Algorithme Génétique
Générer une image

Le TP nous invite à créer un algorithme génétique dont l'objectif est de dessiner la Joconde. Ce document permettra de justifier nos choix ainsi que de mettre en avant les différentes pistes explorées.

Tout d'abord, nous rappellerons les principes des algorithmes génétiques. Ensuite, nous aborderons les technologies employées. Enfin, nous détaillerons notre algorithme pour chaque phase du cycle de vie.

I. Les algorithmes génétiques

Les algorithmes génétiques sont employés pour résoudre des problèmes complexes. En effet, leur puissance réside dans l'évolution d'une population. Comme on ne peut pas prédire le développement de la population initiale, on peut obtenir des résultats tout à fait différents. De même, les chemins explorés pour obtenir une même solution sont multiples.

Les algorithmes simulent un "cycle de vie". Nous devons initialiser une population d'individus. Ces derniers ont un génotype composé de gènes choisis arbitrairement. Par exemple, pour notre cas nous avons 4 gènes : le gène rouge, bleu, vert et alpha.

Pendant le cycle de vie, la première population devient des "parents" et ces individus se reproduisent. Nous retrouvons les principes liés à l'ADN tout au long de cette phase. .Ainsi, nous notons deux procédés. Le premier est de copier un parent en créant un enfant clone. Le second est similaire au crossing-over. On choisit deux parents de la population pour mélanger leurs gènes. Les enfants générés composent une nouvelle population potentielle.

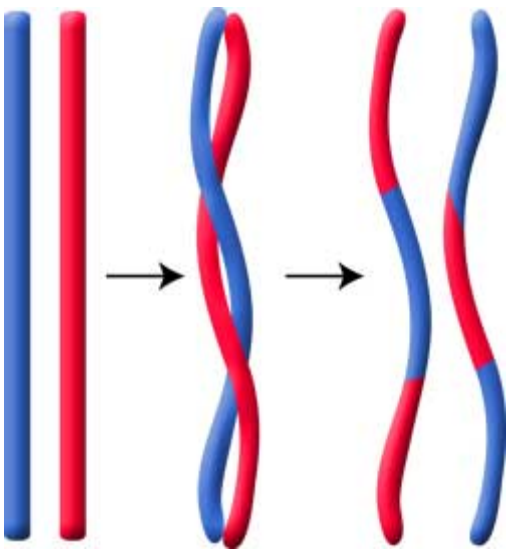


Schéma d'un crossing-over

Après la reproduction, une phase de mutation peut avoir lieu : des enfants sont pris au hasard et leurs gènes sont modifiés de façon aléatoire. Dans la vie réelle, les mutations sont plutôt rares, par contre en informatique, on peut définir un fort taux de probabilité.

On passe ensuite par une phase de survie. Durant cette dernière, on choisit quels parents et enfants composeront la seconde génération. Pour cela, on doit créer un système mesurant la fitness d'un individu : c'est une valeur attribuée par notre algorithme permettant de définir si on est proche ou non de la solution. En fonction de la fitness d'un individu, on peut choisir si on le garde ou non dans la nouvelle génération.

Ce cycle de vie est répété à chaque génération jusqu'à obtenir le résultat souhaité ou une fitness acceptable.

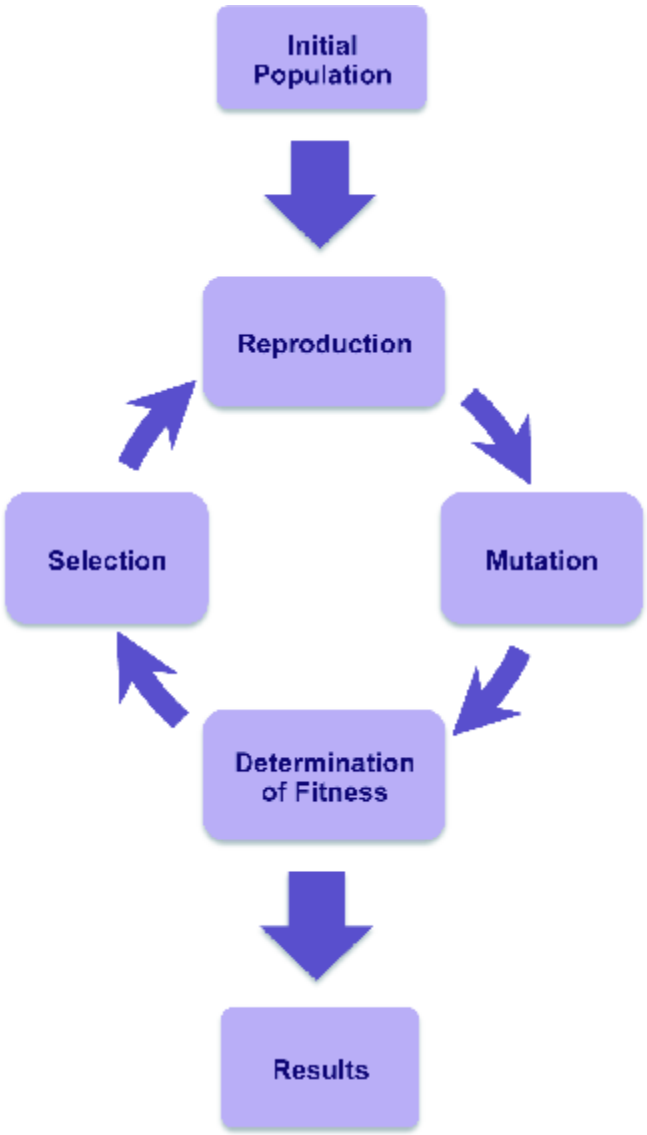


Schéma du cycle de vie

Le choix de la modélisation des chromosomes est crucial. En effet, moins on choisit d'axes et plus il sera simple de trouver une solution. De plus, si les chromosomes sont mal déterminés alors on peut ne pas aboutir à une solution acceptable.

La fonction administrant la fitness est très importante également. Une mauvaise évaluation d'une population peut fausser les résultats.

Plusieurs stratégies peuvent être adaptées :

- On peut opter pour une solution complètement aléatoire si on ne sait pas exactement le résultat qu'on souhaite obtenir.

- Au contraire, si la solution est bien définie, on peut procéder avec le principe du tournois (le meilleur reste en vie) ou avec un raisonnement élitiste (les meilleurs sont sélectionnés pour se reproduire).

Une fois la stratégie choisie, on peut encore jouer sur des paramètre comme le taux de probabilité qu'un crossing-over ou qu'une mutation s'opère sur la population.

Les algorithmes génétiques s'inspirent des principes biologiques liés à l'évolution des espèces. Les résultats obtenus peuvent être très variés pour un même problème.

II. Les choix techniques

A. Choix du langage

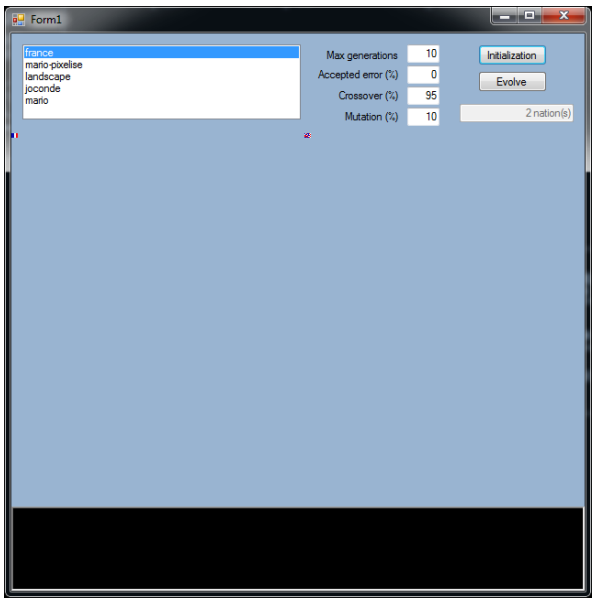
Il fut conseillé en classe d'utiliser du javascript pour suivre l'évolution de la population en temps réel. De part le choix de notre algorithme nous avons opté pour le PHP dans un premier temps. En effet, le javascript s'exécutant au niveau du navigateur, nous avons préféré employer la puissance de calcul des serveurs. Nous aurions pu choisir le C++ pour optimiser au maximum mais nous souhaitions garder l'aspect user-friendly du web.

Nous avons essayé d'optimiser les traitements au maximum. La première version de notre algorithme était structurée sous la logique d'un langage objet. La taille mémoire employée étant devenue trop importante, nous somme alors passés à une version procédurale. L'espace mémoire employé et les temps de traitement ont été grandement réduis, allant de moitié pour une image de moins de 35 pixels (plus les images étaient grandes et plus les économies étaient notables).

Au vue de la lenteur des calculs pour les grandes images, nous avons décidés de nous tourner vers une solution en C#. Ce langage étant plus compilé que le PHP ou le JS, nous avons à disposition une vitesse de traitement plus important. Nous retrouvons une hiérarchie orientée objet, facilitant ainsi la lecture de notre programme. Enfin, nous

gardons l'aspect agréable pour l'utilisateur avec une interface graphique. Nous avons souhaité présenter une interface pour jouer plus facilement avec les paramètres.

Nous avons également noté que C# possédait une bibliothèque très riche pour les traitements d'images, à la différence de C++. Nous avons donc décidé de jouir pleinement de cette opportunité.



L'interface de notre programme

Pour une image de 40*35 pixels, la version web met environ 5 minutes pour 10 générations contre 10 secondes pour la version compilée. Ces résultats nous ont encouragé à préférer le C# pour la suite.

Autre point essentiel, nous avons cherché à rendre les traitements les plus rapides possible. Nous avons donc utilisé un système de thread que nous détaillerons avec notre algorithme un peu plus tard.

B. Stockage des images pour la version PHP

Nous devions analyser tous les pixels de l'image traitée et cela consommait beaucoup de temps dans l'algorithme. Nous avions alors décidé de stocker leurs valeurs dans un fichier texte. Ce dernier était déposé dans le dossier "img_map/". Si l'image avait déjà été étudiée, alors nous récupérions les pixels avec leur code RGBA (Red green Blue Alpha) plus rapidement, au lieu de parcourir tous les pixels et les traiter à chaque nouvelle tentative.

La population solution était utilisée pour générer une image dans le dossier "_results/". On pouvait ainsi comparer plus facilement l'image de départ avec celle créée. Dès qu'un cycle de vie était finie, nous étions averti par un email pour relancer immédiatement après un nouveau test.

Ce système a été abandonné lorsque nous sommes passés sous C#.

III. Notre algorithme

A. Initialisation

Vous trouverez ci-dessous le détail de toutes les décisions choisies. Vous remarquerez que nos choix sont très élitistes puisque nous avions un objectif précis.

1. L'ADN

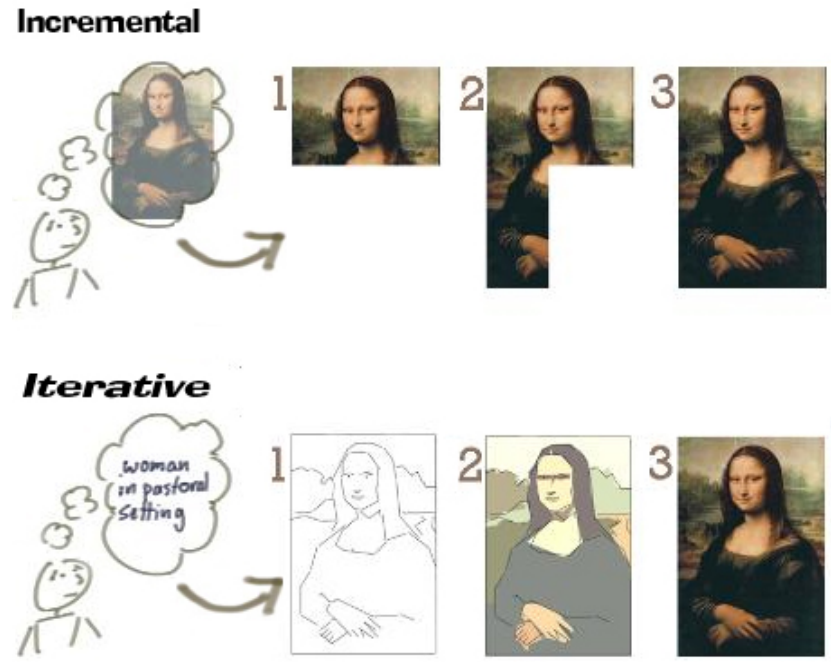
Un individu (Inidivual) possède un génome composé de 4 gènes : red, green, blue et alpha, soient 4 axes sur lesquels jouer. On retrouve le schéma classique ARGB. Cette classe étant déjà implémentée par c#, nous avons remarqué que l'axe alpha de l'opacité était défini sur l'intervalle [0 ; 255] à la différence de l'usuel [0 ; 1]. Nous avons gardé ce modèle, rendant ainsi les 4 axes homogènes.

2. Les nations et les populations

Nous appelons "population" un ensemble d'individus. La taille de la population est égale au nombre de pixels composant l'image.

Face aux grandes images, le temps de traitement devenait exponentiellement proportionnel à leur taille. Nous avons alors choisi la solution suivante : nous découpons l'image en zones, en bandeaux horizontaux pour être plus précis. Nous appelons cela un "continent" pour garder un vocabulaire plus "Humain". Sur chaque continent est créée une population. Cette dernière appliquera notre algorithme génétique et le temps de traitement sera diminué drastiquement. Nous définissons donc une "nation" comme étant un ensemble de populations.

Cette idée de nation est d'autant plus réalisable grâce aux threads. En effet, lorsque nous lançons nos cycles de vie, nous créons un thread par population. L'image générée par l'algorithme se compose de façon incrémentale et pas itérative



Incémentation ou itération ?

3. L'évaluation de la fitness

Le calcul de la fitness est la somme de la distance absolue sur les 4 axes.
Par exemple, si on a le pixel original "pixel_goal" avec un génome { 100, 125, 150, 50 } et un pixel généré "pixel_genetic" { 50, 25, 75, 70 }, la fitness est égale à :

$$fitness = |\ 100 - 50\ | + |\ 125 - 25\ | + |\ 150 - 75\ | + |\ 50 - 70\ | \quad = 245$$

Plus la fitness est basse et plus on est proche du résultat. La fitness totale de la population est la somme de la fitness de chaque pixel comparé avec le pixel_goal du même emplacement.

4. La palette de couleur

La palette de couleurs est le terme que nous avons choisi pour définir les différentes couleurs présentes dans l'image ou dans une population.

Grâce à la palette, nous initialisons la population : création d'autant d'individus que de pixels auxquels on attribue comme ADN une couleur de la palette.

Comme nous sommes pour l'égalité des couleurs, nous créons au moins un individu de chaque couleur de la palette. Nous complétons ensuite la population avec des couleurs de la palette sélectionnées aléatoirement.

B. Sélection et reproduction

Concernant la sélection et la reproduction, nous avons pris en compte les 2 options possibles.

- Le clonage cherche le parent le plus proche du résultat souhaité. Si la couleur solution n'est pas présente dans la population, alors nous nous rapprochons au maximum de cette dernière. Nous choisissons donc le parent avec la meilleure fitness.

- Le crossing-over sélectionne également un parent optimum. Le second parent est obtenu grâce à une fonction random. Nous avions fait l'erreur de choisir les deux parents les plus proches de la solution, mais ceci nous rapprochait sans jamais obtenir la solution. L'aspect aléatoire permet d'espérer que les deux parents puissent compenser mutuellement l'éloignement de l'autre sur chaque axe, créant ainsi un enfant avec le meilleur bagage génétique légué par ses parents.

Nous avons les meilleurs résultats avec une probabilité qu'un crossing-over apparaisse fixé à 95 %.

La phase de mutation a été désignée comme la phase la plus aléatoire. A l'origine, nous laissions l'algorithme choisir sur combien de gène il intervenait et il attribuait une valeur aléatoire. Nous avons alors réfléchi en prenant comme exemple un peinte. Ce dernier utilisera dans tous les cas une couleur de sa palette. Nous avons donc décidé de garder le côté aléatoire mais en prenant une couleur de la palette cette fois-ci. Un taux de mutation à 10 % nous donne des résultats intéressants.

C. Survie

Pour finir, la survie reprend le principe du "tournois". Les deux individus, l'un de la population des parents et l'autre de la population des enfants, situés au même endroit sont comparés grâce à leur fitness. Celui qui est le plus proche du résultat est gardé. En cas d'égalité, nous gardons le plus jeune (ce choix est arbitraire mais sans conséquence).

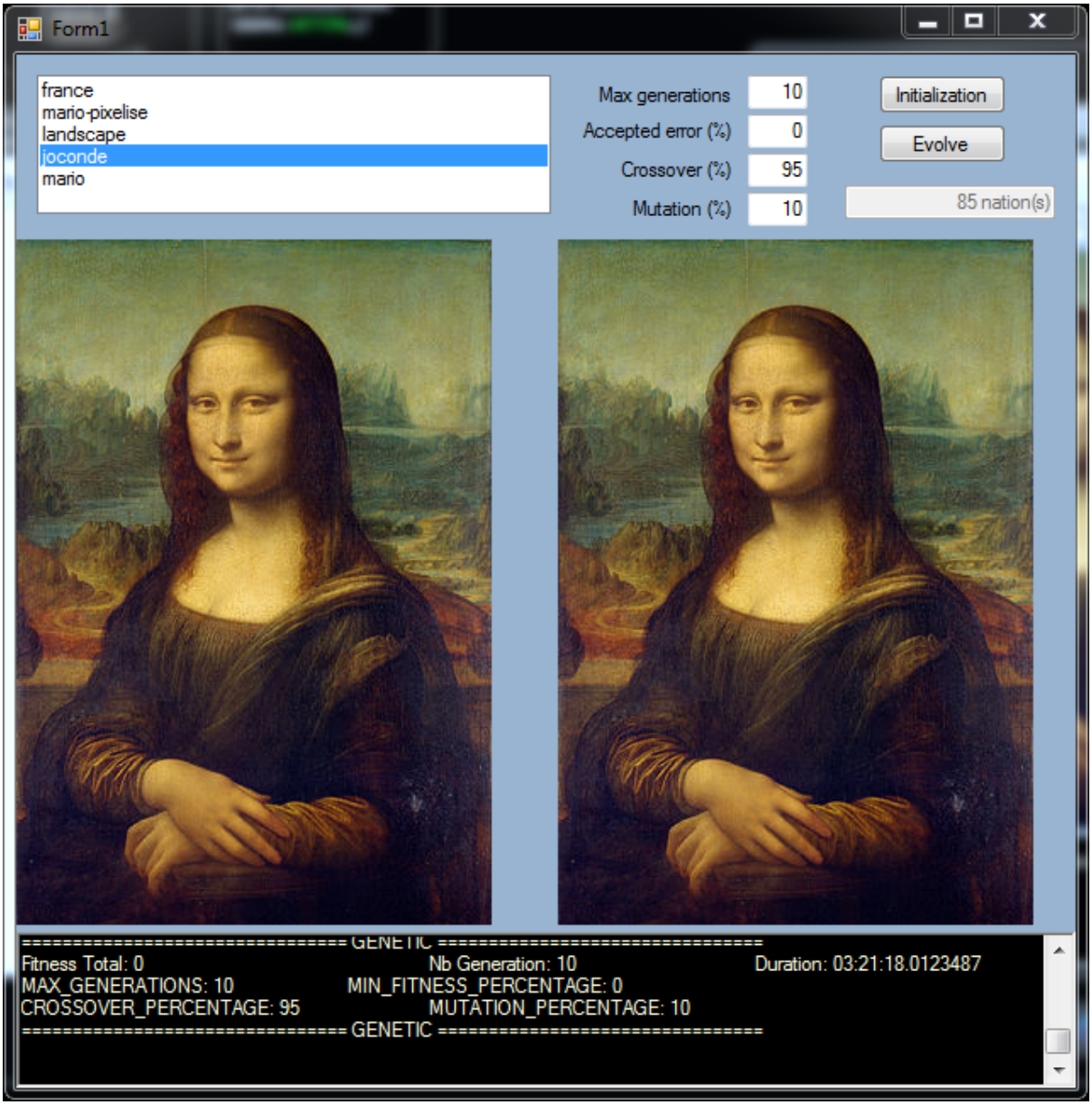
D. Bilan

En termes de résultats, notre algorithme permet de générer parfaitement (fitness à zéro) la Joconde en 3 heures et 21 minutes. Vous pouvez observer ci-dessous le résultat. Nous avons à gauche l'image de départ et à droite l'image générée.

Nous avons parfois une fitness non parfaite. Nous avions un delta de 8000 à 1H30min pour notre premier essai avec la Joconde. Cependant l'écart est réparti sur tous les pixels.

8000 / 280 * 417 = 0,06

Si on répartie l'erreur, on aurait pour tous les pixels une erreur de 0.06 sur un de ses axes (A, R, G ou B). Comme les axes sont une échelle de [0 ; 255], on pourrait dire que l'erreur est négligeable. A l'oeil nu, il est impossible de voir la différence.



Nous avons également testé sur une grande image mais avec moins de nuances de couleur qu'avec la Joconde. Le résultat est intitulé "mario_end_0" et est disponible dans le dossier "results". Nous mettons moins de temps que pour la Joconde puisque la palette de couleur est moins riche. Pour toute notre gamme de tests, nous obtenons une fitness parfaite.

Les algorithmes génétiques sont très intéressants à développer de part la liberté des choix. Les évolutions de chaque population et les résultats peuvent être assez surprenants. Au final, les algorithmes génétiques sont particulièrement adaptés pour

résoudre des problèmes complexes. Une longue phase de réflexion est conseillé pour la modélisation du système.

Arthur LAMBERT et Kim SAVAROCHE