Projet MasterMind

1 Introduction

Ce document est mon rapport sur le projet consistant à étudier le problème de *MasterMind*. J'ai choisi d'étudier la version de *MasterMind* avec n **cases et** n **couleurs**, pour $n \geq 3$. Par la suite, la notation n désignera **toujours** le nombre de cases (et donc aussi le nombre de couleurs).

1.1 Algorithmes étudiés

Dans le cadre du projet, j'ai étudié les algorithmes suivants :

- *Randomized Local Search* (RLS)
- *Evolutionary Algorithms* (EA), plus précisément la **variante** $(\mu + \lambda)$**-EA**. J'ai également implémenté (μ, λ)-EA), mais je ne l'ai pas étudié.
- *Genetic Algorithms* (GA). Il existe de nombreuses variantes. J'ai étudié exclusivement la **variante qui croise des parents avec une probabilité** c**, et qui applique une mutation standard avec une probabilité** $1 - c$.

Il ne s'agit que d'algorithmes élitistes. Etant donné que le problème à maximiser (*MasterMind*) ne possède pas de **maximum local**, il est possible de penser que des algorithmes non élitistes ne peuvent qu'être moins performant relativement à leurs voisins élitistes. C'est pour cette raison que j'ai fait le choix de ne pas les étudier dans mon projet.

1.2 Problématique choisie

Dans le cadre de ce projet, j'ai fait le choix d'étudier les trois algorithmes précédents. Plus spécifiquement, j'ai étudié l'impact des paramètres sur les performances des algorithmes.

Les paramètres des différents algorithmes sont :

- RLS : Aucun paramètre
- EA : μ la taille de la population des parents, λ la taille de la population des enfants, p le taux de mutation.
- GA :μ la taille de la population des parents, λ la taille de la population des enfants, p le taux de mutation, c le taux de croisement, et le type de croisement (en pratique, je n'ai testé que le croisement uniforme).

1.3 Implémentation

J'ai implémenté les algorithmes et les expériences menées en C++. J'ai fait le choix **de privilégier la structure du code**, plutôt que l'optimisation de la performance. Plus précisément, j'ai implémenté les algorithmes étudiées **de manière totalement indépendante du problème étudié** (le *MasterMind*). *MasterMind* n'est, dans mon code, qu'un

paramètre des algorithmes. Je n'ai pas écrit ces algorithmes spécifiquement pour le problème étudié. Bien que j'ai fait attention d'optimiser les performances du code, j'en ai un peu perdu à cause de ce choix.

2 Taux de mutation et de croisement

EA dépend d'un taux de mutation (*mutation rate*), et GA dépend à la fois d'un taux de mutation et d'un taux de croisement (*crossover rate*). Mais quelles valeurs choisir pour ces deux taux ? Faut-il une valeur proche de 1, ou au contraire, une valeur proche de 0 ? C'est la problématique qu'on va étudier dans cette section.

2.1 Formalisation du problème

Considérons qu'on étudie un algorithme $\mathcal{A}$ (dans notre cas, ce sera EA ou GA) qui dépend d'un paramètre Π (dans notre cas, ce sera l'un des deux taux).

Notons T_p la variable aléatoire désignant le nombre de requêtes effectués par l'algorithme $\mathcal{A}$ étudié sur le problème de *MasterMind* avec p comme valeur pour Π. Notons μ_p l'espérance de T_p.

On cherche la valeur du paramètre qui minimise le nombre de requêtes en moyenne, on cherche donc p qui minimise μ_p, *ie* $p^* := \text{argmin}_p \mu_p$.

2.2 Expérience « Optimisation de paramètre »

Pour trouver la valeur optimale du paramètre Π parmi un ensemble E fini, on va en déterminer un intervalle de confiance de $100\alpha\%$ (par exemple, $\alpha = 0.95$). Pour cela, considérons cette expérience :

- Fixer un entier m. Il impactera sur la largeur de l'intervalle.
- Pour chaque valeur $p \in E$:
  - résoudre m instances de *MasterMind* avec l'algorithme $\mathcal{A}$ étudié en prenant p comme paramètre. On obtient $T_p^{(1)}$, $T_p^{(2)}$, ..., $T_p^{(m)}$ les nombres de requêtes des m expériences.
  - Calculer la moyenne empirique $\bar{T_p} := \frac{1}{m}\sum_{i=1}^{m} T_p^{(i)}$ et l'estimateur sans biais de la variance $S^2 := \frac{1}{n-1}\sum_{i=1}^{m}(T_p^{(i)} - \bar{T_p})^2$.
  - Calculer $\alpha_p := \bar{T_p} - \frac{S}{\sqrt{m}} t_{\alpha/2}^{m-1}$ et $\beta_p := \bar{T_p} - \frac{S}{\sqrt{m}} t_{\alpha/2}^{m-1}$ où t_γ^k est le quantile d'ordre $1-\gamma$ de la loi de Student à k degrés de liberté.
- Calculer le seuil $\lambda := \min_p \beta_p$.
- Calculer $p_{\min} := \min\{p \in E : \alpha_p \leq \lambda\}$ et $p_{\max} := \max\{p \in E : \alpha_p \leq \lambda\}$

- Retourner l'intervalle $[p_{\min}, p_{\max}]$.

Explications :

Pour chaque valeur $p \in E$, d'après la **loi de Student** (généralisation du théorème central limite, pour quand la variance théorique est inconnue), on a

$$\mathbb{P}[\alpha_p \leq \mu_p \leq \beta_p] = 1 - \alpha$$

Rappel : on recherche $p^* = \text{argmin}_{p \in E} \mu_p$.

Notons $\hat{p}$ la valeur p tel que $\beta_{\hat{p}} = \lambda$, et calculons $\mathbb{P}[p^* \in [p_{\min}, p_{\max}]]$.

$$\begin{aligned}
\mathbb{P}[p^* \notin [p_{\min}, p_{\max}]] &\leq \mathbb{P}[\alpha_{p^*} > \lambda] \\
&= \mathbb{P}[\alpha_{p^*} > \lambda, \mu_{p^*} \leq \lambda] + \mathbb{P}[\alpha_{p^*} > \lambda, \mu_{p^*} > \lambda] \\
&\leq \mathbb{P}[\alpha_{p^*} > \mu_{p^*}] + \mathbb{P}[\mu_{p^*} > \lambda] \\
&\leq \mathbb{P}[\alpha_{p^*} > \mu_{p^*}] + \mathbb{P}[\mu_{\hat{p}} > \lambda], \text{ car } \forall p \in E, \mu_{p^*} \leq \mu_p \\
&= \mathbb{P}[\alpha_{p^*} > \mu_{p^*}] + \mathbb{P}[\mu_{\hat{p}} > \beta_{\hat{p}}] \\
&= \frac{\alpha}{2} + \frac{\alpha}{2} = \alpha
\end{aligned}$$

donc

$$\mathbb{P}[p^* \in [p_{\min}, p_{\max}]] \geq 1 - \alpha$$

L'expérience construite permet donc de déterminer un intervalle de confiance à 95% pour la valeur optimale du paramètre, si on prend $\alpha = 5\%$.

Limitation de l'expérience. Pour fonctionner, l'expérience a besoin d'un ensemble E ***fini****. Cela implique que les intervalles de confiance générés ne pourront pas plus précis que E. Si E est* $\{0.01, 0.02, ..., 0.99\}$*, l'intervalle de confiance ne pourra pas être plus précis que* 0.01.

2.3 Implementation

J'ai implémenté cette expérience, nommée « Optimisation de paramètre », à travers une classe C++ nommée `ParameterOptimizer`. Cette classe est *abstraite*, et pour l'utiliser, il suffit d'en créer une classe fille en spécifiant l'algorithme et le paramètre étudiés.

2.4 Etude du taux de mutation

Commençons par étudier le taux de mutation du *Evolutionary Algorithm*. Il a été annoncé dans le cours que le choix de $\frac{1}{n}$ comme taux de mutation était le plus commun. Vérifions cela.

J'ai appliqué l'expérience « Optimisation de paramètre » sur mon implémentation de $(1+1)$-EA avec :

- n variant de 3 à 50,
- $(1-\alpha) = 95\%$

Le graphique du haut de la figure 1 montre le résultat de l'expérience (*ie* les intervalles de confiance pour n variant). Avec le graphique du bas, il semble assez clair que la valeur optimale du taux de mutation est proportionnelle à $1/n$. Sachant que la courbe de $1/n$ est dans les intervalles de confiance à 95%, on peut considérer que cette valeur est *validée*.

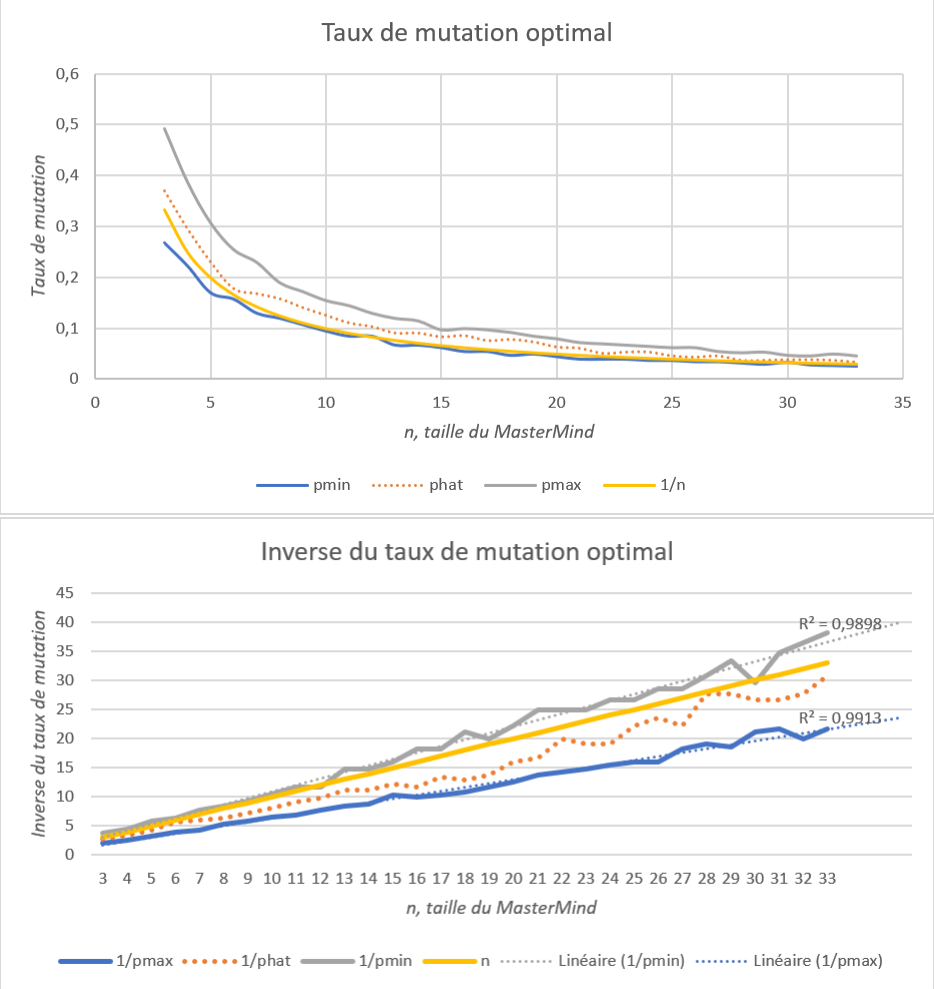

FIGURE 1 – Taux de mutation optimal

2.5 Étude du taux de croisement

A présent, passons aux algorithmes génétiques. Ces algorithmes dépendent de différents paramètres : taille de la population des parents, taille de la population des enfants, taux de mutation (*mutation rate*), taux de croisement (*crossover rate*)...

J'ai déjà étudié le taux de mutation sur les EA. On va considérer que le résultat est le même sur les GA.

Passons au taux de croisement. L'idée est de trouver le taux de croisement qui minimise avec le nombre de requête, pour une taille de problème fixé (pour n fixé) et pour une taille de populations donnée. Par souci de simplification, je vais considérer que la taille des deux populations est égale ($\mu = \lambda$).

Notons $c(n, \lambda)$ le taux de croisement qui minimise, en moyenne, le nombre de requêtes. On souhaite connaître la forme de $c(n, \lambda)$.

2.5.1 Protocole expérimentale

Pour répondre à la problématique, on va appliquer l'expérience « Optimisation de paramètre » pour des n et des λ différents.

A partir de ces données produites, on va déterminer la forme de $\lambda \mapsto c(n, \lambda)$ pour n fixé. Puis, on va regarder l'impact de n sur la fonction précédente pour extrapoler $(n, \lambda) \mapsto c(n, \lambda)$.

2.5.2 Expérience

J'ai fait tourner l'expérience « Optimisation de paramètre » sur $GA(\lambda + \lambda)$, avec n fixé, pour déterminer un intervalle de confiance à 95% de la valeur optimale du taux de croisement :

- n allant de 5 à 19 (inclu)
- λ allant de 5 à 100
- précision de taux de croisement optimale : 0.01

Le temps de calcul de l'expérience a été de 48h pour un coeur de $2,5$ GHz. Cela a produit $15 \times 95 = 1425$ intervalles de confiance. Les données produites sont dans le fichier `data/crossover-rate.csv`.

2.5.3 Taux de croisement pour n fixé

J'ai testé deux modèles de fonctions :

- $c_n(\lambda) = 1 - \frac{\alpha_n}{\lambda}$
- $c_n(\lambda) = 1 - \exp^{-\alpha_n \lambda + \beta_n}$

J'ai donc tracé les courbes pour n fixé (figure 2 pour $n = 10$), et j'ai testé les deux modèles (figure 3 pour $n = 10$). Le modèle de la fonction inverse est peu probable, mais par contre, le modèle de l'exponentiel semble pertinent !

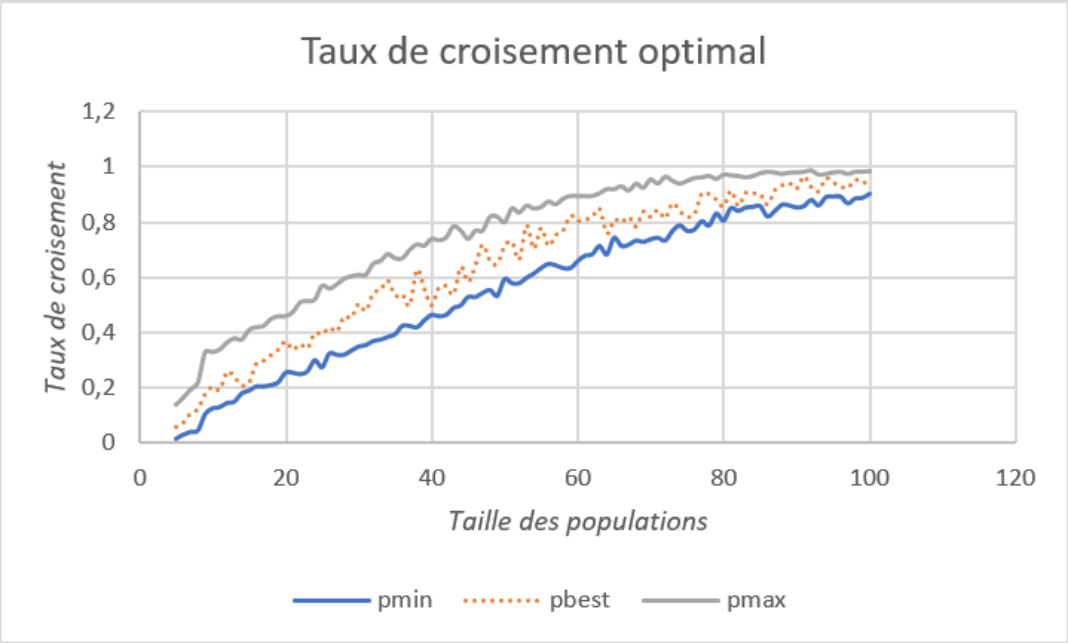

FIGURE 2 – Taux de croisement optimal ($n = 10$)

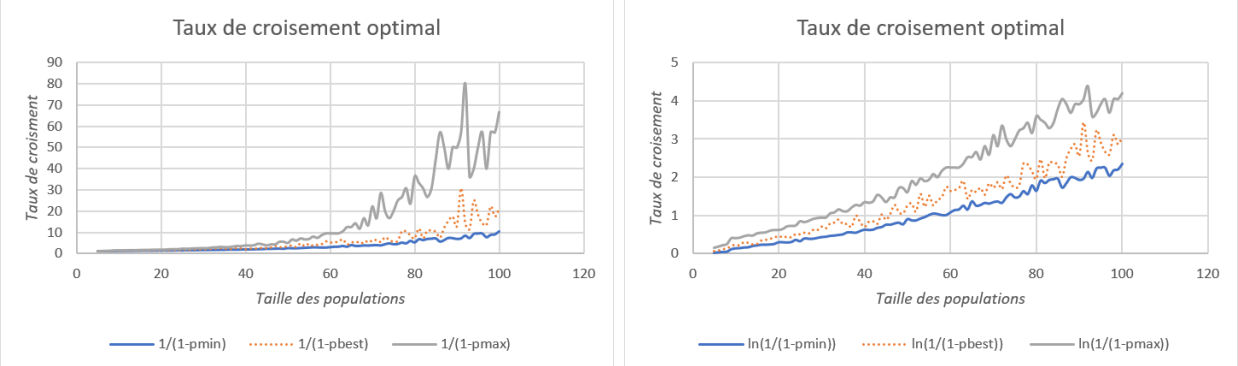

FIGURE 3 – Modélisation du taux de croisement optimal ($n = 10$)

Retenons donc, pour la suite,

$$\forall n, c_n(\lambda) = 1 - \exp^{-\alpha_n \lambda + \beta_n}$$

2.5.4 Taux de croisement pour n variant

En utilisant des régressions linéaires, j'en déduis les coefficients α_n et β_n de $c_n(\lambda) = 1 - e^{-\alpha_n \lambda + \beta_n}$, pour n variant de 5 à 19, pour les deux extrémités des intervalles de confiance. Pour avoir des courbes propres, on retire les données avec un λ et n faible. En effet, comme on peut voir sur la figure 4 ($\lambda > 60$), la borne supérieure est très proche de 1 et comme l'expérience d'optimisation de paramètre dépend d'un ensemble E fini, on tombe sur un problème d'imprécision (*cf* la limitation de l'expérience décrite précédemment).

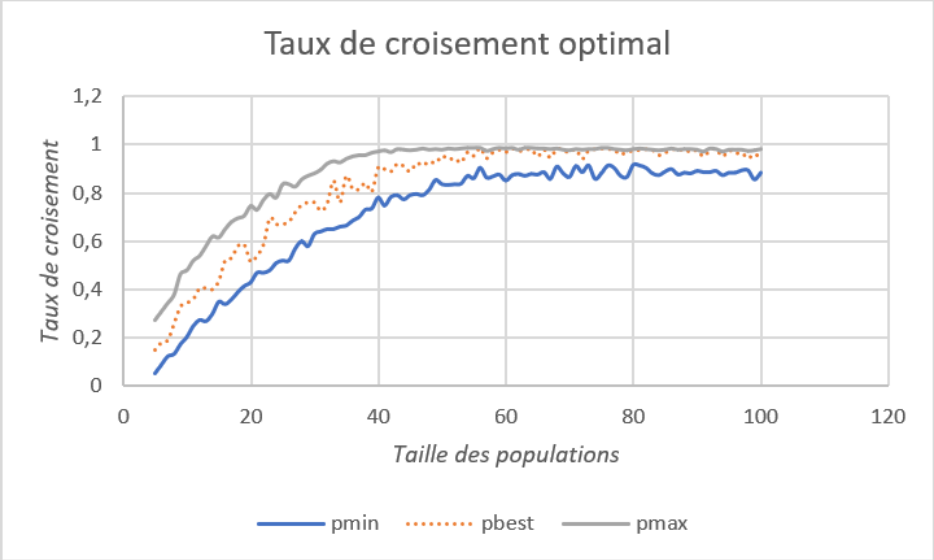

FIGURE 4 – Taux de croisement optimal ($n = 6$)

Si on trace les coefficients α_n, on obtient le graphique de gauche de la figure 5. Et si on trace le logarithme de ces coefficients, on a la figure de droite.

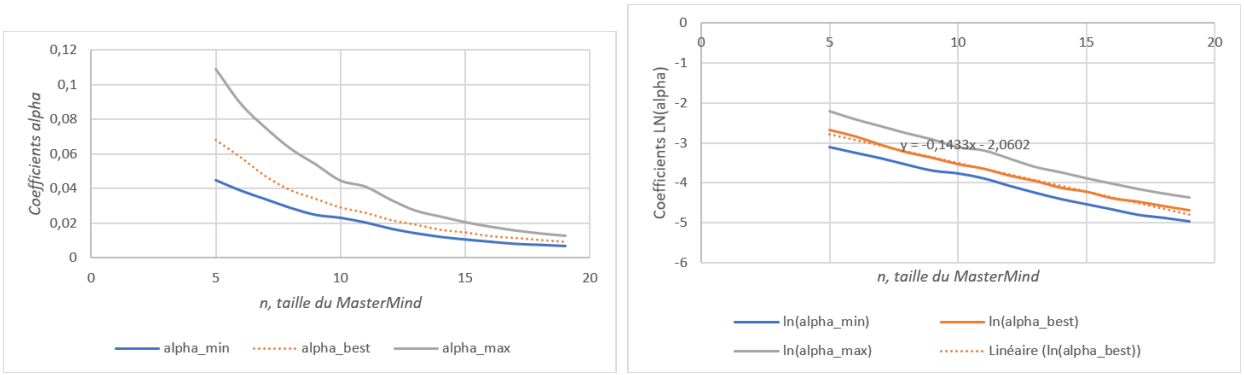

FIGURE 5 – Détermination du coefficient α_n

On obtient de belles droites, on a donc que

$$\ln(\alpha_n) \approx -0,1433n - 2,0602$$

Donc,

$$\alpha_n \approx e^{-0,1433n-2,0602}$$

Pour β_n, la courbe est un peu plus erratique (figure 6). Mais si on ignore les points pour n petit, on peut imaginer un semblant de comportement exponentielle, comme pour α_n. Sous cette hypothèse, avec une régression linéaire, on trouve

$$\beta_n \approx e^{-0,43n-3,4}$$

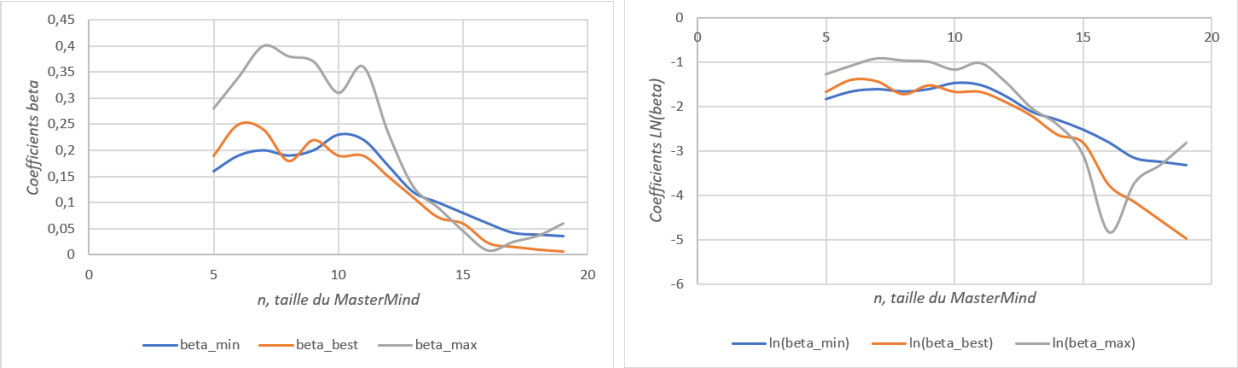

FIGURE 6 – Détermination du coefficient β_n

In fine, on trouve

$$c(n, \lambda) = 1 - \exp(-e^{-0,1433n-2,0602}\lambda + e^{-0,43n-3,4})$$

Testons le modèle. Sur les données de l'expérience, est-ce que la modélisation de $c(n, \lambda)$ se situe dans les intervalles de confiances ? Réponse :

- Pour $93,3\%$ des données, le modèle se situe dans les intervalles de confiance.
- Si l'on retire les donnés où il y a le problème d'imprécision (*cf* limitation de l'expérience), on tombe sur $98,5\%$.

2.5.5 Amélioration des performances de la modélisation

Le modèle de $c(n, \lambda)$ dépend de quatre valeurs. Actuellement, il y a 0.1433, 2.0602, 0.43 et 3.4. Si la pente (0.1433) du coefficient α_n est claire. On peut remarquer que l'ordonnée à l'origine peut varier entre 1.55 et 2.40. De plus, le modèle exponentielle pour β_n n'est pas précis, donc les valeurs optimales de la pente et de l'ordonnée à l'origine pourraient fluctuer.

Pour cette raison, on peut essayer de faire varier un peu les 3 dernières valeurs pour obtenir un meilleur pourcentage. Ainsi, je tombe sur le nouveau modèle

$$c(n, \lambda) = 1 - \exp(-e^{-0{,}1433n-2{,}15}\lambda + e^{-0{,}3n-0{,}22})$$

dont les performances sont :

- Pour $96,8\%$ des données, le modèle se situe dans les intervalles de confiance.
- Si l'on retire les donnés où il y a le problème d'imprécision, on tombe sur $99,9\%$.

2.6 Conclusion sur les valeurs de taux

Le taux de mutation optimal est inversement proportionnelle à la taille du problème. Il devient rapidement faible pour des valeurs élevées de n. Dans le reste du projet, j'utiliserai

$$p(n) = \frac{1}{n}$$

Le taux de croisement optimal est également décroissant avec la taille du problème. Par contre, c'est croissant avec la taille des populations. Pour la suite du projet, j'utiliserai

$$c(n, \lambda) = 1 - \exp(-e^{-0{,}1433n-2{,}15}\lambda + e^{-0{,}3n-0{,}22})$$

3 Complexité des algorithmes

3.1 Randomized Local Search (RLS)

3.1.1 Etude théorique

Quelle est sa complexité du MasterMind ? Utilisons la *fitness level method* pour en avoir une borne supérieur.

Notons p_i une borne inférieure de la probabilité que RLS passe d'un candidat avec i couleurs en commun avec la solution à un candidat strictement mieux.

$$\begin{aligned} p_i &= \mathbb{P}[\text{changer une case pas optimisée}].\mathbb{P}[\text{trouver la bonne couleur}] \\ &= \frac{n-i}{n}\frac{1}{n-1} \end{aligned}$$

Donc,

$$\mathbb{E}[T(RLS, MasterMind)] \leq \sum_{i=0}^{n-1} \frac{1}{p_i} = \sum_{i=0}^{n-1} \frac{n(n-1)}{n-i} = n(n-1)\sum_{i=1}^{n} \frac{1}{i}$$

On en déduit que

$$\mathbb{E}[T(RLS, MasterMind)] = \mathcal{O}(n^2 log(n))$$

3.1.2 Vérification expérimentale

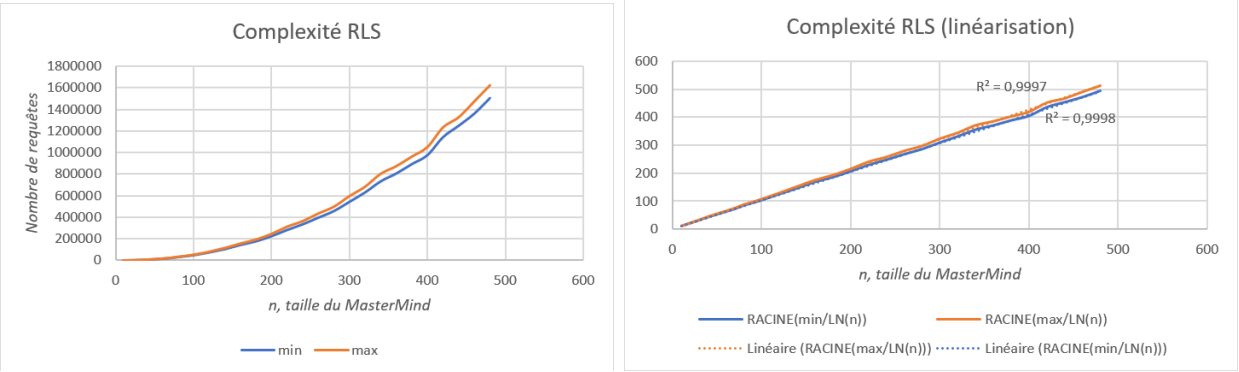

FIGURE 7 – Complexité de RLS

Expérimentalement, on a

$$\mathbb{E}[T_{\text{RLS, } MasterMind}(n)] = (1.092n^2 + 3.135n + 2.25)ln(n)$$

ce qui est en **cohérence** avec la théorie précédente.

3.2 Evolutionary Algorithm (EA)

Reprenons les notations de *RLS*. Conformément à la première section de ce rapport, on va prendre $\frac{1}{n}$ comme taux de mutation.

$$\begin{aligned} p_i &\geq \mathbb{P}[\text{changer qu'une seule case}] \times \mathbb{P}[\text{que celle-ci ne soit pas déjà optimisée}] \\ &\qquad \times \mathbb{P}[\text{trouver la bonne couleur}] \\ &= \binom{n}{1}\left(\frac{1}{n}\right)^1\left(1-\frac{1}{n}\right)^{n-1} \times \frac{n-i}{n} \times \frac{1}{n-1} = \frac{n-i}{n(n-1)}\left(1-\frac{1}{n}\right)^{n-1} \\ &\geq \frac{n-i}{en(n-1)} \end{aligned}$$

Donc,

$$\mathbb{E}[T((\mu+\lambda)EA, MasterMind)] \leq \sum_{i=0}^{n-1} \frac{1}{p_i} = en(n-1) \sum_{i=1}^{n} \frac{1}{i}$$

On en déduit que

$$\mathbb{E}[T((\mu+\lambda)EA, MasterMind)] = \mathcal{O}(n^2 log(n))$$

Concernant l'impact de la taille des populations, on peut directement faire remarquer que :

- Augmenter μ (taille de la population des parents) va diminuer les performances de l'algorithme, car le fait d'avoir plusieurs parents fait qu'on ne part pas forcément du meilleur parent pour engendrer les enfants.
- Augmenter λ (taille de la population des enfants) va diminuer les performances de l'algorithme, car cela signifie que la population des parents sera moins souvent mis à jour, et donc les enfants seront générés à partir de parents moins bons.

3.3 Vérification expérimentale

J'ai fait tourner mon implémentation de EA pour différentes valeurs de n, μ et λ.

- n allant de 5 à 29 (inclu)
- $\mu \in \{1, 3, ..., 49\}$
- $\lambda \in \{1, 3, ..., 49\}$
- Complexité moyennée sur 1000 résolutions, pour chaque triplés de paramètres.

Le temps de calcul a été de 28h pour un coeur de 2,5 GHz. Cela a produit $25 \times 25 \times 25 = 15625$ intervalles de confiance de 95% estimant la complexité moyenne pour un triplé de paramètres fixés. Les données produites sont dans le fichier `data/complexity-ea.csv`.

Comme on peut voir sur la figure 8, lorsque l'on fixe n et une des deux tailles de populations, on tombe sur une droite affine.

On peut donc extrapoler que T est de la forme

$$\mathbb{E}[T_{\text{EA, } MasterMind}(n, \mu, \lambda)] = c_1(n)\mu\lambda + c_2(n)\mu + c_3(n)\lambda + c_4(n)$$

Il ne resterait plus qu'à déterminer l'influence de n sur les coefficients $c_i(n)$. Pour cela, j'ai tracé les $c(i)$ en fonction de n, et j'ai réalisé des régressions.

A l'aide de la figure 9, on déduit que

$$\begin{cases} c_1(n) = (0.0045n - 0.0306)ln(n) \\ c_2(n) = (1.4089n - 3.4084)ln(n) \\ c_3(n) = 1.0153n - 2.9039 \\ c_4(n) = (2.569n^2 - 7,297n + 19.158)ln(n) \end{cases}$$

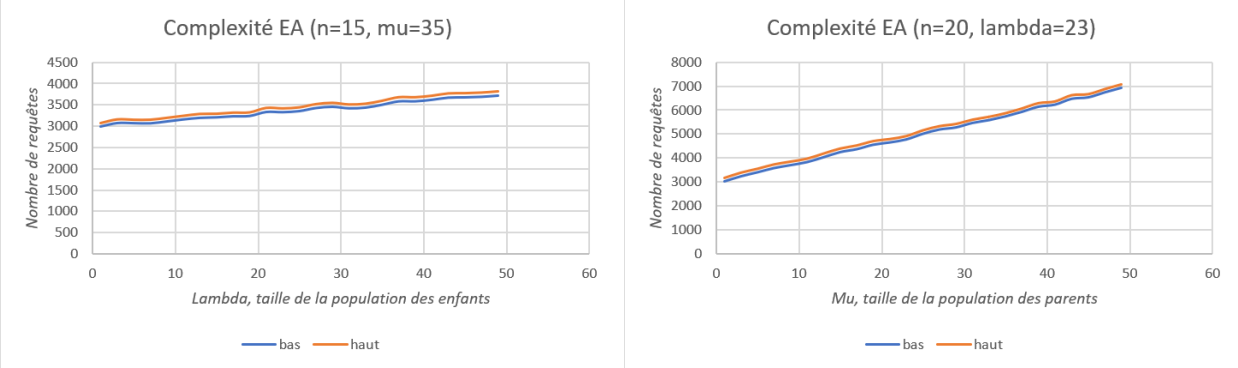

FIGURE 8 – Complexité de *EA* pour deux paramètres fixés sur trois

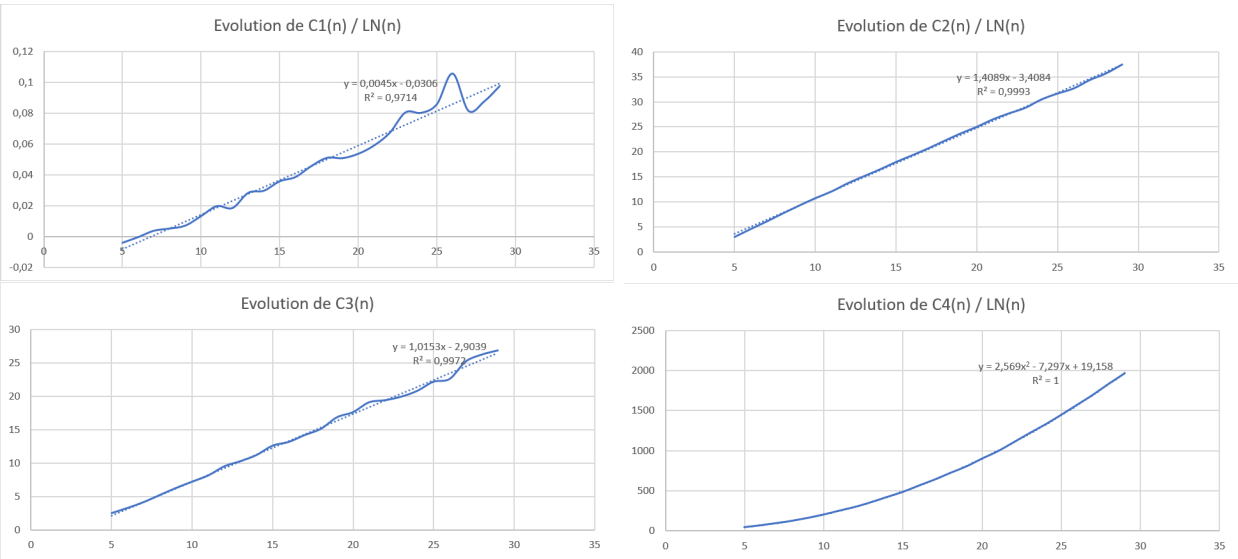

FIGURE 9 – Régression pour déterminer l'influence de n sur les paramètres

Vérifions la performance du modèle (*ie* à quel pourcentage la modélisation est inclue dans les 15625 intervalles de confiance). On obtiens que :

- le modèle intersecte 81.55% des intervalles.
- le modèle intersecte 87.63% des intervalles (si on retire les petits n, *ie* les $n \leq 9$).

Pour améliorer le modèle, j'ai fait tourner **une variante d'un** $(1+1)$**-EA sur les 9 constantes du modèles**, les opérations élémentaires étant une petite fluctuation des constantes (par exemple, augmenter de 10% une constante). J'ai écrit « variante », car mes opérations changent avec le temps (plus l'algorithme tourne, plus les modifications sur les constantes sont petites).

Ainsi, je tombe sur le modèle suivant :

$$\mathbb{E}[T_{\text{EA, } \mathit{MasterMind}}(n,\mu,\lambda)] = c_1(n)\mu\lambda + c_2(n)\mu + c_3(n)\lambda + c_4(n)$$

avec

$$\begin{cases} c_1(n) = (0.0045748n - 0.0325175)ln(n) \\ c_2(n) = (1.4444n - 3.83779)ln(n) \\ c_3(n) = 1.03815n - 2.99155 \\ c_4(n) = (2.52103n^2 - 6.58785n + 16.965)ln(n) \end{cases}$$

dont les performance sont :

- le modèle intersecte 86.37% des intervalles.
- le modèle intersecte 88.37% des intervalles (si on retire les petit n, *ie* les $n \leq 9$).

Et si on optimise directement sans prendre en compte les données avec n faible, on tombe une performance qui se rapproche du 90%.

En résumé, d'après la modélisation, la complexité en moyenne de $(\mu + \lambda)$-EA est en

$$\mathcal{O}(n^2 log(n) + nlog(n)\mu\lambda)$$

3.4 Genetic Algorithm (GA)

3.4.1 Etude théorique

A une constante près (liée au taux de croisement), on a le même résultat de complexité qu'avec *EA*.

$$\mathbb{E}[T((\mu + \lambda)GA, MasterMind)] = \mathcal{O}(n^2 log(n))$$

Quant à l'impact concret de μ et de λ sur la complexité, à cause du croisement des parents, il est plus difficile à prévoir. Certes, augmenter λ diminuera les performances pour les mêmes raisons que pour EA. Mais le croisement de plusieurs parents pourraient avoir un impact positif sur les enfants, donc l'impact de μ sur la complexité n'est pas évident.

3.5 Vérification expérimentale

La figure 10 montre qu'expérimentalement, étant donné qu'on obtient une droite affine quand on applique la fonction $x \mapsto \sqrt{\frac{x}{ln(n)}}$, on a bien

$$\mathbb{E}[T((1 + 1)GA, MasterMind)] = \mathcal{O}(n^2 log(n))$$

Et pour l'impact de μ et de λ ? Déjà, il faut préciser que je ne peux tester expérimentalement que $(\lambda + \lambda)$-GA (et non pas $(\mu + \lambda)$-GA), car je n'ai modélisé le taux de croisement (*crossover rate*) optimal que pour ce cas précis. Il ne serait pas cohérent de comparer si, dans certains cas, on ne prend pas le taux optimal.

Lorsqu'on trace le nombre de requêtes effectuées par $(\lambda + \lambda)$-GA pour une taille n fixée de *MasterMind*, on remarque une forme plutôt linéaire (*cf* figure 11). Néanmoins, je n'ai pas eu le temps d'étudier en détail l'impact de n sur cette courbe linéaire.

Globalement, on remarque que le nombre de requêtes augmente (affinement) quand λ augmente. Donc, si on souhaite minimiser le nombre de requête, il vaut mieux s'intéresser à $(1 + 1)$-GA.

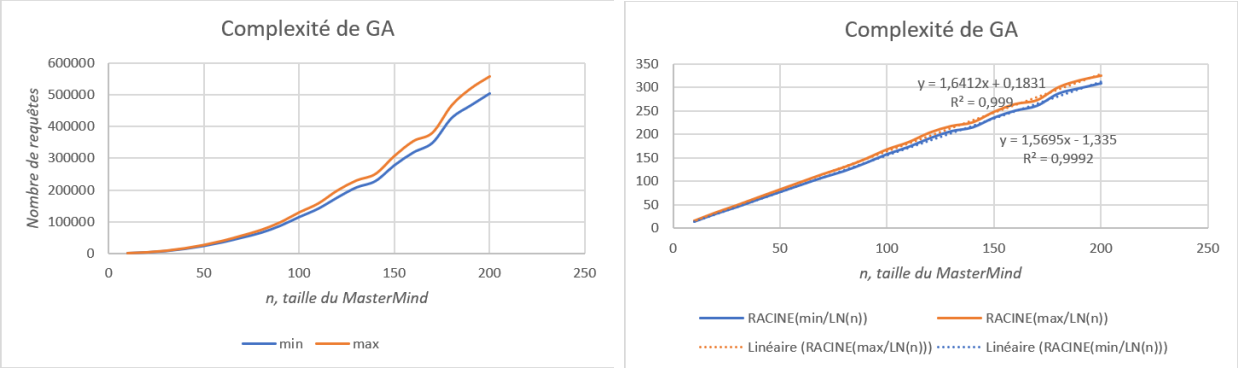

FIGURE 10 – Complexité de $(1 + 1)$-GA, en fonction de n

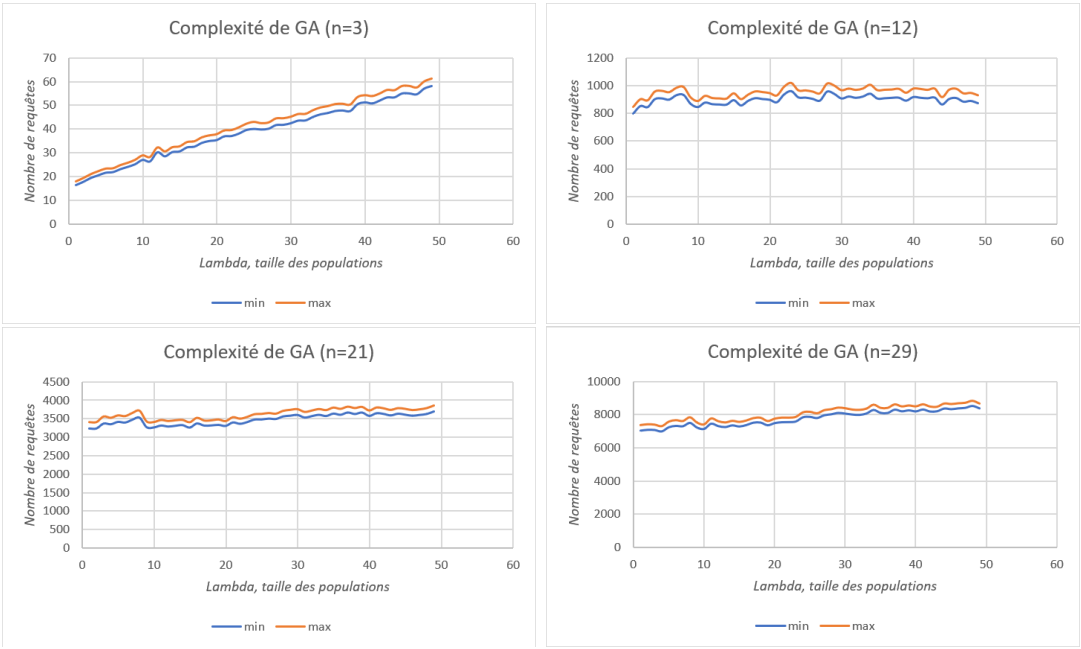

FIGURE 11 – Complexité de $(\lambda + \lambda)$-GA, en fonction de λ

4 Vitesses de convergence des algorithmes

A présent, étudions la vitesse de convergence de plusieurs algorithmes. Nous allons regarder celle du *Randomized Local Search*, celle du *Evolutionary Algorithm* (avec $\mu = \lambda = 1$ puisque cela minimise le nombre de requêtes, *cf* section précédente), celle du *Genetic Algorithm* (même remarque, avec $\mu = \lambda = 1$) et un algorithme personnalisé décrit plus loin.

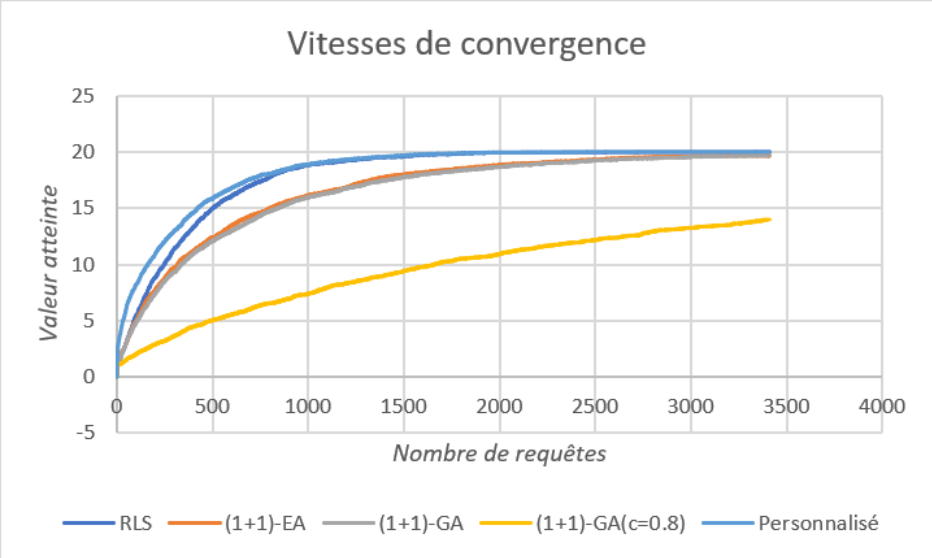

FIGURE 12 – Vitesses de convergence de certains algorithmes

A travers la figure 12 ($n = 20$, convergence moyennée sur 100 tests), on peut voir que c'est le RLS qui a la vitesse de convergence la plus élevée parmi les algorithmes traditionnels. Ensuite, on peut constater que EA et GA ont un comportement très proche l'un de l'autre. Ils sont néanmoins moins performants que RLS. Remarquons que si on choisit mal le *crossover rate* du GA, on peut diminuer fortement les performances. C'est ce que montre la courbe jaune, où j'ai pris arbitrairement le taux de croisement à 80%, au lien de suivre la modélisation du taux optimal. Cela montre l'importance d'avoir étudié le taux de croisement en premier.

Et pour finir, je me suis brievement posé la question s'il était possible de trouver un meilleur algorithme (meilleur que RLS) si on s'autorisait à faire varier les taux (de mutation et/ou de croisement) en fonction de la valeur atteinte. Et la réponse est oui, puisque j'ai trouvé un tel algorithme avec une (légèrement) meilleure vitesse de convergence. Notons x l'avancement en pourcentage. J'ai considéré un mélange d'algorithmes : si $x < 50\%$, c'est $(1+1)$-EA avec un taux de mutation de $\frac{1}{2}\left(\frac{1}{20}\right)^{\frac{20x}{19}}$, et sinon $x \geq 50\%$ c'est un simple RLS. Je n'ai pas eu le temps de pousser mes recherches.

5 Conclusion

Au sein de mon projet, j'ai donc étudié l'impact d'un certain nombre de paramètres sur deux types d'algorithmes (principalement) : les *Evolutionary Algorithms* et les *Genetic Algorithms*.

Dans un premier temps, j'ai étudié quels étaient le taux de mutation (*mutation rate*) et le taux de croisement (*crossover rate*) qui fallaient mettre dans les EA et les GA.

Une fois ces taux optimaux déterminés, j'ai étudié l'impact des paramètres μ (nombre de parents) et λ (nombre des enfants) sur les performances, ainsi que l'impact de la taille n de *MasterMind*.

Et pour terminer, j'ai rapidement regardé les vitesses de convergence des algorithmes en question.