Rapport Mastermind

Thibaut Milhaud

Février 2020

Introduction

L'objectif de ce sujet était d'implémenter différentes heuristiques aléatoires pour résoudre un problème de Mastermind. J'ai choisi de traiter le cas où il y a n couleurs et n cases.

Premièrement, je vais faire une rapide description du sujet. Ensuite je vais essayer d'expliquer les choix d'implémetation puis je vais présenter les expériences que j'ai effectuées et les résultats expérimentaux obtenus.

Le code a été écrit en C avec la librairie standard. Pour le tracé des courbes j'ai utilisé gnuplot. La totalité du travail se trouve sur mon Github.

1 Description du problème

1.1 Mastermind à n cases et n couleurs

Le but du jeu est de deviner le code secret du défenseur en le moins d'étapes possible. Ce code consiste en un vecteur v d'entiers inférieurs à n et de longueur n.

$$v \in (\mathbb{Z}/n\mathbb{Z})^n$$

Il y a donc n^n codes possibles. Pour deviner ce code l'attaquant va soumettre des propositions $p_i \in (\mathbb{Z}/n\mathbb{Z})^n$ au défenseur et il va recevoir en retour le score de sa proposition calculé de la façon suivante :

$$\text{score}(p_i, v) = |\{k \leq n \text{ tels que, } p_i[k] = v[k]\}|$$

.

On peut noter qu'il existe une stratégie déterministe qui trouve la solution en $O(n^2)$ dans le pire des cas et en moyenne. Il s'agit de la stratégie qui consiste à trouver les couleurs case par cases : on teste tout pour la première case (on est sûrs d'avoir trouvé en n coups) et on fait de même pour toutes les cases.

1.2 Heuristiques aléatoires

Ce qui nous intéresse ici est de trouver la solution en utilisant des heuristiques aléatoires et d'étudier les différences de performances entre ces différents algo.

Paramètres Ces algorithmes sont des algorithmes très généraux qui font intervenir de nombreux paramètres tels que :

— Fonction de mutation, fonction qui va donner une proposition à partir d'une ancienne :

1. chaque case à une probabilité p d'être modifiée ;
2. une case est modifiée au hasard.

— Fonction de croisement, fonction qui va créer un nouvel individu à partir de deux parents ou plus.

1. on prend les premières valeurs d'un parent puis les suivante chez l'autre ;
2. chaque case à une probabilité p de provenir du premier parent, sinon elle provient de l'autre.

— Fonction de sélection, dans notre cas, il n'y a pas d'optimum local, on peut donc utiliser la sélection élitiste dans tout les cas.
— λ, μ.

Familles d'algo étudiés Les trois grande familles d'heuristiques dont je vais parler dans ce rapport sont :
— Randomized Local Search(RLS) : à chaque étape on va modifier la valeur d'une case de la proposition actuelle, si elle améliore le score, on la conserve, sinon on garde l'ancienne. On continue jusqu'à avoir trouvé le code.
— $(\lambda + \mu)$Evolutionary ALgorihtms(EA) : ici, on va conserver en permanence une population de μ propositions. Ensuite, à chaque tour de boucle on va construire, à partir de l'ancienne génération, (λ) nouveaux candidats par mutations. Enfin, seul les μ meilleurs de ces $(\lambda+\mu)$ candidats seront conservé pour l'étape suivante. L'algorithme se termine quand le score maximum est atteint.
— $(\lambda+\mu)$Genetic ALgorihtms(GA) : semblable au précédent algorithme à ceci près que les nouveaux nés on une probabilité c d'être le résultat du croisement de deux individus de la génération précédente. Sinon, ils sont comme précédemment obtenus par mutations.

2 Implémentation

2.1 Présentation des structures

La plupart des structures utilisés sont des tableaux d'entiers non signés notamment pour les propositions qui constituent la majorité des objets manipulés dans le code.
— propositions : `unsigned int*` ;
— instance : structure avec une proposition ainsi que n et k les paramètres du problème ;
— result : structure qui contient des données utiles pour l'analyse des différents algorithmes tels dont principalement un tableau de taille n où la case j indique combien d'appels à la fonction score il a fallu pour atteindre un score de j ;
— prop_score : un tuple contenant une proposition et son score. Utilisé pour trier les générations dans EA et GA.

2.2 Meta-algorithme génétique

Les heuristiques que j'ai décidé d'implémenter suivent toutes un schéma similaire en suivant l'ordre suivant (GA > EA > RLS, du plus général au plus spécifique). L'algorithme que j'ai implémenté prend les paramètres suivants :
— a, l'instance du problème ;
— λ, μ ;
— c, la probabilité de croisement ;
— mutation, la fonction de mutation ;
— crossover, la fonction de croisement.
de cette manière, il est possible d'obtenir toutes les fonctions voulues en choisissant les paramètres adaptés. Par, exemple en prenant $c = 0, \lambda = \mu = 1$, crossover = `NULL` et la bonne fonction de mutation on retrouve RLS.

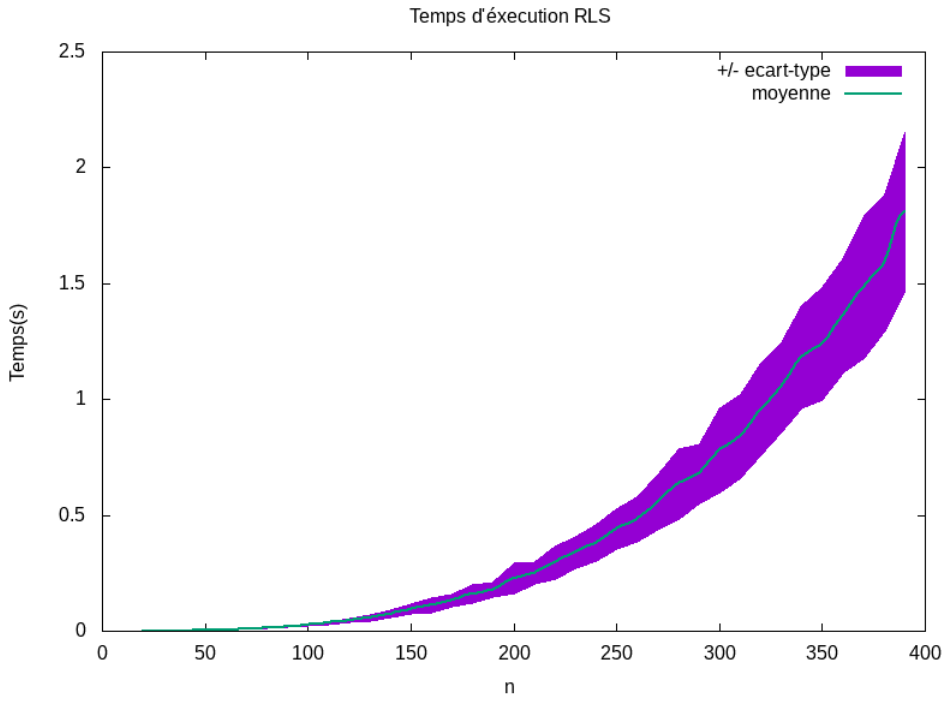

FIGURE 1 – Complexité en temps

3 Expériences

Étant donné la nature hautement probabiliste des algorithmes étudiés, les courbes qui vont suivre présentent la plupart du temps la moyenne (en nombre d'appel à score) sur un nombre d'itérations N de l'ordre de $N = 100$.

3.1 RLS

Évaluation complexité RLS, $N = 100$ D'après 1 on voit que l'aspect aléatoire de l'algorithme sur le temps d'exécution est non négligeable mais pas non plus démesuré. D'après 2 on peut supposer que :

$$K \times n^{2.5} \log n < T(n) < K' \times n^3 \log n$$

BlackBox complexity Ici dans 2 on observe un résultat de manière plus précise, il semblerait que $BB(n) \sim n^2 \log n$.

3.2 Evolutionary Algorithms

Évaluation de l'impact de $(\lambda+\mu)$**,** $N = 200$**, Mutation classique** $p = 1/n$ Difficile de déduire quelque chose de 4, on a dû mal à voir une tendance se dessiner et on se demande si les écarts ne seraient pas dû à la variance induite par l'aléatoire des algorithmes.

Variation de la probabilité de mutation, $N = 200, n = 50$ On observe sur 5 qu'une probabilité de mutation plus importante semble donner des meilleurs résultats au début mais tombe en désuétude par la suite. De plus il est difficile de trouver une façon satisfaisante d'actualiser la probabilité de mutation au cours de l'exécution.

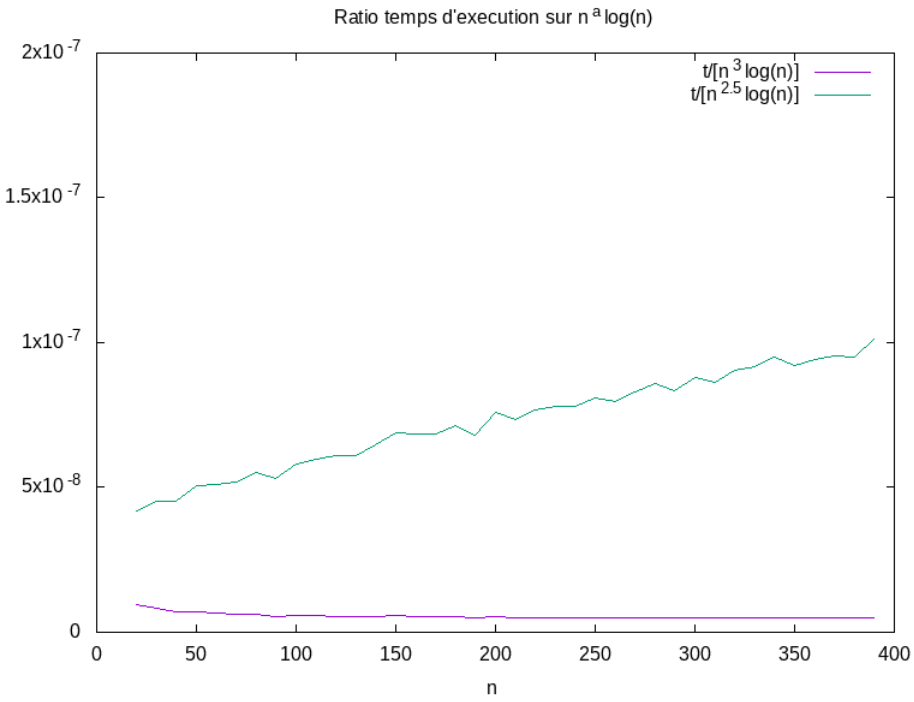

FIGURE 2 – Tentative d'évaluation de la complexité

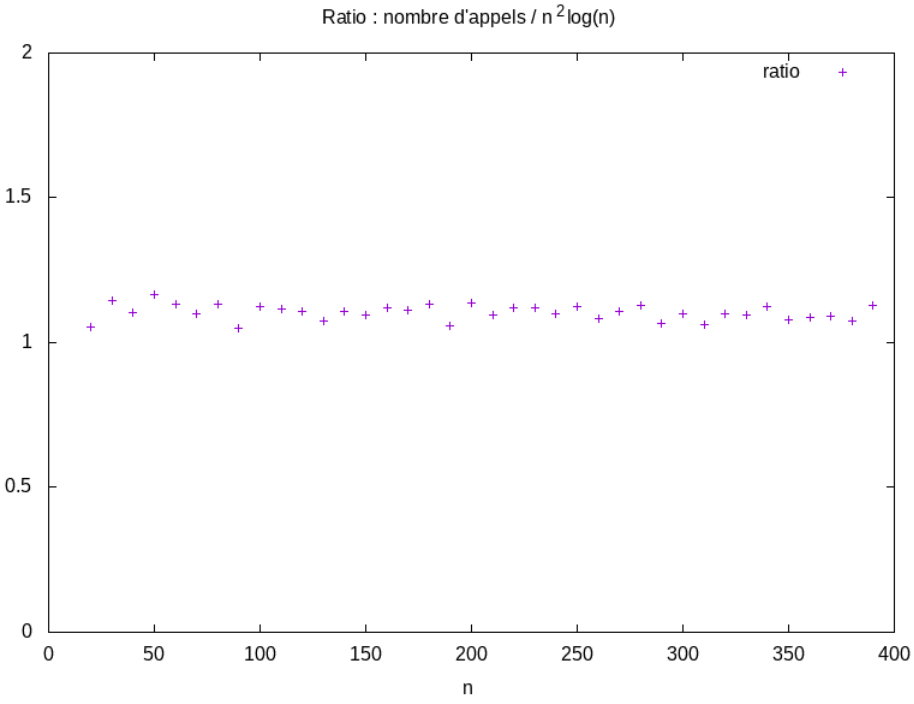

FIGURE 3 – Évaluation blackbox complexity

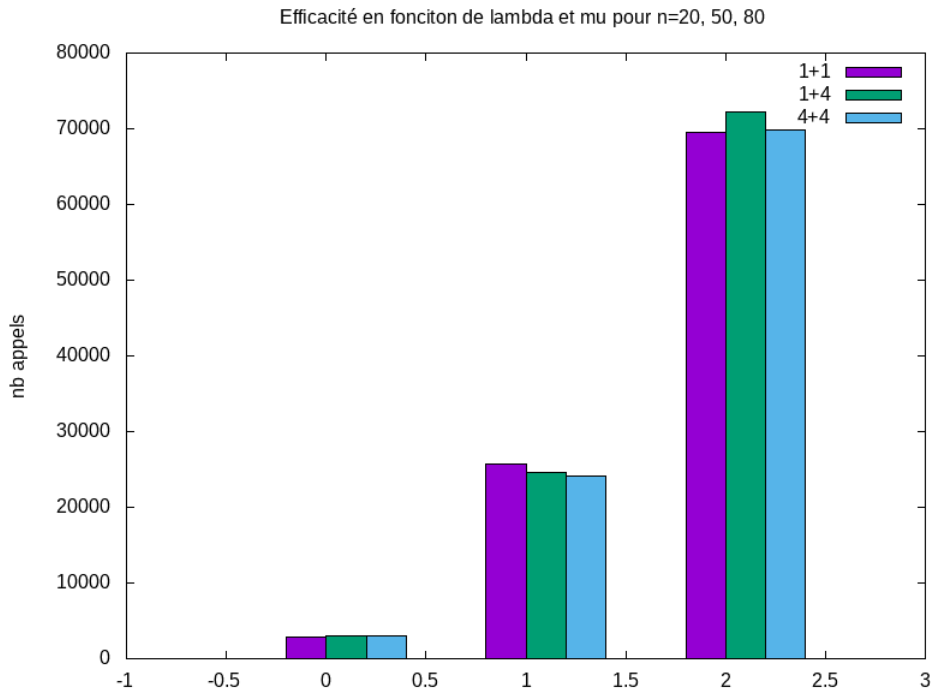

FIGURE 4 – Impact $\lambda + \mu$

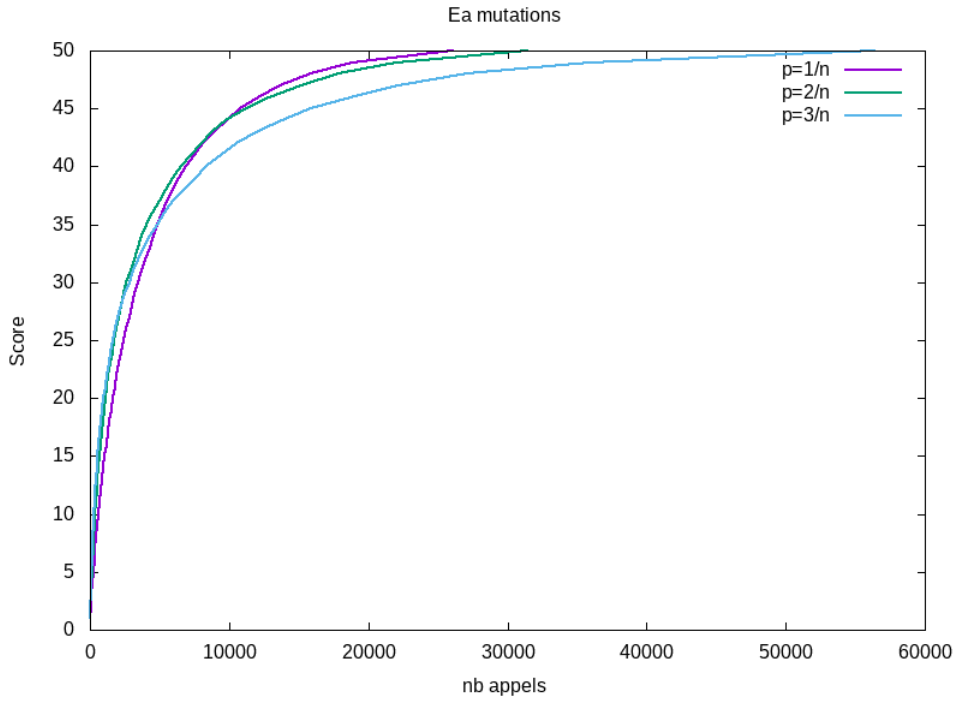

FIGURE 5 – Variation de Probabilité de mutation

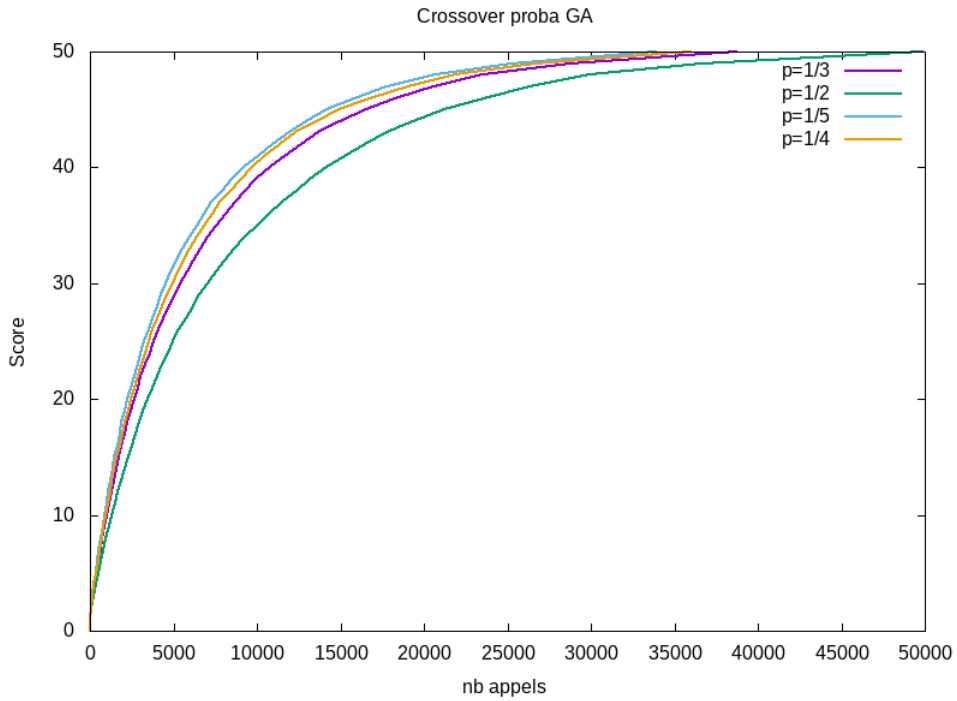

FIGURE 6 – Impact de la proba de Crossover

3.3 Genetic Algorihtms

Impact de la probabilité de croisement, $n = 50, N = 200, \lambda = 10, \mu = 5$ D'après 6 il semblerait qu'il ne faille pas prendre une trop grande probabilité de mutation à voir si il existe une bonne valeur (c'est ce qu'il semblerait) ou bien si les mutations ne sont pas adaptées pour ce problème (dans tout les cas le fils peut avoir un mauvais score).

Comparaison de deux croisements, $n = 60, 65, 70, 75, N = 200, \lambda = 10, \mu = 5$ Dans cette expérience, j'ai tenté de comparer les croisements présenté dans la section 1. De façon analogue à la figure4, on n'obtient pas de résultat évident quand à l'impact de ces différentes fonctions.

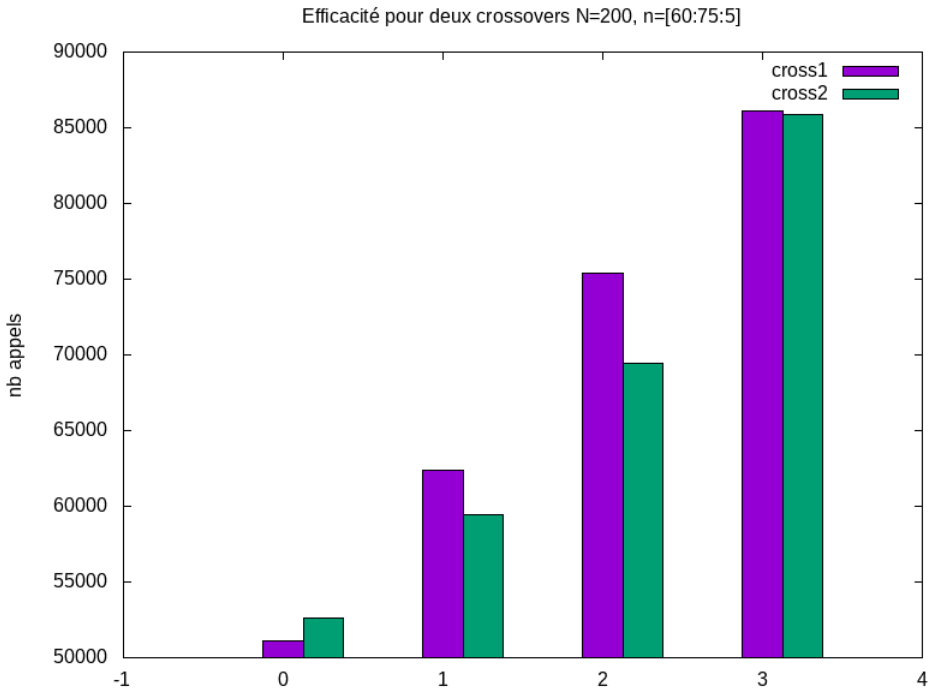

FIGURE 7 – Deux crossovers

Conclusion

Dans ce travail j'ai tenter d'explorer différentes pistes de variations sur 3 algorithmes. Et en traçant des courbes de manière plus ou moins aléatoire, j'ai obtenu comme on pouvait s'y attendre des résultats dont l'intérêt est très variable. Cela dit, en l'état le code permettrait de plus amples investigations et peut-être qu'en réfléchissant un peu plus avant de tracer des trucs il serait possible d'arriver à des données intéressantes.