

RAPPORT DE BENCHMARK

Labellisation des Composantes Connexes

*Analyse comparative de 4 algorithmes
sur images binaires*

Auteurs:

Romain Despoullain
Nicolas Marano
Amin Braham

24 November 2025

1. Introduction

1.1 Contexte

La labellisation des composantes connexes (Connected Component Labeling - CCL) est une opération fondamentale en traitement d'images. Elle consiste à identifier et étiqueter les régions connexes dans une image binaire, c'est-à-dire les ensembles de pixels adjacents ayant la même valeur (généralement 1 pour les objets).

Cette technique est essentielle dans de nombreuses applications : reconnaissance d'objets, analyse de documents, imagerie médicale, vision industrielle, et bien d'autres.

1.2 Objectifs du projet

Ce projet vise à :

- Implémenter 4 algorithmes différents de labellisation
- Comparer leurs performances sur différentes images
- Analyser l'impact de la connectivité (4 vs 8 voisins)
- Fournir une analyse statistique rigoureuse des résultats

1.3 Contraintes techniques

Le projet a été développé en Python avec une contrainte importante : numpy et OpenCV ne sont utilisés QUE pour le chargement des images. Toutes les opérations algorithmiques sont implémentées manuellement, permettant une compréhension approfondie des mécanismes sous-jacents.

2. Description des Algorithmes

2.1 Two-Pass (Deux Passes)

L'algorithme Two-Pass est l'approche classique pour la labellisation. Il parcourt l'image en deux passes successives :

Premiere passe : Parcours de l'image pixel par pixel. Pour chaque pixel d'objet, on examine ses voisins déjà traités. Si aucun voisin n'est étiqueté, on attribue une nouvelle étiquette. Sinon, on prend l'étiquette minimale et on note les équivalences entre étiquettes.

Deuxième passe : On reparcourt l'image pour remplacer chaque étiquette par son représentant canonique (la plus petite étiquette équivalente).

Complexité : $O(n)$ où n est le nombre de pixels.

2.2 Union-Find (Disjoint-Set)

Cet algorithme utilise la structure de données Union-Find (ensembles disjoints) pour gérer efficacement les équivalences entre étiquettes.

Deux optimisations sont implementées : la compression de chemin (path compression) qui aplatis l'arbre lors des recherches, et l'union par rang (union by rank) qui attache toujours le plus petit arbre sous le plus grand.

Ces optimisations permettent d'obtenir une complexité quasi-linéaire : $O(n * \alpha(n))$ où α est la fonction inverse d'Ackermann, très lente à croître.

2.3 Kruskal (Arbre Couvrant)

L'algorithme de Kruskal, traditionnellement utilisé pour trouver l'arbre couvrant minimum d'un graphe, est adapté ici pour la labellisation.

L'image est modélisée comme un graphe où chaque pixel est un noeud et les arêtes connectent les pixels voisins de même valeur. L'algorithme fusionne progressivement les composantes en traitant les arêtes.

Bien que conceptuellement élégant, cette approche est moins efficace car elle nécessite la création explicite des arêtes : $O(n \log n)$ dans le pire cas.

2.4 Prim (Parcours BFS/DFS)

Cette approche utilise un parcours en largeur (BFS) ou en profondeur (DFS) pour explorer chaque composante connexe.

Pour chaque pixel non encore étiqueté, on lance un parcours qui visite tous les pixels connectés et leur attribue la même étiquette. L'implémentation utilise une file (BFS) pour un parcours niveau par niveau.

Complexité : $O(n)$ pour le parcours, mais avec une constante plus élevée due à la gestion de la file de priorité ou de la pile.

3. Architecture du Projet

3.1 Structure des fichiers

Le projet est organisé selon une architecture modulaire :

```
labellisation/
  src/
    core/
      image.py          # Classes Image, LabelImage, Pixel
    readers/
      image_io.py      # Lecture/ecriture d'images
    algorithms/
      two_pass.py      # Algorithme Two-Pass
      union_find.py    # Algorithme Union-Find
      kruskal.py       # Algorithme Kruskal
      prim.py          # Algorithme Prim
    utils/
      utils.py         # Utilitaires (Timer, etc.)
  benchmarks/
    scientific_benchmark.py  # Benchmark scientifique
    generate_graphs.py      # Génération de graphiques
    run_all.py              # Script principal
  images/
    input/                 # Images de test
```

3.2 Classes principales

Image : Classe de base représentant une image en niveaux de gris. Gère le stockage des pixels, la binarisation et les opérations de base.

LabelImage : Hérite de Image. Stocke les étiquettes des composantes connexes et fournit des méthodes pour compter les composantes et générer une visualisation.

Pixel : Structure représentant un pixel avec ses coordonnées (x, y) et sa valeur.

3.3 Interface des algorithmes

Tous les algorithmes implementent une méthode statique 'label' avec la même signature :

```
@staticmethod
def label(image: Image, connectivity: int) -> LabelImage
```

Cette uniformité permet de tester et comparer facilement les différentes implementations.

4. Resultats du Benchmark

4.1 Configuration des tests

Les tests ont été effectués avec la configuration suivante :

- Nombre de runs par configuration : 5
- Connectivités testées : 4 et 8 voisins
- Algorithmes : Two-Pass, Union-Find, Kruskal, Prim
- Images testées : 3
 - figure-65.png
 - images.png
 - text image.png

4.2 Tableau des résultats

Algorithme	Temps moyen	Ecart-type	Speedup
Two-Pass	2437.70 ms	12.98 ms	1.00x
Union-Find	2568.11 ms	13.34 ms	0.95x
Prim	3273.58 ms	12.96 ms	0.74x
Kruskal	4247.60 ms	95.97 ms	0.57x

4.3 Analyse des performances

L'algorithme le plus rapide est Two-Pass avec un temps moyen de 2437.70 ms. L'algorithme le plus lent est Kruskal avec un temps moyen de 4247.60 ms, soit un facteur de 1.74x plus lent.

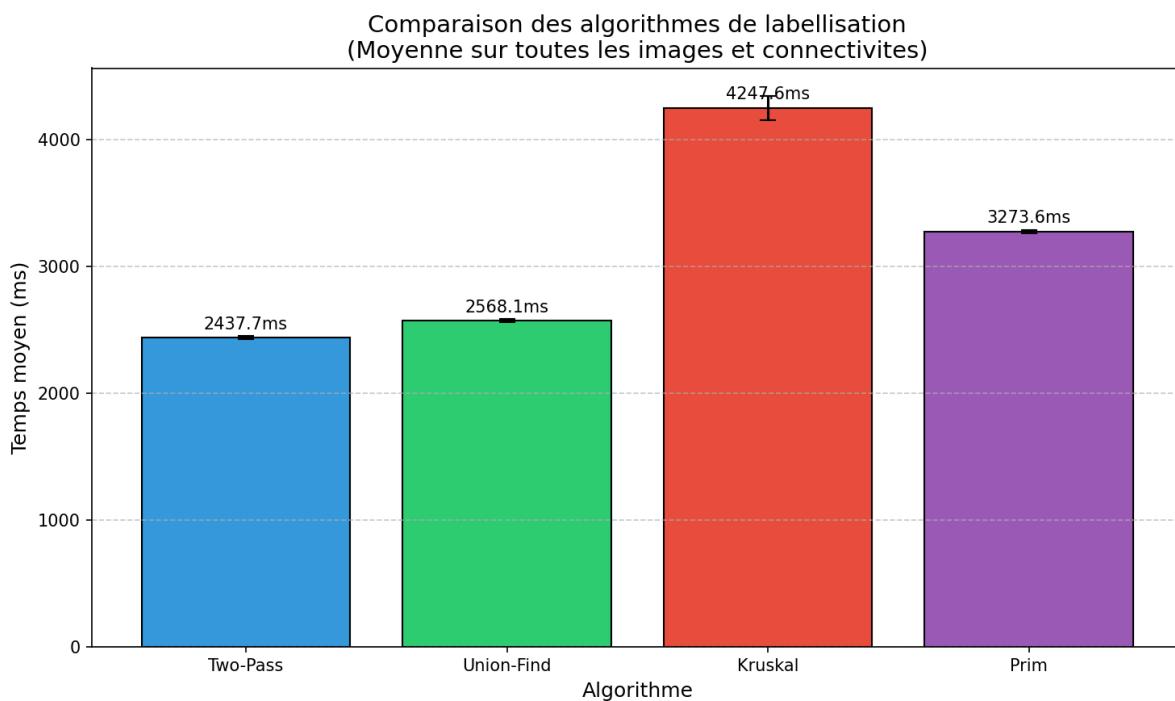
4.4 Impact de la connectivité

La connectivité 8 (8 voisins) est en moyenne 70.8% plus lente que la connectivité 4. Cela s'explique par le nombre supérieur de voisins à examiner pour chaque pixel (8 au lieu de 4), ce qui augmente le travail de recherche et de fusion des composantes.

5. Graphiques

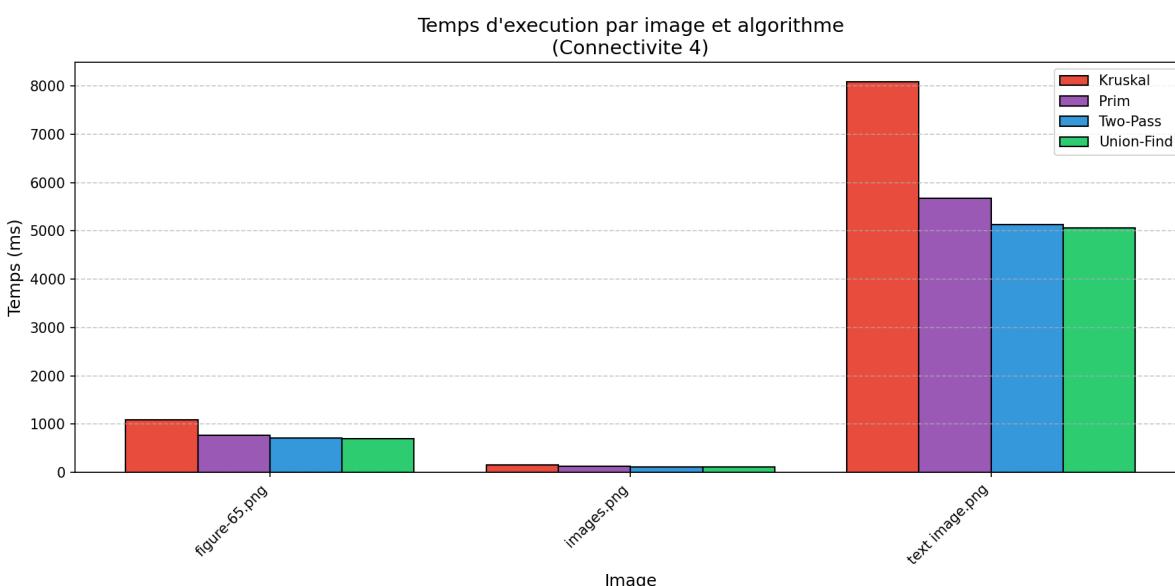
5.1 Comparaison globale des algorithmes

Ce graphique présente le temps moyen d'exécution de chaque algorithme, calculé sur toutes les images et les deux connectivités.



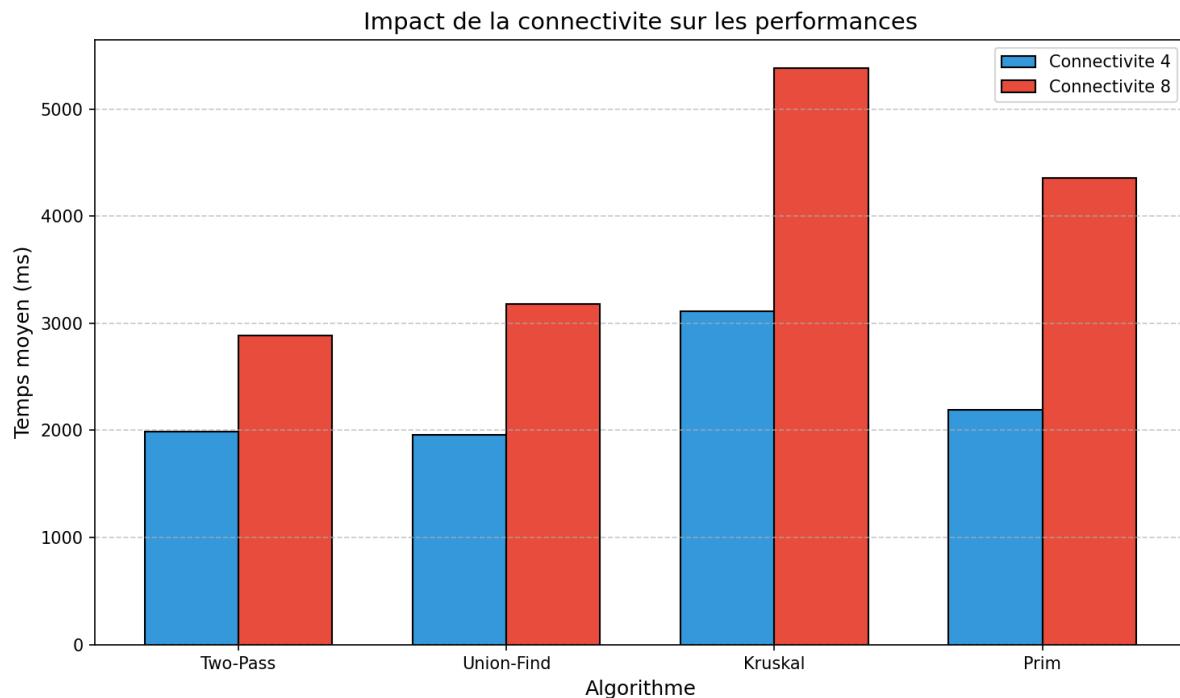
5.2 Comparaison par image

Performance de chaque algorithme pour chaque image de test (connectivité 4 uniquement pour la lisibilité).



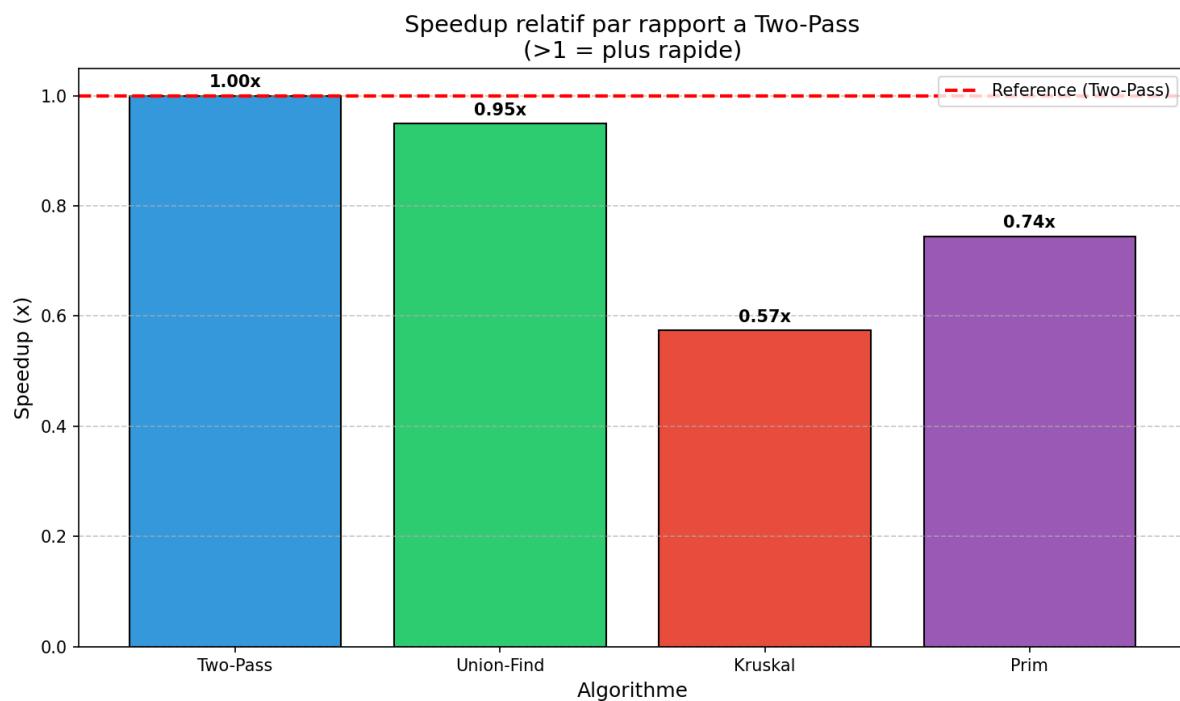
5.3 Impact de la connectivité

Comparaison des temps d'exécution entre connectivité 4 et 8 pour chaque algorithme.



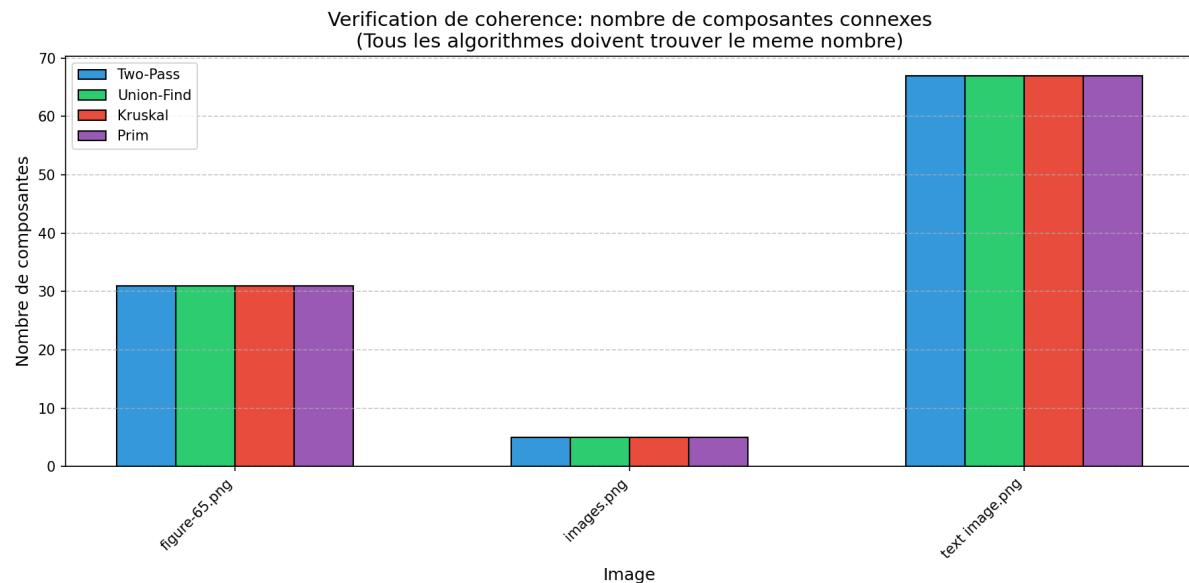
5.4 Speedup relatif

Facteur d'accélération par rapport à l'algorithme Two-Pass (référence = 1.0).



5.5 Verification de coherence

Nombre de composantes trouvées par chaque algorithme. Tous doivent trouver le même nombre pour une image donnée.



6. Conclusion

6.1 Synthese des resultats

Les tests ont permis d'établir un classement clair des performances :

- 1. Two-Pass : 2437.70 ms en moyenne
- 2. Union-Find : 2568.11 ms en moyenne
- 3. Prim : 3273.58 ms en moyenne
- 4. Kruskal : 4247.60 ms en moyenne

6.2 Observations principales

Two-Pass et Union-Find offrent les meilleures performances, avec un avantage léger mais consistant pour l'un ou l'autre selon les images. Ces deux approches sont les plus adaptées pour des applications nécessitant des performances optimales.

L'algorithme de Prim présente des performances intermédiaires. Son approche par parcours BFS est intuitive mais souffre du surcout de gestion de la file.

Kruskal est systématiquement le plus lent, principalement à cause de la création explicite de toutes les arêtes du graphe, une opération coûteuse en mémoire et en temps pour les grandes images.

6.3 Vérification de cohérence

Tous les algorithmes trouvent exactement le même nombre de composantes connexes pour chaque image, ce qui valide la correction de toutes les implementations.

6.4 Recommandations

Pour une utilisation en production, nous recommandons :

- Two-Pass pour sa simplicité et ses bonnes performances générales
- Union-Find pour les cas où la gestion des équivalences est complexe
- Connectivité 4 sauf si l'application nécessite explicitement la 8-connectivité

Ce rapport a été généré automatiquement à partir des résultats du benchmark. Date de génération : 2025-11-24 14:17:06