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 operation fondamentale en traitement d'images. Elle consiste a identifier et etiqueter les regions connexes dans une image binaire, c'est-a-dire les ensembles de pixels adjacents ayant la meme valeur (generalement 1 pour les objets).

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

1.2 Objectifs du projet

Ce projet vise a :

- Implementer 4 algorithmes differents de labellisation
- Comparer leurs performances sur differentes images
- Analyser l'impact de la connectivite (4 vs 8 voisins)
- Fournir une analyse statistique rigoureuse des resultats

1.3 Contraintes techniques

Le projet a ete developpe en Python avec une contrainte importante : numpy et OpenCV ne sont utilises QUE pour le chargement des images. Toutes les operations algorithmiques sont implementees manuellement, permettant une comprehension approfondie des mecanismes 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 deja traites. Si aucun voisin n'est etiquete, on attribue une nouvelle etiquette. Sinon, on prend l'etiquette minimale et on note les equivalences entre etiquettes.

Deuxieme passe : On reparcourt l'image pour remplacer chaque etiquette par son representant canonique (la plus petite etiquette equivalente).

Complexite : O(n) ou n est le nombre de pixels.

2.2 Union-Find (Disjoint-Set)

Cet algorithme utilise la structure de donnees Union-Find (ensembles disjoints) pour gerer efficacement les equivalences entre etiquettes.

Deux optimisations sont implementees : la compression de chemin (path compression) qui aplatit 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 complexite quasi-lineaire : O(n * alpha(n)) ou alpha est la fonction inverse d'Ackermann, tres lente a croitre.

2.3 Kruskal (Arbre Couvrant)

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

L'image est modelisee comme un graphe ou chaque pixel est un noeud et les aretes connectent les pixels voisins de meme valeur. L'algorithme fusionne progressivement les composantes en traitant les aretes.

Bien que conceptuellement elegant, cette approche est moins efficace car elle necessite la creation explicite des aretes : 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 etiquete, on lance un parcours qui visite tous les pixels connectes et leur attribue la meme etiquette. L'implementation utilise une file (BFS) pour un parcours niveau par niveau.

Complexite : O(n) pour le parcours, mais avec une constante plus elevee due a la gestion de la file de priorite ou de la pile.

3. Architecture du Projet

3.1 Structure des fichiers

Le projet est organise 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       # Generation de graphiques
    run_all.py              # Script principal
  images/
    input/                  # Images de test
```

3.2 Classes principales

Image : Classe de base representant une image en niveaux de gris. Gere le stockage des pixels, la binarisation et les operations de base.

LabelImage : Herite de Image. Stocke les etiquettes des composantes connexes et fournit des methodes pour compter les composantes et generer une visualisation.

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

3.3 Interface des algorithmes

Tous les algorithmes implementent une methode statique 'label' avec la meme signature :

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

Cette uniformite permet de tester et comparer facilement les differentes implementations.

4. Resultats du Benchmark

4.1 Configuration des tests

Les tests ont ete effectues avec la configuration suivante :

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

4.2 Tableau des resultats

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.

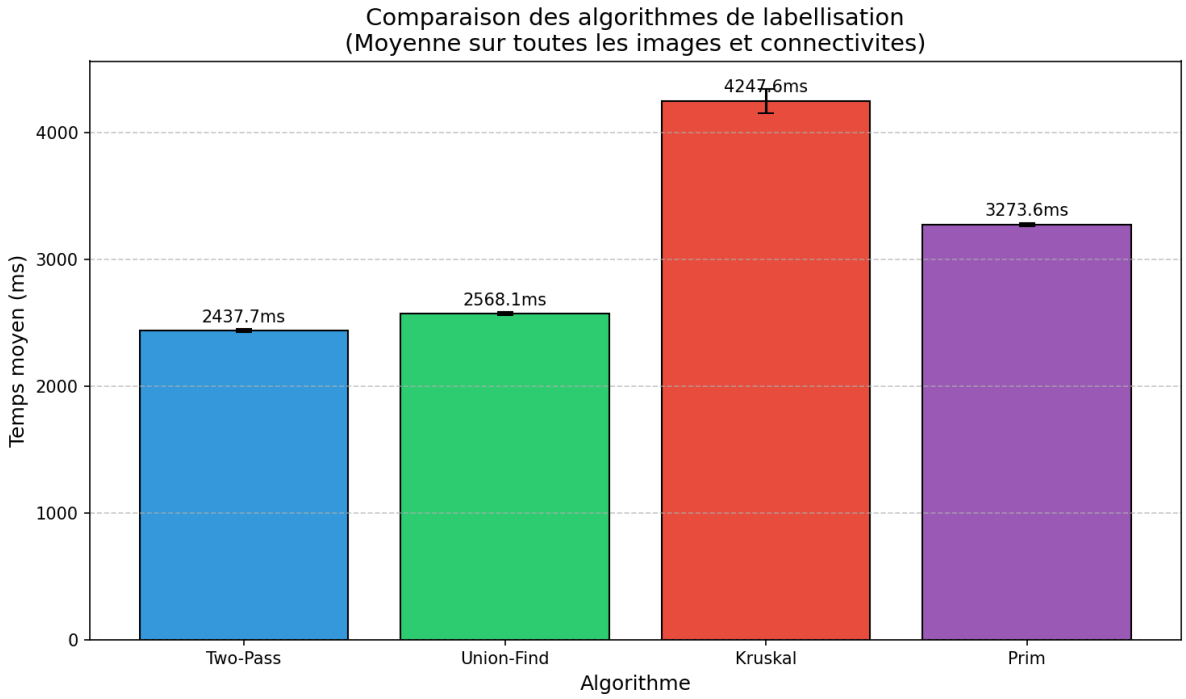
4.4 Impact de la connectivite

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

5. Graphiques

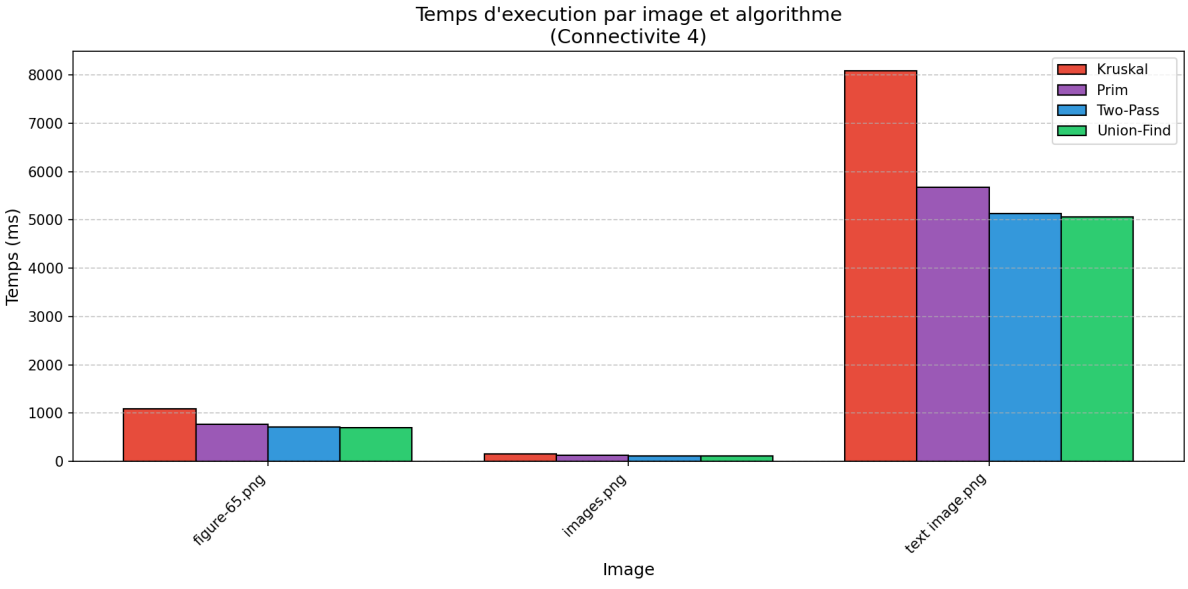
5.1 Comparaison globale des algorithmes

Ce graphique presente le temps moyen d'execution de chaque algorithme, calcule sur toutes les images et les deux connectivites.



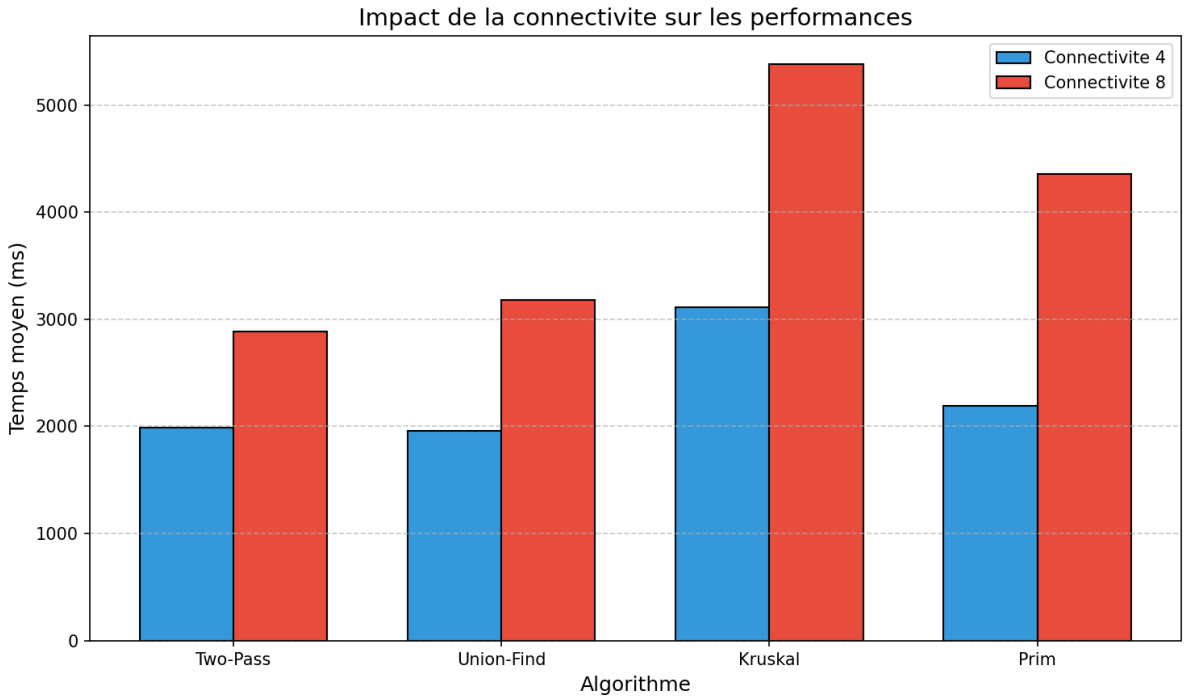
5.2 Comparaison par image

Performance de chaque algorithme pour chaque image de test (connectivite 4 uniquement pour la lisibilite).



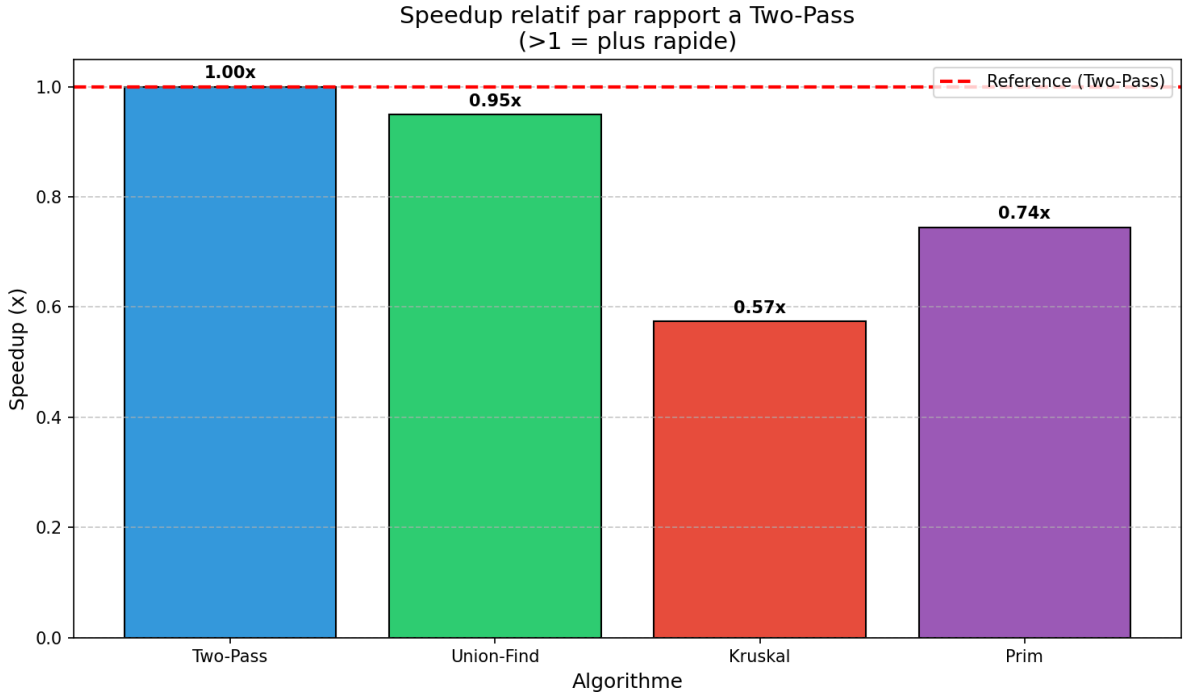
5.3 Impact de la connectivite

Comparaison des temps d'execution entre connectivite 4 et 8 pour chaque algorithme.



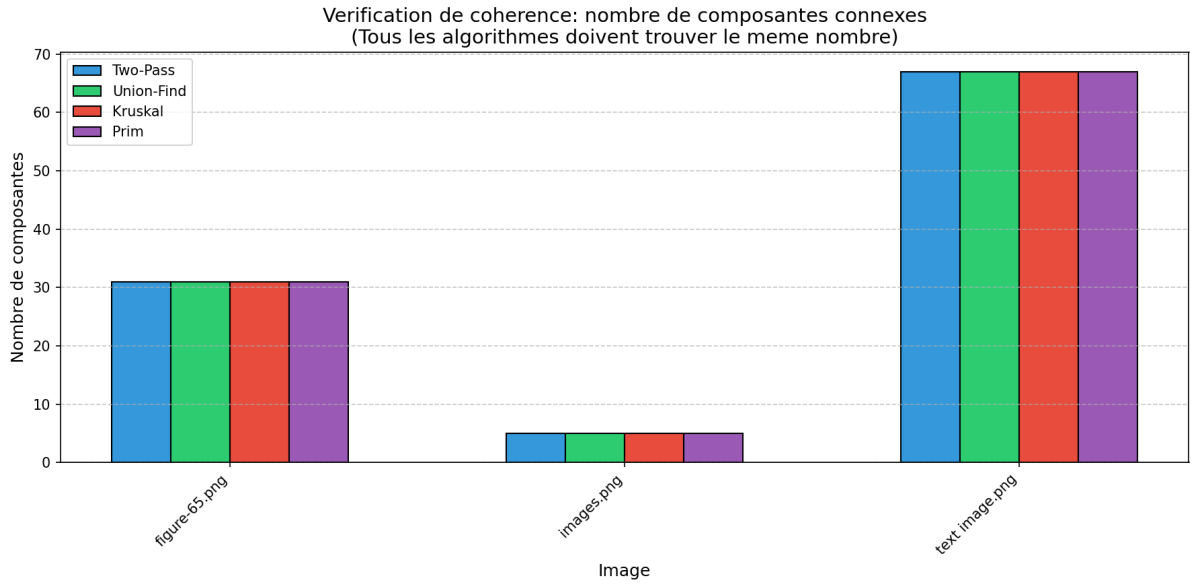
5.4 Speedup relatif

Facteur d'acceleration par rapport a l'algorithme Two-Pass (reference = 1.0).



5.5 Verification de coherence

Nombre de composantes trouvees par chaque algorithme. Tous doivent trouver le meme nombre pour une image donnee.



6. Conclusion

6.1 Synthese des resultats

Les tests ont permis d'etablir 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 leger mais consistant pour l'un ou l'autre selon les images. Ces deux approches sont les plus adaptees pour des applications necessitant des performances optimales.

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

Kruskal est systematiquement le plus lent, principalement a cause de la creation explicite de toutes les aretes du graphe, une operation couteuse en memoire et en temps pour les grandes images.

6.3 Verification de coherence

Tous les algorithmes trouvent exactement le meme 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 simplicite et ses bonnes performances generales
- Union-Find pour les cas ou la gestion des equivalences est complexe
- Connectivite 4 sauf si l'application necessite explicitement la 8-connectivite

Ce rapport a ete genere automatiquement a partir des resultats du benchmark. Date de generation : 2025-11-24 13:28:39