Université de Montpellier, Licence 2 Informatique

Projet TER

Les Villes

Auteurs :
M. Quentin André
M. Yann Caminade
M. Baptiste Darnala
M. Eliott Duverger

Encadrants :
Pr. Phillipe Janssen

3 mai 2017

Introduction

Notre projet consiste en une analyse, décomposition et reconstruction d'une image, plus précisément d'une carte de ville.

Le sujet s'inspire du travail d'une artiste, Armelle Caron, actuellement en résidence à Montpellier. Ce travail intitulé "*Les Villes Rangées*" [1], consiste à réarranger les quartiers d'une ville de façon "esthétique".
Notre travail consiste à les réarranger de manière compacte.

FIGURE 1 – Montpellier réarrangé par Armelle Caron

On cherche donc à isoler les "îlots", à les trier par taille, puis à les replacer sur un fond blanc de manière à recréer une sorte de nouvelle ville plus organisée.

Pour ce faire, nous avons divisé le projet en 2 parties. Premièrement, nous avons travaillé tous ensemble sur l'analyse de l'image, la séparation des îlots et leur tri. Nous avons ensuite chacun choisi une manière différente de réorganiser les îlots que nous avons codées séparément, en mettant toutefois en commun notre travail autant que cela nous a été possible.

Nous avons développé ce projet en C++ et utilisé la bibliothèque EasyBMP afin de manipuler aisément des images au format BMP.

Table des matières

1 Généralités

L'objectif de ce projet sera, comme nous l'avons brièvement présenté lors de l'introduction, de traiter une carte de ville de telle manière à réorganiser les îlots en fonction de leur taille.
Le problème à résoudre est donc double ; d'une part nous devons parvenir à détecter et isoler les îlots de la carte d'origine, et d'autre part nous devons les ranger sur une nouvelle image.

Pour aborder ce sujet, nous nous sommes servis de la bibliothèque EasyBMP. La classe BMP qu'elle offre propose un certain nombre de méthodes permettant la lecture et l'édition d'images. Nous nous sommes servis de cette classe non seulement pour lire l'image d'origine, mais aussi pour stocker les îlots sous la forme d'une liste. On assimile ainsi chaque îlot à un rectangle de la taille de l'image correspondante dans la liste. Cela facilite ainsi grandement la réorganisation des îlots. On se sert à nouveau de cette bibliothèque lorsque l'on construit l'image résultante et qu'on l'enregistre dans un fichier.

À propos de la structure de l'image elle-même, la couleur des pixels est codée en RGB, c'est-à-dire qu'à chaque pixel correspondent 3 valeurs numériques : une pour le rouge, une pour le vert, une pour le bleu. Elles sont toutes 3 comprises entre 0 et 255. Si elles sont toutes 3 égales à 255, le pixel est blanc. Si elles sont égales à 0, le pixel est noir. Grâce à la classe BMP, on peut accéder et modifier individuellement chacune de ces valeurs pour chaque pixel.

2 Détection des îlots

Pour commencer, on doit définir précisément ce qui constitue un îlot. On peut essayer d'assimiler chaque groupe de pixels colorés adjaçents à un îlot, cependant les images utilisées présentent des nuances entre le blanc et la couleur des bâtiments ; il faut ainsi décider dans quelles conditions un pixel sera considéré comme coloré, et dans quelles conditions il sera considéré comme blanc.

Pour déterminer si les pixels sont plus ou moins foncés, on va ainsi faire une moyenne des 3 valeurs RGB. On va ainsi passer l'image en niveaux de gris.
Attention cependant : l'œil humain ne perçoit pas les 3 couleurs codées par les valeurs RGB avec la même intensité. Une lumière verte apparaît ainsi plus claire qu'une lumière rouge, et encore plus qu'une lumière bleue [2].
Pour obtenir des nuances de gris correspondant véritablement à l'image d'origine, on va donc effectuer une moyenne pondérée selon la formule suivante : 0.30 x R + 0.59 x G + 0.11 x B. À noter qu'il existe plusieurs normes et que les coefficients que nous avons utilisés ne sont qu'une solution possible.

On va maintenant pouvoir déterminer quels pixels font partie des îlots. Pour cela on choisit un seuil. Tout pixel dont la valeur en niveau de gris est supérieure à ce seuil fera partie d'un îlot ; on le colore en noir. Sinon, on le colore en blanc. Le choix du seuil est

arbitraire et varie en fonction de l'image. Nous avons choisi 180 comme valeur par défaut, car elle nous permettait d'obtenir de bons résultats sur la plupart des images, mais il faudra parfois en prendre une autre lorsque l'image de base est trop claire ou trop foncée.

On se retrouve donc avec une image sur laquelle les îlots sont clairement délimités : tout pixel noir fait partie d'un îlot, tout pixel blanc fait partie de la séparation entre les îlots.

Il nous faut maintenant savoir différencier les îlots entre eux. Pour cela, on leur associe un numéro à chacun. Pour savoir où un îlot s'arrête et où le suivant commence, on crée donc une matrice d'entier de la taille de l'image. Chaque case correspond alors à un pixel. Si le pixel est blanc, la case contient un 0. Sinon, elle contient le numéro de l'îlot auquel appartient le pixel. Le remplissage de cette matrice est fait de manière récursive (fonction `Locate`).

On peut maintenant séparer les îlots dans des variables BMP séparées. Pour ce faire, on regarde, parmi toutes les cases de la matrice contenant le numéro de l'îlot, la plus à droite, la plus à gauche, la plus haute et la plus basse. On a ainsi exactement délimité notre îlot. On crée donc une image de la bonne taille. Dans l'espace que l'on vient de définir, chaque case contenant le numéro de l'îlot correspond à un pixel noir sur cette nouvelle image, et toute autre case correspond à un pixel blanc. Les îlots sont donc isolés.

On peut ainsi créer la liste d'îlots mentionnée plus haut, que l'on passera en paramètre à nos algorithmes de rangement détaillés dans la partie suivante.

3 Algorithmes de rangement (Bin packing)

Le problème du *Bin Packing* est le suivant : On dispose d'un groupe d'objets rectangulaires (ici, les images contenant les îlots) de taille variable que l'on souhaite placer dans une ou plusieurs boîtes, elles aussi rectangulaires (les côtés des rectangles sont parallèles aux côtés des boîtes). Comment doit-on organiser les rectangles pour occuper le moins de boîtes possibles, et pour laisser le moins d'espace libre possible entre les rectangles ? Il s'agit donc d'un problème d'optimisation d'espace.
Nous allons ici aborder plusieurs solutions possibles [3].

Tous les algorithmes que nous allons aborder ici fonctionnent de manière plutôt similaire. On classe les îlots par hauteur décroissante puis on les organise sur des lignes ou "tranches" du haut vers le bas et de la gauche vers la droite en alignant les îlots sur le haut de la tranche. Lors du 4ème tri uniquement, on ajoutera en plus des îlots dans l'espace restant en bas à droite de la tranche.

3.1 Principe des différents algorithmes

3.1.1 NFDH

Pour l'algorithme de Bin-Packing dit **NFDH** (*Next Fit Decreasing Height*), on s'arrête de remplir la tranche courante dès que l'on trouve un îlot qui ne rentre pas dans l'espace

restant. Lorsque cela se produit, on créé une nouvelle ligne ayant la hauteur de l'îlot courant (pour rappel, tous les îlots qui suivront auront une hauteur inférieure à celui-ci). On ne place des îlots que sur la ligne courante et on ignore les précédentes.

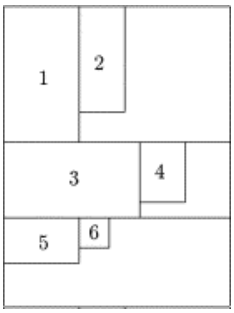


FIGURE 2 – Représentation simplifiée d'un exemple de tri NFDH

Cet algorithme à l'avantage d'être simple à implémenter par rapport aux autres puisqu'il suffit, à chaque placement d'îlot, de vérifier s'il peut rentrer sur la ligne courante puis d'agir en conséquence. Malheureusement il optimise très mal l'espace occupé, en laissant non seulement de l'espace libre en dessous des îlots, mais aussi potentiellement une grande quantité d'espace vide sur la droite de chaque ligne.

3.1.2 FFDH

Pour l'algorithme de Bin-Packing dit **FFDH** (*First Fit Decreasing Height*), on continue de placer des îlots de plus en plus petits de gauche à droite sur des "tranches". Mais à la différence de l'algorithme précédent, on repart de la première tranche pour chaque îlot et on le place sur la première tranche où il rentre. Si l'îlot ne rentre sur aucune ligne, on en créé une nouvelle.

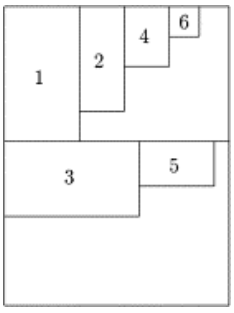


FIGURE 3 – Représentation simplifiée d'un exemple de tri FFDH

Cet algorithme reste assez basique dans l'idée mais il est plus complexe, car on doit passer toutes les tranches en revue à chaque placement, et est donc plus gourmand que le tri précédent.

3.1.3 BFDH

Pour l'algorithme de Bin-Packing dit **BFDH** (*Best Fit Decreasing Height*), on procède de la même manière que pour l'algorithme précédent. Cette fois, cependant, au lieu de placer l'îlot sur la première tranche où l'espace restant est suffisant, on va chercher parmi toutes les tranches sur lesquelles il rentre celle où l'espace restant est minimal.

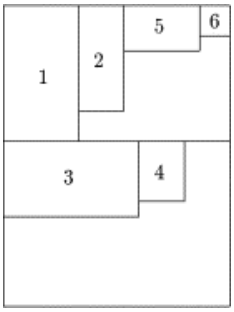


FIGURE 4 – Représentation simplifiée d'un exemple de tri BFDH

Ici par exemple sur la Figure 4, après avoir placé l'îlot 3, on aurait pu choisir de placer l'îlot 4 sur la tranche du haut. Cependant, l'espace restant sur la tranche du bas étant moindre, c'est là que l'on va choisir de le placer.

Cet algorithme présente peu de différences avec le précédent et devrait donner un résultat assez similaire.

3.1.4 FC

Pour l'algorithme de Bin-Packing dit **FC** (*Floor-Ceiling*), on va ranger les îlots dans un étage où l'on va placer les plus grands îlots sur le «plafond» et les plus petits sur le «sol» de l'étage.

On commence par essayer de ranger l'îlot sur la première tranche.

Lorsque l'on tente de placer un îlot sur une tranche, on vérifie que la largeur de l'îlot est inférieure à la largeur de l'espace restant au plafond pour savoir si on peut l'y placer. Si ce n'est pas le cas, on tente de le placer sur le sol.
Pour cela, il faut effectuer 2 vérifications :

- On vérifie que la largeur de l'îlot est inférieure à la largeur de l'espace restant au sol.
- On vérifie si la pièce que l'on veut placer ne va pas chevaucher une pièce du plafond. Pour cela, on va chercher la pièce la plus grande parmi celles situées au dessus de la pièce à placer. Une fois cela fait, on vérifie que la somme des hauteurs de ces 2 pièces (celle du plafond et celle que l'on veut placer) est inférieure à la hauteur de l'étage.

Si ces conditions sont vérifiées alors on place la pièce, sinon on recommence cette démarche à l'étage suivant s'il existe, sinon on créé un nouvel étage de la taille de la nouvelle pièce à placer.

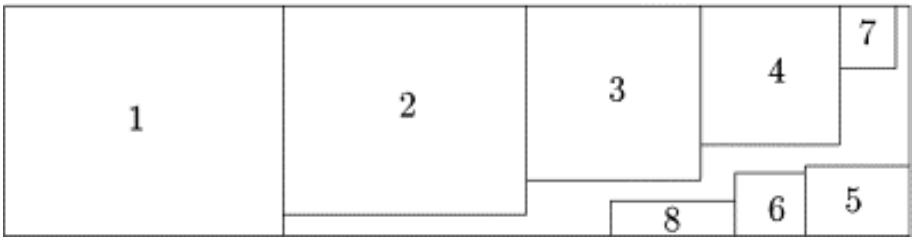


FIGURE 5 – Représentation simplifiée d'un exemple de tri FC

Cet algorithme est très différent des autres et est plus complexe à implémenter, mais il a l'avantage de très bien optimiser l'espace disponible.

3.2 Améliorations apportées

Lors de l'implémentation, plutôt que de simplement placer les îlots côte-à-côte, nous avons décidé de tenter de les rapprocher le plus possible. Cela représente en effet un gain d'espace non négligeable. Pour cela, on se sert de la fonction `calculDecalage`, qui prend en argument l'îlot de gauche et l'îlot de droite et qui renvoie le nombre de pixels dont on peut rapprocher les deux îlots sans qu'ils ne se touchent. Pour ce faire, on regarde chacun des îlots ligne par ligne. Pour l'îlot de gauche, on part du bord droit et on compte les pixels un par un vers la gauche. Pour chaque pixel, on vérifie :

— Le pixel courant est-il noir ?
— Le pixel au-dessus du pixel courant est-il noir ?
— Le pixel en-dessous du pixel courant est-il noir ?

Si l'une des 3 conditions est vérifiée, on s'arrête. Sinon, on continue de compter.
Pour l'îlot de droite, on fait la même chose en partant de la gauche et en allant vers la droite.
À chaque ligne, on additione la valeur ainsi obtenue sur l'îlot de gauche et celle obtenue sur l'îlot de droite. Parmi la liste de valeurs ainsi obtenue, on retient la plus petite.

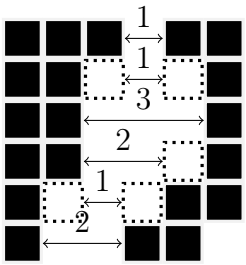


FIGURE 6 – Représentation simplifiée d'un exemple de calcul de décalage

Par exemple ici sur la Figure 6, chaque carré représente un pixel. La valeur minimale obtenue est 1 : cela signifie que l'on peut donc les rapprocher d'exactement 1 pixel.
Cette méthode n'est pas parfaite, mais elle a l'avantage d'être rapide.

De plus, nous avons essayé à chaque placement d'îlot de retourner l'îlot à 90 degrés et de vérifier si le décalage calculé était ainsi plus grand. Pour cela, on stocke le décalage avec l'îlot original dans une variable *decal1*, et le décalage avec l'îlot retourné dans une variable *decal2*. On compare enfin les deux et on retient le plus grand. Si *decal2* est plus grand, on remplace l'îlot par son inverse dans la liste *Ilots*. Sinon, on garde l'îlot d'origine. Le retournement est géré par la fonction `retourner`, qui prend en argument l'îlot d'origine et l'image sur laquelle elle doit copier le résultat.

Nous avions au départ choisi de générer l'image îlot par îlot. Nous créions donc une image blanche, et y ajoutions les îlots au fur et à mesure que nous choisissions où les placer. Lorsqu'une nouvelle tranche était créée, nous augmentions la taille de l'image. Le problème de cette méthode est que lorsque la fonction permettant de changer la taille d'une image est utilisée, elle efface le contenu de l'image. Pour y remédier, nous avons d'abord dû copier l'image sur une image annexe, agrandir l'image d'origine, puis y recopier le contenu de l'image annexe.
Cela est problématique car la copie d'une image est faite pixel par pixel. On y perdait donc en temps de calcul.

Nous avons donc dû changer d'approche. Plutôt que de générer l'image un îlot à la fois, nous avons choisi de d'abord calculer la taille de l'image et l'ordre de placement des îlots, puis de placer tous les îlots en une fois dans l'ordre précédemment établi. Ainsi on évite d'avoir à changer la taille de l'image finale en cours de route.

N.B. : À cause d'un dysfonctionnement avec l'opérateur `=` (affectation), il nous a été impossible de trier la liste *Ilots* elle-même. Pour remédier à cela, nous avons créé une liste *Indice* de même longueur contenant les indices des îlots au sein de la liste *Ilots*. Nous avons ensuite trié cette seconde liste en fonction de la hauteur des îlots correspondants aux indices. C'est-à-dire qu'au sein d'une boucle `for` en fonction de i, pour appeler les îlots du plus haut au moins haut, on utilisera la syntaxe `Ilots[Indice[Count - i]]` où *Count* est le nombre d'îlots.

3.2.1 NFDH, FFDH et BFDH

Ces trois algorithmes sont implémentés, sans surprise, par les fonctions NFDH, FFDH et BFDH. Ils présentent tous trois une structure assez similaire.

C'est pour pouvoir garder en mémoire l'ordre des îlots que nous avons créé la classe Matrix décrite plus haut. Puisque les îlots sont organisés en tranches, on peut essentiellement assimiler le résultat final à une matrice d'îlots. À cette "matrice" (il s'agit en fait plutôt d'une liste de listes), on fait donc correspondre une véritable matrice contenant les indices des îlots. Ainsi, une fois qu'elle est remplie, il nous suffit de la lire en plaçant les îlots correspondant sur l'image finale. Cette méthode est bien plus efficace puisque les opérations de modification d'une matrice sont bien moins coûteuses que les opérations de modification d'une image.

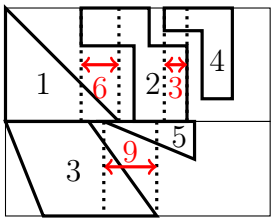


FIGURE 7 – Exemple de rangement d'îlots, décalages notés en rouge

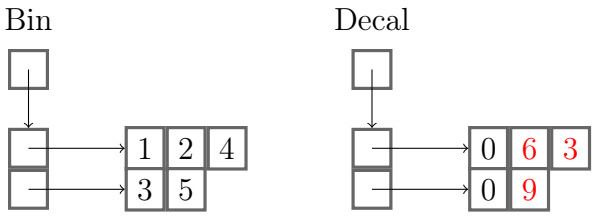


FIGURE 8 – Matrices Bin et Decal correspondant à la Figure 7

En plus de cela, il nous a aussi fallu stocker les décalages entre îlots. Cela est fait de la même manière que pour les indices. On créé une matrice *Decal* dans laquelle on stocke le décalage entre chaque îlot et son voisin (Voir Figures 7 et 8 ci-dessous).

3.2.2 FC

Pour implémenter cet algorithme en C++, nous avons choisi de créer une structure étage, possédant les attributs suivants : la largeur totale des pièces, la dernière pièce que l'on a placée au sol, la liste des pièces et leurs positions. Ces attributs existent en double : d'une part pour le plafond et d'autre part pour le sol. À ces attributs on ajoute les coordonnées du plafond et du sol sur l'image finale, ainsi qu'un pointeur vers l'étage du dessous. Cette structure ne possède, en tant que méthodes, qu'un constructeur paramétré qui reçoit les coordonnées du sol et du plafond. Grâce a cette structure, la génération de l'image finale peut se faire en lisant la structure étage par étage dans une fonction copie.

Pour le rangement en lui-même, la fonction place_pièce va recevoir un indice d'îlot et va d'abord regarder s'il peut être placé au plafond, si oui une fonction sera appelée pour modifier les différentes valeurs de la structures qui représente le plafond de l'étage. Si le placement au plafond est impossible, une autre fonction sera appelé pour vérifier si le placement au sol est possible, si oui elle modifiera les valeurs de la structure, de la même manière que pour le plafond.
Si aucun placement n'est possible à cet étage, on réessaye avec l'étage suivant. La boucle s'arrête dès qu'un placement sera réalisé.

3.3 Complexité et Efficacité

Un calcul des complexités nous révèle que les 4 algorithmes possèdent tous des complexités similaires. On note *Count* le nombre d'îlots et A l'aire de l'image d'origine :

	NFDH	FFDH	BFDH	FC
Complexité	O(A*Count)	O(A*Count^2)	O(A*Count^2)	O(A*Count^2)

Table 1 – Complexité pour chacun des algorithmes

Cela est compréhensible du fait des similarités présentées par les différents algorithmes. Pourtant, ces complexités semblent en désaccord avec les temps de calcul observés :

	Ville				
Algorithme	Montpellier	Berlin	Bordeaux	Istanbul	Paris
NFDH	0.54 s	0.25 s	1.55 s	0.33 s	0.2 s
FFDH	3.73 s	1.62 s	3.74 s	2.31 s	1.72 s
BFDH	3.89 s	1.63 s	3.85 s	2.36 s	1.76 s
FC	6.04 s	2.73 s	7.19 s	4.57 s	2.92 s

Table 2 – Temps de calcul par ville et par algorithme

En effet, l'algorithme FC prend beaucoup plus de temps que les algorithmes FFDH et BFDH. On peut supposer que cela s'explique par le fait que dans l'algorithme FC, beaucoup d'opérations sont répétées en double ; une fois pour le sol et une fois pour le plafond.

On s'intéresse maintenant à la taille des images générées par les différents algorithmes :

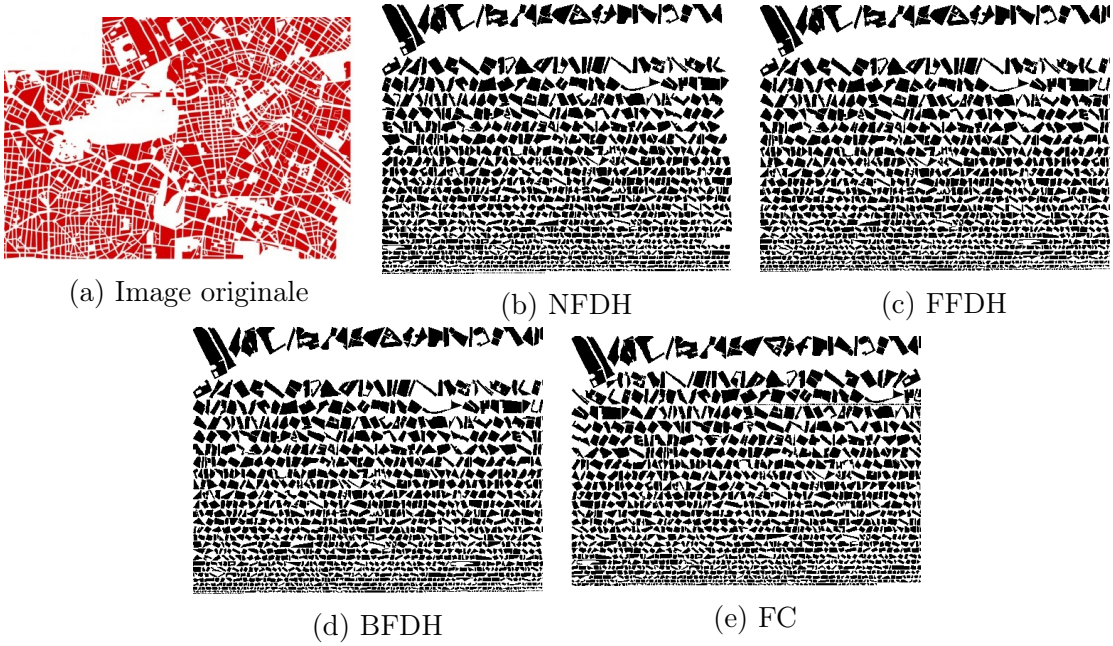
	Ville				
Algorithme	Montpellier	Berlin	Bordeaux	Istanbul	Paris
NFDH	146%	111%	178%	130%	87%
FFDH	142%	110%	172%	129%	86%
BFDH	142%	109%	172%	128%	86%
FC	140%	102%	170%	120%	81%

Table 3 – Taux de gain en hauteur par algorithme et par ville

Les valeurs de la Table 3 ci-dessus représentent le rapport entre la hauteur de la nouvelle image et la hauteur de l'image d'origine. Comme on pouvait s'y attendre, FC est l'algorithme qui offre le plus grand gain d'espace. On remarque que seules les images obtenues pour Paris sont plus petites que l'image d'origine. Cela s'explique par le fait que l'image d'origine pour Paris est en grande partie blanche (voir Section 4.1 ci-après), et que nos algorithmes auront tendance à remplir ces espaces blancs.

4 Résultats et Conclusion

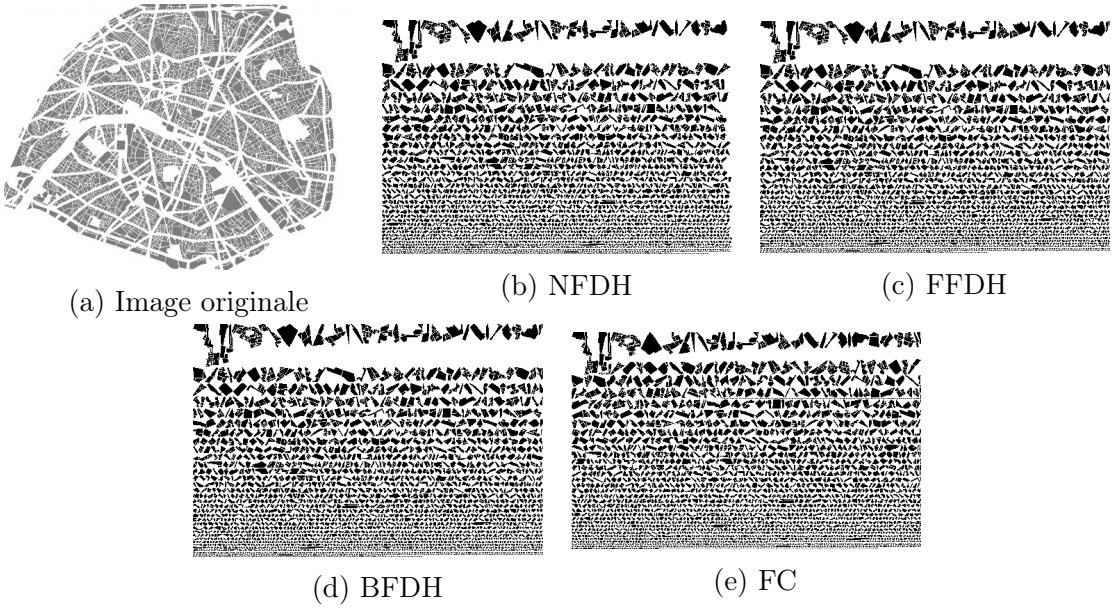
4.1 Résultats



(a) Image originale (b) NFDH (c) FFDH

(d) BFDH (e) FC

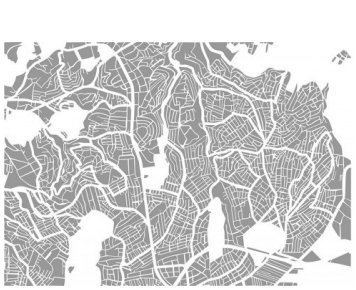
FIGURE 9 – Carte de la ville de Berlin et résultats



(a) Image originale (b) NFDH (c) FFDH

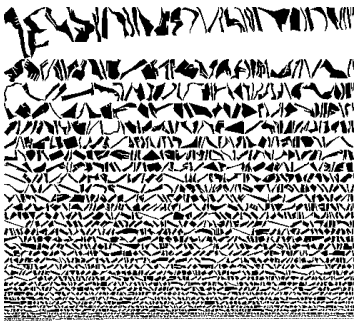
(d) BFDH (e) FC

FIGURE 10 – Carte de la ville de Paris et résultats

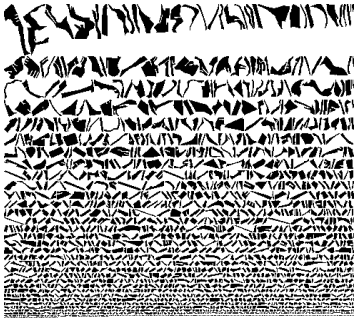


(a) Image originale

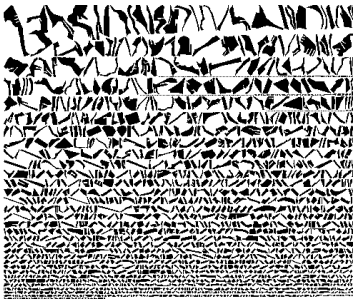
(b) NFDH



(c) FFDH



(d) BFDH



(e) FC

FIGURE 11 – Carte de la ville d'Istanbul et résultats

4.2 Perspectives d'amélioration

Pour améliorer nos algorithmes en matière d'espace occupé, une solution possible serait d'ignorer les îlots dont la taille est inférieure à un certain seuil, ou bien de tenter de les placer au milieu des îlots les plus grand, là où l'espace vide est le plus important. On éliminerait ainsi quelques tranches.

Nous aurions aussi pu essayer d'implémenter plus d'algorithmes de rangement. Nous nous sommes limités à des algorithmes rangeant les îlots par tranche, mais il en existe d'autres qui fonctionnent selon des concepts complètement différent. L'algorithme **FBL** (*Finite Bottom-Left*), par exemple, va juste chercher à placer l'îlot courant le plus en bas à gauche possible jusqu'à remplir l'image. Ou pourrait aussi, par exemple, trier les îlots par aire plutôt que par hauteur.

4.3 Conclusion

Nous avons abordé lors du développement de ce projet un certain nombre de notions avec lesquelles nous n'étions jusque là que peu familiers (Notamment le traitement d'images et le Bin Packing).
De ce fait, même si ce projet n'a pas de véritables applications en dehors du domaine esthétique, il fut tout de même intéressant à traiter.

Références

[1] Armelle Caron, *Les Villes Rangees*. Disponible sur :
http ://www.armellecaron.fr/works/les-villes-rangees/

[2] *Niveau de gris*, Wikipédia. Disponible sur :
https ://fr.wikipedia.org/wiki/Niveau_de_gris

[3] A. Lodi, S. Martello et D. Vigo, *Recent advances on two-dimensional bin packing problems*, Discrete Applied Mathematics, 2002. Disponible sur :
http ://www.sciencedirect.com/science/article/pii/S0166218X0100347X