Création d'images par algorithme génétique avec référence

Rapport de projet

Lucien Cartier-Tilet

March 25, 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 :

$$\sqrt{\sum_{i=0}^{n} V_i^2 + W_i^2}$$

`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 Thinkpad x220, disposant d'un processeur Intel® Core™ i5-2540M à 2.6GHz, composé de deux cœurs supportant chacun deux threads, et de 4Go 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 la ligne de commande utilisée depuis le répertoire `build` afin de pouvoir obtenir un temps d'exécution :

```
perf stat -r {nombreExécutions} -B ./bin/genetic-image \
     -i ../img/mahakala-monochrome.jpg -o output.png   \
     -n {nombreIterations} -m 1
```

Les deux éléments entre accolades sont à remplacer par leur valeur, par exemple afin d'exécuter dix fois le programme avec vingt améliorations, il faudrait exécuter ceci :

```
perf stat -r 1 -B ./bin/genetic-image \
     -i ../img/mahakala-monochrome.jpg -o output.png -n 20 -m 1
```

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	Temps d'exécution (s)	Variation (s)	Nb d'exécutions
10	0.060847	0.000498	200
50	0.29823	0.00453	100
100	0.7093	0.0135	50
200	1.9584	0.0559	20
500	8.739	0.291	10
1000	27.930	0.582	5

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 Annexe 1.1 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	Temps d'exécution (s)	Variation (s)	Nb d'exécutions
10	0.074951	0.000533	200
50	0.26385	0.00401	100
100	0.6385	0.0148	50
200	1.6145	0.0348	20
500	6.747	0.235	10
1000	19.608	0.327	5

On peut remarquer une très nette amélioration de la rapidité d'exécution du logiciel. 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.

2.4 Concurrence entre threads

Une utilisation na

3 Annexes

3.1 Images

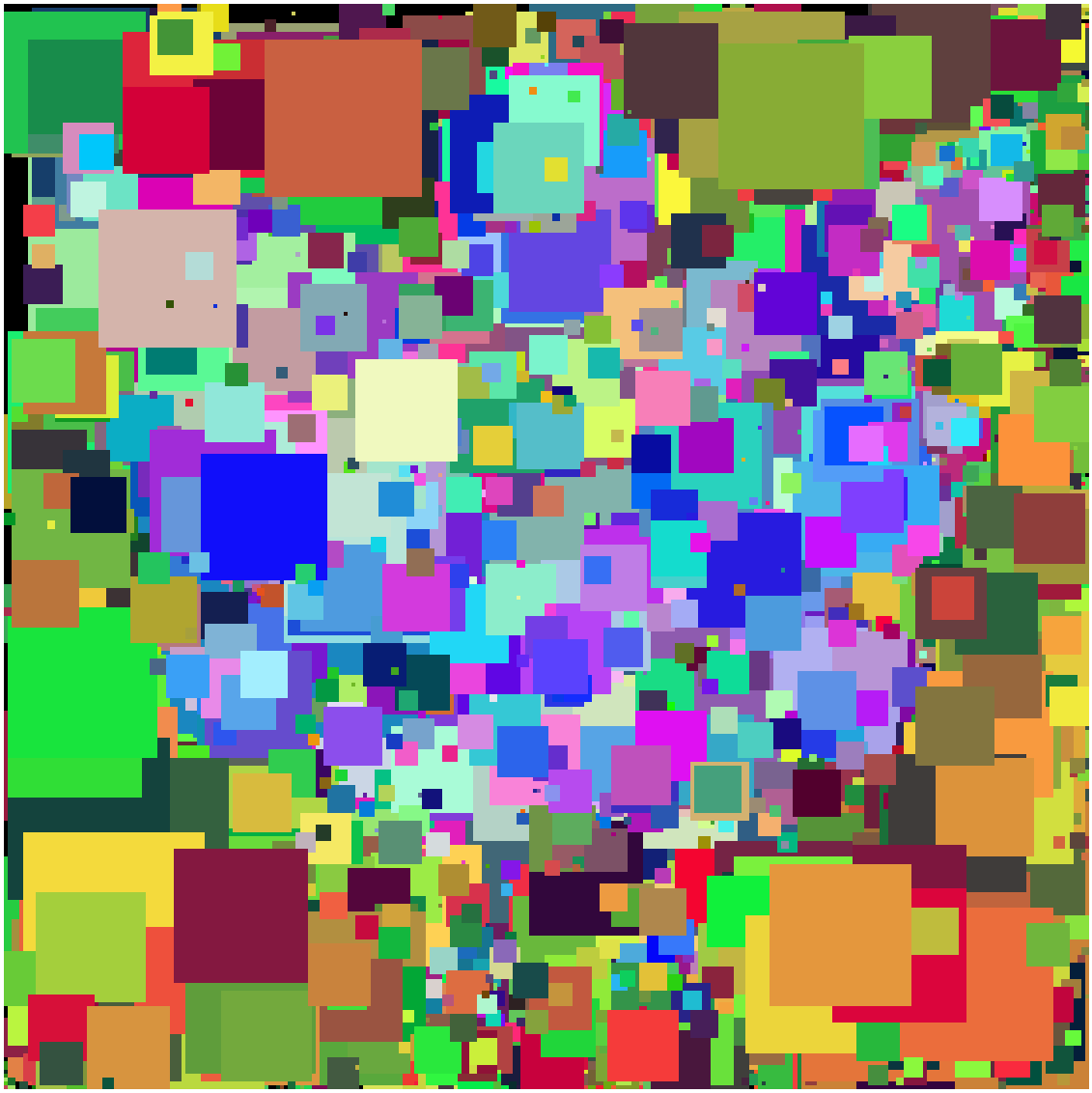

Figure 1: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la première méthode

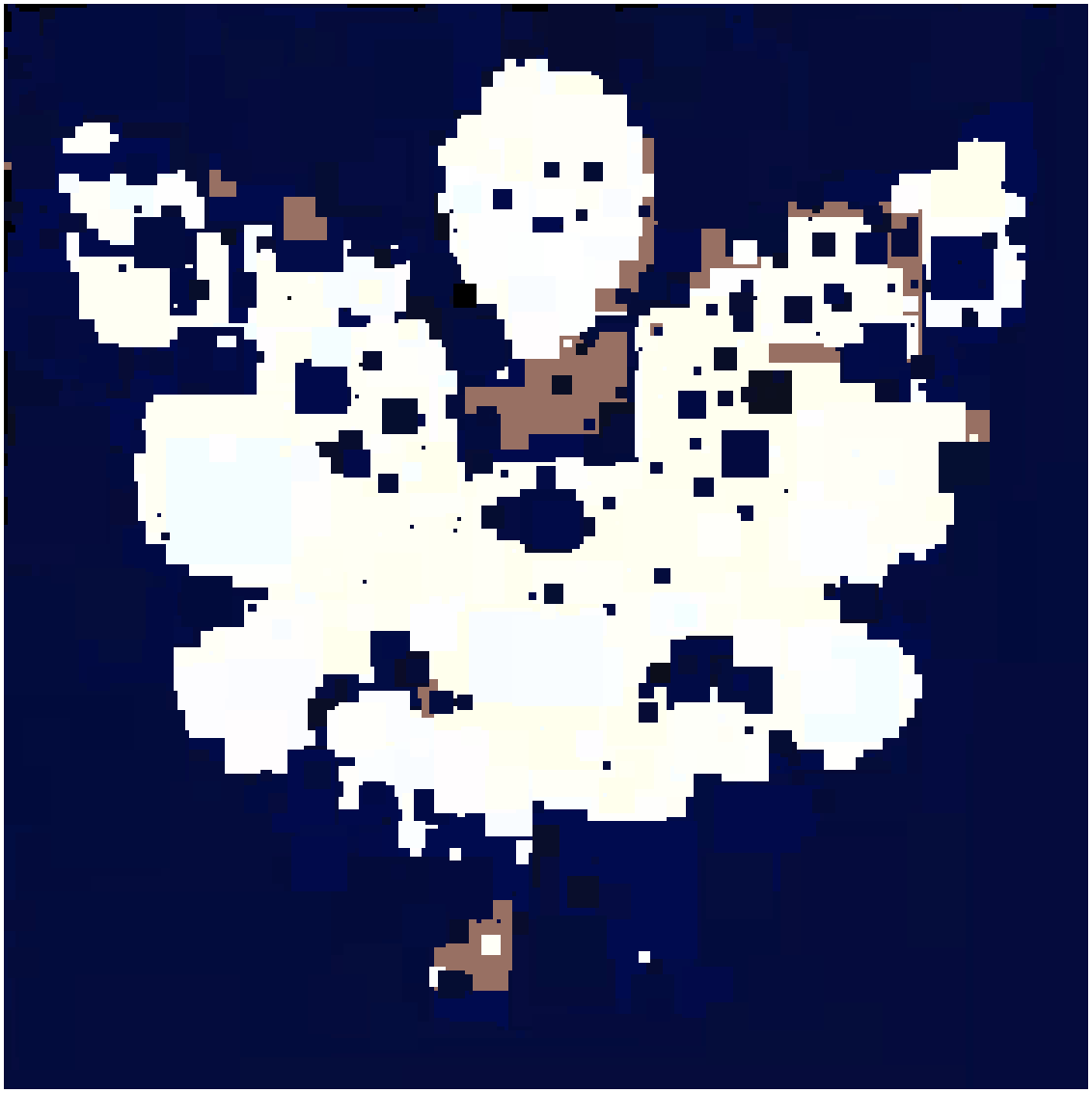

Figure 2: Image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations avec la seconde méthode