



# Compte Rendu Mini-Projet

Algorithmique Avancée - L3

**FEKIH HASSEN Yassine**

Enseignants : M. LOMENIE Nicolas, M. MAHE Gael

# **Dijkstra et Image**

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Contexte . . . . .	3
<b>2</b>	<b>Traitement d'Image comme un Graphe pour le Calcul du Plus Court Chemin</b>	<b>3</b>
2.1	Représentation Graphique d'une Image . . . . .	3
2.2	Coût Basé sur la Différence de Luminosité . . . . .	3
<b>3</b>	<b>Représentation d'une Image comme un Graphe</b>	<b>3</b>
3.1	Connexité dans une Image . . . . .	4
3.2	4-Connexité . . . . .	4
3.3	8-Connexité . . . . .	4
<b>4</b>	<b>Calcul du Plus Court Chemin dans une Image</b>	<b>5</b>
4.1	Types de Fonctions de Coût . . . . .	5
4.2	Choix de la Fonction de Coût . . . . .	5
4.3	Choix de l'Espace Colorimétrique Lab* pour le Projet . . . . .	5
4.3.1	Caractéristiques de l'Espace Lab* . . . . .	6
<b>5</b>	<b>Fonctions de calcul de coûts</b>	<b>6</b>
5.1	Fonction de Coût Basée sur l'Espace Lab* . . . . .	6
5.2	Fonction de Coût Basée sur la Différence de Luminosité . . . . .	6
5.3	Fonction de Coût Basée sur l'Intensité Lumineuse . . . . .	6
5.4	Fonction de Coût Basée sur le Contraste Local . . . . .	6
5.5	Nécessité d'Appliquer des Filtres sur les Images . . . . .	7
5.5.1	Filtre de Lissage . . . . .	7
5.5.2	Avantages pour le Calcul du Plus Court Chemin . . . . .	7
<b>6</b>	<b>Dijkstra pour le Calcul du Plus Court Chemin</b>	<b>8</b>
6.1	Implémentation de l'Algorithm de Dijkstra . . . . .	8
6.2	Fonction pour Trouver et Afficher le Chemin le Plus Court . . . . .	9
6.3	Note sur les Fonctions de Coût . . . . .	10
<b>7</b>	<b>Optimisation de la Fonction Dijkstra dans le Projet</b>	<b>10</b>
7.1	Changement de PriorityQueue à heapq . . . . .	10
7.1.1	Raisons du Changement . . . . .	10
7.1.2	Impact sur l'Algorithm . . . . .	10
7.2	Conclusion . . . . .	10
<b>8</b>	<b>Création du Graphe à partir de l'Image</b>	<b>10</b>
8.1	Fonction create_graph . . . . .	11
8.1.1	Visualisation des Sommets . . . . .	11
8.2	Autres Méthodes pour l'Interface Graphique . . . . .	11
<b>9</b>	<b>Exécution du programme</b>	<b>12</b>

# 1 | Introduction

## 1.1 | Contexte

Le présent compte rendu se penche sur un exercice centré sur la manipulation d'images et l'application de concepts de théorie des graphes. Notre objectif principal est de traiter une structure de graphe, une grille image.

Nous considérons les pixels d'une image comme les sommets d'un graphe, tandis que les relations de voisinage entre ces pixels sont interprétées comme des arêtes. Une caractéristique essentielle de ce graphe est que ses arêtes sont pondérées en fonction de la différence d'intensité entre les pixels voisins. En d'autres termes, plus la différence d'intensité est grande, plus le poids de l'arête correspondante est élevé. Le processus initial de ce projet consiste à construire un tel graphe à partir de n'importe quelle image. Cette étape implique la conversion d'une grille d'image en une structure de graphe. Une fois ce graphe généré, nous passons à la phase suivante, qui consiste à concevoir une interface utilisateur graphique. L'interface utilisateur doit permettre aux utilisateurs de sélectionner deux pixels spécifiques, l'un servant de point de départ et l'autre de point d'arrivée. Ensuite, cette interface doit utiliser l'algorithme de Dijkstra pour trouver le plus court chemin entre ces deux sommets dans le graphe. Enfin, elle doit présenter une visualisation de ce chemin sur l'image d'origine, fournissant ainsi une représentation visuelle du résultat.

# 2 | Traitement d'Image comme un Graphe pour le Calcul du Plus Court Chemin

Dans cette approche, nous traitons une image 2D comme un graphe pour calculer le chemin le plus court, en mettant l'accent sur la connectivité des pixels et le coût associé à leurs déplacements.

## 2.1 | Représentation Graphique d'une Image

Une image est considérée comme un graphe où chaque pixel est un nœud. Chaque pixel est connecté à ses quatre voisins les plus proches : haut, bas, droite, gauche. Cette structure est similaire à celle d'une grille régulière.

## 2.2 | Coût Basé sur la Différence de Luminosité

Contrairement à une grille régulière où le coût de déplacement est uniforme, dans ce graphe, le coût de déplacement d'un pixel à l'autre dépend de la différence de luminosité entre ces deux pixels.

# 3 | Représentation d'une Image comme un Graphe

Une image peut être vue comme une grille de pixels, chaque pixel ayant une position unique représentée par ses coordonnées  $(i, j)$  où  $i$  est le numéro de ligne et  $j$  est le numéro de colonne. Pour résoudre le problème du plus court chemin dans une image, nous pouvons représenter cette image comme un graphe non orienté. Dans cette représentation, chaque pixel devient un sommet du graphe, et les relations de voisinage entre les pixels deviennent les arêtes du graphe. Pour formaliser cela, définissons un graphe  $G = (V, E)$  où  $V$  est l'ensemble des sommets représentant les pixels et  $E$  est l'ensemble des arêtes qui relient les sommets en fonction des relations de voisinage.

### 3.1 | Connexité dans une Image

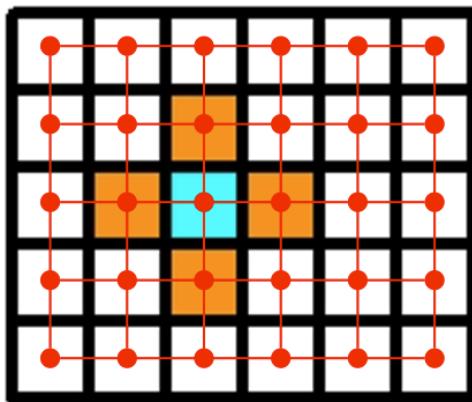
La connexité d'une image se réfère à la façon dont les pixels sont reliés les uns aux autres. Deux types de connexités couramment utilisées sont la 4-connectivité et la 8-connectivité.

### 3.2 | 4-Connexité

Dans le cas de la 4-connectivité, deux pixels sont considérés comme voisins s'ils partagent un côté, c'est-à-dire qu'ils sont reliés horizontalement ou verticalement. Mathématiquement, deux pixels  $(i_1, j_1)$  et  $(i_2, j_2)$  sont 4-connectés si :

$$|i_1 - i_2| + |j_1 - j_2| = 1$$

Cela signifie que les pixels sont adjacents par le haut, le bas, la gauche ou la droite.



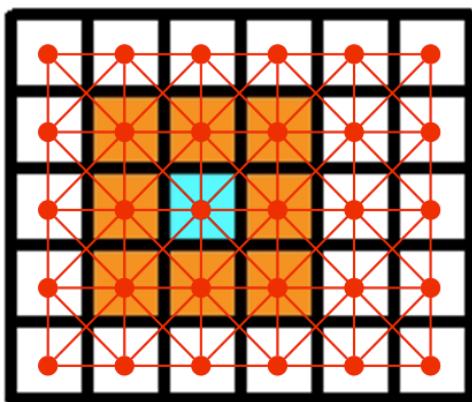
**Figure 1:** Exemple de 4-connectivité.

### 3.3 | 8-Connexité

La 8-connectivité étend la notion de voisinage en incluant les diagonales. Deux pixels  $(i_1, j_1)$  et  $(i_2, j_2)$  sont 8-connectés si :

$$\max(|i_1 - i_2|, |j_1 - j_2|) = 1$$

Cela signifie que les pixels sont adjacents par le haut, le bas, la gauche, la droite ou l'une des quatre diagonales.



**Figure 2:** Exemple de 8-connectivité.

## 4 | Calcul du Plus Court Chemin dans une Image

Pour calculer le plus court chemin entre deux pixels dans une image représentée comme un graphe, nous utilisons l'algorithme de Dijkstra. Cet algorithme explore le graphe de manière itérative en choisissant toujours le chemin le plus court depuis un point de départ jusqu'à ce qu'il atteigne le point d'arrêt. La fonction de coût entre deux pixels détermine la longueur de l'arête entre eux.

Mathématiquement, la fonction de coût entre deux pixels  $(i_1, j_1)$  et  $(i_2, j_2)$  peut être définie comme suit :

$$\text{coût}(i_1, j_1, i_2, j_2) = f(\text{image}, (i_1, j_1), (i_2, j_2))$$

Où  $f$  est une fonction dépendante du problème spécifique. Par exemple, pour une image en niveaux de gris,  $f$  pourrait être basé sur la différence d'intensité lumineuse entre les pixels. Dans mon programme j'ai implémenté plusieurs fonctions de coûts.

L'algorithme de Dijkstra maintient une file de priorité qui stocke les pixels à explorer, en commençant par le pixel de départ. Il met à jour les distances entre les pixels à mesure qu'il explore le graphe, en choisissant toujours le pixel avec la plus courte distance pour l'exploration suivante. Une fois le pixel d'arrêt atteint, l'algorithme reconstruit le chemin le plus court en suivant les liens parent créés pendant l'exploration.

Pour déterminer le plus court chemin sur une image, il est essentiel d'utiliser une fonction de calcul de coût qui évalue les arêtes en fonction de divers critères. La section ci-dessous présente plusieurs types de fonctions de coût et leurs applications.

### 4.1 | Types de Fonctions de Coût

**Coût basé sur la luminosité ou le contraste** : Idéal pour le traitement d'images, en particulier la segmentation, où la distinction entre différentes régions de luminosité ou de contraste est cruciale.

**Coût basé sur la couleur** : Utilisé principalement en vision par ordinateur pour la classification et le suivi d'objets, où la couleur est un facteur déterminant.

**Coût basé sur la texture** : Pertinent pour la reconnaissance de motifs ou la classification de surfaces, utile lorsque la texture est un attribut clé de l'image.

**Coût basé sur la distance euclidienne** : Simple et efficace pour les parcours dans des espaces de faible complexité, adapté aux applications nécessitant une approximation rapide.

**Coût basé sur le chemin géodésique** : Convient aux applications exigeant des mesures précises sur des surfaces irrégulières ou courbes.

etc...

### 4.2 | Choix de la Fonction de Coût

Le choix de la fonction de coût dépend largement de l'application spécifique et des caractéristiques de l'image. Par exemple, dans le domaine médical pour l'analyse d'images IRM, un coût basé sur la texture ou le contraste peut être plus approprié. Pour la navigation robotique, un coût basé sur la distance euclidienne ou le chemin géodésique pourrait être préféré etc...

### 4.3 | Choix de l'Espace Colorimétrique Lab\* pour le Projet

Dans le cadre de notre projet, nous avons opté pour l'utilisation de l'espace colorimétrique Lab\*, un choix stratégique pour plusieurs raisons. L'espace Lab\*, également connu sous le nom de CIELAB, est un espace colorimétrique tridimensionnel conçu pour être plus perceptuellement uniforme que d'autres espaces colorimétriques. Autrement dit, une modification d'une même quantité dans une valeur de couleur Lab\* correspond à peu près à la même modification perçue dans la couleur, quelle que soit la région de l'espace.

#### 4.3.1 | Caractéristiques de l'Espace Lab\*

L'espace Lab\* se compose de trois axes :

**L\*** pour la luminosité, allant de 0 (noir) à 100 (blanc).

**a\*** allant de vert à rouge, avec des valeurs négatives indiquant le vert et des valeurs positives le rouge.

**b\*** allant de bleu à jaune, avec des valeurs négatives pour le bleu et des valeurs positives pour le jaune.

Ainsi, l'utilisation de l'espace Lab\* dans notre projet s'aligne sur notre objectif de traitement d'images précis et conforme à la perception humaine des couleurs.

## 5 | Fonctions de calcul de coûts

Les fonctions de calcul de coûts dans notre programme sont adaptées au type d'image traitée et aux informations disponibles sur cette image. Plusieurs fonctions ont été implémentées pour tenir compte de divers aspects tels que la couleur, la luminosité et le contraste local.

### 5.1 | Fonction de Coût Basée sur l'Espace Lab\*

Cette fonction calcule la différence de couleur et de luminosité entre deux pixels dans l'espace Lab\*. Elle est définie comme suit :

```
def cost_function_lab(point1, point2, image_lab):
    L1, a1, b1 = image_lab[point1[0], point1[1]].astype(int)
    L2, a2, b2 = image_lab[point2[0], point2[1]].astype(int)
    return np.sqrt((L1 - L2) ** 2 + (a1 - a2) ** 2 + (b1 - b2) ** 2)
```

### 5.2 | Fonction de Coût Basée sur la Différence de Luminosité

Cette fonction prend en compte uniquement la différence de luminosité entre les pixels, ignorant la chrominance. Elle est particulièrement utile pour des images avec beaucoup de bruit :

```
def cost_function_labDif(point1, point2, image_lab):
    L1, a1, b1 = image_lab[point1[0], point1[1]].astype(int)
    L2, a2, b2 = image_lab[point2[0], point2[1]].astype(int)
    return abs(L1 - L2)
```

### 5.3 | Fonction de Coût Basée sur l'Intensité Lumineuse

Cette fonction normalise les valeurs de luminance et calcule la différence d'intensité lumineuse entre les pixels :

```
def cost_function_intensity(point1, point2, image_lab):
    intensity1 = image_lab[point1[0], point1[1], 0] / 255.0
    intensity2 = image_lab[point2[0], point2[1], 0] / 255.0
    return abs(intensity1 - intensity2)
```

### 5.4 | Fonction de Coût Basée sur le Contraste Local

Cette fonction peut être plus coûteuse en termes de calcul, surtout sur des images de grande résolution. Elle calcule le contraste local en se basant sur la moyenne de la luminance autour des points :

```
def cost_function_local_contrast(point1, point2, image_lab, window_size=5):
    # [Code pour calculer le contraste local]
    return local_contrast
```

Il est important de noter qu'il existe d'autres fonctions de coût et heuristiques qui peuvent être utilisées pour le calcul du plus court chemin dans les images. Le choix de la fonction de coût appropriée dépend fortement de la nature du problème à résoudre et du type d'image traitée.

Des heuristiques basées sur des caractéristiques spécifiques telles que les bords, les motifs ou les textures peuvent être employées pour des images complexes ou pour des applications nécessitant une analyse plus détaillée. De même, dans le cas d'images avec des variations de couleur ou de luminosité moins prononcées, des fonctions de coût simplifiées peuvent être plus efficaces.

Le développement de fonctions de coût personnalisées ou l'adaptation de méthodes existantes est également possible et peut être avantageux, surtout lorsque les images présentent des caractéristiques uniques ou lorsqu'il y a des exigences spécifiques en termes de précision ou de performance du traitement d'images. En résumé, bien que nous ayons implémenté des fonctions de coût spécifiques dans notre programme, l'exploration et l'expérimentation avec d'autres méthodes peuvent fournir des résultats plus optimisés, adaptés aux besoins spécifiques du projet et à la nature des images traitées.

## 5.5 | Nécessité d'Appliquer des Filtres sur les Images

Dans certains cas, l'application de filtres sur une image est nécessaire pour améliorer l'efficacité et la précision du calcul du plus court chemin. Cela est particulièrement vrai pour les images avec un niveau élevé de bruit, comme l'image du Portrait de Mona Lisa. Le bruit peut introduire des irrégularités et des variations de couleur ou de luminosité qui compliquent le calcul du chemin le plus court.

### 5.5.1 | Filtre de Lissage

Un exemple pertinent est l'utilisation d'un filtre de lissage. Ce type de filtre réduit le bruit en moyennant les valeurs des pixels environnants, ce qui a pour effet de lisser les transitions de couleur et de luminosité. En appliquant un filtre de lissage, on peut obtenir une image plus homogène où les différences de couleur et de luminosité sont moins abruptes.

### 5.5.2 | Avantages pour le Calcul du Plus Court Chemin

L'avantage d'appliquer un tel filtre avant le calcul du plus court chemin est double :

- 0. Réduction du Bruit :** Le lissage aide à minimiser les effets du bruit, rendant le chemin calculé plus représentatif de la structure réelle de l'image.
- 0. Simplification du Calcul :** En lissant les variations mineures, le calcul du chemin le plus court devient moins complexe et plus rapide, ce qui est particulièrement bénéfique pour les images de grande taille ou de haute résolution.

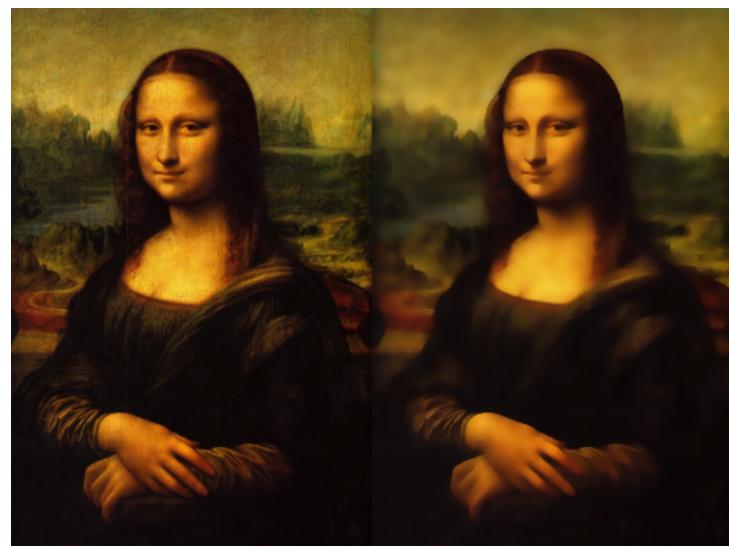
En conclusion, l'application de filtres appropriés, comme le filtre de lissage, peut être une étape cruciale pour améliorer la performance des algorithmes de calcul du plus court chemin, surtout dans les cas où l'image d'origine présente des défis tels que le bruit élevé ou des variations de couleur complexes.

La fonction suivante a été développée pour appliquer un filtre gaussien, plus précisément un filtre bilatéral, sur une image :

```
def apply_bilateral_filter(image_rgb, d=9, sigmaColor=75, sigmaSpace=75):  
    return cv2.bilateralFilter(image_rgb, d, sigmaColor, sigmaSpace)
```

Voici un exemple de ce filtre appliqué sur l'image de Mona Lisa :

Avec ce filtre, nous avons réduit le bruit sur l'image.



**Figure 3:** A gauche l'image originale, à droite l'image lissée.

## 6 | Dijkstra pour le Calcul du Plus Court Chemin

### 6.1 | Implémentation de l'Algorithme de Dijkstra

L'algorithme de Dijkstra est implémenté pour trouver le chemin le plus court dans une image. La fonction principale est la suivante :

```

1 def dijkstra(image_lab, start, end, cost_function):
2     height, width = image_lab.shape[:2]
3     visited = np.full((height, width), False, dtype=bool)
4     distance_map = np.full((height, width), np.inf)
5     parent_map = np.full((height, width, 2), -1, dtype=int)
6     distance_map[start] = 0
7     priority_queue = [(0, start)]
8     while priority_queue:
9         dist, current_node = heapq.heappop(priority_queue)
10        if visited[current_node]:
11            continue
12        visited[current_node] = True
13        if current_node == end:
14            break
15        for dx, dy in [
16            (0, 1),
17            (1, 0),
18            (0, -1),
19            (-1, 0),
20            (1, 1),
21            (-1, 1),
22            (1, -1),
23            (-1, -1),
24        ]:
25            # 8-connexité
26            new_x = current_node[1] + dx
27            new_y = current_node[0] + dy
28            neighbor = (new_y, new_x)
29
30            if 0 <= new_x < width and 0 <= new_y < height and not visited[neighbor]:
31                cost = cost_function(current_node, neighbor, image_lab)
32                new_dist = dist + cost
33
34                if new_dist < distance_map[neighbor]:
35                    distance_map[neighbor] = new_dist
36                    parent_map[neighbor] = current_node
37                    heapq.heappush(priority_queue, (new_dist, neighbor))
38
39        path = [end]
40        while path[-1] != start:
41            parent = tuple(parent_map[path[-1]])
42            path.append(parent)
43
44    return path[::-1]

```

Cette fonction utilise une image transformée en espace Lab, des points de départ et d'arrivée, ainsi qu'une fonction de coût spécifique pour calculer le chemin le plus court.

Dans mon programme, l'algorithme de Dijkstra ainsi que les autres fonctions nécessaires au calcul du chemin le plus court sont appliqués sur l'image après l'avoir transformée dans l'espace colorimétrique Lab et après l'avoir lissée. Cependant, l'utilisateur ne visualise pas cette version modifiée de l'image, car le programme affiche l'image originale lors du rendu des résultats.

## 6.2 | Fonction pour Trouver et Afficher le Chemin le Plus Court

En plus de l'algorithme de Dijkstra, une fonction supplémentaire est utilisée pour trouver et afficher le chemin le plus court sur l'image :

```

def find_shortest_path():
    # Traitement des points de départ et d'arrivée
    # Appel de l'algorithme de Dijkstra avec une fonction de coût spécifique
    # Dessin du chemin sur l'image et affichage
    # Affichage des instructions et du chemin dans le terminal

```

Cette fonction gère la transformation des coordonnées, appelle l'algorithme de Dijkstra avec une fonction de coût potentiellement coûteuse en temps de calcul, comme `cost_function_local_contrast`, dessine le chemin trouvé sur l'image originale et met à jour l'affichage.

## 6.3 | Note sur les Fonctions de Coût

Il est à noter que différentes fonctions de coût peuvent être utilisées selon les besoins spécifiques de l'application. Par exemple, `cost_function_local_contrast` peut être longue à s'exécuter sur des images de grande taille ou complexes. L'utilisateur doit donc choisir la fonction de coût en fonction des caractéristiques de l'image et des exigences de performance.

# 7 | Optimisation de la Fonction Dijkstra dans le Projet

Dans le cadre de l'amélioration de notre programme de traitement d'image, nous avons effectué une modification significative dans l'implémentation de l'algorithme de Dijkstra.

## 7.1 | Changement de PriorityQueue à heapq

Initialement, notre fonction Dijkstra utilisait la structure de données `PriorityQueue` pour gérer la file de priorité. Cependant, pour optimiser la performance, nous avons décidé de remplacer `PriorityQueue` par `heapq`.

### 7.1.1 | Raisons du Changement

Les raisons principales de ce changement sont :

**Performance Améliorée** : En pratique, `heapq` offre une meilleure performance pour notre cas d'utilisation. Étant donné que notre application est mono-thread et ne nécessite pas les fonctionnalités de sécurité des threads offertes par `PriorityQueue`, `heapq` est plus rapide pour manipuler les petites opérations de file de priorité.

**Simplicité et Efficacité** : `heapq` est plus simple à utiliser et plus direct dans son approche. Cette simplicité se traduit par une efficacité accrue dans la gestion de la file de priorité, ce qui est crucial pour l'algorithme de Dijkstra, en particulier lors du traitement d'images de grande taille.

### 7.1.2 | Impact sur l'Algorithme

Ce changement a eu un impact positif sur l'exécution de l'algorithme de Dijkstra dans notre programme. Nous avons observé une accélération notable dans le calcul du plus court chemin, ce qui est particulièrement bénéfique dans le cadre de notre projet, où de nombreuses images doivent être traitées efficacement.

## 7.2 | Conclusion

Le passage de `PriorityQueue` à `heapq` dans notre fonction Dijkstra illustre notre engagement à optimiser constamment notre code pour une meilleure performance et efficacité. Ce choix technique reflète notre attention aux détails et notre volonté d'adopter des solutions qui améliorent l'expérience utilisateur finale.

# 8 | Création du Graphe à partir de l'Image

Une partie cruciale de notre programme implique la conversion d'une image en un graphe pour le calcul du plus court chemin. La fonction `create_graph` joue un rôle central dans ce processus.

## 8.1 | Fonction `create_graph`

La fonction `create_graph` transforme une image dans l'espace colorimétrique Lab en un graphe où chaque pixel est un sommet. Voici un aperçu de son fonctionnement :

```
def create_graph(image_lab):
    graph = {}
    height, width = image_lab.shape[:2]

    for y in range(height):
        for x in range(width):
            neighbors = []
            if x > 0:
                neighbors.append((y, x - 1))
            if y > 0:
                neighbors.append((y - 1, x))
            graph[(y, x)] = neighbors
    return graph
```

Dans cette fonction, chaque pixel est connecté à ses voisins immédiats pour former le graphe. Les coordonnées (x, y) de chaque pixel sont utilisées comme clés dans un dictionnaire

### 8.1.1 | Visualisation des Sommets

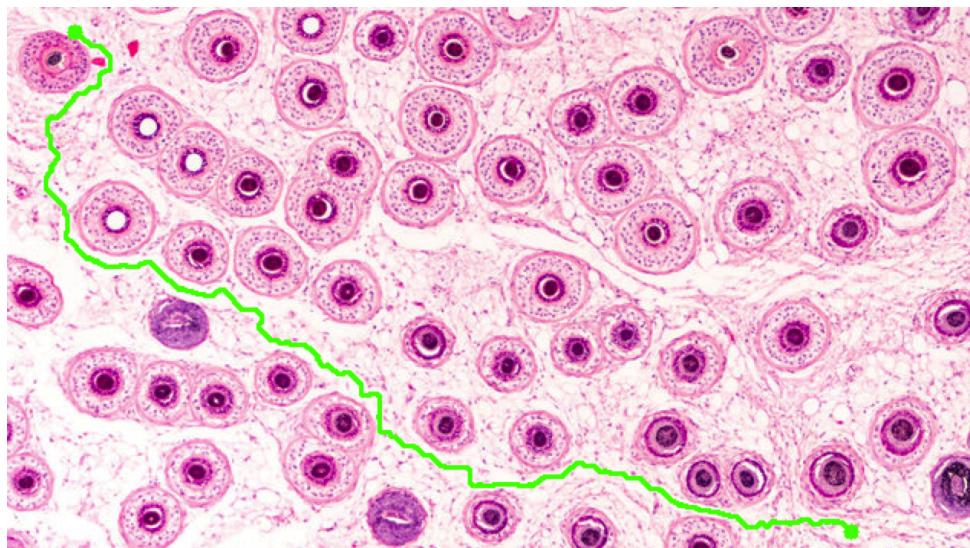
Pour des fins de débogage ou de visualisation, il est possible d'afficher chaque sommet (pixel) en vert sur l'image originale. Cependant, cette étape est généralement omise dans l'exécution standard du programme pour des raisons de performance.

## 8.2 | Autres Méthodes pour l'Interface Graphique

Outre `create_graph`, notre programme comprend d'autres méthodes dédiées à la gestion de l'interface graphique. Ces méthodes facilitent l'interaction avec l'utilisateur et permettent une visualisation plus intuitive des résultats du traitement d'image. Bien que ces fonctions ne soient pas détaillées ici, elles sont essentielles pour offrir une expérience utilisateur complète et sont disponibles dans le code source du programme.

## 9 | Exécution du programme

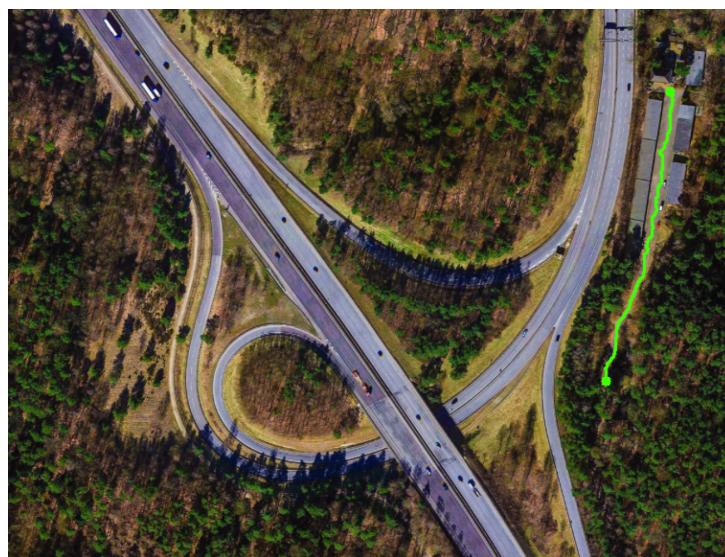
Voici un exemple de à quoi ressemble l'exécution de notre programme avec l'image ci-dessous :



**Figure 4:** Illustration du plus court chemin calculé sur une image microscopique.

L'image microscopique traitée démontre l'application d'un algorithme de chemin le plus court qui exploite l'intensité des pixels comme fonction de coût. L'algorithme parvient à tracer un chemin, représenté par une ligne verte, qui navigue en quelque sorte autour des cellules, des structures roses. Ce choix de fonction de coût permet à l'algorithme de distinguer les régions de haute intensité et d'opter pour des passages à travers des zones présentant une densité moindre. Cette méthode d'évitement reflète la capacité de l'algorithme à identifier et à contourner des obstacles.

Autre exemple pour également bien voir cette démonstration :



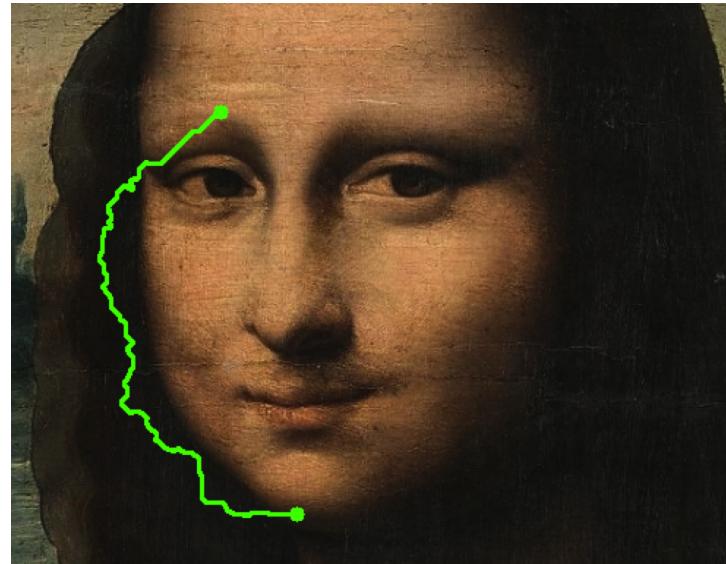
**Figure 5:** Illustration du plus court chemin calculé sur une image satellite.

Ici on peut bien voir que le chemin suit bien le chemin de terre sur cette vue satellite, et évite de passer par les zones de plus hautes intensité, c'est-à-dire les regroupements d'arbres.



**Figure 6:** Illustration du plus court chemin calculé sur une image de labyrinthe.

Ici l'algorithme trouve bien le chemin le plus court, afin de sortir du labyrinthe.



**Figure 7:** Illustration du plus court chemin calculé sur une image de la Joconde.

Sur cette image de la Joconde, on peut voir que le chemin calculé va éviter de parcourir les pixels à forte intensité, zone où le bruit est beaucoup présent en raison de la nature principale de cette oeuvre riche en détails. Le chemin va donc se détourner vers des régions plus homogènes et uniformes de l'image, notamment les cheveux, où la variabilité des pixels est moindre.

# References

- [1] <https://perso.esiee.fr/~perretb/I5FM/TAI/connexity/index.html>.
- [2] <https://medium.com/@sampeach/how-to-use-pathfinding-algorithms-with-satellite-images-part-2-77562d4a94f3>.
- [3] <https://dsc-courses.github.io/dsc30-2023-su/projects/shortest-paths/>.