UNIVERSITÉ
CAEN
NORMANDIE

Stratégies de remplissage de forme

Hugo JEAN
21602196

Steve GENDARME
21700149

Chef de projet : M. Youssef CHAHIR

Projet tutoré de M1

Master 1 Informatique

2020/2021

Table des matières

Introduction

Ce projet met en avant deux composantes du Master Informatique de Caen, en effet d'un côté nous avons une partie majeure qui concerne l'imagerie (IDM) et de l'autre nous avons une dimension de résolution de contrainte ou d'optimisation (DOP). L'imagerie aujourd'hui est l'un des secteurs les plus étendus de l'informatique moderne avec plusieurs branches elle mêmes très importantes et porteuses. On peut scinder l'imagerie en trois sous-groupes :

— La synthèse d'image, principalement de la création d'images de synthèse (3D, réalité virtuelle) ou non comme la réalité augmentée.

— Le traitement d'image, le traitement par ordinateur d'image ou de vidéo. On attend une donnée différente de celle mise en entrée, par exemple de la recolorisation ou alors de la compression d'images.

— L'analyse d'image, on cherche à extraire des informations à partir des images. C'est ici où l'on va retrouver tous les projets de deep-learning mais aussi par exemple de robotisation comme avec le robot Curiosity sur Mars.

L'idée de remplir des formes avec certaines contraintes (type de forme, taille de forme, nombre de formes restreint, etc ...) peut et doit être interprétée comme un problème d'optimisation sous contrainte, ce genre de problème peut trouver son utilité dans la logistique par exemple, comment remplir un avion ou bateau en laissant le moins d'espace vide avec un inventaire donné. Mais aussi comme vous pourrez le voir au cours de notre projet comment supprimer des objets de certaines images tout en gardant une structuration d'image cohérente.

Nous allons donc au cours de ce projet, aborder et travailler avec plusieurs de ces concepts. Tout d'abord nous utiliserons le code de Freeman qui est une technique de compression d'image et essaierons d'amener à une solution ou une partie de solution avec celui-ci. Nous aborderons ensuite la résolution par contrainte de problème et la difficulté de la modélisation de ce problème et les contraintes de calcul et de temps que cela implique. Enfin nous terminerons sur de l'inpainting, champ de l'imagerie déjà bien développé et efficace, celui-ci permettra de mettre en avant des résultats visuels et donc de rendre compte de notre travail.

1.1 Point technique

Ce projet est réalisé en Python, ci dessous toutes les librairies et modules utilisés pour ce projet.

— Numpy : Pour de la manipulation d'array

— MatPlotLib : Pour tout ce qui est affichage de graphique ou d'image

— PIL : importation d'image vers des formats utilisables pour Python (i.e. array Numpy)

— SciPy : Pour des fonctions avancées optimisées (Par exemple la labelisation des zones d'une image)

— OpenCV : Une librairie de traitement d'image très complète et rapide

— Sklearn : Ici juste utilisé pour créer des patchs dans une image

— OR-tools : Pour la résolution de contraintes

Objectifs

L'objectif original était de proposer une IA pour remplir des formes à partir d'un set de formes plus petites, comme par exemple les formes disponibles dans le célèbre jeu Tetris.

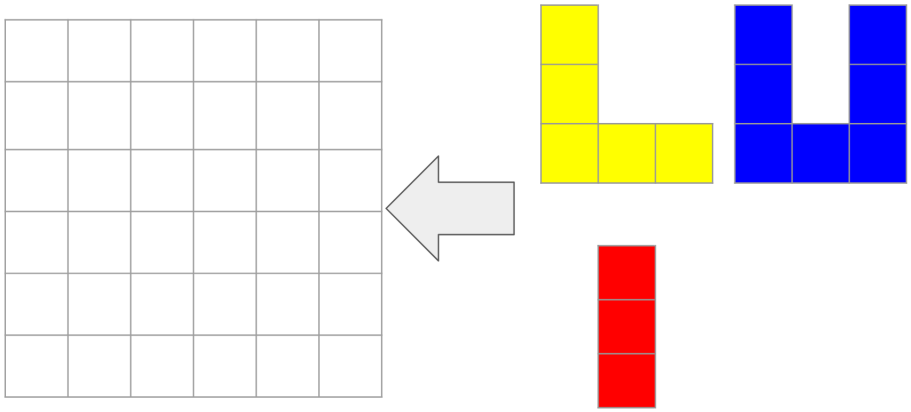

FIGURE 2-1: Problème original

Bien évidemment la forme à remplir peut être de n'importe quelle forme et pas forcément un rectangle et la même chose est vraie pour les formes, elles peuvent être quelconques. Ainsi ce problème n'est pas ré-solvable facilement. Et trouver une combinaison parfaite dans ce genre de problème n'est tout simplement pas possible dans un temps efficace, c'est pourquoi il faut se laisser une marge d'erreur, par exemple on peut définir qu'une surface remplie à 95% suffit et remplit les conditions.

Cependant, notre projet fût mis en pause pendant un peu plus de 2 mois, nous avons donc décidé avec notre professeur qu'il serait plus intéressant d'explorer plusieurs pistes de recherche pour résoudre ce problème. Et ce sans proposer d'interface graphique pour l'utilisateur, rendre la partie interactive beaucoup plus petite qu'originalement prévu, tout en gardant la partie de recherche et d'expérimentation toujours aussi importante.

Codage de Freeman

3.1 Qu'est-ce que c'est ?

L'un des moyens le plus facile pour représenter une forme dans une image binaire est d'utiliser des "Chain code". Ces dits 'Chain code' fonctionnent de la façon suivante ;

— On sélectionne un pixel de départ

— On se déplace dans son voisinage en ajoutant le code défini à la pile (On répète l'étape jusqu'au retour au pixel de départ)

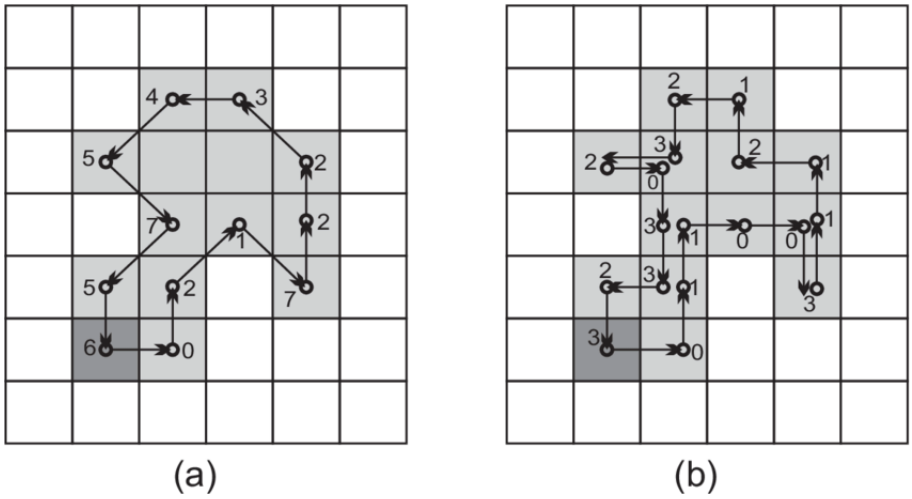

FIGURE 3-1: a : Codage de Freeman(8) b : Codage de Freeman(4)

La figure 3-1 présente le codage de Freeman créé par Freeman en 1961 [1] est souvent cité comme le plus simple des "chain code" et le plus compressé [6], en effet Freeman définit son code avec une connectivité des pixels. C'est à dire que dans un voisinage de pixel on considère le pixel central et selon que l'on soit dans un code de Freeman (4) ou (8), ses 4 voisins c'est à dire sans les diagonales ou ses 8 voisins. Ainsi on pourra coder Freeman sur 2 bits pour le code avec 4 voisins et sur 3 bits pour les 8 voisins.

La compression d'image est un des axes majeurs de la recherche en image, on peut ainsi trouver beaucoup d'articles sur le code de Freeman et ses utilisations. Par exemple une thèse très intéressante qui parle des propriétés géométriques du code ainsi que mathématiques *"Étude de paramètres géométriques à partir du code de Freeman"* de Xavier TROUILLOT [2].

Parce que ce genre d'algorithme de compression ne marche que sur des images binaires, il est souvent utilisé dans le stockage des documents provenant de scanners. On peut citer les formats *JBIG2* ou *DjVu*, utilisant le "Chain code" dans leurs algorithmes de compression.

3.2 Notre solution

Le code de Freeman peut aussi être vu comme une suite de changements d'angle. Considérons le code suivant :

$$0,0,0,1,1 \tag{3.1}$$

On réalise trois sauts à droite puis deux en haut, mais on peut aussi dire que l'on a pris un angle de 90° au bout de trois pixels. On peut donc décrire une forme, comme par exemple un L avec le code précédent, c'est à dire décrire la longue bar du L avec les 0 ici et la plus petite avec les 1.

Il nous est donc venu à l'idée d'utiliser les expressions régulières pour retrouver ces patterns à l'intérieur du code. Si l'on reprend le code (3.1) qui représente donc un L, on peut le modifier pour qu'il soit modulable à n'importe quelle rotation du L. On va donc créer l'expression régulière suivante :

$$(aaa)+(bb)+,\ avec\ a,b \in \{0,1,2,3\}\ et\ a \neq b \tag{3.2}$$

On obtient ainsi un L avec une barre longue de taille d'au moins 4 pixels et la barre courte d'au moins 3. Grâce à cela on peut donc positionner des formes dans le code et aussi récupérer le nombre de chaque type de forme utilisée. On peut aussi citer par exemple l'expression régulière pour un U :

$$(aaa)+(bb)+(ccc)+,\ avec\ a,b,c \in \{0,1,2,3\}\ ,\ a \neq b\ ,\ |a-c| = 2 \tag{3.3}$$

Le retour de notre programme est le suivant :

$$\{code\}Position : [Pixel]\ jusqu'\ [Pixel] \tag{3.4}$$

3.3 Est-elle satisfaisante ?

Cette solution fut l'une des premières abordées et sans doute celle la moins intéressante à explorer. En effet, la découverte des "Chain code" nous a intéressé mais comme le prouve le nombre de thèses et travaux de recherche réalisés sur ce sujet, il est paradoxalement assez facile à appréhender mais travailler dessus nécessite un réel investissement et de réelles recherches, or de par la nature de notre projet nous n'étions pas capable de consacrer tout notre temps sur une seule solution. Ainsi notre solution ne reste qu'une esquisse de ce que nous pourrions en faire. On peut penser à par exemple factoriser les formes à l'intérieur du code, c'est à dire regrouper des sous-formes pour former une plus grande comme par exemple un L suivi d'un I peut se factoriser en U. Or cela implique encore une fois beaucoup travail et nous n'avions pas forcément le temps pour ça. On peut aussi par exemple souligner que cette solution peut permettre de donner le nombre de formes et ainsi permettre à la résolution par contrainte d'être beaucoup plus simple vu que l'on connaît le nombre de formes.

Résolution de contraintes

4.1 Qu'est-ce que c'est ?

La résolution de contraintes, aussi appelée programmation linéaire, est un paradigme de programmation permettant de maximiser ou de minimiser la solution à un problème en le modélisant avec des variables et des contraintes définies par des fonctions linéaires.

Le projet nécessite de maximiser le nombre de formes à utiliser pour remplir une surface. Nous avons donc tenté de résoudre ce problème en le modélisant en problème de programmation linéaire.

4.2 Création du modèle

Pour chaque forme, nous créons une nouvelle variable pour chaque position qu'elle peut prendre dans la surface. Pour que les formes ne se superposent pas, nous créons une contrainte limitant la présence d'une seule forme par pixel. Dans le pire cas possible, cela nous donnera pour une image de x pixels sur y pixels et n formes un modèle avec :

— $x \times y \times s$ variables.

— $x \times y$ contraintes.

Pour éviter les cas où il n'est pas possible de remplir la totalité de la surface avec les formes disponibles, nous pouvons assouplir la contrainte en autorisant les pixels à ne contenir aucune forme, cependant, cela ne marchera que pour la maximisation, car cela nous donnerait un remplissage de zéro forme en minimisation. Nous obtenons donc le modèle suivant :

Soit P l'ensemble des pixels et F l'ensemble des formes :

$$max \sum_{(x,y)\in P} \sum_{f\in F} p_{f,x,y} \qquad \text{(Maximiser le nombre de forme)}$$

$$\forall (x,y) \in P, \sum_{f\in F} p_{f,x,y} = 1 \qquad \text{(Une seul forme par pixel)}$$

$$\forall (x,y) \in P, \forall f \in F, \ 0 \leq p_{f,x,y} \leq 1 \qquad \text{(Variable booléenne)}$$

4.3 Réalisation

Nous avons utilisé la librairie OR-tools, qui est une collection d'outils permettant de faire de la résolution de contraintes.

Nous allons maintenant essayer de remplir une image de 10 pixels sur 10 pixels en utilisant 5 formes (Figure 4-1). Pour chaque forme, nous créons une variable pour chaque emplacement où nous pouvons la placer. Nous stockerons cette variable dans un dictionnaire indexé par la position pour tous les pixels où cette forme sera présente, cela nous permettra ensuite de créer une contrainte pour chaque pixel de l'image.

FIGURE 4-1: Caption

En résolvant ce modèle, nous obtenons une liste des formes à appliquer. Dans notre exemple, nous obtenons 23 formes en minimisant et 33 formes en maximisant (Figure 4-2).

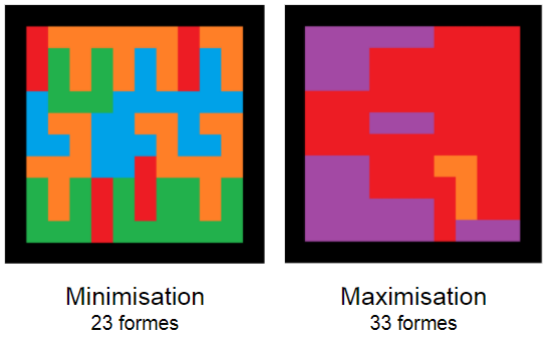

FIGURE 4-2: Caption

4.4 Problème rencontré

Cette méthode nous donne de bons résultats, cependant, la résolution de contrainte est un problème de la classe NP Difficile, le temps de calcul devient donc trop important avec l'augmentation du nombre de pixels et le nombre de formes. Nous avons trouvé plusieurs solutions pour remédier à ce problème. Nous pouvons par exemple découper l'image en plusieurs patchs que nous traiterons séparément. Nous pouvons aussi spécifier au solveur un temps maximal de recherche, une fois ce temps dépassé, il retournera la meilleure solution qu'il aura trouvée. Ces deux solutions réduisent énormément le temps de calcul, mais elles ne nous donnent qu'une approximation de la solution optimale (Voir figure 4-3).

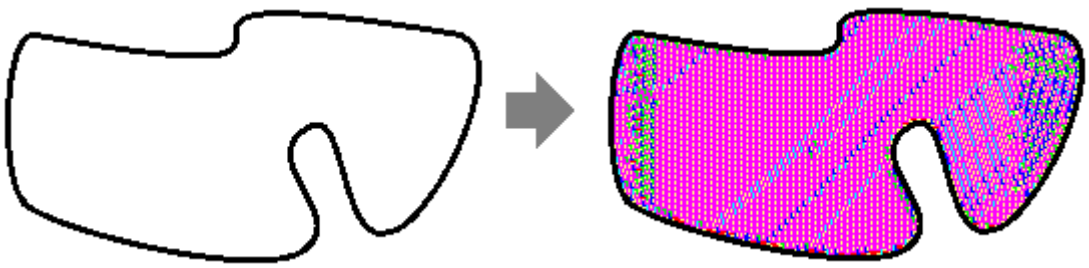

FIGURE 4-3: Approximation de minimisation

Inpainting

5.1 Qu'est-ce que c'est ?

L'inpainting est principalement utilisé pour réparer des images abîmées ou alors enlever des objets dans une image, bien que cela ne revienne en fait qu'à endommager l'image puis de la réparer sans l'objet. Cette technologie est largement adoptée aujourd'hui, elle est par exemple présente dans l'outil G'MIC du GREYC.

Il existe différents types d'algorithmes d'inpainting, on peut en citer au moins deux :

— Remplissage par Patch (local ou non)

— "Diffusion based" par utilisation d'EDP (équation de dérivée partielle)

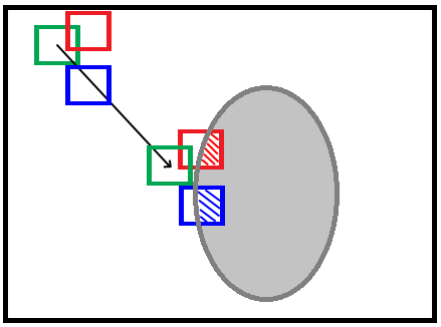

FIGURE 5-1: Présentation du principe de remplissage par patch

Ici dans notre projet nous avons utilisé le remplissage par Patch, pour faciliter la compréhension nous allons définir quelques notions.

— Patch : rectangle de $m * n$ pixels

— Fonction d'erreur : Fonction utilisée pour quantifier la réussite ou non de notre algorithme

5.2 Notre solution

Comme dit précédemment nous avons utilisé dans notre projet le principe de Patch-match pour remplir les zones souhaitées. Nous avons utilisé comme fonction d'erreur la fonction d'**Erreur quadratique moyenne** ou MSE en anglais.

$$MSE = \frac{1}{N * M} \sum_{i=1}^{N} \sum_{j=1}^{M} \left[I(i,j) - I'(i,j) \right]^2 \tag{5.1}$$

Cette fonction d'erreur reste très simple puisque qu'elle réalise en fait la distance entre les images, elle est très rapide mais malheureusement peut être trompeuse. En effet deux images avec les mêmes formes mais de couleurs différentes seront très éloignées selon cette fonction. Nous avons donc essayé de trouver d'autres index comme par exemple le SSIM [5]. Cependant nous avons été satisfaits du résultat des MSE et nous ne disposions pas du temps nécessaire pour bien appréhender le SSIM bien que l'option soit disponible dans notre implémentation.

Notre solution suit le pseudo code suivant :

Data : Image I
Fonction de coût F
Result : Meilleur patch P
listePatch ←creerListePatch()
for *patch* ∈*listePatch* **do**
  compteur ← 0
  for *voisin* ∈ *getVoisins*() **do**
    compteur += $F(voisin, patch)$
  end
end
Retourner le patch avec min(*compteur*)

FIGURE 5-2: Pseudo code de notre implémentation

creerListePatch() fait référence à la fonction de scikit-learn : sklearn.feature_extraction.image.extract_patches_2d qui permet de créer un certain nombre de patch de taille définie dans une image. On peut noter que cette fonction induit une notion d'aléatoire dans la création de ses patchs, ainsi si l'on sélectionne un nombre de patch trop faible pour une grande zone les résultats seront différents à chaque utilisations.

Ce pseudo code utilise les zones voisines de la zone à remplir pour remplir, cependant nous avons aussi réalisé une version ou l'on peut donner à l'algorithme un set de forme, ainsi l'algorithme trouvera parmi ces formes le meilleur candidat, cela permet donc de rentrer un peu plus dans la résolution de la problématique.

5.3 Validation

L'inpainting est de toutes les solutions proposées la plus probante, en effet de par sa nature celle-ci propose un résultat graphique comme présenté plus bas.

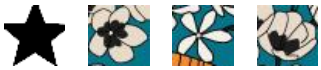

FIGURE 5-3: Pattern proposé dans le set

FIGURE 5-4: Image originale

FIGURE 5-5: Image locale avec trou

En utilisant notre programme avec le set d'entrée et comme fonction de coût la *MSE* nous obtenons comme meilleur candidat dans le set l'image suivante :

FIGURE 5-6: Meilleur motif sélectionné par notre algorithme

FIGURE 5-7: Image avec motif implémenté

Comme vous pouvez le voir sur la 5-7 incrustation est plutôt satisfaisante, cependant on peut voir que l'algorithme ne sélectionne pas le pattern qui pourrait recréer l'image initiale, c'est dû au fait qu'on ne sait pas quel était le patch original. Sinon cela n'aurait pas de sens.

Ce genre de pipeline de travail sur image permet une utilisation parfaite pour des algorithmes de machine-learning avec des réseaux de neurones. Un des sujets sur lequel nous aurions aimé nous pencher un peu plus, pour pouvoir mettre en parallèle plusieurs des cours que nous avons eus cette année. On remarque aussi que l'inpainting avec réseau de neurones semble est un sujet de recherche assez apprécié et assez récent comme peuvent le montrer ces articles tous assez récents [4][3].

Nous avons testé la rapidité de notre algorithme avec différents paramètres et noté tous les résultats dans le tableau ci-dessous :

	MSE (s)	SSIM (s)
Set de pattern (4 images)	0,028374	0,0287498
Voisinage(10 patchs)	0,0057	0,02815
Voisinage(20 patchs)	0,01	0,0978
Voisinage(50 patchs)	0,03922	0,5876
Voisinage(100 patchs)	0,143	2,3233

FIGURE 5-8: Temps d'exécution moyen

Ainsi nous avons pu tracer le graphique suivant, on peut noter que la taille des patchs est de 40x40 pixels :

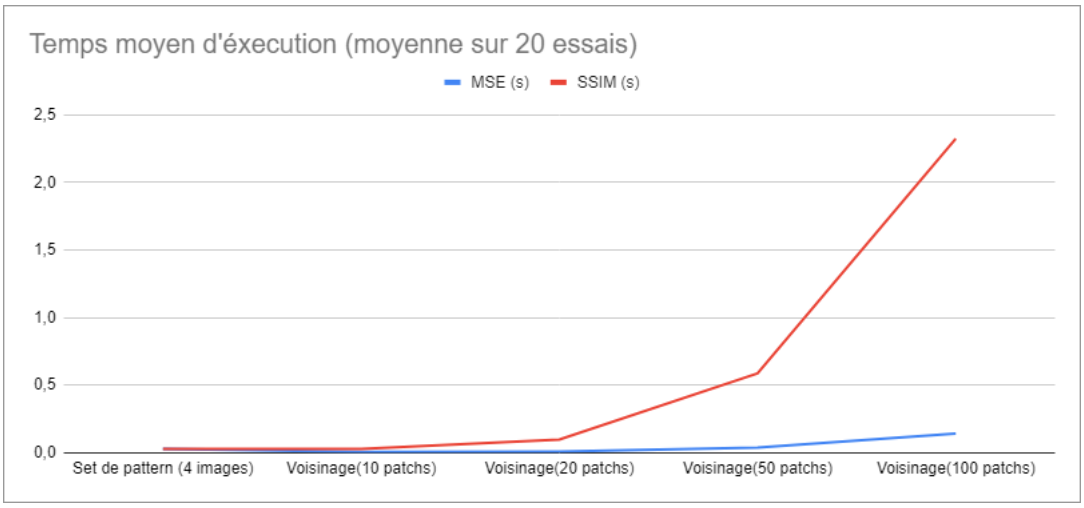

FIGURE 5-9: Graphique du temps d'exécution selon le nombre de patchs

Grâce à ce graphique on peut voir que la fonction SSIM rend le temps d'exécution exponentiellement plus lent. Et un tel coût n'a pas forcément montré sa nécessité, nous n'avons pas vu de différence énorme entre le résultats avec la MSE et un résultat avec la SSIM, même parfois de pires résultats si par exemple on utilise les fonctions sur le set de pattern.

Conclusion

Ce projet fût pour nous deux très intéressant et permit d'entrecroiser nos deux spécialisations de Master, cependant celui-ci fût réalisé pendant la crise sanitaire du COVID-19, ainsi le temps consacré n'était pas forcément optimal sachant que la crise a forcé la majorité de nos examens et CC à devenir des projets à part entière. Il faut aussi noter qu'une grande partie de ce projet était de s'approprier le sujet et la littérature qui va avec, nos bases en imagerie dataient de la L2 et n'étaient pas aussi poussées que les sujets abordés. Ce n'est pas pour autant que nous ne sommes pas satisfaits de nos résultats. En effet nous avons des résultats concluants qui peuvent être améliorés mais ceci était un projet de recherche où ce que nous avons découvert est sans doute bien plus intéressant que la réalisation d'une interface graphique. Ce projet fut également très enrichissant d'un point de vue de gestion projet, nous avions des réunions hebdomadaires pour faire un point sur nos avancées et définir la direction pour la semaine d'après. Nous tenons donc à remercier M Chahir pour toute son aide et ses disponibilités malgré des conditions sanitaires pas forcément optimales. Ainsi nous avons quelques idées d'approfondissement, la première serait de pouvoir lier notre solution de résolution de contraintes et notre analyse du Code de Freeman pour optimiser la résolution linéaire. Ensuite on peut penser à l'utilisation d'un algorithme de réseau de neurone profond pour améliorer notre algorithme d'inpainting car celui-ci s'y prête extrêmement bien. Finalement onpourrait avec plus de temps s'attaquer à une interface utilisateur pour une visualisation des algorithmes utilisés en temps réel. Enfin nous aimerions noter que ce projet nous a permis d'aborder un pipeline de projet classique ; (1) prise en main du projet et définition d'objectifs ; (2) réunions hebdomadaires ; (3) travail à distance sur le site de gestion de projet de l'université.

References

[1] H. FREEMAN. "On the Encoding of Arbitrary Geometric Configurations". In : *IRE Transactions on Electronic Computers* EC-10.2 (1961), p. 260-268. DOI : 10.1109/TEC.1961.5219197.

[2] Xavier TROUILLOT. "Étude de paramètres géométriques à partir du code de Freeman". 172 pages. Theses. Ecole Nationale Supérieure des Mines de Saint-Etienne, déc. 2008. URL : https://tel.archives-ouvertes.fr/tel-00496290.

[3] Yi WANG et al. *Image Inpainting via Generative Multi-column Convolutional Neural Networks*. 2018. arXiv : 1810.08771 [cs.CV].

[4] Chao YANG et al. "High-Resolution Image Inpainting Using Multi-Scale Neural Patch Synthesis". In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Juil. 2017.

[5] Xue Dong YANG. *A comprehensive assessment of the structural similarity index*. 2011. URL : https://ece.uwaterloo.ca/~z70wang/publications/ssim.html.

[6] Borut ŽALIK et Niko LUKAČ. "Chain code lossless compression using move-to-front transform and adaptive run-length encoding". In : *Signal Processing : Image Communication* 29.1 (2014), p. 96-106. ISSN : 0923-5965. DOI : https://doi.org/10.1016/j.image.2013.09.002. URL : https://www.sciencedirect.com/science/article/pii/S0923596513001513.