de segmentation d'image 2D, la preuve assistée par ordinateur de sa correction totale, et la dérivation d'un programme optimal écrit en langage impératif. Problème classique en imagerie, la segmentation est résolue ici de manière générale pour toute subdivision colorée finie du plan. Une telle subdivision est un complexe de cellules de dimensions 0, 1 et 2 − sommets, arêtes et faces colorées −, munies de relations d'incidence et d'adjacence. Dans notre problème, les formes géométriques et la localisation des cellules dans le plan sont sans importance : la segmentation est alors un pur problème de topologie combinatoire. Pour décrire la topologie des subdivisions, nous avons retenu ici le modèle générique des hypercartes.

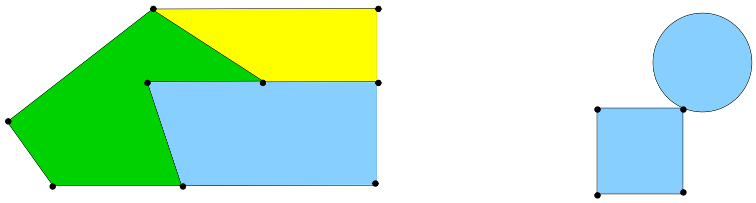
Nous montrons comment les hypercartes et les preuves de leurs propriétés, notamment celles de la segmentation, sont entièrement formalisées de manière constructive. Etant donné le volume des développements, il est hautement souhaitable, pour des raisons de confort et de sécurité, que ce processus soit assisté par ordinateur. Nous avons opté pour l'assistant de preuves Coq de l'INRIA dont nous faisons un usage intensif. En Sections 2, 3 et 4, nous faisons des rappels et évoquons les travaux connexes. En Sections 4, 5 et 6 nous présentons des spécifications formelles conduisant aux hypercartes. En Sections 7 et 8, nous spécifions de manière fonctionnelle la segmentation et prouvons sa correction. En Section 9, nous dérivons un programme impératif. En Section 10, nous discutons nos choix et nous concluons en Section 11.

2. Segmentation et partitions

Soit π un plan euclidien supportant une image continue. L'ensemble des *partitions* de π est ordonné comme suit : une partition B est *plus grossière* qu'une partition A si chaque classe de B contient une classe de A. En supposant que toutes les partitions considérées satisfont un *critère* donné, la segmentation consiste, à partir d'une partition initiale A, à trouver une partition plus grossière que



a. Une subdivision planaire colorée.



b. Segmentation de la subdivision en (a).

FIG. 1 – Subdivision planaire finie et sa segmentation.

A et *maximale* pour cet ordre [Ser06]. Selon le critère, avec des images complexes ou bruitées, et des embarras dus au seuillage, cela peut être extrêmement difficile [HS85]. Mais nous considérons ici un problème de segmentation très simple opérant sur des partitions du plan en subdivisions finies.

Définition 1 *(*Subdivision finie du plan*)*
Une subdivision finie *de π est un triplet $S = (V, E, F)$ composé d'un ensemble fini V de points appelés* sommets*, d'un ensemble fini E d'arcs de Jordan ouverts appelés* arêtes*, et d'un ensemble fini F de régions ouvertes connexes appelées* faces*, tels que : (i) sommets, arêtes et faces de S, appelés aussi ses k-*cellules *pour la dimension $k = 0, 1$ et 2, forment une partition de π ; (ii) chaque arête de S est bordée par 1 ou 2 sommets de V.*

Un exemple est en Fig. 1(a), où l'on remarque des arêtes *pendantes* et un sommet *isolé.* Pour avoir une partition de π, il faut considérer qu'existe exactement une face non bornée. Le critère retenu est la *préservation* de la couleur en chaque point, la *coloration homogène* de l'intérieur de chaque cellule, et la *conservation des sommets et arêtes frontières* entre faces de couleurs différentes. Dans les faces, chaque couleur (ou niveau de gris) est codée par un entier. Par convention, la couleur de chaque sommet ou arête est un rationnel défini comme la moyenne des couleurs de ses faces adjacentes. Alors, nous disons que la subdivision est *bien colorée.* La Fig. 1(b) donne la segmentation, unique pour le critère retenu, de la subdivision initiale en Fig. 1(a).

3. Travaux connexes

En comparaison de la grande sophistication de certains problèmes de segmentation [Ser06, HS85], le nôtre est particulièrement simple. Pour être bien formalisé, il nécessite juste un modèle mathématique convenable pour les subdivisions. Comme l'atteste leur utilisation en modélisation géométrique [BD94] et en imagerie, notamment pour la segmentation [BD99, BFP99], les modèles de cartes combinatoires sont bien indiqués. Nous avons retenu le plus général d'entre eux, les hypercartes [Cor70]. Souvent, les algorithmes de segmentation basés sur les modèles de cartes partent d'une grille de pixels ou voxels. Ils reposent sur un *ordre* de parcours spécifique des cellules et testent des configurations locales de pixels

brin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
α0	2	3	1	7	9	5	8	4	10	6	14	11	12	13	17	16
α1	11	2	4	5	6	3	1	12	8	17	7	13	20	16	14	15

brin	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
α0	20	19	18	15	23	24	21	25	22	29	28	27	26	31	30	32
α1	10	18	19	9	22	21	31	23	28	27	26	29	25	24	30	32

FIG. 2 – Permutations α_0, α_1 de l'hypercarte de la Fig. 3.

ou voxels, appelées *précodes* [Dam01]. Nous verrons que notre algorithme travaille sur des subdivisions générales, n'impose pas d'ordre de parcours, et fait des tests en nombre limité. Cependant, il réalise seulement une segmentation de base, et, pour l'instant, juste en 2D. Une version préliminaire en Ocaml, sans preuve et fortement améliorée ici, est dans [DP00].

Notre cadre formel est le *Calcul des constructions inductives* [CH85, Pau93]. Il est implanté dans l'assistant de preuves interactif Coq qui est notre support [Coq04, BC04]. On trouve quelques expériences de preuves assistées par Coq en algorithmique et en modélisation géométriques [PB01, DD04]. Des preuves du théorème du genre et de la formule d'Euler pour les polyèdres avec des cartes et Coq sont présentées dans [Duf07]. Une spécification en B événementiel de l'appariement de droites avec des cartes généralisées est dans [MD05], avec un raffinement de structures de données et des obligations de preuves partiellement automatisées. Des hypercartes sont axiomatisées dans [Gon05] pour prouver avec Coq l'impressionnant théorème des quatre couleurs. Les spécifications qui vont suivre sont différentes et semblent plus simples. Enfin, à notre connaissance, aucune expérience n'a jamais été menée en preuve assistée par ordinateur pour des algorithmes d'imagerie.

4. Aspects mathématiques

La topologie d'une subdivision planaire finie peut être modélisée par une hypercarte, qui permet de définir, orienter et parcourir aisément toutes les cellules.

Définition 2 *(*Hypercarte*)*
(i) Une hypercarte *est une structure algébrique $M = (D, \alpha_0, \alpha_1)$, où D est un ensemble fini dont les éléments sont appelés des* brins*, et α_0, α_1 sont des permutations dans D.*
*(ii) Si $y = \alpha_k(x)$, y est le k-*successeur *de x, x est le k-*prédécesseur *de y, et x et y sont dits k-*liés.

La Fig. 2 tabule des *permutations* α_0 et α_1 dans $D = \{1, \ldots, 32\}$. Alors, $M = (D, \alpha_0, \alpha_1)$ est une hypercarte, tracée sur le plan à la Fig. 3 en associant tout brin à un arc de courbe orienté d'une *pastille* à une *barre* : les brins 0-liés (*resp.* 1-liés) partagent la même barre (*resp.* pastille). M modélise la subdivision en Fig. 1(a), dont les arêtes sont devenues des brins et les sommets des pastilles ou des barres, autour desquelles les k-successeurs tournent dans le sens rétrograde. Un ajout de sommets ou d'arêtes est parfois nécessaire pour la consistance de l'hypercarte. Les cellules topologiques d'une hypercarte, qui modélisent celles de la subdivision sous-jacente, peuvent être facilement définies.

Définition 3 *(*Orbites et cellules d'hypercarte*)*
*(i) Soient D un ensemble et $f_1, \ldots, f_n$ des fonctions dans D. L'*orbite *de $x \in D$ pour ces fonctions est le sous-ensemble de D noté $< f_1, \ldots, f_n > (x)$ des éléments accessibles depuis x par n'importe quelle composition des fonctions.*
*(ii) Dans l'hypercarte $M = (D, \alpha_0, \alpha_1)$, $< \alpha_0 > (x)$ est la 0-orbite ou l'*arête *du brin x, $< \alpha_1 > (x)$ sa 1-orbite ou son* sommet, *$< \alpha_1^{-1} \circ \alpha_0^{-1} > (x)$ sa* face, *et $< \alpha_0, \alpha_1 > (x)$ sa* composante connexe. *Sommets, arêtes, et faces sont aussi appelés k-*cellules topologiques, *pour $k = 0, 1$ et 2.*

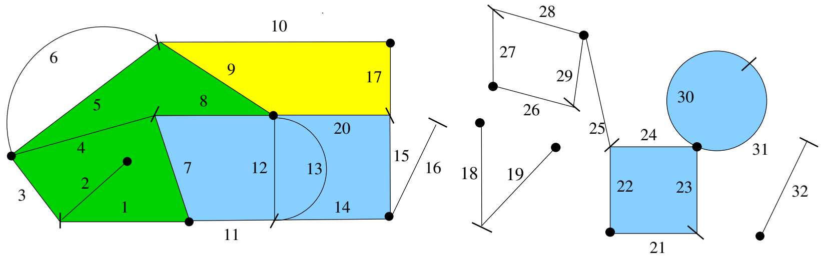


FIG. 3 – Un exemple d'hypercarte.

L'hypercarte en Fig. 3 a 13 arêtes (barres), 13 sommets (pastilles), 14 faces et 4 composantes connexes. La représentation d'une hypercarte dans le plan euclidien est son *plongement.* Il correspond à une projection des cellules topologiques sur des cellules géométriques : sommets et arêtes sur des points, brins sur des arcs de Jordan ouverts (et vides pour les points isolés), et faces sur des régions ouvertes connexes. Dans le plongement, le parcours de toute k-cellule par la fonction qui la caractérise se fait dans le sens rétrograde. En outre, pour respecter les définitions, le plongement de chaque composante connexe doit être une subdivision finie planaire, avec exactement une face non bornée. Ainsi, le plongement de l'hypercarte en Fig. 3 possède 10 faces bornées, comme $\{5, 9\}$ ou $\{26, 28\}$, et 4 faces non bornées, comme $\{6, 17, 16, 14, 1\}$ ou $\{32\}$. Notons qu'on peut modéliser la même subdivision avec différentes hypercartes. Par exemple, nous pourrions représenter toute arête de la subdivision originale par une arête de l'hypercarte composée de deux brins liés par α_0, ce qui serait sans doute plus intuitif. L'hypercarte serait alors une *carte combinatoire orientée* [BD99, DP00], où nos résultats resteraient valides. Mais elle comporterait approximativement le double de brins.

5. Spécifications de base

En Gallina, langage de spécification de Coq, nous déclarons le type inductif `dim` des dimensions, avec les *constructeurs* `zero` et `one` codant les 2 dimensions :

```
Inductive dim:Set:= zero: dim | one: dim.
```

Le type `dim` a lui-même le type `Set` "de tous les types". On doit d'abord prouver la *décidabilité* de l'égalité = dans `dim`. En effet, la logique de Coq étant intuitionniste, le *tiers-exclu* n'y est pas valide et la décidabilité des prédicats doit y être prouvée. Le prédicat d'égalité = est prédéfini pour chaque type, mais pas sa décidabilité. Pour `dim`, elle peut être établie comme un lemme :

```
Lemma eq_dim_dec:forall i j:dim,{i=j}+{~i=j}.
```

En Gallina, la décidabilité de `i=j` est par convention écrite comme la somme `{i=j}+{~i=j}`. Une fois établie, la preuve est une fonction, appelée ici `eq_dim_dec`, avec 2 arguments, `i j :dim`, et un résultat du type somme ci-dessus, dont les objets peuvent être testés dans une expression conditionnelle `if...then...else...`. Ce lemme est prouvé interactivement avec l'aide de tactiques qui ne sont pas données ici. Le raisonnement est une induction structurelle sur `i` et `j`, ici un simple raisonnement par cas. En effet, à partir de toute définition inductive de type, Coq engendre un *principe d'induction*, utilisable soit pour prouver des propositions soit pour construire des fonctions totales sur le type. Les *couleurs* de faces sont considérées ici comme des entiers naturels sans nouveau type spécifique, et les couleurs rationnelles des sommets et arêtes n'interviennent jamais.

Le type `dart` des brins est `nat`, type prédéfini des entiers naturels. La décidabilité `eq_dart_dec` de l'égalité des brins est un renommage de `eq_nat_dec`, décidabilité de l'égalité dans `nat`. Un brin `nil` pour gérer les exceptions est un renommage de `0` :

```
Definition dart:= nat.
Definition eq_dart_dec:= eq_nat_dec.
Definition nil:= 0.
```

Dans un premier temps, les hypercartes sont approchées par un type `fmap` plus général de *cartes libres* (ou *free maps*) :

```
Inductive fmap:Set:=
      V : fmap
    | I : fmap->dart->nat->fmap
    | L : fmap->dim->dart->dart->fmap.
```

A nouveau, c'est un type inductif avec 3 constructeurs `V`, `I` et `L`, respectivement pour la carte vide, l'insertion d'un brin dans une carte (avec la couleur de sa face), et la liaison de deux brins à l'intérieur d'une carte. En Coq, `fmap` est le plus petit ensemble de termes clos construits avec `V`, `I` et `L`, considérés comme des injections indépendantes. A nouveau, à partir de ces constructeurs, Coq engendre un principe d'induction pour conduire des preuves ou construire des fonctions sur `fmap`. Une partie de l'hypercarte en Fig. 3 est décrite par un terme où sont composés les constructeurs. Elle est représentée en Fig. 4, où les liaisons effectives par `L` sont matérialisées par des arcs de cercle autour des barres et des pastilles. Par convention, chaque couleur de brin (orienté) est portée à sa gauche. Comme c'est la couleur de la face du brin, ceci engendre entre tous les brins d'une face un problème de consistance dont il faudra se préoccuper.

Des *observateurs* de cartes libres sont à présent définis. Le prédicat `exd` teste si un brin existe dans une carte libre. Le mot-clé `Fixpoint` indique une définition récursive. Un filtrage sur `m` s'écrit `match m with...`. L'annotation `{struct m}` indique à Coq que les appels récursifs se font sur des termes plus petits que `m`, ce qui certifie la terminaison. Le résultat est `False` ou `True`, les constantes de `Prop`, type prédéfini des propositions. Les termes sont en notation préfixe, et `_` symbolise la place d'un argument inutilisé. La décidabilité `exd_dec` de `exd` en dérive directement :

```
Fixpoint exd(m:fmap)(z:dart){struct m}:Prop:=
 match m with
       V => False
    |  I m0 x _ => z=x \/ exd m0 z
    |  L m0 _ _ _  => exd m0 z
 end.
```

Une version `A` de l'opération α_k, complétée par `nil`, s'écrit, avec des fonctions de décidabilité :

```
Fixpoint A(m:fmap)(k:dim)(z:dart){struct m}:dart:=
 match m with
      V => nil
    | I m0 x _ => A m0 k z
    | L m0 k0 x y =>
         if (eq_dim_dec k k0)
         then if (eq_dart_dec z x) then y else A m0 k z
         else A m0 k z
 end.
```

La définition de l'inverse de `A`, notée `A_1`, est similaire. De même, une function `col` retourne la couleur de chaque brin d'une carte libre. Les prédicats auxiliaires `succ` et `pred` testent si un brin a un `k`-successeur et un `k`-prédécesseur (non `nil`), avec les fonctions de décidabilité `succ_dec` et `pred_dec`.

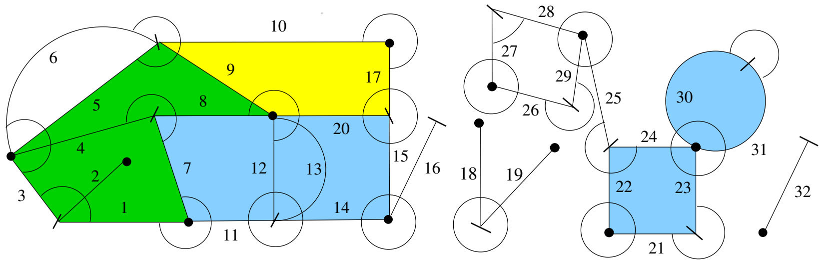


FIG. 4 – Quasi-hypercarte pour l'hypercarte en exemple.

Enfin, des *destructeurs* sont définis récursivement. Premièrement, `D` efface la dernière insertion d'un brin par `I`. Deuxièmement, `B` et `B_1` rompent le dernier `k`-lien inséré par `L` pour un brin, vers l'avant et vers l'arrière. Cependant, pour qu'on ait des hypercartes consistantes, les opérations doivent être contraintes par des préconditions.

6. Quasi-hypercartes et hypercartes

Chaque précondition se présente comme un prédicat sur les paramètres de l'opérateur contraint. Ainsi, nous avons pour `I` et `L` :

```
Definition prec_I(m:fmap)(x:dart):= x <> nil /\ ~ exd m x.
Definition prec_L(m:fmap)(k:dim)(x y:dart):=
    exd m x /\ exd m y /\ ~ succ m k x /\ ~ pred m k y.
```

Si `I` et `L` sont toujours utilisés dans ces conditions, alors l'objet construit est facile à interpréter comme une hypercarte avec certaines k-orbites ouvertes : nous disons que c'est une *quasi-hypercarte*. Le type `qhmap` correspondant satisfait l'invariant `inv_qhmap` :

```
Fixpoint inv_qhmap(m:fmap){struct m}:Prop:=
  match m with
      V => True
    | I m0 x _ => inv_qhmap m0 /\ prec_I m0 x
    | L m0 k x y => inv_qhmap m0 /\ prec_L m0 k x y
  end.
```

La Fig. 4 contient en fait une quasi-hypercarte qui modélise la subdivision en Fig. 1(a). On démontre que, pour toute quasi-hypercarte `m` et dimension `k`, `(A m k)` et `(A_1 m k)` sont des *injections* inverses l'une de l'autre. Alors, une *hypercarte* est une quasi-hypercarte *complète*, *i.e.* munie de k-orbites closes. Ceci est exprimé par l'invariant `inv_hmap` du type `hmap` des hypercartes :

```
Definition inv_hmap(m:fmap):Prop:= inv_qhmap m /\
  forall(x:dart)(k:dim), exd m x -> succ m k x /\ pred m k x.
```

L'hypercarte qui clôt la quasi-hypercarte en Fig. 4 est en Fig. 5. Alors, nous pouvons prouver que, pour toute hypercarte `m` et toute dimension `k`, `(A m k)` et `(A_1 m k)` sont des *permutations*. Enfin, savoir s'il est possible d'aller d'un brin à un autre dans une face est crucial pour la suite. C'est le rôle du prédicat `expf : fmap -> dart -> dart -> Prop`, défini par induction sur `fmap`, la *décidabilité* de `expf m x y` étant exprimée par `expf_dec m x y`.

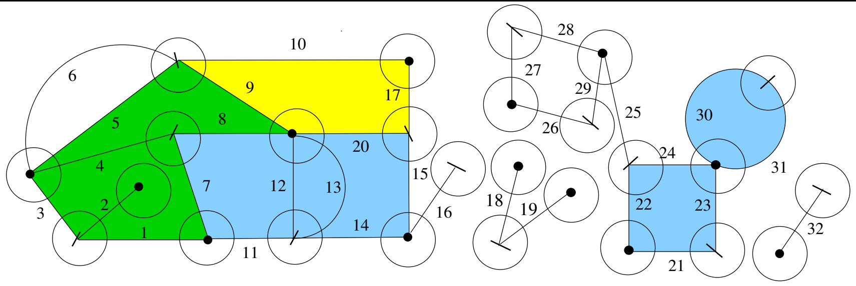
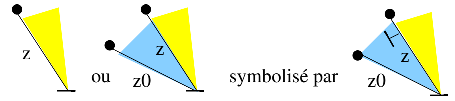
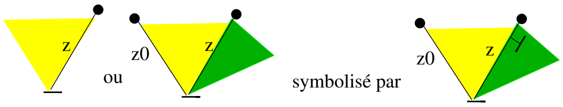


FIG. 5 – Hypercarte pour la subdivision en exemple.



a. up_border0 m z (col m z0)



b. low_border0 m z (col m z0).

FIG. 6 – Bords supérieur et inférieur à la dimension 0.

7. Un algorithme fonctionnel de segmentation

La segmentation est conçue pour des quasi-hypercartes colorées, mais définie inductivement sur `fmap` en deux étapes appelées `seg1` et `seg2`. La première isole les brins qui ne sont pas des frontières, et la deuxième les supprime.

7.1. Première étape de segmentation : seg1

D'abord, il faut définir la notion de *frontière* dans une quasi-hypercarte colorée `m`. Pour `zero` et `one`, nous distinguons des frontières *supérieures (upper-borders)* et *inférieures (lower-borders)* : `(up_border0 m z a)` exprime que, dans `m`, le brin `z` n'a pas de 0-successeur ou a un 0-successeur de couleur différente de `a`. Les prédicats `low_border0`, `up_border1` et `low_border1` sont similaires :

```
Definition up_border0(m:fmap)(z:dart)(a:tag):Prop:=
    ~succ m zero z \/ col m (A m zero z) <> a.
```

Pour `a = col m z0` et `z0 = A m zero z`, la notion de frontière à la dimension 0 est illustrée en Fig. 6, où un T symbolise l'absence possible de 0-successeur ou de 0-prédécesseur. Ces prédicats sont immédiatement prouvés décidables, avec `up_border0_dec`, `low_border0_dec`, `up_border1_dec`, `low_border1_dec`. Une définition fonctionnelle récursive de `seg1 m`, qui conserve les frontières et isole les autres brins, s'ensuit :

```
Fixpoint seg1(m:fmap):fmap:=
  match m with
      V => V                               // 1
    | I m0 x c => I (seg1 m0) x c          // 2
    | L m0 zero x y  =>                    // 3
      let m1 := seg1 m0 in
      let a := (col m0 y) in
      if eq_dart_dec x y then m1           // 3.1
      else
        let x_0:= A_1 m1 zero x in
        let y0 := A m1 zero y in
        if up_border0_dec m1 y a
        then
         if low_border0_dec m1 x a         // 3.2
         then L m1 zero x y
         else L (B_1 m1 zero x) zero x_0 y  // 3.3
     else
      if low_border0_dec m1 x a            // 3.4
      then L (B m1 zero y) zero x y0
      else                                 // 3.5
       if eq_dart_dec x_0 y                // 3.5.1
       then B m1 zero y
       else                                // 3.5.2
    L (B (B_1 m1 zero x) zero y) zero x_0 y0
 | L m0 one x y  =>                        // 4
    (* ... analogue ... *)
end.
```

Les cas de filtrage 1 :(`m = V`) et 2 :(`m = I m0 x c`) sont triviaux. Pour 3 :(`m = L m0 zero x y`), cinq sous-cas sont à considérer, de 3.1 à 3.5, illustrés en Fig. 7. Le cas 4 :(`m = L m0 one x y`) est analogue. La raison profonde des `k`-ruptures et `k`-liaisons ci-dessus est que l'*invariant* suivant doit toujours être maintenu : dans une quasi-hypercarte segmentée, il n'y a jamais deux faces de même couleur adjacentes par un sommet ou une arête. Cette propriété sera prouvée par la suite. Pour la quasi-hypercarte `m` en Fig. 4, `(seg1 m)` est en Fig. 8(a). Ce résultat n'est pas complètement satisfaisant car il contient des brins pendants non isolés. Avec l'application de `seg1` à la Fig. 5, qui contient une vraie hypercarte, ce défaut disparaît, comme on le voit en Fig. 9(a).

7.2. Deuxième étape de segmentation : seg2

La seconde étape de segmentation `seg2` supprime tous les brins sans `k`-lien, pour `k = 0,1`, *i.e. isolés* selon le prédicat `isolated` :

```
Definition isolated(m:fmap)(z:dart):Prop:=
  ~succ m zero z /\ ~pred m zero z /\ ~succ m one z /\ ~pred m one z.
```

La décidabilité `isolated_dec` étant aisément prouvée, une opération auxiliaire `seg2_aux` supprime les brins isolés d'une carte libre `m`, par filtrage sur `m`. Elle utilise une carte fixe appelée `mr`, toujours égale à la carte originale `m` et servant de *référence*. Alors, `seg2` est immédiatement définie, ainsi que `segmentation` composant `seg1` et `seg2` pour réaliser la segmentation complète :

```
Fixpoint seg2_aux(m mr:fmap){struct m}:fmap:=
  match m with
```

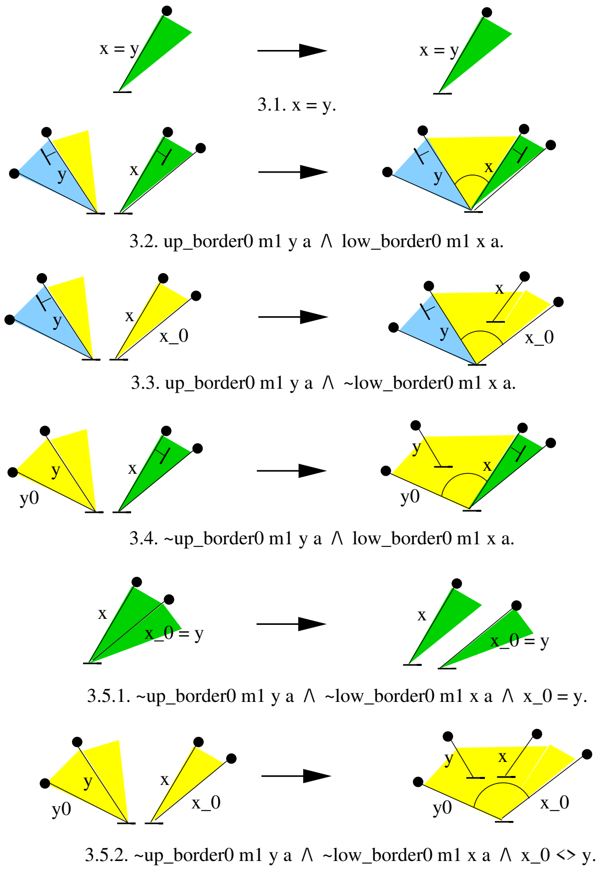
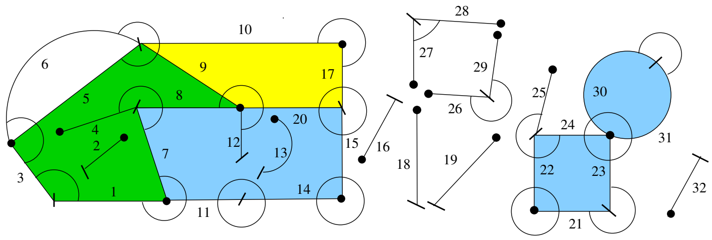
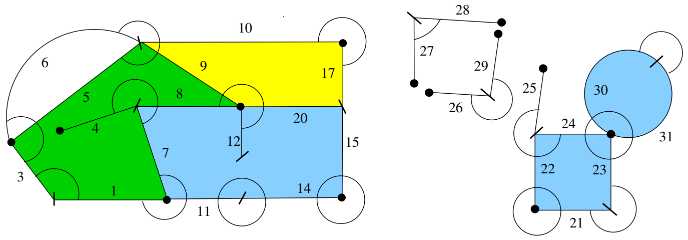


FIG. 7 – Cas pour la segmentation (étape 1 : seg1, 0-liaisons).



a. Segmentation de m, étape 1 : (seg1 m) .



b. Segmentation de m, étapes 1 et 2 : (segmentation m).

FIG. 8 – Segmentation de la quasi-hypercarte en Fig. 4.

```
      V => V
    | I m0 x c =>
       if isolated_dec mr x then seg2_aux m0 mr
       else I (seg2_aux m0 mr) x c
    | L m0 k x y =>  L (seg2_aux m0 mr) k x y
  end.
Definition seg2(m:fmap):= seg2_aux m m.
Definition segmentation(m:fmap):= seg2 (seg1 m).
```

Le résultat de `seg2` est d'abord illustré en Fig.8(b) sur la quasi-hypercarte de la Fig. 8(a). Nous gardons des défauts dus aux brins pendants. La même opération appliquée sur la Fig. 9(a), résultat de `seg1` sur la Fig. 5, conduit au bon résultat, en Fig. 9(b), correspondant exactement à la Fig. 1(b). Obtenir des résultats satisfaisants sur des jeux d'essai est indispensable mais insuffisant. Une *certification*, ou preuve de *correction totale*, est nécessaire pour des algorithmes aussi intrigants.

8. Certification de la segmentation

8.1. Correction de la première étape : seg1

• a. Coq assure que `seg1` est à *terminaison finie* parce que, durant le filtrage sur un terme `m :fmap`, les appels récursifs de `seg1` s'appliquent toujours sur des sous-termes stricts de `m`.
• b. Notre spécification est mise à l'épreuve sur quelques propriétés simples, comme la préservation par `seg1 m` de tous les brins de `m` avec leurs couleurs. Les preuves sont menées par induction structurelle sur `m`, comme d'ailleurs la plupart de celles qui suivent.
• c. La *préservation de type* est assurée :

```
Theorem inv_qhmap_seg1:forall(m:fmap),
  inv_qhmap m -> inv_qhmap (seg1 m).
```

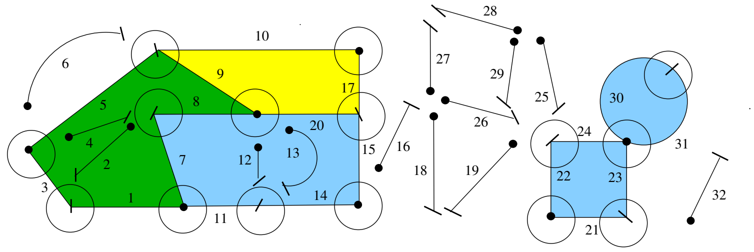
Techniquement, il faut en même temps prouver une autre propriété importante : les brins sans `k`-successeurs ou `k`-prédécesseurs restent dans le même état après `seg1`.
• d. L'*invariant* mentionné en liant dans `seg1` est satisfait, ce qui donne, à la dimension 0 :

```
Lemma succ0_pred0_seg1:forall(m:fmap)(z:dart),
 inv_qhmap m -> let m1 := seg1 m in
    succ m1 zero z -> pred m1 zero z -> col m (A m1 zero z) <> col m z.
```

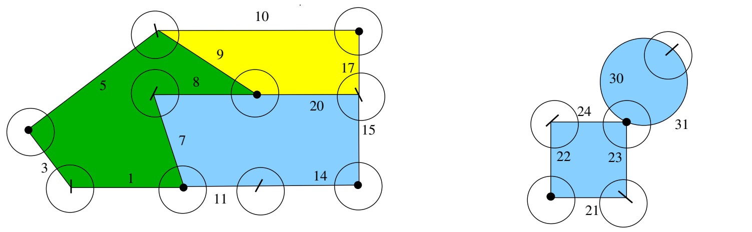
• e. Nous donnons une *caractérisation* des 0-liens résiduels dans `seg1 m`. les brins avec un 0-successeur dans `(seg1 m)` sont exactement ceux qui en avaient un dans `m` et qui étaient frontières inférieures à la dimension 0. Les résultats sont similaires pour les prédécesseurs avec `pred` et pour la dimension 1.

```
Theorem succ0_seg1_lb0:forall(m:fmap)(z:dart),
 inv_qhmap m -> let m1 := seg1 m in
  succ m1 zero z <-> (succ m zero z /\ low_border0 m z (col m (A m zero z))).
```

• f. Pour être plus précis, nous définissons une opération qui, pour une quasi-hypercarte `m` et un brin `z`, renvoie le premier brin de la `k`-orbite de `z` qui est frontière supérieure à la dimension `k` (éventuellement `nil`). Pour la dimension 0, cette fonction est nommée `uA0`. Ainsi, dans la quasi-hypercarte `m` en Fig. 4, nous avons `uA0 m 1 = 3`, `uA0 m 3 = 5`, `uA0 m 14 = 11`, `uA0 m 17 = 10`, `uA0 m 6 = nil`, `uA0 m 18 = nil`. Avec `uA0`, nous caractérisons complètement le résultat de `seg1` à la dimension 0 :



a. Segmentation de m, étape 1 : (seg1 m).



b. Segmentation de m, étapes 1 et 2 : (segmentation m).

FIG. 9 – Segmentation de l'hypercarte en Fig. 5.

```
Lemma not_succ_uA0_nil : forall(m:fmap)(z:dart),
  inv_qhmap m -> ~ succ m zero z -> uA0 m z = nil.
Theorem A_seg1_uA0:forall(m:fmap)(z:dart),
  inv_qhmap m -> let m1:=seg1 m in succ m1 zero z -> A m1 zero z = uA0 m z.
```

Le lemme affirme que les brins sans 0-successeur dans `m` ont une 0-frontière supérieure `nil`. Le théorème dit que le 0-successeur dans `seg1 m`, s'il existe, est fourni par `uA0` dans `m`. Avec les propriétés similaires pour la dimension 1, nous affirmons ici la *maximalité* de la première étape de segmentation.
• g. Il faut examiner la question de la couleur, qui laisse encore planer un doute. En fait, la segmentation concerne uniquement les quasi-hypercartes *bien colorées* :

```
Definition well_coloured(m:fmap):=
  forall(z t:dart), expf m z t -> col m z = col m t.
```

Nous prouvons alors le résultat fondamental affirmant la préservation de la bonne coloration par `seg1` :

```
Theorem correct2_seg1:
  forall m:fmap, inv_qhmap m -> well_coloured m -> well_coloured (seg1 m).
```

Ceci met un point final à la preuve de *correction totale* de `seg1`.

8.2. Correction de la deuxième étape : seg2

• a. La *terminaison* de `seg2_aux` est immédiate, à cause de l'induction structurelle.
• b. Nous prouvons de petits lemmes sur `seg2_aux`, tels que la préservation de l'existence des brins de `m` non isolés, et du résultat de l'opération `A` (et de celui de `A_1`).
• c. Nous prouvons que `segmentation` est bien typée : elle préserve les invariants des quasi-hypercartes et des hypercartes.
• d. Une hypercarte segmentée préserve les couleurs des brins. Alors, quand l'hypercarte `m` est bien colorée, de bonnes propriétés de `m1 :=segmentation m` émergent, comme le fait que tout brin de `m1`

est complètement lié.
• e. Ceci contribue à établir le théorème fondamental affirmant qu'une hypercarte bien colorée est segmentée en une hypercarte bien colorée :

```
Theorem well_coloured_segm:forall(m:fmap),
  inv_hmap m -> well_coloured m-> well_coloured (segmentation m).
```

• f. Enfin, nous prouvons aisément que, dans une hypercarte, `up_border` est équivalent à `low_border`. Ainsi, à la dimension 0 :

```
Theorem up_border0_low_border0:forall(m:fmap),
  inv_hmap m -> exd m z
    -> let a:= col m (A m zero z) in (up_border0 m z a  <-> low_border0 m z a).
```

Sous les mêmes conditions, avec une hypercarte bien colorée, nous avons immédiatement une équivalence entre les frontières aux dimensions 0 and 1, par exemple :

```
Theorem up_border0_low_border0:forall(m:fmap),
  inv_hmap m -> well_coloured m -> exd m z
   -> let a:= col m (A m zero z) in (up_border0 m z a  <-> low_border1 m z a).
```

Alors, les brins de `(segmentation m)` sont ceux qui étaient `low_border0` (ou `up_border0`), ou `low_border1` (ou `up_border1`) dans `m`. Ceci achève la preuve de *correction totale* de l'opération `segmentation`.

9. Implantation réelle et programme impératif

9.1. Implantation chaînée des hypercartes en C

Il est clair qu'une *extraction* par Coq d'un programme fonctionnel – inefficace – est immédiate en Ocaml à partir de notre spécification. Même si l'on peut espérer bientôt une extraction vers C, un programme performant nécessite de programmer avec *mutations* sur une structure de données d'hypercarte bien adaptée, dont la découverte automatique semble encore improbable.

Il n'est pas nécessaire d'implanter les cartes libres, mais seulement les quasi-hypercartes (et les hypercartes). De manière classique [BD94], nous les représentons par une liste doublement chaînée d'enregistrements pour les brins. Les dimensions sont simplement les entiers `0` et `1`. En C, une couleur est une valeur de type `unsigned int`. Un brin de type `dart` est un pointeur sur un enregistrement contenant une couleur et les quatre pointeurs nécessaires pour réaliser en accès direct `A` et `A_1` aux dimensions 0 et 1, `nil` étant représenté par `NULL`. Les quasi-hypercartes et les hypercartes sont en fait de type `dart`. Les *constructeurs* de cartes sont écrits avec effets de bord. L'égalité des brins est juste une égalité de pointeurs. Les prédicats sont transformés en fonctions booléennes et, comme il se doit, les préconditions restent en dehors du code. L'implantation des *destructeurs* de cartes avec effets de bord, des *observateurs*, fonctions et prédicats préliminaires à la segmentation, est élémentaire.

9.2. Fonction de segmentation en langage C

Les spécifications de `seg1` et `seg2` sont *indéterministes* : elles sont fondées sur un *raisonnement local* concernant les brins et les liaisons de l'hypercarte argument, sans qu'*aucun ordre* ne soit imposé pour leur parcours. Alors, un programme réalisant `seg1` ou `seg2` peut être un simple parcours de la liste des brins, *dans n'importe quel ordre*, avec un traitement local des brins et liaisons. Ici, nous écrivons directement `segmentation` comme une itération de la tête à la queue de la liste. En nommant `x` le brin courant, en conservant nos précédentes notations et en programmant avec effets de bord sur la quasi-hypercarte `m`, nous avons en C :

```
qhmap segmentation(qhmap m)
 { dart x = m,y,x_0,y0,x_1,y1,xs; colour a;
   while(x!=nil)
   {y = A(m,0,x);
    if (y!=nil)
     {m = B(m,0,x);
      if (x!=y)
       {a = col(m,y);
        x_0 = A_1(m,0,x);
        y0 = A(m,0,y);
        if (up_border0(m,y,a))
         {if (low_border0(m,x,a)) m = L(m,0,x,y);
          else m = L(B_1(m,0,x),0,x_0,y);
         }
        else
         {if (low_border0(m,x,a))
            m = L(B(m,0,y),0,x,y0);
          else
           {if (x_0==y) m = B(m,0,y);
            else m = L(B(B_1(m,0,x),0,y),0,x_0,y0);
           }
         }
       }
     }
     // ... ici, traitement similaire en dimension 1 ...
      xs = (x->s==x||x->s==m)?nil:x->s;
      if (isolated(m,x)) m = D(m,x);
      x = xs;
   }
   return m;
 }
```

Ce programme est tellement proche des expressions fonctionnelles précédentes de `seg1` et seg2 qu'il est inutile de le commenter. Il a été testé avec succès, notamment sur les exemples des Fig. 8 et 9. De plus, il est très efficace. En effet, la complexité en temps des opérations élémentaires, `A, B, D, L, low_border0`, etc., étant clairement en $\Theta(1)$ à cause des pointeurs, la complexité globale en temps de `segmentation` dans le pire des cas est en $\Theta(n)$, n étant le nombre de brins dans la quasi-hypercarte. Enfin, le programme n'utilise comme mémoire que quelques pointeurs et entiers en nombre fixe : sa complexité en espace dans le pire des cas est en $\Theta(1)$. Il est donc *optimal* en temps et en espace.

10. Discussion

Notre choix de codage d'une hypercarte par un mélange de brins et de liaisons dans une structure arborescente de terme peut intriguer car bien d'autres représentations paraissent plus immédiates. En calquant la définition mathématique, on peut par exemple retenir un triplet $M = (D, \alpha_0, \alpha_1)$ offrant l'avantage d'une vision ensembliste simple qu'apprécieraient certainement les zélateurs des méthodes Z ou B. Ceci oblige tout de même à travailler à deux niveaux : celui des composants D, α_0 et α_1, et celui de M où doivent s'exprimer des contraintes d'intégrité, comme la bijection des α_k sur D, et des opérations globales sur M. En bonne démarche par *abstraction*, celles-ci doivent cacher les opérations ensemblistes et l'on devra garantir dans des *obligations de preuves* la préservation des contraintes. C'est une sur-spécification, puisque l'on *implante* les opérations sur M par des opérations sur un modèle ensembliste. Les preuves en seraient peut-être simplifiées, mais ceci demanderait à être vérifié. De plus, dans une programmation réelle suivant le même schéma, on est conduit à représenter séparément D, α_0 et α_1, ce qui n'est pas toujours souhaitable, par exemple dans notre implantation par une seule liste chaînée. C'est bien pire si l'on doit gérer encore une troisième permutation redondante avec les deux premières, comme dans [Gon05].

Le choix des cartes libres avec les constructeurs `V`, `I` et `L` résulte d'une longue expérience de formalisation et de développement de logiciel à base de spécifications algébriques avec des modèles de cartes [BD94]. Ces opérateurs sont *atomiques*, contrairement à ceux de [Gon05], ils n'occasionnent aucune sur-spécification et permettent d'exprimer immédiatement les contraintes globales des hypercartes. Le principal mode de raisonnement engendré est une induction structurelle simple qui est la trame de la plupart de nos définitions, preuves et algorithmes, notamment celui de segmentation. Enfin, l'implantation des hypercartes n'est pas biaisée : une représentation chaînée, comme nous l'avons vue et comme on la pratique dans les modeleurs géométriques [BD94], est immédiate pour les constructeurs et les opérations de plus haut niveau. Mais nous aurions pu choisir, avec plus de difficulté cependant, une représentation contiguë dans des tableaux.

S'il est bien clair que Coq autorise toute cette variété de modes d'expressions, ses bibliothèques favorisent néanmoins la réutilisation d'ensembles et de relations, plutôt pour faire des preuves. Notre objectif de développement réel nous contraint à adopter un mode de pensée constructif très poussé dès le début avec les cartes libres. Mais il y aurait encore beaucoup à dire sur les propriétés de notre langage de cartes et d'hypercartes. Ainsi, tester l'égalité de deux cartes nécessite un point de vue *observationnel* par les α_k, ou des *formes canoniques*, que Coq n'offre pas à l'heure actuelle. Ceci nous handicape dans certaines preuves et il faut contourner la question. Il ne semble cependant pas que le problème serait beaucoup mieux résolu avec le parti pris ensembliste évoqué plus haut.

11. Conclusion

Ce travail s'appuie sur une base de spécifications d'hypercartes de près de 10 000 lignes de Coq, qui a servi à prouver des résultats fondamentaux [Duf07]. Le développement pour la segmentation en représente environ 5 000, avec une centaine de définitions, théorèmes et lemmes. C'est le prix à payer pour la rigueur et la sécurité. Le rapport spécifications/lignes de preuves est d'environ 1/15. Rien n'est automatisé à part ce qui l'est dans les tactiques standard de Coq.

Le Calcul des constructions inductives et le système Coq sont plutôt bien adaptés aux spécifications et preuves sur les hypercartes, grâce à nôtre schéma d'induction. Le programme impératif a été écrit "à la main" depuis l'algorithme fonctionnel. Même s'il en est étonnamment proche, pour la beauté du geste, il faudrait aussi le prouver. Ainsi, nous devrons examiner la possibilité de preuves de programmes impératifs avec des pointeurs, comme dans Why et Caduceus [FM04]. Il faudra alors étudier la transformation de nos programmes fonctionnels en programmes de ce type avec une *fonction d'interprétation* des listes chaînées vers les quasi-hypercartes et le transfert des obligations de preuves. Peut-être parviendrons-nous aussi à spécialiser à notre cas les techniques d'extraction de programmes à partir de preuves [BC04].

Par ailleurs, l'investigation du problème de segmentation peut se poursuivre pour optimiser l'hypercarte résultat, passer à la dimension 3 [Dam01], et travailler sur des surfaces quelconques. Ces questions sont essentiellement du ressort de l'algorithmique topologique. De vrais problèmes d'algorithmique géométrique concernant les plongements, les nombres réels et les erreurs d'arrondi, sont plus difficiles à appréhender. Il faut des axiomatiques adaptées comme celle de D. Knuth [Knu92] utilisée dans [PB01]. Enfin, la géométrie discrète, qui renouvelle la problématique de l'imagerie, sera abordable par nos techniques, quand on disposera d'une axiomatique constructive des courbes et surfaces discrètes.

Références

[BC04] Bertot, Y., Castéran, P. : Interactive Theorem Proving and Program Development - Coq'Art : The Calculus of Inductive Construction. Text in TCS, An EATCS Series, Springer-Verlag (2004).

[BD94] Bertrand, Y., Dufourd, J.-F. : Algebraic specification of a 3D-modeler based on hypermaps. Graphical Models and Image Processing **56** :1 (1994) 29–60.

[BFP99] Bertrand, Y., Fiorio, C., Pennaneach, Y. : Border map : a topological representation for nD image analysis. DGCI'99 Conf., LNCS **1568**, Springer-Verlag (1999) 241–257.

[BD99] Braquelaire, J.-P., Domenger, J.-P. : Representation of segmented images with discrete geometric maps. Image and Vision Computing **17** :10 (1999) 715–735.

[CH85] Coquand, T., Huet, G. : Constructions : A higher order proof system for mechanizing mathematics. EUROCAL (1985), LNCS **203**, Springer-Verlag.

[Coq04] The Coq Team Development - LogiCal Project : The Coq Proof Assistant Reference Manual - Version 8.0, INRIA, France (2004). http ://coq.inria.fr/doc/main.html.

[Cor70] Cori, R. : Un Code pour les Graphes Planaires et ses Appl. Astérisque **27**, SMF (1970).

[Dam01] Damiand, G. : Définition et étude d'un modèle topologique minimal de représentation d'images 2D et 3D. PhD Thesis, U. de Montpellier 2 (2001).

[DD04] Dehlinger, C., Dufourd, J.-F. : Formalizing the trading theorem in Coq. Theoretical Computer Science **323** (2004) 399–442.

[DP00] Dufourd, J.-F., Puitg, F. : Fonctional specification and prototyping with combinatorial oriented maps. Computational Geometry - Theory and Applications **16** (2000) 129–156.

[Duf07] Dufourd, J.-F. : A hypermap framework for computer-aided proofs in space subdivisions - Genus theorem and Euler's formula. ACM SAC'07 - Geometric Computing and Reasoning (2007).

[FM04] Filliâtre, J.-C., Marché, C. : Multi-Prover Verification of C Programs. 6th ICFEM'04, LNCS **3308**, Springer-Verlag (2004).

[Gon05] Gonthier, G. : A computer-checked proof of the Four Colour Theorem. Microsoft Research, Cambridge, http ://coq.inria.fr/doc/main.html (2005) 57 pages.

[HS85] Haralick, R.M., Shapiro, L.G. : Image segmentation techniques. CVGIP **29** (1985) 100–132.

[Knu92] Knuth, D.E. : Axioms and Hulls. LNCS **606**, Springer-Verlag (1992).

[MD05] Mota, J.-M., Dubois, C. : Raffinement événementiel pour les algorithmes géométriques. Conf. AFADL'2006, Paris (2006).

[Pau93] Paulin-Mohring, C. : Inductive Definition in the System Coq - Rules and Properties. *In* Typed Lambda-calculi and Applications. LNCS **664**, Springer-Verlag (1993) 328–345.

[PB01] Pichardie, D., Bertot, Y. : Formalizing Convex Hulls Algorithms. Theorem Proving in HOL Conf. (2001). LNCS **2152**, Springer-Verlag, 346–361.

[Ser06] Serra, J. : A Lattice Approach to Image Segmentation. J. of Mathematical Imaging and Vision **24** (2006), 83–130.