Transformation d'images par application quasi-affines discrètes

Maxime PETITJEAN, Guillaume HAYETTE

30 janvier 2008

Résumé

Ce document est le rapport sur le travail effectué par Maxime PETITJEAN et Guillaume HAYETTE sur le sujet du TER concernant la transformation d'image par application quasi-affines discrètes.

Mots-clés : Application affine discrète, Traitement d'image, Algorithme, TER.

Table des Matières

1. Introduction

Chaque personne de Master informatique connait déjà le monde des affaires grâce au stage effectué en fin de licence. Mais peu d'entre eux ont pu connaitre le monde de la recherche. Dans cette optique, il nous a été proposé d'effectuer un stage aux côtés d'un enseignant de l'université, afin de nous donner l'opportunité d'avoir un aperçu du travail de chercheur.
De nombreux sujets ont été proposés, sur des thèmes très diversifiés, et c'est en discutant avec Mr David Coeurjolly que nous avons trouvé celui qui nous conviendrait le mieux. Mr Coeurjolly nous proposa d'implémenter un algorithme de transformation d'image en nombre entier. En effet, les transformations d'images sont souvent très complexes, et nécessitent de grands temps de calculs. L'algorithme qu'il nous a été demandé de coder propose une manière optimisée de faire ces opérations, en utilisant que des nombres entiers.
Ce travail ne se limitera pas à un simple codage. Il nous faudra faire de l'analyse de document, de la découverte de l'art de transformer les images, de la compréhension des différentes étapes de l'algorithme, et finalement de l'implémentation d'algorithme écrit en pseudo code.

2. Le sujet

Le traitement d'images est une opération présente dans de nombreux logiciels, et utilisée par de nombreuses personnes, professionnelles ou amateurs. De plus, la rapidité de son exécution dépend de la taille de l'image, de la complexité de l'opération demandée, et surtout de l'optimisation de l'algorithme utilisé pour effectuer la transformation. C'est pourquoi les chercheurs tentent de trouver toutes les optimisations possibles à faire sur ces algorithmes.

L'absence de division, l'utilisation de nombres entiers plutôt que de flottants, la réduction du nombre d'opérations, tous ces détails sont à prendre en compte pour l'optimisation d'un algorithme. Notre travail consistait à analyser deux documents expliquant une technique optimisée, employée pour effectuer une transformation sur une image, et à coder l'algorithme décrit dans ces documents.

A. Application quasi-affine : définition

Une application quasi-affine est une application discrète définie par :

$$x' = \left\lfloor \frac{\mathrm{ax} + \mathrm{by} + \mathrm{e}}{\omega} \right\rfloor$$
$$y' = \left\lfloor \frac{\mathrm{cx} + \mathrm{dy} + \mathrm{f}}{\omega} \right\rfloor$$

Avec x, y, x', y', a, b, c, d, e, f, ω des entiers, et ω > 0

Une application quasi-affine peut donc être définie par une matrice, son coefficient, et un vecteur :

$$A = \frac{1}{\omega}\begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad V = (e \quad f)$$

Les applications quasi-affines sont séparée en deux groupes, qu'il est nécessaire de différencier, car les opérations à effectuer changent selon le groupe auquel elles appartiennent :

- les applications dilatantes : l'image finale obtenue avec une telle application est plus grande que l'image initiale. Par conséquent, chaque pixel de l'image initiale donne un ou plusieurs pixels dans l'image finale. Il apparaît que cet ensemble de pixel se positionne toujours en bloc, et c'est pourquoi nous l'appellerons par la suite un "pavé". Ainsi, le pavé de pixel obtenu dans l'image finale par l'application de la transformation sur le pixel de coordonnées (i, j) dans l'image initiale sera noté $P_{i,j}$.
- Les applications contractantes : l'image finale est plus petite que l'image initiale. C'est donc le processus inverse qui est opéré : un pavé de l'image initiale donnera un seul pixel dans l'image finale.

Une application est contractante si :

$$\omega^2 > ad - bc$$

Une application est appelée équivolume si

$$\omega^2 = ad - bc$$

Sinon, elle est dilatante.

Les équations définissant l'application quasi-affine (décrites au début du paragraphe) permettent de trouver les coordonnées d'un point de l'image finale (x', y') à partir d'un point dans l'image initiale (x, y)

Une application peut-être inversible. Pour toute application inversible, son application inverse permet de revenir à l'image initiale sans perte. Une application est inversible si

$$\delta \geq \omega \sup(|\mathrm{d}|, |b|, |a| + |c|)$$
ou si
$$\delta \geq \omega \sup(|\mathrm{a}|, |\mathrm{c}|, |d| + |b|)$$

avec $\delta = \mathrm{ad} - \mathrm{bc}$ représentant le déterminant de la matrice

B. Transformation d'image

Le but de cet algorithme est d'obtenir une image finale, c'est à dire un ensemble de pixels à partir des pixels de l'image initiale. La méthode à utiliser est donc différente selon si l'application qu'on veut appliquer à l'image est dilatante ou contractante.

a. Les pavés

Dans le cas d'une application dilatante, le nombre de pixels dans l'image d'arrivée est supérieur au nombre de pixels dans l'image de départ. Par conséquent, à chaque point de l'image initiale est associé un ensemble de pixels dans l'image d'arrivée. Ces ensembles de pixels sont groupés, et forment des pavés de pixels. Un des principes de cet algorithme est de trouver le pavé pour un point donné, de manière rapide et optimisée.

Le pavé obtenu par l'application quasi-affine à partir d'un point (i, j) sera noté $P_{i,j}$.

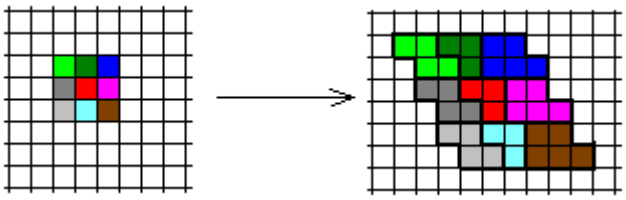

Figure 1

b. La transformation

Si l'application est dilatante, nous parcourons les points de l'image d'origine (à gauche dans le schéma figure 2). Pour chaque point, nous déterminons le pavé correspondant par l'application inverse, et nous appliquons à chaque point de ce pavé la couleur du pixel de départ.

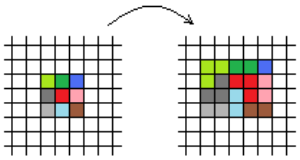

Figure 2

En revanche, si l'application est contractante, on détermine les quatre coins de l'image finale (opération marquée par (1) dans la figure 3). Puis, on parcourt les points de la figure formée par ces quatre coins. Pour chaque point, on détermine avec l'application son pavé correspondant, qui se trouve alors dans l'image initiale. On applique ensuite au point une couleur dépendant de la couleur des point du pavé. Dans la conception de notre programme, nous avons choisi d'appliquer la couleur moyenne des pixels du pavé. Mais il est possible d'utiliser d'autres critères, comme la couleur la plus foncé, la plus claire etc.

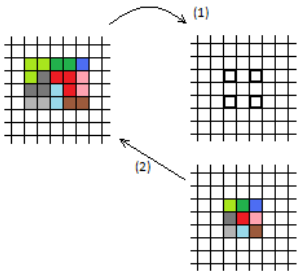

Figure 3

c. Déterminer la taille de l'image

Connaitre la taille de l'image finale est nécessaire pour pouvoir allouer la place suffisante à cette image. En effet, utiliser un système d'image dynamique serait extrêmement couteux en temps d'exécution, et nous perdrions tout l'avantage d'utiliser un algorithme optimisé.

La méthode utilisée pour déterminer la taille de l'image finale change selon si l'application est dilatante, ou contractante. Dans les deux cas, nous utiliserons les coins de l'image initiale pour déterminer la position des coins de l'image finale, et ainsi la taille de l'image

Pour une application contractante, nous avons vu que le calcul des quatre coins de l'image finale est la première opération à faire. Ils sont calculés par les équations de l'application quasi-affine :

$$x' = \left\lfloor \frac{ax + by + e}{\omega} \right\rfloor$$
$$y' = \left\lfloor \frac{cx + dy + f}{\omega} \right\rfloor$$

Pour chaque coins (x, y) de l'image initiale, nous obtenons le coin de l'image finale (x', y') correspondant, et nous obtenons ainsi la taille de l'image.

Pour une application dilatante, nous savons que le pavé correspondant à un coin de l'image initiale sera également situé à un coin de l'image finale. Il faut donc calculer le pavé de chaque coin, puis déterminer la taille de l'image en recherchant l'abscisse minimale trouvée dans les quatre pavé, l'abscisse maximale, et de même pour les ordonnées.

C. Construction des pavés

Grâce à l'application quasi-affine, il nous est possible de définir les équations des quatre droites discrètes. En effet, on sait que $x' = \frac{ax+by+e}{w}$. Le pavé du point (x, y) est donc entouré de deux droites, que nous définition par l'équation

$$\omega x' \leq ax + by + e < \omega (x' + 1)$$

De même, $y' = \frac{cx+dy+f}{\omega}$ donne l'inéquation $\omega y' \leq cx + dy + f < \omega (y' + 1)$

Le pavé d'un point (x, y) est donc borné par ces quatre droites (représentées en rouge dans la figure 4). Chaque pixel présent à l'intérieur de ce parallélogramme fait partie du pavé

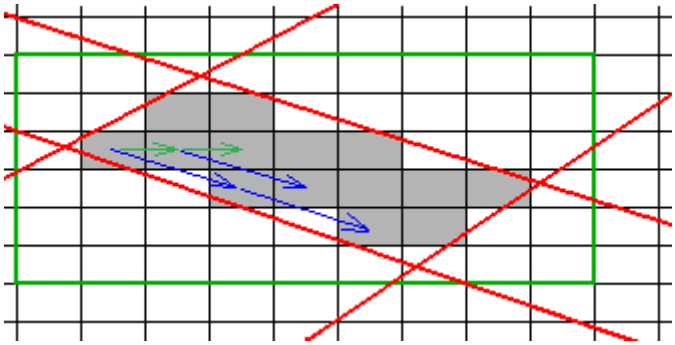

Figure 4

Le protocole à suivre pour déterminer les points appartenant au pavé utilise des translations successives de deux vecteurs (flèches vertes et bleues dans la figure 4). Pour cela, et pour chacun des deux vecteurs, il faut connaitre le nombre de fois qu'il faut répéter cette translation. Si nous allons trop loin, nous sortons du parallélogramme. Si nous n'allons pas assez loin, des points sont oubliés

Par conséquent, l'algorithme de détermination de pavé contient deux boucles : une pour chaque vecteur utilisé. L'arrêt de la première boucle (celle des flèches vertes) se définit par deux variables, appelée M et N, qui dépendent de variables découlant de la matrice et du vecteur de l'application. L'arrêt de la seconde boucle se définit par deux autres variables, appelées C_k et D_k, dont les formules dépendent également de variables découlant de la matrice et du vecteur de l'application.

Une fois ces quatre variables connues, les itérations des deux boucles imbriquées nous permettent d'obtenir les points appartenant au pavé.

D. Optimisation : périodicité des pavés

Nous avons vu comment créer un pavé à partir d'un pixel par application de la transformation souhaité, et cela en n'employant que des nombres entiers. Nous avons donc déjà une application très optimisée et rapide pour effectuer une transformation sur une image.

Mais sur une image suffisamment grande, certains pavés de l'image sont identiques, c'est à dire contiennent des pixels positionnées de la même façon. Un pavé répété plusieurs fois dans une image n'a donc besoin d'être calculé qu'une seule fois, puis déplacé par translation aux autres endroits où il doit être présent. Nous obtiendrions ainsi un algorithme faisant encore moins d'opérations que le précédent.

Cette répétition de pavé est périodique, et forme un pavage. Ce pavage est arithmétiquement explicable, par un modulo dépendant de ω. Cela signifie que les pavés géométriquement identiques se répètent de manière régulière, et peuvent donc être définis par translation de vecteur. Par conséquent, l'image finale est formée d'un « super pavé », contenant l'ensemble des pavés distincts (les super pavés sont entourés de traits gras dans la figure 5). Pour chaque pavé qu'on veut créer, on cherche donc le pavé qui lui est identique à l'intérieur du super pavé, et le vecteur correspondant à la translation de ce pavé vers le pavé que l'on veut créer.

Appelons P l'ensemble de ces pavés distincts. Pour chaque point (i, j) de l'image initiale, nous déterminons le point (i', j') et le vecteur V tel que :

- $P_{i,j}$ est l'image de $P_{i',j'}$ par translation du vecteur V
- $P_{i',j'}$ appartient à P

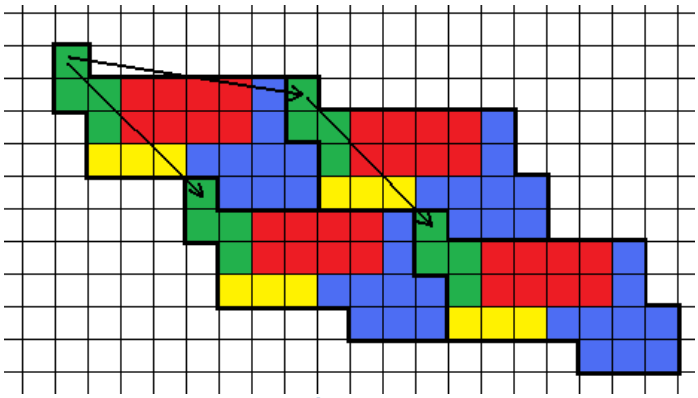

Figure 5

Ainsi, on ramène chaque point à un point dont le pavé à déjà été calculé. Inutile de faire de nouveau le calcul du pavé, il suffit juste de translater le pavé avec le vecteur obtenu. Nous obtenons ainsi la même technique que la précédente, mais le nombre d'opérations à considérablement diminué. Si cela ne se ressent pas sur les petites images, plus l'image est grande, plus le nombre de pavés semblables est important, et donc plus cette méthode apparait comme optimisée.

E. Algorithme obtenu

Le calcul d'une image finale obtenue par une application quasi-affine s'effectue donc par cet algorithme :

Données : Image initiale I, application quasi-affine f
si f contractante
alors
    F = application quasi-affine associée à f
    Calculer les quatre sommets de l'image finale
    min_i, max_i = abscisse minimale et maximale de ces sommets
    min_j, max_j = ordonnées minimale et maximale de ces sommets
Sinon, f est dilatante
    F = application quasi-affin inverse de f (notée f^{-1})
    min_i, max_i = abscisse minimale et maximale de l'image initiale
    min_j, max_j = ordonnées minimale et maximale de l'image initiale

Déterminer l'ensemble P des pavés distinct
Pour i de min_i à max_i faire
    Pour j de min_j à max_j faire
        Déterminer i', j' et le vecteur V tels que :
        - $P_{i,j}$ soit l'image de la translation de Pi', j' par le vecteur V
        - $P_{i',j'}$ appartienne à P

        Si f contractante
            Affecter à (i, j) une couleur dépendant des couleurs des points de $P_{i,j}$
        Sinon
            Affecter aux points de $P_{i,j}$ la couleur du point (i, j)

Voici un exemple de transformation d'image obtenu avec l'application suivante :

$$A = \frac{1}{4}\begin{pmatrix} -3 & -4 \\ 4 & -3 \end{pmatrix} \quad V = (0 \quad 0)$$

Image initiale

Image finale

3. Déroulement du TER

Mr Coeurjolly nous a fourni deux extraits expliquant le fonctionnement de ces applications quasi-affines, et donnant l'algorithme associé écrit en pseudo-code. Nous étions en présence de deux documents :

- L'extrait d'un article, rédigé en anglais, détaillant un algorithme optimisé.
- L'extrait d'un livre, rédigé en français, qui s'inspire du document précédent, et qui explique, de manière plus détaillée et lisible, certains points du premier article.

L'analyse des documents était la partie la plus importante pour nous, car n'ayant aucune expérience dans ce domaine, le vocabulaire employé, ainsi que la rapidité des explications données ne permettaient pas une compréhension rapide. C'est pourquoi nous avons dû plusieurs fois faire appel à Mr Coeurjolly pour nous aider. Celui-ci nous offrit des explications qui nous débloquèrent souvent, ou des compromis, permettant de contourner les parties qui nous posaient trop de problèmes. Les algorithmes étant vaguement expliqués, puis rapidement donnés. Il était souvent difficile pour nous de comprendre, étant donné le peu d'expérience que nous avions.

Notre TER s'est donc déroulé en trois phases :

- Analyse du document, qui nous pris le plus de temps
- Codage d'outils de bases nécessaires pour l'application de l'algorithme
- Codage de l'algorithme, qui s'est avéré plus long que prévu.

A. Les outils de base

Nous avons vu comment créer un pavé à partir d'un pixel par application de la transformation souhaitée, et cela en n'employant que des nombres entiers. Nous avons donc déjà une application très optimisée et rapide pour effectuer une transformation sur une image.

Ces outils ont été codés en parallèle avec la lecture approfondie des documents. Ils correspondent aux outils que nous savions indispensables. Ainsi, lorsque nous avons commencé à coder l'algorithme, nous disposions de :

- une classe "matrice", nous permettant de faire diverses opérations sur les matrices (multiplication, calcul du déterminant ...)
- une classe "vecteur", proposant le même genre d'opération
- une classe "image", permettant d'ouvrir un fichier image, de le sauvegarder, et cela dans trois formats différents. De plus, une classe "pixel" permet de décomposer l'image en pixel, facilitant son parcourt pour l'application de la transformation de l'image.
- d'autres classes permettant la gestion des images et des pixels, qu'il n'est pas nécessaire de détailler ici

B. Transformation d'une image

Nous avons commencé à coder l'algorithme de transformation d'une image, sans prendre en compte l'optimisation par l'utilisation des pavés périodiques. Nous nous sommes contentés d'essayer d'obtenir un pavé pour chaque pixel, et de le placer au bon endroit dans l'image finale.

Pour cela, nous avons suivit l'algorithme expliqué dans le paragraphe 2. C. Nous parcourions les points de l'image initiale, et nous effectuions les opérations selon la convergence de l'application.

Cependant, lors de la programmation de cette section de code, nous nous sommes rendu compte que les valeurs trouvées pour les variables C_k et $D_{k,}$ qui correspondent à l'arrêt de la boucle (cf. paragraphe 2. C.) ne correspondaient pas à ce qui était attendu. Elles ne permettaient pas de construire le pavé correctement, nous faisait sortir du pavé, ou nous donnait un nombre de pixel nul. De plus, un exemple explicité dans le livre dans lequel l'auteur donnait les résultats obtenus pour le calcul de C_k et D_k nous montrait que nos résultats étaient erronés.

Nous avons passé beaucoup de temps à essayer de comprendre où était notre erreur. Le déroulement de l'algorithme à la main sur papier donnait rigoureusement les même résultats que ceux que nous trouvions par exécution de notre code, et nous étions sûrs, pour les avoir calculées de nombreuses fois, des valeurs des variables que nous utilisions. Sur conseil de Mr Coeurjolly, nous avons utilisé une autre méthode pour déterminer les pixels appartenant aux pavés.

Nous avons vu qu'un pavé est entouré par un parallélogramme, définit par quatre droites discrètes dont nous connaissons les équations. Ces équations sont obtenues à partir des inéquations vues dans le paragraphe 2. c. :

$$\omega x' <= ax + by + e < \omega\,(x' + 1)$$
$$\omega y' <= cx + dy + f < \omega\,(y' + 1)$$

Chaque point dont les coordonnées (x', y') répondent à ces inéquations appartiennent donc au pavé $P_{x,\,y}$.

En utilisant le carré bornant le parallélogramme (carré représenté en vert dans la figure 4), nous parcourrons chaque point susceptible d'appartenir à $P_{x,y}$. Pour chaque point, si ses coordonnées répondent aux conditions d'appartenance au pavé, c'est qu'il est à l'intérieur du parallélogramme, et donc qu'il appartient au pavé.

En appliquant cette méthode à l'ensemble des points de l'image initiale, nous trouvons les pavés de chaque point, et donc une image finale correcte.

Nous avons enfin obtenu un résultat. Beaucoup d'applications affines furent testées, les résultats étaient satisfaisants. Bien que la méthode pour les trouver ne respecte pas tout à fait l'algorithme expliqué dans l'article, nous avons codé un programme de transformation d'image n'utilisant que des nombres entiers.

Par la suite, nous avons étudié l'algorithme écrit en annexe de l'article en pseudo code, qui permet de déterminer le pavé d'un point. Cet algorithme reprenait la même méthode que celle décrite dans l'article, avec quelques variantes. Mais surtout, nous nous sommes rendu compte que les formules de certaines variables n'étaient pas exactement identiques. Parenthèses déplacées, signes inversés, les deux formules donnaient des résultats différents. Nous avons alors implémenté cet algorithme, en faisant les changements qui nous paraissaient logiques.

Les valeurs trouvées alors pour les variables Ck et Dk sont apparues plus plausibles, et certaines applications ont fonctionnées avec cette méthode. Néanmoins, certaines applications continuaient à renvoyer de mauvais résultats.

Nous nous sommes donc penchés sur les éléments restant incertains de notre algorithme, nous avons étudié les différences entre les deux documents, essayant de comprendre et de trouver des éventuelles erreurs de raisonnement de notre part, ou des imprécisions de l'article qui auraient conduit à des conclusions hâtives. L'élément le plus aléatoire de notre algorithme repose sur deux variables appelées v_0 et v_1. Ces variables représentent les vecteurs utilisés pour la recherche des points appartenant à un pavé. Ces vecteurs sont représentés par les flèches vertes dans la figure 4. V_0 et v_1 doivent être définies telles que :

$$c' v_0 + d' v_1 = 1$$

avec c' et d' connus

C'est par le théorème de Bézout que nous calculons ces valeurs. Néanmoins, plusieurs couples d'entiers sont possibles, et une seule condition est données dans l'article pour vérifier que nous avons le bon couple : v_0 et v_1 doivent être choisis tels que

$$p = a v_0 + b v_1 < \delta_1$$

avec δ_1 connu

Nous calculions donc v_0 et v_1, et nous testions si la condition était vérifiée. Puis, nous appliquions ces deux variables au reste de l'algorithme.

Or, nous nous sommes rendus compte que la condition de validation des variables v0 et v1 n'était pas suffisante. En effet, plus p est proche de δ_1, plus les résultats obtenus par la suite par notre code sont proches des bons résultats. Nous avons donc modifié la condition en disant que v_0 et V_1 doivent être choisis tels que p doit être le plus proche entier inférieur à δ_1. Voici le détail du calcul permettant de faire cela :

D'après le théorème de Bézout, il existe un entier k tel que :

$$c'(v_0 - kd') + d'(v_1 + kc') = 1$$

Nous voulons :

$$a(v_0 - kd') + b(v_1 + kc') < \delta_1$$
$$av_0 + bv_1 - \delta_1 < k(ad' - bc')$$

$Or, \delta_1 = ad' - bc'$

$$d' \text{ où } k = \frac{av_0 + bv_1}{\delta_1}$$

Il est maintenant aisé de trouver les valeurs correctes de v_0 et v_1.

En appliquant cette nouvelle condition, les résultats obtenus deviennent correctes. Les deux méthodes que nous avons codées retournent exactement la même image résultat.

C. Périodicité

Comme nous l'avons vu précédemment, cette phase d'optimisation comporte plusieurs étapes :

- Trouver P, l'ensemble des pavés distincts de l'image finales
- Pour chaque point (i, j) de l'image initiale, déterminer le point (i', j') et le vecteur $\vec{V}$ tels que $P_{i',j'}$ soit la translation de $P_{i,j}$ par le vecteur $\vec{V}$, et que Pi',j' appartienne à P.
- Appliquer la translation du pavé, et donner au pavé la couleur du point (i, j).

Lors de la programmation de cette partie de l'algorithme, nous avons eu un problème similaire à ceux rencontrés précédemment. Un exemple donné dans l'article permettait de connaître les résultats que nous étions supposés obtenir. Les formules étaient données, les variables utilisées étaient juste, mais le calcule du vecteur $\vec{V}$ qui découlait de ces variables par une formule donnée renvoyait un mauvais résultat. Il nous était alors impossible d'avancer plus loin.

Afin de contourner cette difficulté, Mr Coeurjolly nous a proposé une solution. Nous pouvons déterminer les pavés identiques grâce à leurs restes. Deux pavés ayant les mêmes restes sont géométriquement identiques. Le reste d'un pavé correspond aux morceaux de pixels présents à l'intérieur du parallélogramme, mais dont les pixels n'appartiennent pas au pavé.

Le but de l'algorithme est de calculer les pavés identiques qu'une seule fois, et d'effectuer des translations de ces pavés pour remplir l'image finale. Pour contourner cette méthode, nous allons déterminer les pavés un à un, et retenir leur restes et leur positions. Dès que deux pavés ayant le même reste seront trouvés, le vecteur les liant sera facilement déterminable, et nous pourrons l'utiliser pour placer ce pavé dans le reste de l'image.

Ainsi, dans l'exemple de la figure 6, lors du calcul du pavé du point marqué d'un 4, le programme saura que le pavé du point 1 est le même. Le vecteur permettant d'aller du pavé 1 au pavé 4 (représenté par la flèche rouge) est alors calculé, et il est possible de copier ce pavé aux bons endroits.

Chaque pavé ne sera alors calculé que deux fois. Cette méthode, bien que moins optimisée que celle proposée à l'origine, permettrait d'utiliser les propriétés de périodicité des pavés, et d'obtenir un programme optimisé. De plus, plus l'image est importante, plus la différence de temps sera intéressante.

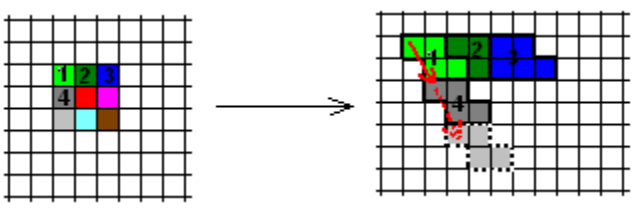

Figure 6

4. Conclusion

Notre TER a faillit se solder par un échec, avec l'impossibilité de coder l'algorithme, les résultats que nous trouvions étant faux. Mais nous avons persévéré et percé les raisons de ces erreurs, découvrant petit à petit l'algorithme à implémenter.

La partie analyse de document à été très longue, et nous a montré à quel point il peut être difficile de comprendre les uniques sources que nous avons pour effectuer le travail demandé. Dans le monde des affaires, un codeur se trouvant face à un problème a toujours le choix d'aller se documenter, et de trouver une solution. Un chercheur fait souvent des travaux précurseurs, pour lesquels les documentations et les aides sont inexistantes. C'était notre cas ici. Nous avions une documentation, mais dès que sa compréhension nous posait problème, il nous était impossible de nous documenter ailleurs. La seule chose qui peut faire que les obstacles soient franchis est la recherche minutieuse dans la totalité du problème.

C'est un aspect de l'informatique qui nous était inconnu, et il nous a fallu nous y adapter. Mais finalement, nous avons corrigé nos erreurs en continuant à chercher, et nous pensons avoir découvert le véritable comportement du chercheur dans ces moments là. C'est par cela qu'il nous est possible de présenter un résultat au lieu de calculs erronés.

Nous avons découvert qu'à force d'analyse, toutes les erreurs, de nous ou des documents, peuvent être trouvées et corrigées. Mais cela prend du temps, que nous n'avons plus. Alors que ce TER s'achève, nous commencions juste à découvrir le comportement à avoir face à ce genre d'erreur, qui est différent de tout ce que nous avons connu jusque là. Notre travail reste donc inachevé, mais nous comptons continuer à analyser la fin des documents, afin de trouver le bon code, qui permettra de terminer le travail.

5. Références bibliographiques

JACOB Marie-Andrée, « transformation of digital images by discrete affine application », *Computer & graphics*, vol 19 p. 373-389, 1995

ANDRES Eric, JACOB Marie-Andrée, « Géométrie discrète et image numérique », p.173-190