Projet Inpainting - Arthur Evain - Raphaël Legrand

Résumé de l'article et de l'algorithme de Criminisi :

L'algorithme de Criminisi (ou Image Inpainting based on the Fast Marching Method) est une méthode de synthèse de texture par patchs utilisée pour combler des régions masquées ("trous" Ω) dans une image. Il fonctionne de manière itérative, en propageant l'information de l'extérieur vers l'intérieur du trou.

1. Structure Itérative

L'algorithme comble la zone masquée (mask) en choisissant et en copiant les meilleurs patchs source disponibles dans la région connue. La boucle se poursuit jusqu'à ce que le masque soit entièrement rempli.

2. Formule de Priorité (Choix du Patch Cible p)

À chaque itération, l'algorithme doit décider quel point p sur la frontière du trou ($\delta\,\Omega$) sera rempli en priorité. Cette décision est basée sur une formule combinant deux termes :
$P(p)=C(p)D(p)$

- Terme de Confiance $C_{(p)}$: Mesure la fiabilité des pixels autour de p. Il est initialisé à 1.0 dans la source et 0.0 dans le trou, et décroît ($0<C<1.0$) au fur et à mesure que les zones sont synthétisées. Il favorise les patchs près de la source originale.
- Terme de Données $D(p)$: Mesure la force de la structure (lignes, bords) qui converge vers p. Il est basé sur le produit scalaire entre le vecteur normal à la frontière et le vecteur isophote (gradient). Il favorise la continuité des contours.

3. Recherche du Meilleur Patch Source

Une fois le point p sélectionné, l'algorithme recherche le meilleur patch source Ψ_q dans la zone connue qui minimise la distance au patch cible Ψ_p.

Nos modifications

Nous avons effectué quelques modifications par rapport à l'algorithme décrit dans le papier :

- Le patch possède une couronne qui intervient pour déterminer quel patch est le meilleur. Les pixels dans cette couronne servent à ajouter du contexte mais ne sont pas copié dans la zone à inpaint.

- Le calcul des distances, censé être une simple distance qudratique, est pondéré en ajoutant un poids au canal L dans la représentation CIELab.
- Un seuil est ajouté pour obliger le patch source à avoir un certain score de confiance. Cela permet d'éviter que des artefacts de patch se propagent dans la zone à inpaint.

Explication du code

Importations et variables globales

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

PATCH_RADIUS = 3
COURONNE = 4
```

Nous avons choisi d'utiliser le librairie OpenCV pour gérer les images.

PATCH_RADIUS = 3 -> Signifie que les patchs que l'on copiera, centré en un pixel, sont de côté $2 $ PATCH_RADIUS $ + 1 = 7$

COURONNE = 4 -> On aggrandit la taille du patch avec une "couronne" de largeur 4 pixels. Ceci ne sert que de comparaison entre les patchs, et le contenu de la couronne n'est pas copié.

COURONNE est un ajout de notre part. Cela nous permet d'obtenir plus de précision sur les comparaisons entre patchs afin d'éviter certaines abérrations (morceaux de ciel qui se retrouvent dans l'herbe par exemple) dans le rendu final. Nous expliquons plus comment nous faisons nos comparaisons entre les patchs.

Fonctions d'affichage

```
def overlay_mask(img, mask):
    """
    Affiche le masque en rouge sur l'image
    """
    overlay = img.copy()
    overlay[mask == 255] = (0, 0, 255)
    return cv2.addWeighted(img, 0.7, overlay, 0.3, 0)

def resize_for_display(img, target_width=1000):
    """
    Redimensionne l'image pour l'affichage
    """
    h, w = img.shape[:2]
    if w > target_width:
        aspect_ratio = h / w
        target_height = int(target_width * aspect_ratio)
```

```
        return cv2.resize(img, (target_width, target_height))
    if w <= target_width:
        new_width = max(w, target_width)
        aspect_ratio = h / w
        target_height = int(new_width * aspect_ratio)
        return cv2.resize(img, (new_width, target_height))
    return img

def select_mask(img):
    points = []
    img_display = img.copy()

    def select_polygon(event, x, y, flags, param):
        nonlocal points, img_display # Chargement des variables de
select_mask
        if event == cv2.EVENT_LBUTTONDOWN:
            points.append((x, y))
            if len(points) > 1:
                cv2.line(img_display, points[-2], points[-1], (0, 255,
0), 2)
            cv2.circle(img_display, (x, y), 1, (255, 0, 0), -1)

    cv2.namedWindow("Selection")
    cv2.setMouseCallback("Selection", select_polygon)

    while True:
        cv2.imshow("Selection", img_display)
        key = cv2.waitKey(1) & 0xFF
        if key == 13: # Entrée = validation du masque
            break
        elif key == 27: # Échap = réinitialisation du masque
            points = []
            img_display = img.copy()

    cv2.destroyWindow("Selection")

    if len(points) < 3:
        return np.zeros(img.shape[:2], dtype=np.uint8) # Masque vide

    mask = np.zeros(img.shape[:2], dtype=np.uint8)
    cv2.fillPoly(mask, [np.array(points)], 255)
    return mask
```

La zone Ω à inpaint est représentée par un masque avec des 255 aux positions de ses pixels (et des 0 aux positions des pixels connus de l'image). Ce masque est sélectionné à la main grâce à la fonction **select_mask**, où l'utilisateur place des points qui formeront le polygone délimitant Ω.

overlay_mask sert à l'affichage de l'évolution en temps réel du masque.

resize_for_display permet d'ajuster la taille des images affichées (car on travaille avec des images petites, l'algorithme étant lent).

Fonctions de calcul vectoriel

```
def compute_gradients(gray):
    """
    Calcule les gradients de l'image en niveaux de gris à l'aide de
l'opérateur Sobel
    """
    Ix = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    Iy = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    return Ix, Iy

def compute_isophote(Ix, Iy):
    """
    Calcule les isophotes à partir des gradients.
    """
    return np.dstack((-Iy, Ix))

def compute_normals(mask):
    """
    Calcule les vecteurs normaux à la frontière du masque.
    """
    mask_float = mask.astype(np.float32)/255.0
    Nx = cv2.Sobel(mask_float, cv2.CV_64F, 1, 0, ksize=3)
    Ny = cv2.Sobel(mask_float, cv2.CV_64F, 0, 1, ksize=3)
    N = np.dstack((Nx, Ny))
    norm = np.linalg.norm(N, axis=2, keepdims=True) + 1e-8
    return N / norm
```

Ces fonctions calculent les termes apparaissant dans l'expression de la priorité.

`compute_gradients` utilise l'opérateur **Sobel** pour calculer le gradient.

`compute_isophote` effectue une rotation de 90° sur le gradient pour calculer l'isophote.

`compute_normals` calcule le vecteur normal à la frontière du masque, qui est nécessaire pour le produit scalaire dans le calcul $D(p)$.

Calcul de la priorité

```
def compute_priority(img_gray, mask, C, patch_radius=PATCH_RADIUS,
alpha=255.0):

    Ix, Iy = compute_gradients(img_gray)
    isophote = compute_isophote(Ix, Iy)
    N = compute_normals(mask)

    priorities = np.zeros_like(img_gray, dtype=np.float32)
```

```
    # Calcul de la bordure
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
    border = cv2.morphologyEx(mask, cv2.MORPH_GRADIENT, kernel)
    border_points = np.where(border > 0)
    h, w = img_gray.shape
    r = patch_radius

    for y, x in zip(*border_points):

        # Calcul de C
        x1, x2 = max(0, x - r), min(w, x + r + 1)
        y1, y2 = max(0, y - r), min(h, y + r + 1)
        patch_conf = C[y1:y2, x1:x2]

        # Empêcher les priorités fantômes : si presque aucun pixel
n'est connu -> priorité = 0
        if np.sum(patch_conf > 0) < 4:
            continue

        C_p = np.mean(patch_conf)

        # Calcul de D

        # Utiliser la valeur normalisée du gradient pour le calcul
D_raw
        D_raw = np.dot(isophote[y, x], N[y, x])
        if np.isnan(D_raw):
            continue

        D_p = abs(D_raw) / alpha # Alpha = 255.0 est la borne max du
gradient

        # Calcul de P(p) = C(p) * D(p)
        priorities[y, x] = C_p * D_p

    return priorities
```

Cette fonction calcule la priorité $P(p)$ du point de remplissage optimal sur la frontière ($\delta\Omega$).

Élément Analysé	Description	Conformité Criminisi
$P(p)=C(p)\times D(p)$	Implémentation de la formule combinée **Confiance (C)** et **Données (D)**.	**Base Théorique :** C'est la signature mathématique de l'algorithme.
Terme de Confiance ($C(p)$)	Calculé comme la **moyenne des valeurs** C dans le patch cible Ψ_p.	Respecte l'esprit du Terme de Confiance, favorisant les zones bien entourées.
Terme de Données ($D(p)$)	Déterminé par le produit scalaire entre le **gradient**	Respecte le Terme de Données, privilégiant la

Élément Analysé	Description	Conformité Criminisi
	isophote et la **normale** de la frontière.	continuité des structures.

.

Détermination du meilleur patch : Comparaison Robuste et Flexibilité

```
def find_best_patch(img_inpaint_lab, mask, C, target_patch,
patch_radius=PATCH_RADIUS, C_threshold=0.9, w_L=2.0):

    h, w = img_inpaint_lab.shape[:2]
    y, x = target_patch
    r = patch_radius + COURONNE

    t_y1 = max(0, y-r); t_y2 = min(h, y+r+1)
    t_x1 = max(0, x-r); t_x2 = min(w, x+r+1)
    t_h = t_y2 - t_y1; t_w = t_x2 - t_x1

    # Patch CIELab
    tgt_patch_lab = img_inpaint_lab[t_y1:t_y2,
t_x1:t_x2].astype(np.float32)
    tgt_mask = mask[t_y1:t_y2, t_x1:t_x2]

    valid = (tgt_mask == 0)
    n_valid = np.sum(valid)
    if n_valid == 0:
        return None

    best_patch = None
    best_dist = float("inf")

    y_min, y_max = r, h - r - 1
    x_min, x_max = r, w - r - 1

    for yy in range(y_min, y_max + 1):
        for xx in range(x_min, x_max + 1):

            if mask[yy, xx] != 0:
                continue

            s_y1 = yy - t_h//2; s_y2 = yy + (t_h - t_h//2)
            s_x1 = xx - t_w//2; s_x2 = xx + (t_w - t_w//2)

            src_conf_patch = C[s_y1:s_y2, s_x1:s_x2]

            if np.min(src_conf_patch) < C_threshold: # Seuil de
confiance sur le patch source
                continue

            src_patch_lab = img_inpaint_lab[s_y1:s_y2,
s_x1:s_x2].astype(np.float32)
```

```
            if src_patch_lab.shape != tgt_patch_lab.shape:
                continue

            src_valid = src_patch_lab[valid]
            tgt_valid = tgt_patch_lab[valid]

            # Calcul de la distance de texture pondérée
            diff_L = (src_valid[:, 0] - tgt_valid[:, 0])**2 * w_L
            diff_a = (src_valid[:, 1] - tgt_valid[:, 1])**2
            diff_b = (src_valid[:, 2] - tgt_valid[:, 2])**2

            total_diff = diff_L + diff_a + diff_b

            if n_valid == 0:
                continue

            # MSD sur 3 canaux (normalisation par n_valid * 3)
            dist = np.sum(total_diff) / (n_valid * 3)

            if dist < best_dist:
                best_dist = dist
                best_patch = (yy, xx)

    return best_patch
```

Cette fonction trouve le meilleur patch source Ψ_q, mais avec des critères d'acceptation et une métrique de distance optimisés.

Élément Analysé	Description	Avantage Personnel
Espace Couleur	Le calcul de la distance utilise l'espace $L^{\iota}a^{\iota}b^{\iota}$ (`img_inpaint_lab`) et non BGR.	**Robustesse aux Couleurs :** Utilisation de $L^{\iota}a^{\iota}b^{\iota}$ avec **pondération** w_L plus forte sur L^{ι} pour prioriser la **texture** sur la teinte.
Métrique de Distance	La distance est calculée comme la **Mean Squared Difference (MSD)**, normalisée par le nombre de pixels valides.	**Stabilité :** Évite que la distance soit faussée par le nombre variable de pixels valides dans le patch cible.
Seuil de Confiance	Le patch source est accepté seulement si $np.min(C[\dots]) \geq C_{\text{threshold}}$.	**Flexibilité de Source :** Permet d'utiliser des **zones déjà synthétisées** comme source (où $0<C<1.0$), crucial pour les grandes régions.

Patch par défaut

```
def get_next_point(mask):
    """
    Retourne un point sur la bordure du masque, dans le cas où on ne
trouve pas la priorité maximale.
    """
    border = cv2.Canny(mask, 100, 200)
    border_points = np.where(border > 0)
    if len(border_points[0]) > 0:
        idx = np.random.randint(0, len(border_points[0]))
        y, x = border_points[0][idx], border_points[1][idx]
        return y, x
    return None, None
```

Cette fonction retourne un point aléatoire sur la bordure $\delta \Omega$ dans le rare cas où le calcul du point de priorité maximale échoue.

Logique de Mise à Jour

```
def run_inpainting(img, mask, C):

    img_inpaint = img.copy()
    r = PATCH_RADIUS

    # On passe l'image dans le domaine CIELAB
    img_inpaint_lab = cv2.cvtColor(img_inpaint, cv2.COLOR_BGR2LAB)

    cv2.namedWindow("Inpainting progress", cv2.WINDOW_NORMAL)
    cv2.resizeWindow("Inpainting progress", 800, 600)

    while np.any(mask == 255):
        gray = cv2.cvtColor(img_inpaint, cv2.COLOR_BGR2GRAY)
        priorities = compute_priority(gray, mask, C)

        if np.max(priorities) > 0:
            y, x = np.unravel_index(np.argmax(priorities),
priorities.shape)
        else:
            y, x = get_next_point(mask)
        if y is None:
            break

        y1, y2 = max(0, y - r), min(C.shape[0], y + r + 1)
        x1, x2 = max(0, x - r), min(C.shape[1], x + r + 1)
        patch_conf_at_p = C[y1:y2, x1:x2]
        C_p_value = np.mean(patch_conf_at_p)

        best = find_best_patch(img_inpaint_lab, mask, C, (y, x))
```

```
        if best is None:
            break

        yy, xx = best

        t_y1, t_y2 = y-r, y+r+1
        t_x1, t_x2 = x-r, x+r+1
        s_y1, s_y2 = yy-r, yy+r+1
        s_x1, s_x2 = xx-r, xx+r+1

        tgt_mask = mask[t_y1:t_y2, t_x1:t_x2]

        src_patch_bgr = img_inpaint[s_y1:s_y2, s_x1:s_x2]
        img_inpaint[t_y1:t_y2, t_x1:t_x2][tgt_mask == 255] =
src_patch_bgr[tgt_mask == 255]

        src_patch_lab = img_inpaint_lab[s_y1:s_y2, s_x1:s_x2]
        img_inpaint_lab[t_y1:t_y2, t_x1:t_x2][tgt_mask == 255] =
src_patch_lab[tgt_mask == 255]

        mask[t_y1:t_y2, t_x1:t_x2][tgt_mask == 255] = 0
        C[t_y1:t_y2, t_x1:t_x2][tgt_mask == 255] = C_p_value

        cv2.imshow("Inpainting progress",
resize_for_display(overlay_mask(img_inpaint, mask)))
        cv2.waitKey(1)

    cv2.destroyWindow("Inpainting progress")
    return img_inpaint.copy()
```

Cette fonction effectue les différentes étapes de l'algorithme dans cet ordre :

- Conversion de l'image en CIELab
- Détermination du pixel de priorité maximale (avec solution par défaut si échec)
- Détermination du meilleur patch pour ce point
- Copie du patch et mise à jour du masque et de la confiance

et ce tant que le masque représentant Ω n'est pas vide

Élément Analysé	Rôle	Avantage Personnel
Propagation de C	C est mis à jour avec $C_{p_value} = np.mean(patch_c on$.	**Correction Critique :** Assure que C devient une carte de **valeurs continues** ($0 < C < 1.0$), reflétant fidèlement la fiabilité des pixels synthétisés.
Mise à Jour LAB/BGR	L'image BGR et l'image LAB sont mises à jour **simultanément** après la	**Cohérence :** Maintient l'alignement des données pour la précision de la

Élément Analysé	Rôle	Avantage Personnel
	copie de chaque patch.	recherche de patchs basée sur LAB.

.

Gestion des itérations

```
if __name__ == "__main__":
    img_path = "img/arbre.jpg"
    img = cv2.imread(img_path)
    cv2.imshow("Image d'origine", resize_for_display(img))
    if img is None:
        raise SystemExit(f"Impossible de charger l'image: {img_path}")

    img_working = img.copy()

    print("\n➡ Tracez le masque (Entrée pour valider) \n")
    mask_initial = select_mask(img_working)

    if np.all(mask_initial == 0):
        print("Pas de zone masquée. Terminé.")
        cv2.destroyAllWindows()
        exit()

    print("\n Nouvelle passe d'Inpainting (Entrée/espace) Q = quitter\
n")

    # Boucle d'amélioration itérative sur la MÊME zone
    iteration_count = 0
    while True:
        iteration_count += 1
        print(f"--- Début de l'itération {iteration_count} ---")

        # Réinitialisation du masque pour la passe
        mask_current = mask_initial.copy()

        # Réinitialisation de la carte de Confiance C :
        C = np.ones_like(mask_initial, dtype=np.float32)
        C[mask_current == 255] = 0.0

        # Exécution de l'inpainting
        img_result = run_inpainting(img_working.copy(), mask_current,
C)

        # Mise à jour de l'image de travail
        img_working = img_result.copy()

        cv2.imshow("Resultat courant",
resize_for_display(img_working))
```

```
        key = cv2.waitKey(0) & 0xFF

        if key in (ord('q'), ord('Q')):
            break

    cv2.imwrite("result.png", img_working)
    cv2.destroyAllWindows()
```

On peut relancer l'algorithme plusieurs fois sur le même masque en appuyant sur une touche qu'autre que q. On autorise la copie des patchs dans Ω à partir de la deuxième itération.

Observations et résultats

Nous avons testé l'algorithme sur des images du papier et d'autres images qui nous ont semblé intéressantes. Nous avons effectué une seule itération au début, puis sur les images plus difficiles nous avons fait plusieurs itérations où on redéfinit un masque à chaque fois pour obtenir des résultats convaincants. A noter que les images sont en basse résolution pour que l'algorithme ne prenne pas trop de temps à terminer.

Résultats sur une itération

Commençons par une image simple : un dégradé qui contient un watermark. L'algorithme est très efficace et propage naturellement les sctructures.

Image de dégradé Image de dégradé inpaintée

Confrontons maintenant cette propagation des textures aux fameux triangles de Kanizsa. Comme dans le papier, un pic apparaît, mais systématiquement sur le cercle du haut et pas sur les deux autres, qui sont complétés naturellement. C'est sans doute dû au manque de contexte : il y a peu de pixels en bas des autres ronds noirs, donc aucun patch ne peut compléter correctement le bas du rond noir du haut. Réduire la taille des patchs empêche d'avoir une bonne recontruction. La solution serait d'ajouter une bordure planche d'au moins 3 pixels de large autour de l'image.

Image de Kanizsa triangle Image de Kanizsa triangle inpaintée

Testons maintenant l'algorithme sur une image complexe utilisé dans le papier. Les lignes sont correctement propagées, mais les erreurs s'accumulent et l'algorithme n'arrive pas à garder un résultat cohérent jusqu'au bout.

Image de femme devant une fontaine Image de femme devant une fontaine inpaintée

Résultats sur plusieurs itérations

Testons maintenant de redéfinir plusieurs fois le masque. En ciblant spéficiquement et progressivement les zones, on arrive à un résultat réaliste. On trouve quand même des artefacts, provoqués par des textures complexes (haut de la fontaine) ou des besoins de cohérence trop longs (lignes longues).

Image de femme devant une fontaine Image de femme devant une fontaine inpaintée

Voici une autre image du papier. Plusieurs itérations successives ont été nécessiares pour propager le poteau du panneau. Le bas du panneau du haut comporte des artefacts difficiles à effacer.

Image de panneaux Image de panneaux inpaintée

Enfin, voici une de nos images. Les vagues ont provoqué des artefacts car la progressivité de leur forme est difficile à intuiter pour l'algorithme. La proximité des deux chevaux a rendue difficile la suppression des artefacts à coté de la queue du cheval.

Image de chevaux sur le plage Image de chevaux sur le plage inpaintée

---

Discussions pour améliorations

Notre code aurait besoin d'optimisation pour être plus rapide, ce qui permettrait de le tester sur des images de meilleure qualité. On pourrait l'implémenter sur un autre langage ou optimiser la recherche de patch. Sinon, on pourrait s'inspirer de papiers plus avancés comme PatchWorks pour pallier à des problèmes de propagation de textures observés (par exemple, on pourrait affiner le calcul de la distance entre deux patchs). Enfin, l'inpainting passe aujourd'hui par les réseaux de neurone : ceux-ci constitueraient indéniablement une nette amélioration dans les résultats.