

# Détection du réseau routier

---

GARCIA Guillaume, LEBLANC Thibault, TALLANDIER Benoît

## Module TOP : Détection du réseau routier

Date de rendu : 07/01/16



## Table des matières

I.	Introduction.....	3
	Rappel du sujet : .....	3
	Démarche : .....	3
II.	Retravailler l'image.....	4
1.	Méthodes existantes : .....	4
2.	Travail réalisé : .....	4
III.	Identifier les routes .....	6
1.	Raisonnement : .....	6
2.	Travail réalisé : .....	6
IV.	Coloration des routes .....	9
V.	Résultats et Conclusion .....	10
VI.	Annexe.....	11
	Document 1 .....	11
	Document 2 .....	11
	Document 3 .....	12
	Document 4 .....	13
	Document 5 .....	15
VII.	Bibliographie.....	16

# I. Introduction

## Rappel du sujet :

Le but du projet était d'écrire un algorithme, divisé en plusieurs fonctions, permettant la reconnaissance de routes sur des images de différents formats et de définitions différentes. La recherche du réseau routier ne doit pas se baser uniquement sur la couleur. Nous avons travaillé sur la bibliothèque d'image test (*figure 1*) ainsi qu'avec l'API fournie.

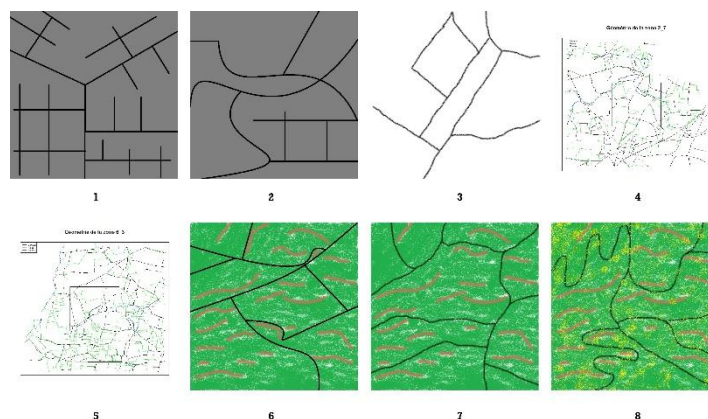


Figure 1

## Démarche :

Afin de répondre au problème posé nous avons tout d'abord divisé notre travail en trois grandes parties. La première étape consistant à retravailler l'image de manière à la rendre exploitable. La seconde étape consiste alors à isoler les routes potentielles afin de pouvoir les colorier.

Nous nous sommes aussi répartis les tâches en fonction de nos capacités ainsi que de nos préférences et avons effectué un suivi de notre projet à l'aide d'un diagramme de Gantt (voir annexe document 1 & 2). Ce dernier fut particulièrement utile pendant les périodes de partiels et de vacances. Une des autres initiatives que nous avons pris a été de se répartir les responsabilités de manière à avoir 3 postes : l'un sur l'algorithme, les autres sur la recherche et la création du rapport. Cela nous a permis de mieux travailler en équipe, puisque chacun a pu relancer les autres membres du groupe concernant les éléments de sa propre partie.

## II. Retravailler l'image

Afin de pouvoir travailler sur toutes les images nous avons choisi de les passer en niveau de gris. Le gros avantage du niveau de gris étant qu'il va permettre l'utilisation de filtres, nous permettant ainsi de détecter les fortes variations de couleurs, et donc des contours.

### 1. Méthodes existantes :

Il existe en effet de nombreux filtres permettant de faire de la détection de contours. Le plus basique d'entre eux étant le filtre gradient. Ce dernier va simplement comparer les niveaux de gris entre les pixels adjacents.

Exemple détaillé : Gradient horizontal (*figure 2*)

- 1) Initialement on se trouve en position B
- 2) Dans un second tableau, on va donner au pixel B la valeur suivante :

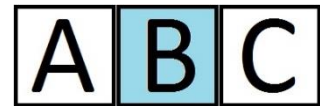


Figure 2

$$\text{valeur de B} = \frac{|\text{valeur de A} - \text{valeur de C}|}{2} \quad (\text{en niveau de gris})$$

- 3) Ainsi s'il n'y a pas de variation forte autour de B, la valeur de B est presque nulle. Une forte variation de couleur va donc se caractériser par un niveau de gris élevé après traitement.

D'autres filtres utilisent le même principe comme les filtres de Sobel, Prewitt, ou Kirsch (que nous avons testé). Des filtres de lissage permettent aussi de limiter l'impact du bruit comme celui de Nagao (détaillé par la suite).

### 2. Travail réalisé :

En premier lieu nous avons passé l'image en **niveau de gris**. Le seul point de difficulté de cette étape consiste à dissocier les niveaux des 3 couleurs (RVB) contenus. La méthode la plus simple consiste à utiliser un ET logique bit à bit :

```
12      var rouge: Int = ((x & 0xFF0000)/math.pow(16,4)).toInt
```

On sait que chaque octet de x contient la transparence pour le premier octet, ou la valeur attribuée à une couleur pour les octets suivants. En utilisant un ET logique entre x et 00FF0000 on est certain de récupérer la valeur attribuée à la couleur rouge. On la divise ensuite pour obtenir un entier entre 0 et 255. Il ne reste donc plus qu'à faire la moyenne des 3 couleurs pour obtenir le niveau de gris du pixel concerné. Il ne reste plus qu'à appliquer ce traitement à chaque pixel de l'image.

La principale difficulté du prétraitement de l'image a été de trouver, puis de mettre en place, un filtre permettant d'éviter les problèmes liés au flou et aux nuages de pixels jaunes présents sur les 3 dernières images.

Le problème étant qu'en utilisant un algorithme de flou classique (moyenne de la couleur du pixel central avec celle des pixels voisins) les contours des routes étaient détériorés. **Le filtre de Nagao** a quant à lui la particularité permettre un lissage de l'image en préservant les contours.

Fonctionnement :

Le filtre de Nagao se base sur neuf masques pour établir la couleur du pixel central (*figure 3*). On a donc 4 fois le masque 1 et 4 fois le masque 2 (rotation d'un angle de 90°) et l'on procède de la manière suivante :

1. On fait la moyenne du niveau de gris des pixels de chaque masque
2. On calcule la variance de chaque masque (c'est-à-dire l'écart à la moyenne)
3. La valeur moyenne (en niveau de gris) du masque possédant la plus petite variance devient la nouvelle valeur du pixel central.

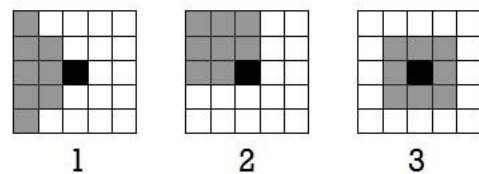


Figure 3 (image extraite de la bibliographie)

Après un passage du filtre de Nagao sur les images, nous n'avons pas vu de différence notable entre un filtre gradient et un filtre de Sobel. Nous avons donc utilisé **un filtre gradient** que nous avons légèrement modifié, ce dernier étant plus simple et fonctionnant tout aussi bien, nous n'avons pas jugé bon de le changer.

Fonctionnement :

Il se base sur le même modèle que le filtre gradient. On donne simplement au pixel B (*figure 4*) la valeur suivante :

$$\text{valeur de B} = |\text{valeur de B} - \text{valeur de C}| + |\text{valeur de B} - \text{valeur de A}| \quad (\text{en niveau de gris})$$

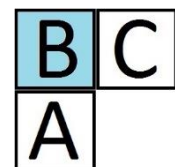


Figure 4

On mesure ainsi l'écart de teinte avec le pixel de droite, mais aussi avec celui du dessous. Il est à noter que nous n'avons pas utilisé de coefficient pour diviser le résultat du pixel B : C'est inutile puisque nous avons inclus à notre gradient un système de **seuillage**. Si la valeur après le passage de notre filtre gradient est trop élevée, c'est-à-dire si l'écart constaté avec ses voisins est significatif, le pixel devient blanc, sinon il devient noir.

Nous n'avons cependant pas réussi à trouver de manière formelle pour calculer une valeur de seuil idéale. Nous avons donc déterminé celle-ci de manière empirique (voir annexe document 3) et l'avons fixé à une valeur de 40.

### III. Identifier les routes

L'identification des routes a certainement été la partie qui nous a posé le plus de problème, à la fois du point de vue du raisonnement, mais aussi du point de vue du codage.

#### 1. Raisonnement :

Initialement nous avions deux options :

- Tracer d'abord la totalité des routes potentielles et ensuite les trier.
- Rechercher les vraies routes dès le début.

Le premier cas nous imposait non seulement de chercher une large multitude de routes possibles mais il fallait ensuite les trier. Après quelques tentatives nous avons abandonné cette idée car le temps d'exécution de nos algorithmes était beaucoup trop long par rapport aux résultats observés.


Après réflexion, le second cas est apparu plus simple à traiter. En effet, à condition que le prétraitement de l'image nous permette d'avoir des contours relativement propres, il suffit ensuite de ne rechercher que le début des routes. Les routes étant forcément reliées soit entre elles, soit à un des bords de l'image, il suffit ensuite de colorier intelligemment à partir des débuts de routes identifiés précédemment pour obtenir la totalité des routes.

#### 2. Travail réalisé :

Afin de détecter le début des routes nous avons utilisé 4 programmes, deux programmes principaux et deux programmes d'aide. Parmi les deux programmes d'aide, le premier, **contient**, retourne un booléen si un élément donné est compris dans la liste rentrée en paramètre. Le second, **cercle**, renvoie les coordonnées des points contenus dans un disque (de centre et de rayon donnés) dans une liste.

Un des deux programmes principaux est le programme **check**. C'est un programme récursif faisant appel au backtracking et ayant pour but de retourner les coordonnées de pixels blancs formant un chemin de longueur n. Cette fonction va recevoir en entrée les paramètres suivants :

```
326 def check(table:Array[Array[Int]],colb:Int,rowb:Int,col:Int,row:Int,n:Int,chemin:List[List[Int]],ok:Boolean):List[List[Int]]={
```



- 1) Image à traiter
- 2) Coordonnées du pixel traité précédemment
- 3) Coordonnées du pixel à traiter
- 4) Longueur du chemin désiré
- 5) Chemin en cours (de longueur comprise entre 0 et n)
- 6) Un booléen pour permettre une sortie rapide de la boucle.

Nous avons codé notre algorithme de telle sorte qu'ils prennent en priorité certaines directions plutôt que d'autres, en fonction du pixel traité précédemment.

Fonctionnement (figure 5):

- 1) A l'étape initiale on se trouve sur le pixel « A », le pixel « a » étant celui défini comme son prédécesseur (il n'est pas obligé d'exister physiquement). Il est impossible de faire demi-tour (pixels rouges). On va donc se déplacer sur la case numérotée 1 si le pixel est blanc. A défaut on recommencera le test sur la case numérotée deux, et ainsi de suite.
- 2) Le pixel est valide l'appel récursif suivant nous place donc en position (2,1) et le pixel A est pris comme nouveau prédécesseur.
- 3) Puisque qu'on se retrouve dans la même configuration que précédemment on réattribue les mêmes priorités concernant les directions à envisager.
- 4) Dans le cas d'un déplacement en diagonale les priorités seront définies ainsi.

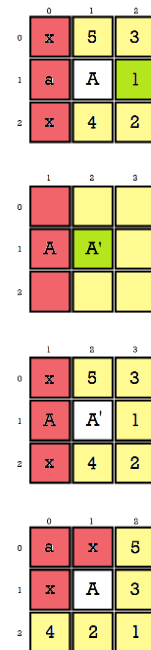
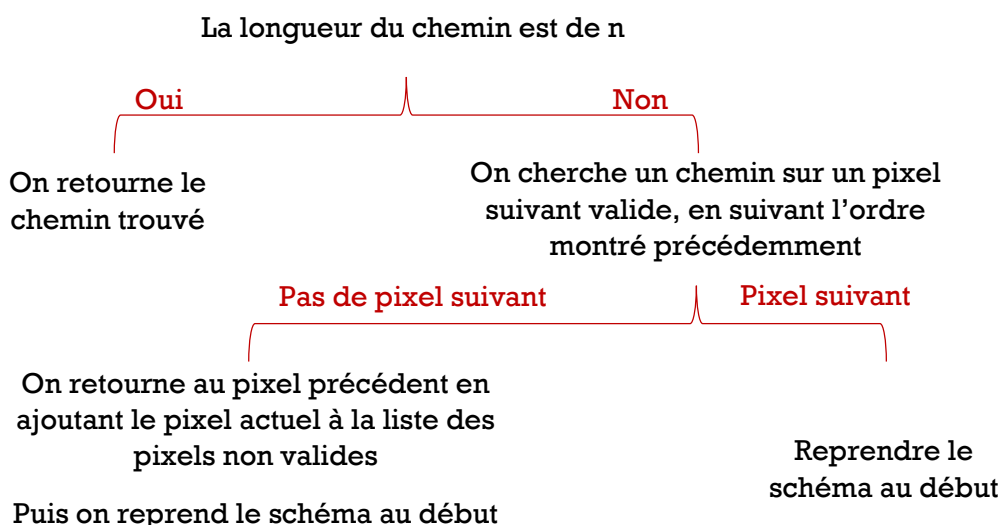


Figure 5

En utilisant un système de différence avec la position du pixel précédent on peut faire de même quel que soit la position du pixel précédent.

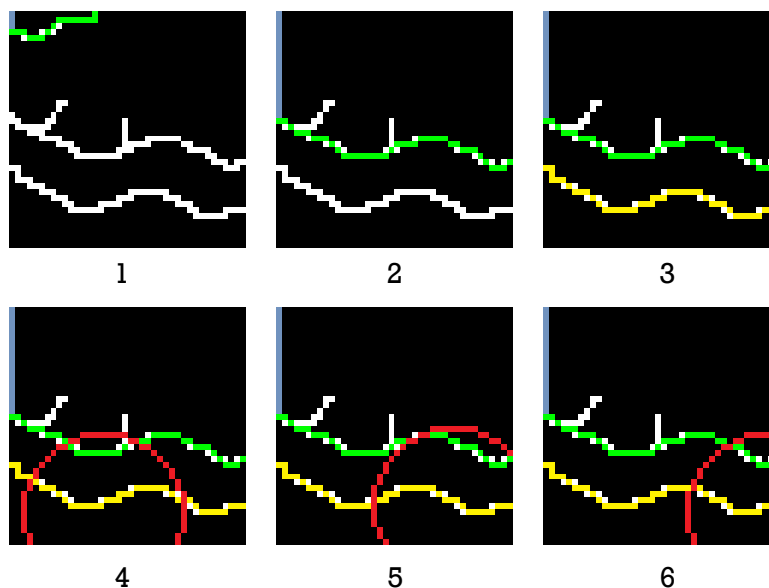
Le fonctionnement global de l'algorithme est cependant un peu plus complexe puisqu'il doit prendre en compte le backtracking, en effet il revient à la position précédente s'il ne trouve pas de chemin. Une liste de « pixels morts » a aussi été utilisée pour cet algorithme. Elle permet de lister les pixels qui n'ont pas aboutis afin de ne plus les prendre en compte ultérieurement. L'exécution se déroule donc de la manière suivante :



Ainsi à la fin de l'exécution de check soit tous les pixels qu'il était possible d'emprunter ont été ajoutés à la liste des pixels morts. Dans ce cas le programme retourne la liste vide. Dans le cas contraire, on renvoie le premier chemin de longueur n trouvé. En prenant n assez grand on trouve quasi systématiquement une route.

On utilise ensuite une fonction nommée **debut\_route** Il en existe deux versions, pour les chemins horizontaux et verticaux mais nous nous contenterons d'expliquer uniquement **debut\_route\_vertical**. Ce programme est celui qui va permettre de localiser les débuts des routes. A l'aide d'un test sur la colonne de départ on peut déterminer un sens de recherche (de la gauche vers la droite ou inversement). On se placera donc dans le cas où l'on part de la gauche de l'image.

Fonctionnement :



- 1) On commence à parcourir la première colonne en cherchant un pixel blanc suivi d'un pixel noir. Lorsque l'on en a trouvé un, on lance la recherche d'un chemin de longueur n assez grande (n=50 dans le cas de nos test). Dans le cas présent il n'y en a pas. On continue à chercher sur la première colonne.
- 2) Sur la seconde image on trouve bien un chemin de longueur n. On en cherche un second qui est proche du précédent (moins de 10 pixels).
- 3) S'il y en a un on le garde en mémoire pour pouvoir le parcourir.
- 4) 5) et 6) On déplace un cercle de rayon 10 pixels sur le chemin 2. Si après avoir parcouru le chemin deux, 90% des pixels du chemin 1 sont apparus dans le cercle. On considère que l'on a une route entre les deux chemins.

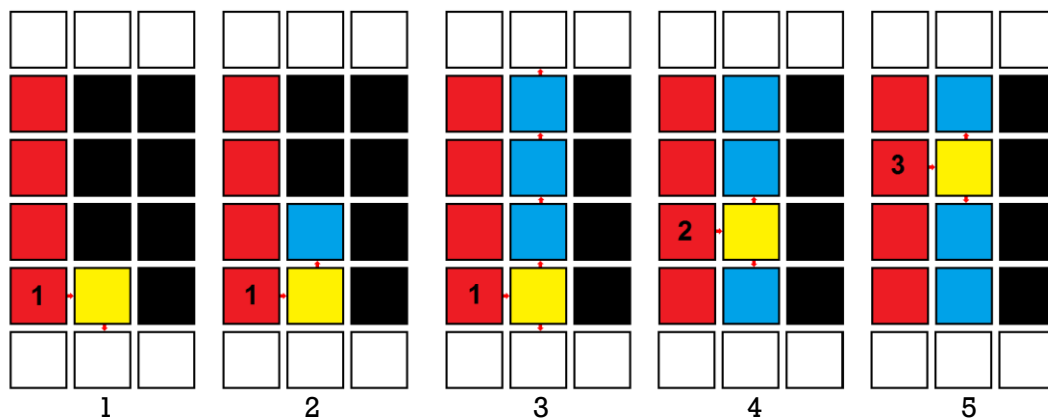
On va donc pouvoir récupérer les endroits où il y a bien une route à colorier. Il suffit ensuite d'utiliser les fonctions débuts roue sur les 4 bords de l'image.



## IV. Coloration des routes

Pour la coloration des routes nous sommes partis des débuts de routes identifiés précédemment. Nous les avons utilisés comme point de départ et avons mis au point un algorithme capable de retourner la liste des pixels à colorier. Le programme **coloration** va en fait chercher les pixels qu'il peut colorier dans la route. Pour des raisons pratiques nous avons réalisés un algorithme `coloration_vertical` et un algorithme `coloration_horizontale`. On se contentera d'expliquer l'algorithme **coloration\_vertical** qui avance colonne par colonne dans le tableau.

Fonctionnement :



- 1) On dispose initialement de la liste des pixels coloriés précédemment. On va commencer par se placer à la position 1 (dernier pixel colorié). S'il est possible de colorier le pixel voisin (jaune), on le colorie. On cherche ensuite à colorier tous les pixels qui ne sont pas blancs ou déjà coloriés se trouvant en dessous.
- 2) On se remet sur le pixel jaune puis on colorie de la même manière les pixels au-dessus.
- 3) Dans ce cas toute la largeur du chemin est déjà coloriée.
- 4) Mais il faut tout de même répéter l'opération en position 2. Cependant puisque les cases sont coloriées l'algorithme va passer directement à la position 3.
- 5) Et ainsi de suite jusqu'à ce que tous les pixels coloriés à la colonne précédente soient passés.

Si le fait de passer en revue tous les pixels coloriés peut paraître inutile dans cet exemple, il est en revanche nécessaire dans le cas où l'on arrive à un embranchement. En effet, puisque l'on part de la position des pixels précédemment coloriés et que l'on colorie au maximum au-dessus et en dessous de chacun d'entre eux. Il n'y a donc pas de problème si les pixels se retrouvent sur des chemins différents.

En parcourant l'image dans les deux sens (vertical et horizontal) de cette manière on parvient donc à colorier l'intégralité des routes. A noter, que pour éviter les trous de 1 pixel de large dans les contours, nous avons mis en place une tolérance

## V. Résultats et Conclusion

Les résultats concernant le prétraitement de l'image sont, de manière générale, assez satisfaisants (*voir annexe document 4*). En effet à l'exception des images 4 et 5, la totalité des routes est toujours présente après le passage du gradient. Et n'y a que les images 7 et 8 où le flou constitue encore un obstacle à la restitution des routes. Il faut par ailleurs noter que sans la présence du filtre Nagao, le filtre gradient rendait l'image 8 complètement inutilisable. Ce filtre agit donc particulièrement sur les images 6, 7, et 8 qui contiennent toutes trois bon nombre de pixels parasites.

En dépit de ces résultats la détection des débuts de routes pose problème, particulièrement pour les images 7 et 8 dont les contours sont clairsemés après traitement. En effet, si l'on réduit trop la longueur des chemins que l'on recherche quasiment tous les amas de pixels sont susceptibles d'être des routes. Et le fait que les contours soient clairsemés nous empêche d'utiliser des chemins de taille suffisante. Cela rend donc le coloriage de la route impossible.

Un des autres problèmes qui subsiste toujours est de pouvoir prendre en compte des routes dont la largeur n'est que d'un pixel, puisque nous cherchons à en colorier l'intérieur. Cela pose notamment problème sur les images 3, 4, et 5.

En revanche les résultats obtenus avec les images 1, 2, et 6 sont plus qu'encourageants puisque la totalité des routes est restituée. En effet lorsque l'on observe ces 3 images on peut remarquer que les contours des routes sont fortement marqués initialement et après passage des filtres. Dès lors notre programme n'a plus aucun problème à colorier les routes.

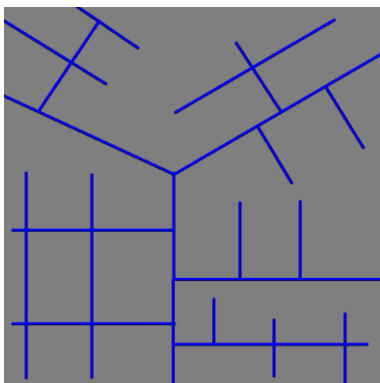


Image 1

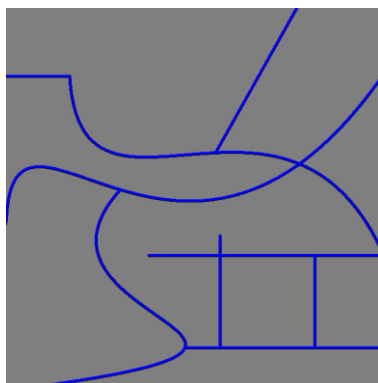


Image 2

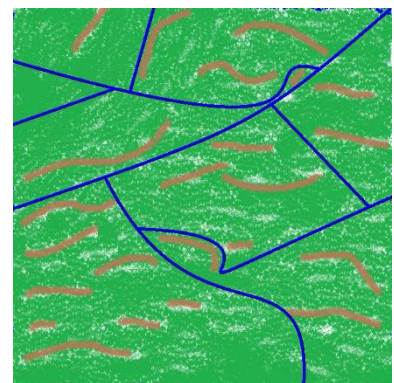


Image 6

Il est aussi à noter que des légendes, cadres, ou noms de rues font aussi obstacle à la détection du réseau routier. En effet pour la plupart des images issues de google maps (bibliothèque surplus), la coloration des routes est arrêtée par un nom de rue, qui est reconnu comme un contour par notre algorithme (*voir annexe document 5*).

# VI. Annexe

## Document 1

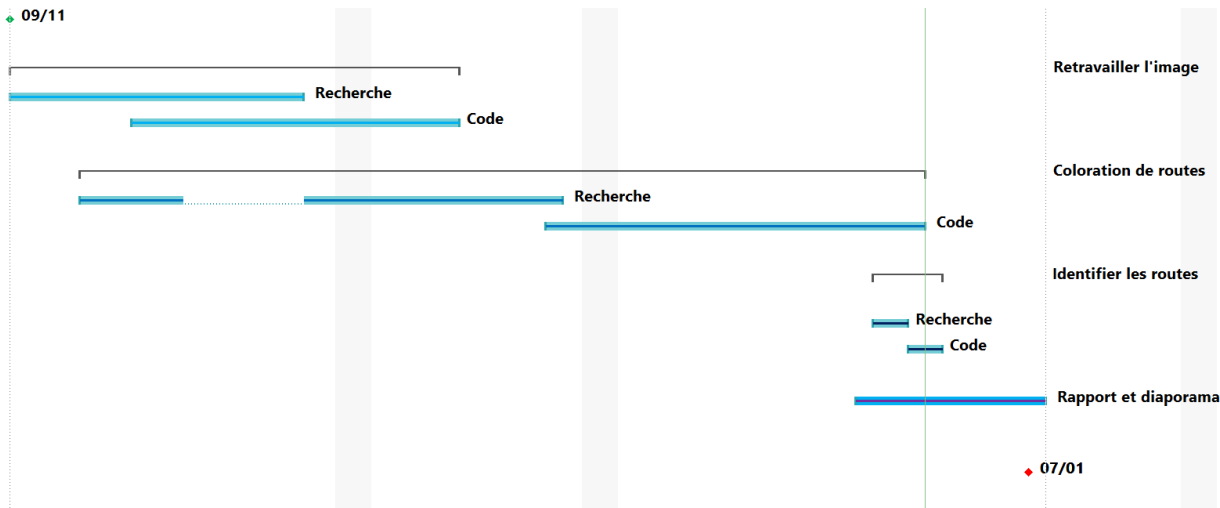


Diagramme de Gantt

## Document 2

Benoît    Guillaume    Thibault

### Retravailler l'image

Recherche	2	7	6	15
Code	12	8	8	28

### Coloration des routes

Recherche	4	9	5	18
Code	13	7	4	24

### Identifier les routes

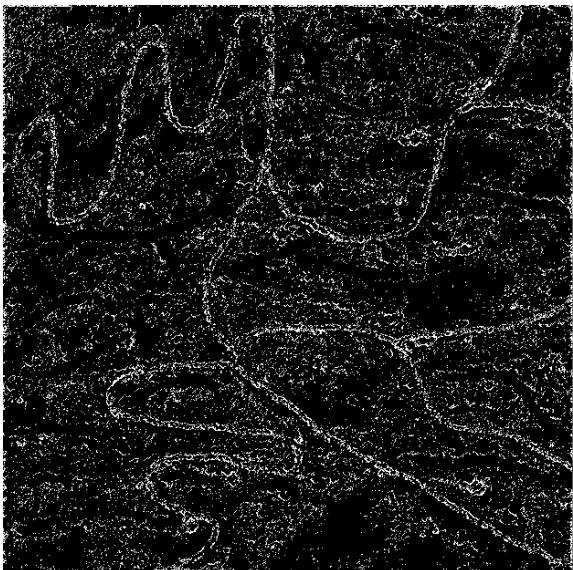
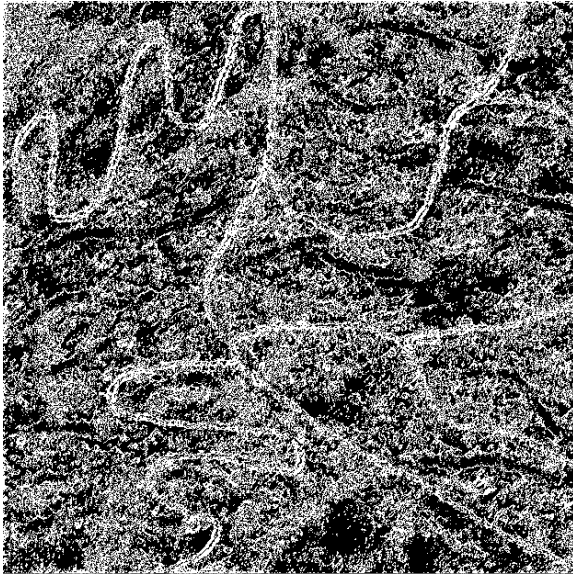
Recherche	3	8	4	15
Code	22	6	7	35

### Rapport

	4	6	22	32
	60	51	56	167

Tableau de répartition des heures

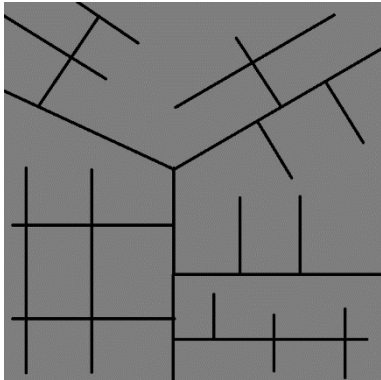
### Document 3



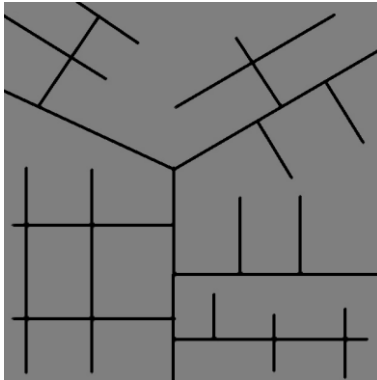
Gradients de l'image 8 avec des seuils respectifs de 10, 20, 30, et 40

## Document 4

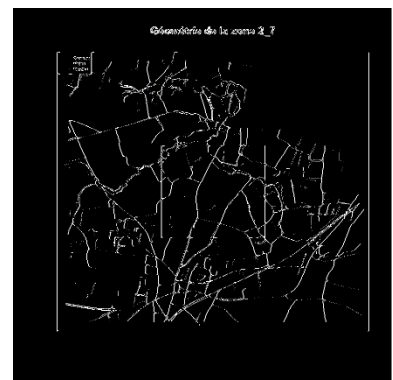
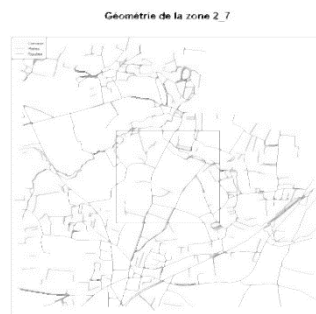
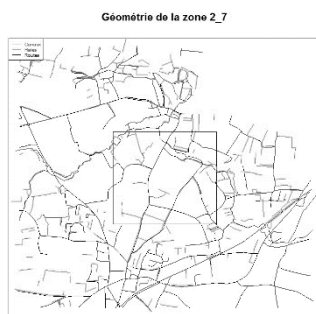
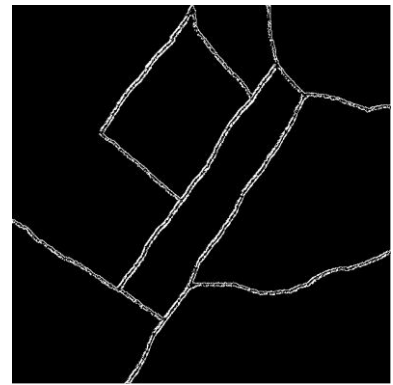
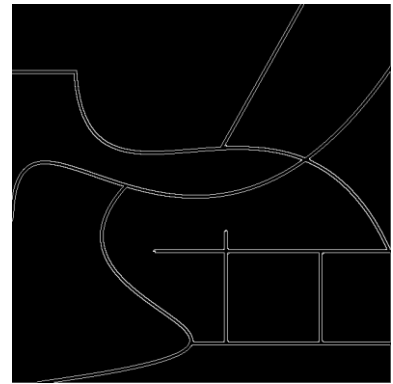
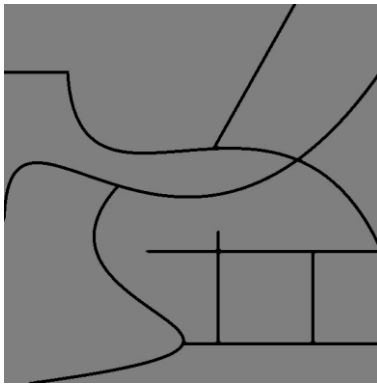
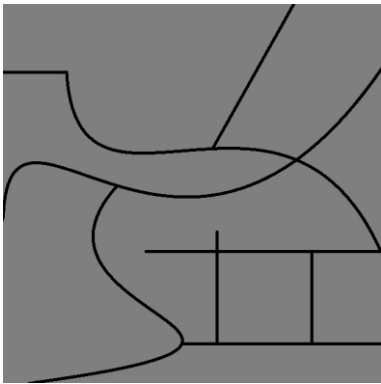
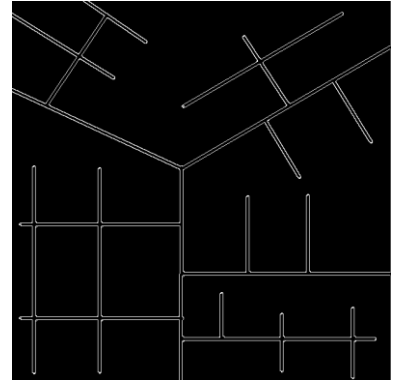
Niveau de gris



Nagao

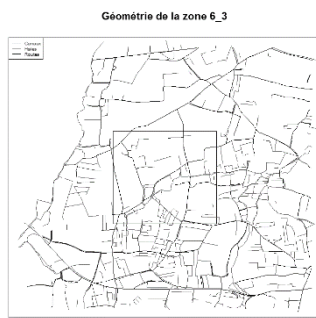


Gradient

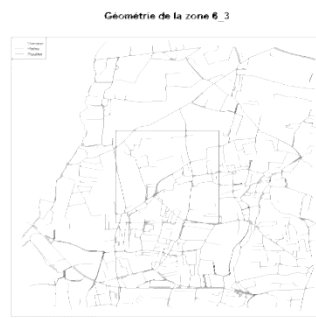




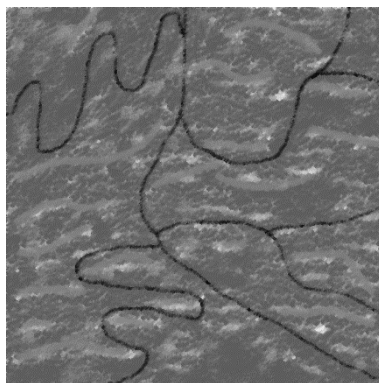
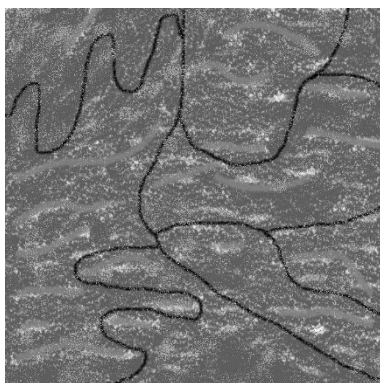
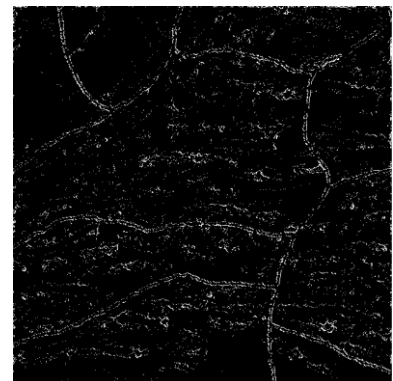
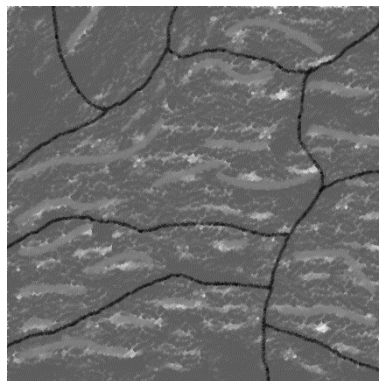
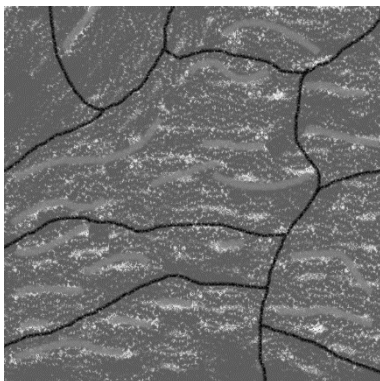
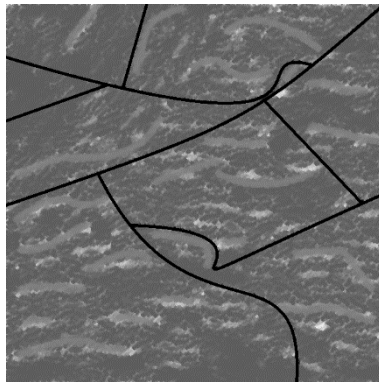
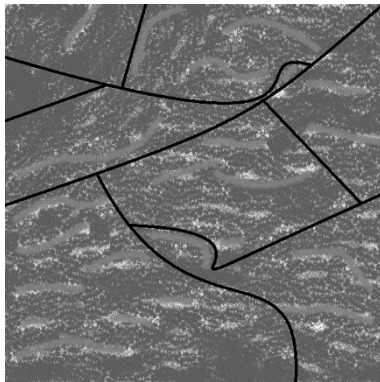
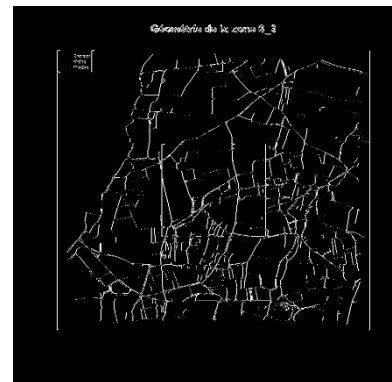
## Niveau de gris



## Nagao



## Gradient



## Document 5



Image extraite de google maps

## VII. Bibliographie

ET bit à bit :

<http://www.bien-programmer.fr/bits.htm>

Niveau de gris :

[https://fr.wikipedia.org/wiki/Niveau\\_de\\_gris](https://fr.wikipedia.org/wiki/Niveau_de_gris)

Filtre gradient et similaires :

[http://bnazarian.free.fr/MyUploads/IN\\_GBM\\_04\\_CONTOURS.PDF](http://bnazarian.free.fr/MyUploads/IN_GBM_04_CONTOURS.PDF)

[https://fr.wikipedia.org/wiki/Filtre\\_de\\_Prewitt](https://fr.wikipedia.org/wiki/Filtre_de_Prewitt)

[https://fr.wikipedia.org/wiki/Filtre\\_de\\_Sobel](https://fr.wikipedia.org/wiki/Filtre_de_Sobel)

Filtre de Nagao :

<http://www.tsi.telecom->

[paristech.fr/pages/enseignement/ressources/beti/Tomita/tests.htm](http://www.tsi.telecom-paristech.fr/pages/enseignement/ressources/beti/Tomita/tests.htm)

[http://ensiwiki.ensimag.fr/images/9/9a/Janviera\\_rapport.pdf](http://ensiwiki.ensimag.fr/images/9/9a/Janviera_rapport.pdf)