Création d'images par algorithme génétique avec référence

Rapport de projet

Lucien Cartier-Tilet

April 9, 2019

1 Sujet

Le sujet de ce projet est la création d'un logiciel pouvant recréer une image fournie grâce à des générations aléatoires et successives de formes aux, positions, couleurs et taille aléatoires. L'algorithme commence par créer une image vide aux dimensions identiques à l'image de référence, puis applique une de ces formes aléatoires. Si la ressemblance de l'image ainsi générée augmente par rapport à sa version précédente par rapport à l'image de référence, alors cette modification est conservée, sinon elle est annulée. Répéter jusqu'à satisfaction.

2 Les méthodes utilisées

Plusieurs approches au problème sont possibles, allant de la simple implémentation naïve du problème à des moyen pouvant au moins décupler la vitesse de génération de l'image. Sauf indication contraire, j'ai utilisé dans l'implémentation de chaque méthode des carrés comme forme d'éléments appliqués aléatoirement à l'image.

Pour évaluer la ressemblance entre deux image, j'évalue une distance euclidienne entre le vecteur de leurs pixels qui peut se résumer à ceci :

$$\sqrt{\sum_{i=0}^{n}(v_i - w_i)^2}$$

`V` étant le vecteur de pixels de l'image de référence, `W` étant le vecteur de pixels de l'image générée, et `n` la taille de ces deux vecteurs.

Les tests de temps sont réalisés sur un Lenovo Ideapad Y700, disposant d'un processeur Intel® Core™ i7-6700HQ à 2.6GHz et un turbo à 3.5GHz, composé de quatre cœurs supportant chacun deux threads, et de 16Go de RAM. Le programme est compilé avec les options d'optimisation `-O3` et `-flto`.

Voici également ci-dessous la liste des options et arguments possibles concernant l'exécution du logiciel.

```
$ ./bin/genetic-image -h
Allowed options:
  -h [ --help ]           Display this help message
  -i [ --input ] arg      Input image
  -o [ --output ] arg     Image output path (default: input path + "_output")
  -m [ --method ] arg     Method number to be used (default: 1)
  -n [ --iterations ] arg Number of iterations (default: 5000)
  -v [ --verbose ]        Enables verbosity
```

Voici le script grâce auquel les valeurs de temps d'exécution ont été obtenues :

```
#!/usr/bin/env fish
# This script was written to work with the fish shell. If you do not have the
# fish shell installed on your system, please install it before executing this
# script.
# The results will be stored in the output file `results.txt`
set nb_amelioration 10 50 100 200 500 1000
```

```
set nb_execution 200 100 50 20 10 5
set available_methods 1 2 3 4
set output results.txt
rm -f $output
for method in $available_methods
    set nb_options (count $nb_execution)
    echo "Method $method:" >> $output
    echo "|                    / | <                | <                       |" >> $output
    echo "| Nb d'améliorations | Nb d'exécutions | Temps d'exécution (s) |" >> $output
    echo "|--------------------+-----------------+-----------------------|" >> $output
    for i in (seq $nb_options)
        set total_exec_time 0
        set startexec (date +%s.%N)
        for j in $nb_execution[$i]
            ./build/bin/genetic-image -i ./img/mahakala-monochrome.jpg \
            -n $nb_amelioration[$i] -m $method
        end
        set endexec (date +%s.%N)
        set total_exec_time (math "$total_exec_time+($endexec-$startexec)/$i")
        echo "|$nb_amelioration[$i]|$nb_execution[$i]|$total_exec_time|" >> $output
    end
    echo "" >> $output
end
```

Quelques-unes de ces lignes commençasont là uniquement pour de la mise en forme des données afin que je puisse

2.1 Méthode naïve

J'ai tout d'abord implémenté la méthode naïve afin d'avoir une référence en matière de temps. Cette dernière est implémentée dans `src/methods.cc` avec la fonction `method1()`. Comme ce à quoi je m'attendais, cette méthode de génération d'images est très lente, principalement dû au fait que l'algorithme en l'état essaiera d'appliquer des couleurs n'existant pas dans l'image de référence, voire complètement à l'opposées de la palette de couleurs de l'image de référence.

Voici les moyennes de temps d'exécution selon le nombre d'itérations réussies sur le nombre d'exécutions indiqué.

Nb d'améliorations	Nb d'exécutions	Temps d'exécution (s)
10	200	0.065881
50	100	0.130041
100	50	0.186012
200	20	0.385982
500	10	1.437486
1000	5	3.608983

Naturellement, la variation en temps d'exécution croît en même temps que le nombre d'améliorations nécessaires à apporter à l'image à améliorer, dû à la nature aléatoire de l'algorithme. Cependant, on constate également une croissance importante du temps d'exécution suivant également ce nombre d'itérations réussies.

Vous trouverez en Annexes (§3.1.2) un exemple d'image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations via cette méthode.

2.2 Réduction du panel des couleurs

Constatant que la majorité des échecs d'ajout de formes de couleur par la première méthode échouent dû à une couleur incorrecte, voire n'appartenant pas à l'image de référence, j'ai décidé de restreindre les possibilités de couleurs parmis lesquelles le hasard peut choisir à la liste des couleurs présentes dans l'image de référence uniquement. Ce choix se fait donc via l'implémentation d'un set de valeurs uniques représentant

les couleurs trouvées dans l'image de référence, leur détection étant réalisée avec des threads parallèles pour plus de rapidité à l'exécution. Cette méthode est celle implémentée dans la fonction `method2()` dans `src/methods.cc`.

Voici les moyennes de temps d'exécution selon le nombre d'itérations réussies sur le nombre d'exécutions indiqué.

Nb d'améliorations	Nb d'exécutions	Temps d'exécution (s)
10	200	0.072979
50	100	0.114426
100	50	0.157965
200	20	0.290475
500	10	0.785426
1000	5	2.664046

On peut remarquer une amélioration quant à la rapidité d'exécution du logiciel. Cependant, le résultat n'est pas aussi important qu'escompté. Je suppose que cela est dû au fait que l'algorithme précédent peut considérer un rapprochement d'une zone déjà colorée vers la couleur d'origine comme une amélioration, avec une possibilité plus large sur ce plan-là que pour le second algorithme qui se doit d'être plus précis concernant les couleurs. Une nette amélioration du résultat est toutefois visibles, voir Annexes (§3.1.3) pour une image générée à partir de `img/mahakala-monochrome.png` via la méthode 2 et avec 2000 améliorations.

Étant donné que cette modification ne sera à priori pas en conflit avec d'autres méthodes, cette amélioration sera conservée pour toutes les autres avancées suivantes.

2.3 Une taille des formes aléatoire mais contrôlée

Une autre méthode peut être de contrôler la taille des éléments en spécifiant une taille minimale et maximale selon le nombre d'éléments posés et le nombre total d'éléments à poser. Ainsi, on pourrait privilégier tout d'abord de grandes formes en début de génération pour encourager petit à petit les formes à réduire en taille. Cela permettrait d'obtenir rapidement une représentation grossière de l'image pour ensuite pouvoir progressivement afiner les détails. La taille de la forme à appliquer est définie comme suit :

$$coef = \frac{nbIterRestantes}{totalIter}$$

$$tailleMinimale = coef\frac{min(Width, Height)}{2}$$

$$tailleMaximale = tailleMinimale * 2 + 1$$

$$taille = Rand(\llbracket tailleMinimale; tailleMaximale \rrbracket)$$

Voici les moyennes de temps d'exécution selon le nombre d'itérations réussies sur le nombre d'exécutions indiqué.

Nb d'améliorations	Nb d'exécutions	Temps d'exécution (s)
10	200	0.082068
50	100	0.244236
100	50	0.418075
200	20	1.453703
500	10	4.777205
1000	5	20.33209

Cette version du logiciel est nettement plus lente que ses versions précédentes du fait de la contrainte de taille pour les formes pouvant potentiellement améliorer l'image, cependant la qualité des images générées est plus haute que celle des version précédentes, voir en Annexes (§3.1.4).

Cette méthode ne me semble que moyennement concluante, certes la vitesse d'exécution du logiciel est beaucoup plus faible, mais il est également vrai que la qualité des images générées est supérieure aux deux autres méthodes. Ainsi, il sera possible d'utiliser les modifications apportées par cette méthode en utilisant une option `-s [ --size ]` avec les méthodes suivantes pour activer cette modification de l'algorithme.

2.4 Concurrence entre threads

Une utilisation de calculs parallèles pourrait être intéressante afin d'accélerer la génération des images : l'utilisation de threads mis en concurrence. À chaque tentative d'amélioration de l'image, plusieurs threads sont lancés, et chacun créera sa propre amélioration possible de l'image. Ces résultats sont récupérés et évalués, et parmi les résultats améliorant l'image générée, celle avec le meilleur score est conservée. Cela permet ainsi de multiplier les chances d'avoir une amélioration de l'image par tentative.

Voici les benchmarks d'exécution de cette méthode sans contrôle de la taille des formes aléatoires :

Nb d'améliorations	Nb d'exécutions	Temps d'exécution (s)
10	200	0.080525
50	100	0.139892
100	50	0.169113
200	20	0.273342
500	10	0.610812
1000	5	1.403816

Et voici les benchmarks d'exécution de cette même méthode avec contrôle de la taille des formes aléatoires :

Nb d'améliorations	Nb d'exécutions	Temps d'exécution (s)
10	200	0.085981
50	100	0.156099
100	50	0.29183
200	20	0.59844
500	10	2.513782
1000	5	6.457168

Pour résumer, ces deux tableaux montrent la parallélisation de la seconde méthode et de la troisième méthode respectivement via des threads concurrentiels. On peut remarquer que le temps d'exécution s'est nettement amélioré, avec un temps d'exécution à peu près deux fois plus rapide pour l'exécution sans contrôle de taille des formes que la seconde méthode, et pouvant être jusqu'à trois fois plus rapide que la troisième méthode avec le contrôle de la taille des formes activée. On a donc une véritable amélioration significative avec cette nouvelle version parallèle.

2.5 Collaboration entre threads

Une différente approche au parallélisme peut être réalisée : plutôt que d'essayer de mettre en concurrence plusieurs threads, il serait possible d'essayer de plutôt les mettre en collaboration. Cela implique par exemple de diviser l'image d'entrée en plusieurs zones sur laquelle chacun des threads lancés travailleraient, appliquant chacun le nombre d'améliorations demandé sur sa zone dédiée. Puis, une fois que chacun des threads a terminé son travail, les différentes zones sont unifiées en une seule image.

3 Annexes

3.1 Images

3.1.1 Image de référence

3.1.2 Méthode 1

3.1.3 Méthode 2

3.1.4 Méthode 3

3.1.5 Méthode 4

Taille des formes non contrôlée

Taille des formes contrôlée

Figure 1: Image de référence utilisée pour les tests du logiciel

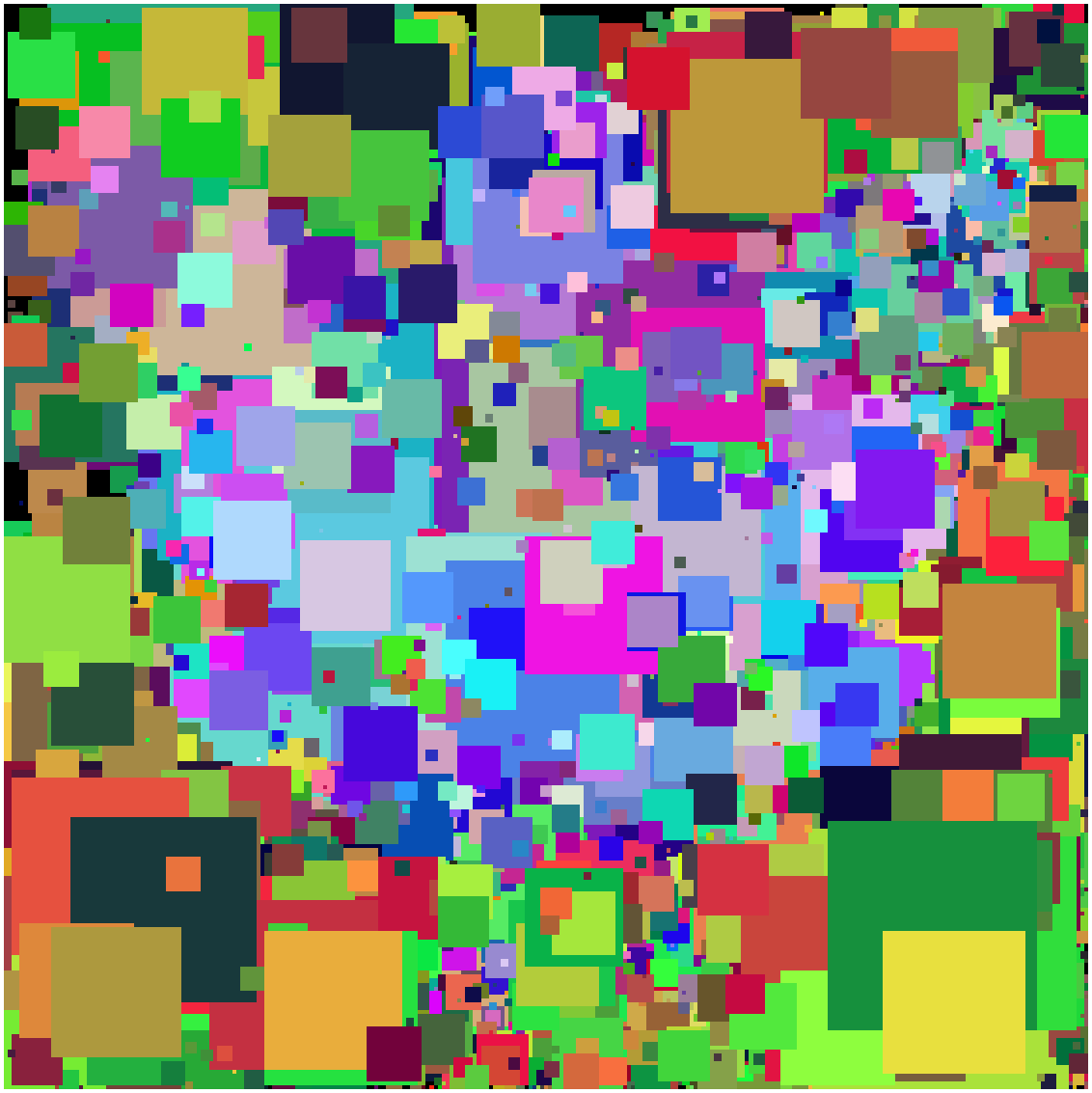

Figure 2: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la première méthode

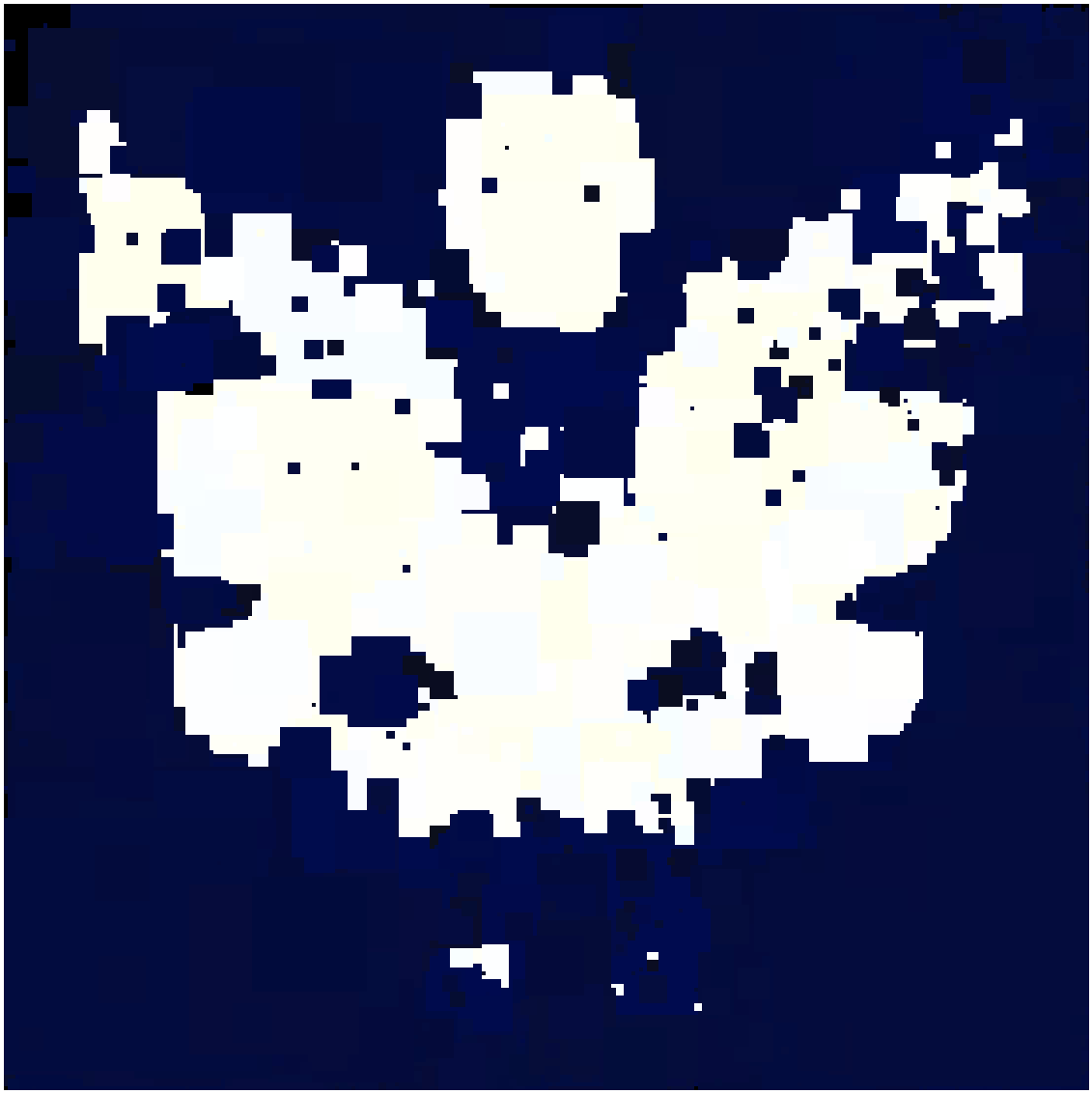

Figure 3: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la seconde méthode

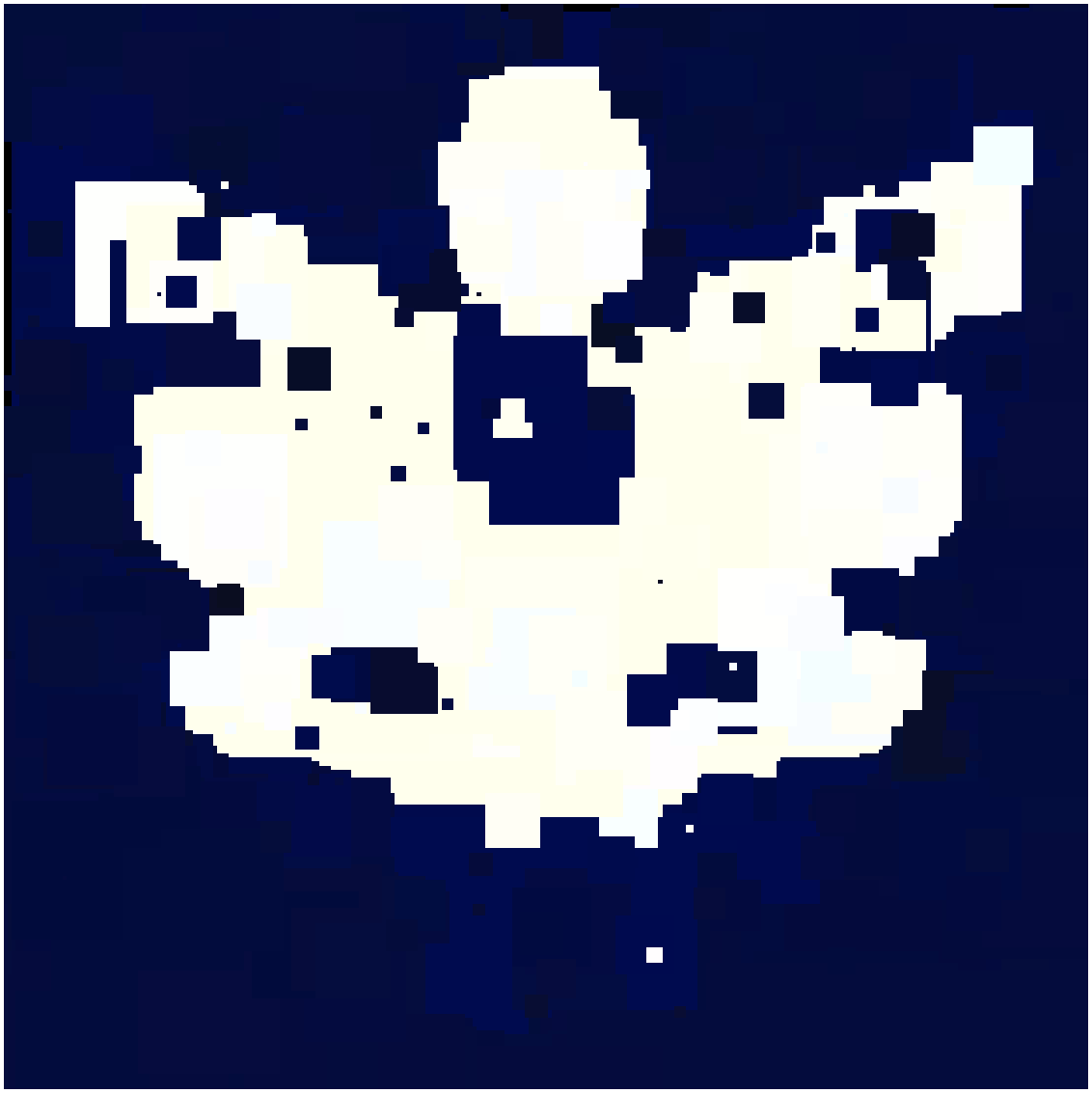

Figure 4: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la troisième méthode

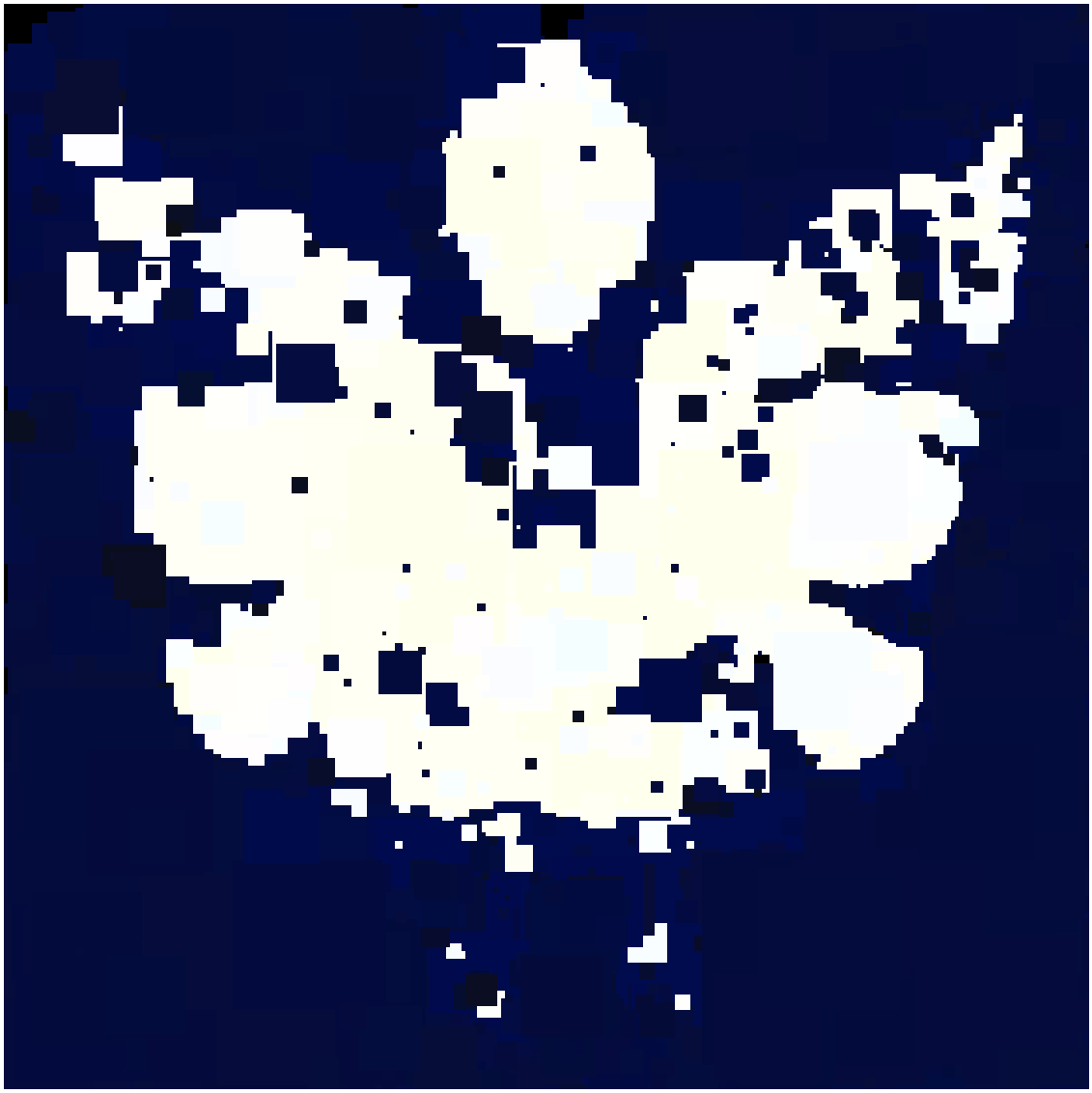

Figure 5: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la quatrième méthode sans l'option `-s`

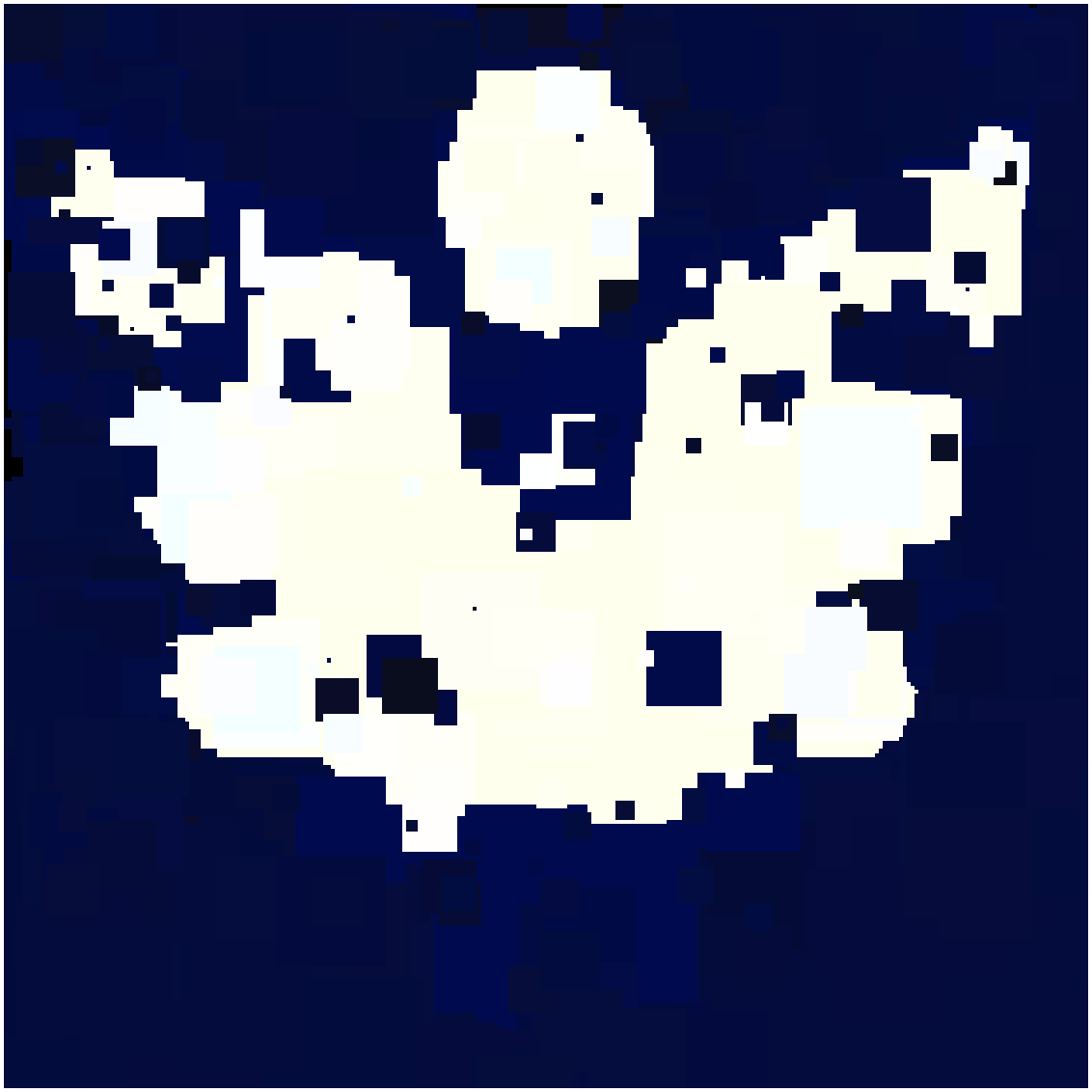

Figure 6: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la quatrième méthode avec l'option `-s`