

Projet Algorithmique, Redimensionnement d'image





Sommaire

Première partie : coccinelle et puceron	3
Deuxième partie : Seam carving	5
Qu'est-ce que le Seam carving ?	5
Notre approche du problème	6
Comment travailler avec l'image ?	7
Le problème du filtre	7
Solution retenue pour le filtre	9
Réduction vertical ou horizontal	10
Chemin de coût minimum	11
Matrice M	11
Le chemin minimum	12
Supprimer un chemin	13
Boucle de réduction	13
Les limites du seam carving	13
L'estimation du temps de calcul	13
Les commandes d'exécutions	14
Conclusion	14
Exemples d'exécution	14
Bibliographie	24

Première partie : coccinelle et puceron

Le but de cet algorithme est de calculer le chemin optimal, c'est-à-dire trouver le chemin avec le plus de valeur. Ici nous parlerons de pucerons au vu du problème.

1. Combien de pucerons la coccinelle a-t-elle mangé lors de son périple sur la grille ?
2. Quel a été ce périple ?

Dans le but de sélectionner le maximum de pucerons la coccinelle choisiras la suite suivante : (0,3)(1,2)(2,2)(3,1)(4,0)(5,0)(6,1)(7,0)

Pour un nombre total de : $9 + 15 + 4 + 36 + 46 + 89 + 72 + 3 = 279$ pucerons.

Afin de résoudre le premier problème de la coccinelle qui était de parcourir la grille composée de pucerons en ramassant un maximum d'entre eux sachant que celle-ci pouvait se déplacer d'une case à droite, à gauche ou tout droit à chaque déplacement. Nous avons donc calculé la valeur maximale parmi les cases comptant un maximum obtenue avec les pucerons déjà consommés. Puis nous avons additionnés pour chaque chemin possible la valeur du maximum et nous avons stockés ces valeurs dans une matrice de la taille de notre tableau initial.

Par la suite, afin de pouvoir connaître le chemin maximal et pouvoir afficher celui-ci nous avons supposé le problème résolu en partant de notre matrice finale. Nous avons comparés tous les derniers éléments de notre matrice finale afin de nous positionner sur la valeur la plus grande de notre dernière ligne. Ensuite, nous avons reculé en regardant les cases à une distance de 1 par rapport à notre case la plus élevée, il a suffit ensuite d'afficher la ligne et la colonne puis de réitérer le procédé. Enfin, on retourne la liste inversée afin de récupérer notre chemin dans l'ordre.

PS

C:\Users\lebon\OneDrive\Documents\GitHub\projet_algo\coccinelle
e> java Coccinelle

Tableau M[L][C] de terme général M[l][c] = m(l,c) :

279 277 278 149 145

205 276 145 140 140

204 142 124 134 125

115 71 98 114 122

64 69 51 90 107

16 28 35 50 77

5 14 24 10 11

2 4 3 9 6

La coccinelle a mangé 279 pucerons.

Elle a suivi le chemin :

(0,3) (1,2) (2,2) (3,1) (4,0) (5,0) (6,1) (7,0)

Case d'atterrissage = (0,3)

Case de l'interview = (7,0)

Deuxième partie : Seam carving

Un redimensionnement d'image intelligent.

Dans notre code source nous avons ajouté des commentaires ainsi que la javadoc pour brièvement expliquer l'utilité de chaque méthode.

Qu'est-ce que le Seam carving ?

Le seam carving est une technique qui permet de redimensionner une image sans la déformer (Figure 1) ni la rogner (Figure 2). Cela nous permet de garder uniquement les éléments importants, c'est-à-dire que l'algorithme est « conscient » des éléments qui composent l'image (Figure 3).



Figure 1



Figure 2



Figure 3



Image sans modification [1]

L'impression que l'image redimensionnée est similaire à l'original est obtenu en modifiant le moins possible les formes, en effet l'oeil humain est avant tout sensible aux formes.

Nous avons d'ailleur réalisé avec notre programme une vidéo qui permet de visualiser l'action du seam carving sur une image.

<https://youtu.be/-6wNd0pUUIQ>

Notre approche du problème

Nous nous sommes principalement appuyé sur l'étude “*Seam carving for media retargeting*” de Ariel Shamir et Shai Avidan [2].

Pour cela il nous faut calculer l'importance de chaque pixel de l'image. Nous parlerons ici « d'énergie », plus le pixel est important plus nous lui accorderons une énergie importante. L'importance des pixels est calculée par rapport à la différence de couleurs entre lui et ses voisins, plus la différence est grande, plus le changement de couleurs est important. Connaître les changements de couleurs brutaux permet au programme de savoir où se situent les bords des formes, et dans la majeure partie des cas ces bords correspondent aux éléments importants de l'image.

Ensuite une fois cette étape faite il ne nous reste plus qu'à trouver la ligne de pixel la moins importante, c'est la ligne de coût minimum. Cette ligne doit passer par un maximum de pixels de basse énergie. Une fois cette ligne trouvée elle est supprimée. En effet elle comporte le moins de pixel de haute énergie et sa suppression n'aura qu'un impact réduit sur les éléments importants de l'image.

On recommence cette dernière étape autant de fois que le nombre de pixel en largeur ou hauteur que nous souhaitons supprimer à l'image.

Cet algorithme permet de redimensionner l'image sans perdre de contenu ou déformer l'image. Mais comme nous allons le voir plus tard cette méthode a ses limites.

Créer cet algorithme nous a confronté à plusieurs problématiques. Comment détecter les bords de des formes d'une image ? Et donc par extension comment calculer l'intensité, le gradient, d'un pixel ? Comment trouver le chemin de coût minimum ? Quel chemin supprimer en premier et comment supprimer cette ligne de pixel de coût minimum ?

Dans ce rapport nous apporterons les réponses à ces questions tout en détaillant les techniques utilisées mais aussi les difficultés que nous avons rencontrées.

Comment travailler avec l'image ?

Pour travailler avec l'image nous utilisons une bibliothèque intégrée à java qui permet de lire la valeur RGB de chaque pixel. Et nous la mettons dans un tableau. Par la suite nous utiliserons ce tableau comme "image".

En tout nous utilisons 4 structures de données :

- Un tableau 2D avec les valeurs RGB de l'image.
- Un tableau 2D avec l'énergie de chaque pixel.
- Un tableau 2D M avec le chemin de coût minimum.
- Une Stack qui permet de stocker les coordonnées de nos point du chemin de coût minimum.

Nous reviendrons sur les 3 derniers plus tard dans ce rapport.

Le problème du filtre

Pour détecter les bords des formes dans nos images à traiter nous sommes passé par plusieurs étapes différentes.

Dans un premier temps nous avons imaginé un système de comparaison à 2 seuils. En vérifiant la différence de couleur entre le pixel actuel avec ses voisins direct. (Figure 4) Il est possible de détecter les différences brutales de couleur.

x-1,y-1	x,y-1	x+1,y-1
x-1,y	x,y	x+1,y
x-1,y+1	x,y+1	x+1,y+1

Figure 4

On compare l'expression $(\text{rouge} + \text{vert} + \text{bleu}) / 3$ (rouge étant la valeur RGB pour le rouge etc) de la case voisine avec la case actuelle. En fonction de la différence il y a plusieurs possibilités :

- Si la différence est inférieur à 1 on met la couleur blanche dans la case RGB (0, 0, 0)
- Si la différence est entre 1 et 3 on met la couleur grise dans la case RGB (122, 122, 122)
- Sinon on laisse du noir RGB (255, 255, 255)

Ensuite on donne une valeur à chaque couleur, ce qui compose notre matrice d'énergies. Avec cette méthode nous avons essayé divers paramètres. Notamment celui ci a la figure 5

$x-22,y-22$	$x,y-22$	$x+22,y-22$
$x-22,y$	x,y	$x+22,y$
$x-22,y+22$	$x,y+22$	$x+22,y+22$

Figure 5

Avec ce paramètre on fait la même opération qu'au dessus mais en vérifiant la 22ème case au lieu du voisin. Nous nous sommes rendu compte que la détection de bords était bien plus précise comparé à la vérification du voisin seulement. Mais cette méthode est globalement de mauvaise qualité car elle ne prend pas en compte les petites objets et elle n'a pas assez de flexibilité (seulement 3 cas possibles).

Notre solution n'étant pas viable nous nous sommes penchés vers les filtres. En appliquant un filtre sur notre matrice nous calculons une valeur pour chacun des pixels, cette valeur dépend des pixels voisins.

Dans le cadre du filtre laplacien nous calculons cette valeur selon la figure 6.

Pour ce faire nous divisons par 3 la somme des valeurs R,G et B de chaque pixel utilisé. Ensuite on applique le coefficient défini dans le filtre. Pour finir on additionne toute les composantes du filtre dans la case actuelle.

0	1	0
1	-4	1
0	1	0

Figure 6

Après un premier essaie de ce filtre (Figure 7) nous l'avons abandonné comme peut en témoigner le résultat.



Figure 7

Solution retenue pour le filtre

Après de multiple test de filtre nous en avons déduit qu'il fallait regarder la différence entre les pixels opposés, comme avec le filtre de Prewitt (Figure 8). Or comme nous travaillons avec les composantes R,G et B, nous devons manipuler uniquement de INT.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * \mathbf{A}$$

Figure 8 [3]

Or pour utiliser le filtre de Prewitt nous devons affecter à notre pixel actuel la valeur $\sqrt{G_x^2 + G_y^2}$. Manipuler des puissances et des carrés en java demande d'utiliser des FLOAT ce qui n'était pas compatible avec nos valeurs RGB.

A la place nous avons tout calculé en valeur absolu ce qui nous dispense de mettre au carré puis à la racine carré.

Notre implémentation final est légèrement différente du filtre de Prewitt, il prend en compte tous les pixels sur un rayon de 2. Et de plus nous comparons (c'est à dire nous faisons la différence) en "passant" par le pixel où nous sommes actuellement. La figure 9 est une représentation graphique de ce filtre.

Indices de chaque valeurs				
x-2,y-2	x-1,y-2	x,y-2	x+1,y-2	x+2,y-2
x-2,y-1	x-1,y-1	x,y-1	x+1,y-1	x+2,y-1
x-2,y	x-1,y	x,y	x+1,y	x+2,y
x-2,y+1	x-1,y+1	x,y+1	x+1,y+1	x+2,y+1
x-2,y+2	x-1,y+2	x,y+2	x+1,y+2	x+2,y+2

Figure 9

Chaque couleur correspond à 2 valeurs que l'on compare.

Exemple : $\text{Abs}(\text{T}[x][y-1] - \text{T}[x][y+1])$

Ce filtre que nous avons imaginé nous permet d'avoir une grande précision par il prend en compte beaucoup de voisins au pixel étudié et donc d'avoir une meilleur résolution de détection de bord.

Réduction vertical ou horizontal

Dans ce programme nous donnons la possibilité à l'utilisateur de réduire son image en hauteur mais aussi en largeur. On parlera alors de chemins verticaux pour la largeur et horizontaux pour la hauteur.

Dans le rapport ci dessous nous préciserons les petits changement entre ces deux, ils sont minimes. Généralement il suffit juste de changer l'incrémentation d'indice.

Notre programme effectue d'abord le redimensionnement verticale puis horizontale.

Chemin de coût minimum

Matrice M

La première étape est de calculer la matrice M, cette matrice est calculée selon le schéma de la figure 10.

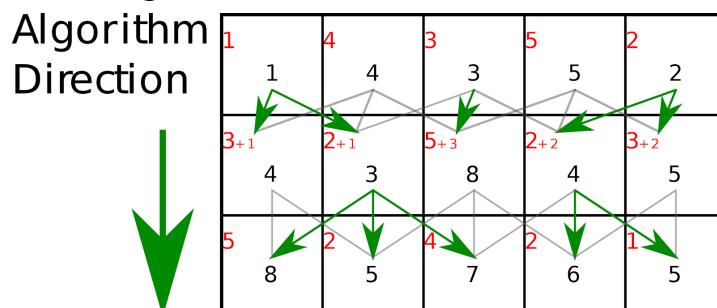


Figure 10 [4]

Cette matrice contient des chemins à bas coup mais particulièrement le chemin de coût minimum. Pour faire cela on additionne l'énergie du pixel actuel plus l'énergie de celui en bas à gauche, en bas ou en bas à droite. (Ici nous sommes dans la cadre des seams verticaux, pour les horizontaux la technique est la même seul le sens diffère). Ensuite si notre somme est la plus petite, on met le résultat dans la case avec laquelle nous avons ajouté notre pixel actuel. Sinon on ne fait rien. Une fois que l'on a traité toute la ligne on traite la suivante, mais en reprenant les valeurs assigné aux cases, puis on recommence pour faire cela sur toute les lignes.

Le chemin minimum

Identify Min.				Backtrack for Seam			
2	3	5	4	2	3	5	4
8	3	10	12	8	3	10	12
5	10	4	12	5	10	4	12
15	10	11	12	15	10	11	12

Illustration du chemin de coût minimum, extrait de “A Performance Analysis for Seam Carving Algorithm”[5]

Une fois notre matrice M créée il suffit d'y localiser les coordonnées de chaque point appartenant au chemin de coût minimum. Pour cela nous parcourons la dernière ligne (ou colonne si on est dans le cas des seams verticaux) et on sélectionne la valeur la plus petite, au hasard si il y en a plusieurs; la sélection aléatoire améliore un peu le rendu final de l'image redimensionnée. Notre fonction étant conçu en récursif nous avons utilisé un Stack (Figure 11), qui est une structure de donnée intégrée à java. En haut de la pile nous mettons le dernier point, c'est à dire le plus en bas de l'image ou le plus à droite en fonction de notre type de chemin. Le reste suit l'ordre du dernier au premier. Nous avons utilisé cette technique du stack car elle nous permet de ne pas avoir d'indice en plus à gérer. En effet il suffit juste de faire `maStack.pop()` pour récupérer votre objet en haut de la pile et le supprimer le tout sans se préoccuper de l'indice. Et de même pour en rajouter `maStack.push()` pour l'ajouter en haut.

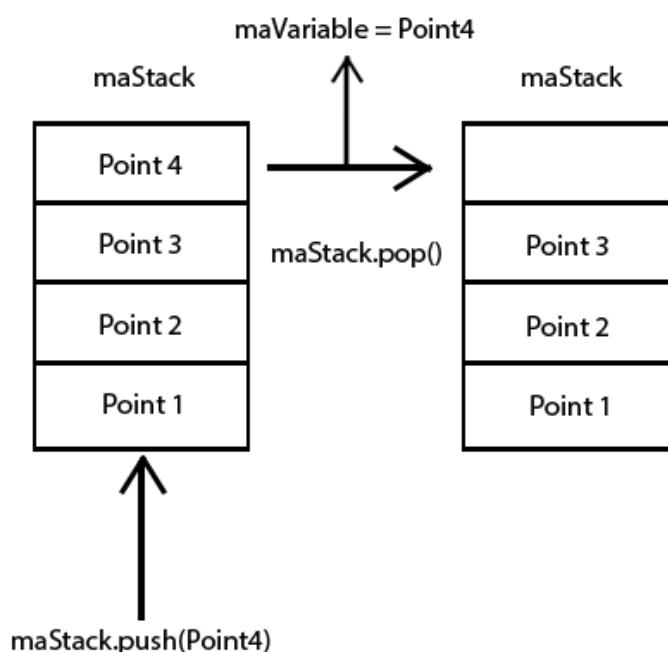


Figure 11

Supprimer un chemin

Pour retirer un chemin il nous suffit d'utiliser la stack créé précédemment. On récupère notre le point le plus haut sur la stack et on retire le pixel à sa coordonnée. Ensuite on décale chaque pixel de 1 tel que `pixel(x, y) == pixel(x+1, y)` (pour les seam horizontaux nous aurons `pixel(x, y) == pixel(x, y+1)`). On prend bien soin de s'arrêter à `x = largeur_de_limage - 2` pour bien réduire l'image de 1 pixel.

D'un point de vu plus technique pour réduire notre tableau 2D de 1 colonne ou ligne, il faut au préalable créer un tableau plus petit de moins 1 colonne. Cette opération simplifi la suppression d'une colonne.

Boucle de réduction

Une fois toute ces étapes réalisées nous avons notre image qui a été réduite de 1 pixel de largeur ou hauteur, nous réduisons aussi notre matrice qui contient les énergies de 1 case de largeur ou hauteur. Et pour finir on recommence toutes les étapes depuis le calcul de la matrice M autant de fois qu'il y a de pixel à retirer.

Les limites du seam carving

En réalisant cet algorithme nous avons rapidement été confrontés aux limites de Seams Carving. En effet, selon le type de filtre utilisé l'aperçu du rétrécissement de l'image change considérablement d'une image à l'autre. Pour une image utilisant un filtre en particulier cela ne conviendra pas forcément à une autre image d'une taille équivalente ou différente. De plus, le fait de devoir recalculer les chemins de coût plus faible à chaque redimensionnement d'image augmente le temps de calcul considérablement.

L'estimation du temps de calcul

Pour estimer le temps de calcul nous récupérons le temps de calcul nécessaire pour retirer 50 seams, puis nous le multiplions par le nombres de chemins restants. Cela nous permet d'avoir une estimations du temps restant presque fiable. En effet certains chemins sont plus rapides à traiter que d'autre, ce qui peut fausser notre estimation de quelques secondes à quelques dizaines de secondes en fonction de la taille de l'image.

Les commandes d'exécutions

Pour exécuter le programme il faut se placer dans le répertoire `src` du dossier de notre projet. Puis exécuter la commande :

```
java SeamCarving nomdufichier.sonextention y% x%
```

Où `x` représente le pourcentage de réduction en largeur et `y` en hauteur.

Une fois l'exécution du programme terminée, nous avons 2 fichiers `nomdufichier_edge.png` qui est une image représentant le premier calcul de notre algorithme de détection de bord (énergie des pixels) et `nomdufichier_resized_y_x.png` qui est notre image redimensionnée.

Conclusion

Cet algorithme est très intéressant car au delà de son aspect esthétique, il permet aussi de mettre en pratique des notions abstraite acquise en cours. Mais il a tout de même des défauts comme son temps d'exécution très élevé et ses erreurs sur certaines images (notamment les êtres humains). Ces problèmes peuvent venir autant du contenu de l'image que de ses couleurs, ses formes etc...

Ces problèmes peuvent être réglé en utilisant un filtre différent plus adapté à l'image que nous voulons traiter. De plus lorsque l'on retire plus 50% d'une composante de l'image, une forte déformation est constatée. Le degré de cette déformation dépend aussi du contenu de l'image.

Pour aller plus loin nous pourrions ajouter une détection automatique du filtre le plus adapté à utiliser pour l'image choisie. En effet "A Performance Analysis for Seam Carving Algorithm"^[5] par Zehra Karapinar Senturk et Devrim Akgün suggère que en fonction du filtre choisi les gains de performances peuvent être améliorés de 12% en plus du gain esthétique.

Exemples d'exécution

Commande écrite pour l'exécution

1^{er} image est l'original

2^{eme} image est celle avec l'énergie des pixels

3^{eme} image redimensionnée

Toutes les images ci dessous sont disponible dans le .zip du rapport.

```
java SeamCarving seamcarvingsdemo.png 0 50
```



Image d'origine



Energie de chaque pixel



Réduite de 0% et 50%

```
java SeamCarving baie_de_somme.png 20 60
```



Image d'origine



Energie de chaque pixel

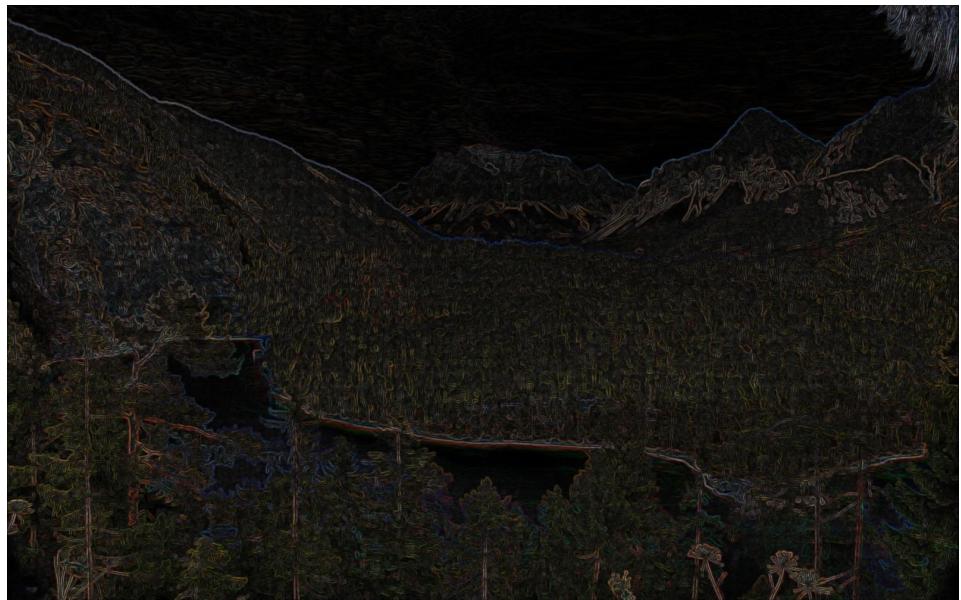


Réduite de 20% et 60%

```
java SeamCarving paysage.png 10 80
```



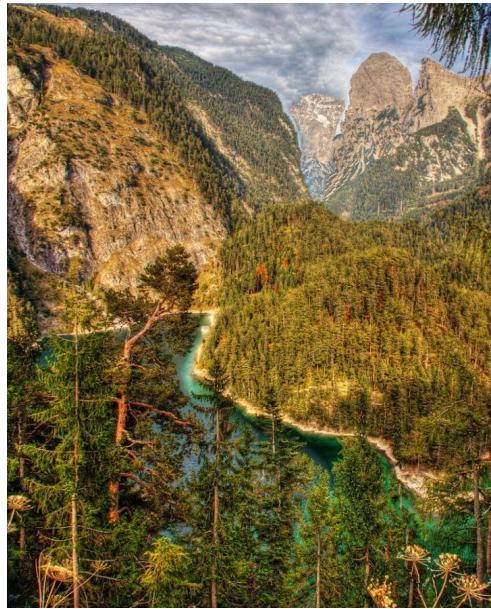
Image d'origine



Energie de chaque pixel



Réduite de 10% et 80%

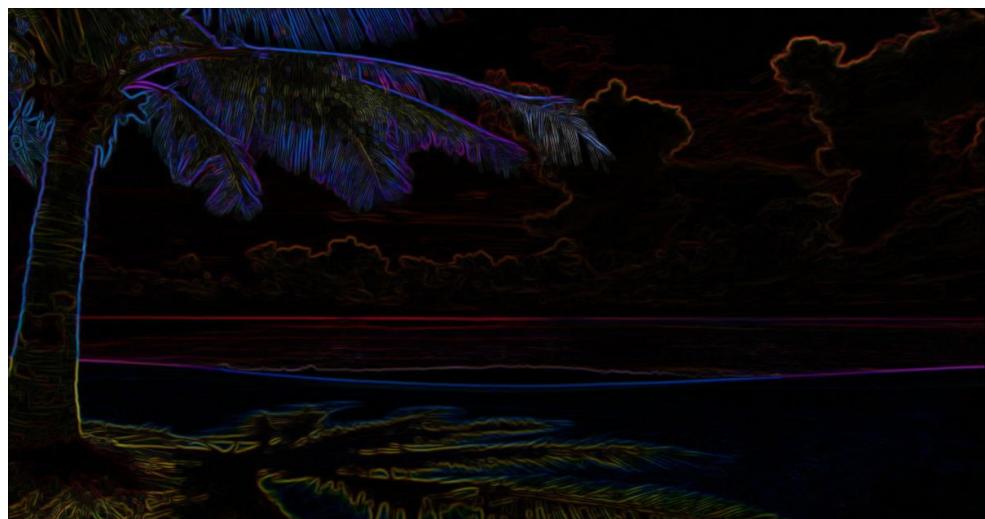


```
java SeamCarving paysage.png 10 55  
Réduite de 10% et 55%
```

```
java SeamCarving plage.png 20 30
```



Image d'origine



Energie de chaque pixel

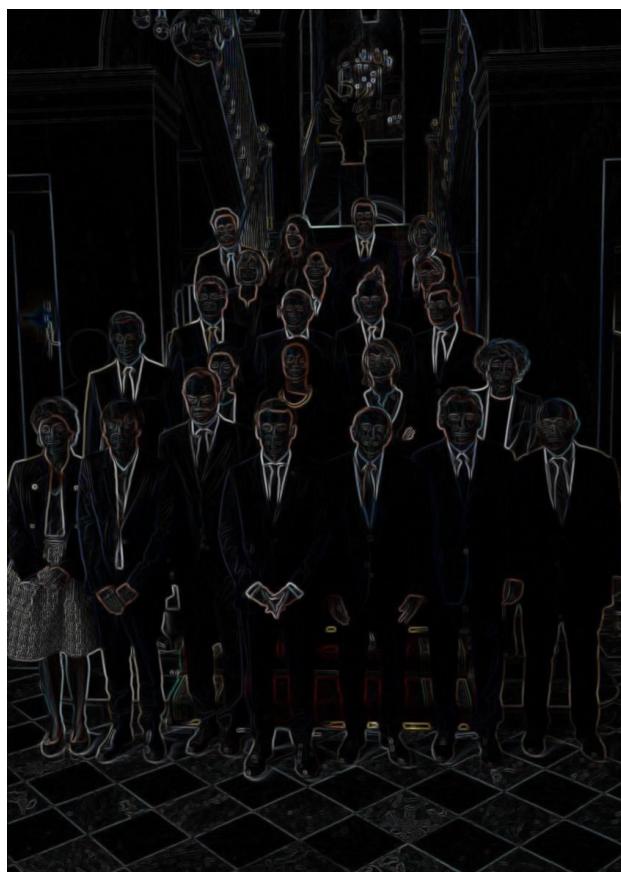


Réduite de 20% et 30%

java SeamCarving gouvernement_macron.png 40 30



Image d'origine



Energie de chaque pixel



Réduite de 40% et 30%

Bibliographie

[File:Broadway tower pls.jpg](#) [1] Image utilisée par Wikipédia pour illustrer le seam carving.

<https://dl.acm.org/doi/abs/10.1145/1435417.1435437> [2] Seam carving for media retargeting

[Filtre de Prewitt](#) [3]

[Seam carving](#) [4]

[\(PDF\) A Performance Analysis for Seam Carving Algorithm](#) [5]