Projet de résolution de problèmes : Satisfaction de contraintes pour le Master Mind

Etudiant :
Hakim CHEKIROU

Mai 2020

Introduction

L'objet de ce projet est de développer et tester des méthodes de satisfaction de contraintes et un algorithme génétique pour la résolution d'un problème de *master mind*.
Le jeu consiste pour un joueur (le codeur) à choisir un code qu'il garde secret et pour l'autre joueur (le décodeur) à deviner ce code. Le code secret à découvrir se compose de n caractères alphanumériques (chiffres ou lettres deux à deux distincts) pris parmi l'ensemble D_p constitué des p premiers éléments de l'ensemble $\{0, ..., 9, A, .., Z\}$ (on pourra se restreindre au cas où $(4 \leq n \leq 18$ et $p = 2n)$. Pour obtenir de l'information le décodeur peut faire un essai, c'est-à-dire proposer un code et le codeur devra alors indiquer d'une part combien de caractères du code proposé sont corrects et bien placés et d'autre part combien de caractères sont corrects mais mal placés. Le décodeur doit tenter un nouvel essai jusqu'à obtenir le bon code. Le but du jeu et de trouver le bon code avec le minimum d'essais possibles.
Dans ce projet, on s'intéresse à réaliser un programme qui, à chaque étape du jeu, est capable de proposer un nouveau code à essayer qui soit compatible avec toutes les informations accumulées lors des essais précédents (on s'interdit de tester des combinaisons incompatibles avec l'information disponible, même si elle sont informatives).

Question 0

Chaque code proposé est compatible avec toutes les informations accumulées lors des essais précédents ce qui permet de créer une séquence d'essais qui converge nécessairement vers la solution car, en respectant à chaque étape les contraintes accumulés depuis le début du jeu, on réduit les possibilités pour un bon code et donc in fine réduire les possibilités jusqu'à n'avoir que le bon code.

Modélisation et résolution par CSP

Question 1.1

Dans cette partie, nous implémentons 3 algorithmes de résolution de contraintes. Chaque algorithme propose un code compatible avec les contraintes accumulés depuis le début du jeu. Nous explorons ces trois algorithmes:

- A1 : Engendrer et tester
- A2 : Retour arrière chronologique
- A3 : Retour arrière chronologique avec forward checking (le forward checking n'utilisant que les contraintes all-diff)

Pour qu'une combinaison proposée soit consistante avec les contraintes. Il faut d'abord que le code proposé soit composé de caractères deux à deux distincts et que le nouveau code soit diffèrent de toutes les anciennes propositions.
Pour chaque ancienne proposition, il faut respecter les indices donnés par le codeur. Le nombre de caractères qui restent à la même place par rapport au nouveau code doit être égale au nombre de pions rouge à cette étape du jeu. Le nombre de caractères qui changent de place doit aussi être égale au pions blancs. De cette manière, on s'assure que chaque nouvelle instanciation respecte les informations précédentes.
Pour la consistance locale, c'est la même méthodologie, sauf qu'on tient compte des variables qui n'ont pas encore été instanciées. Par exemple, si à un moment du jeu, un code proposé a retourné 2 pions rouges et que dans l'instanciation partiel, aucun caractère n'est resté à sa place et qu'il ne reste plus qu'une seule variable non instanciée, alors on ne pourra jamais respecter la contrainte. Ce qui fait que cette instanciation est non consistante. on vérifie aussi que le nombre de pièces qui restent à leurs places ne dépasse pas le nombre de pions rouges. On procède de la même manière avec les pions blancs.

Temps moyens d'exécutions

Nous avons testé le temps d'exécution nécessaire aux trois algorithmes pour résoudre le problème sur 20 instances de taille 4 et de vocabulaire 8. Les temps d'exécutions moyennés sont représentés dans la figure 1. Tous les tests réalisés dans ce projet sont effectué avec un processeur Intel Xeon @ 2.20GHz.

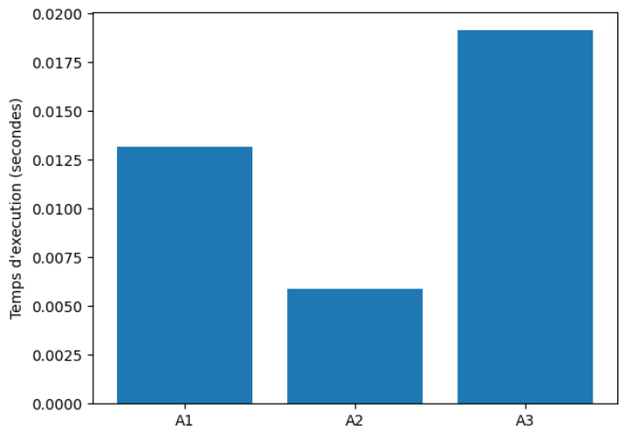

Figure 1: Temps moyens d'exécutions sur 20 instances de taille n=4 et p=8

L'algorithme le plus rapide est celui utilisant le retour arrière chronologique, vu qu'on explore moins de possibilités. Dans l'algorithme A3, on rajoute l'opération "check forward" au retour arrière chronologique, le temps d'exécution est donc plus élevé que l'algorithme A2.

Évolution du temps d'exécution

Nous lançons des tests pour analyser l'évolution du temps moyens d'exécution lorsque n et p augmentent. Dans la figure suivante, on a testé les trois algorithmes pour différentes valeurs de n et $p = 2 * n$. Nous nous sommes arrêté quand l'exécution devenait trop lente.

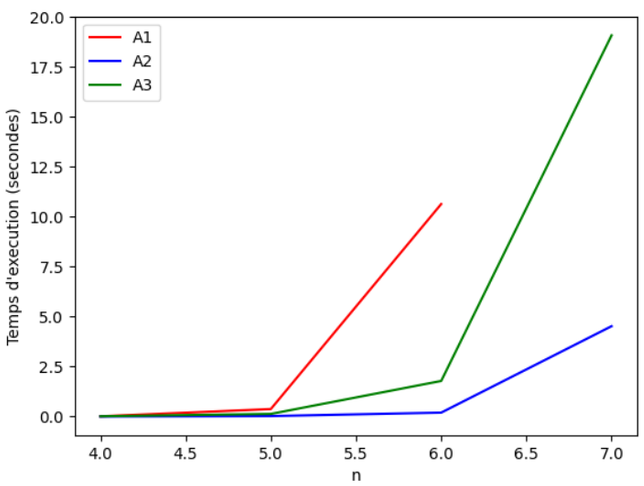

Figure 2: Évolution du temps moyen d'exécutions en fonction de n avec $p = 2 * n$

On peut voir que les temps d'exécutions des trois algorithmes suivent des courbes exponentielles. Néanmoins, Le temps d'exécution de l'algorithme Engendrer et tester devinent très important dès que n atteint 6, nous arrêtons donc l'exécution ici. L'algorithme A3 est plus lent que le simple retour arrière chronologique car on rajoute l'étape check-forward. Il n'utilise que la condition all-diff, qui ne réduit pas assez le domaine des variables pour éliminer assez de possibilités et impacter les temps d'exécutions. Dans la figures suivante, nous traçons le nombre d'essais moyen nécessaires a trouver le bon code par les trois algorithmes en fonction de la taille de l'alphabet et en fixant n à 5.

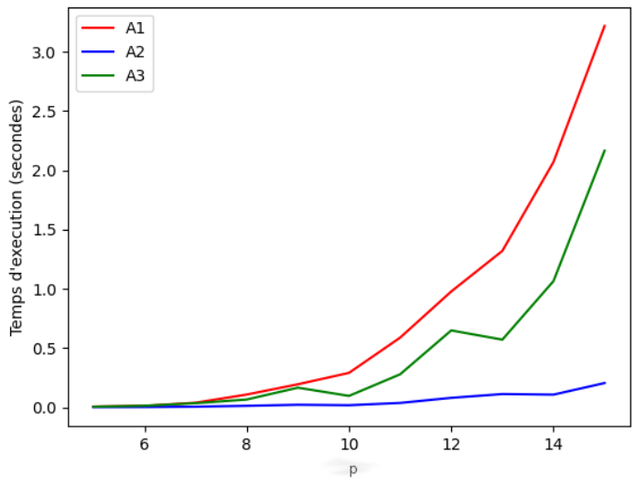

Figure 3: Évolution du temps moyen d'exécution en fonction de p

Comme pour la fois précédente, le temps d'exécution augmentent exponentiellement. L'algorithme A2 reste plus rapide. Le forward checking ne semble pas plus performant que le simple retour arrière chronologique car on ne diminue pas assez les domaines pour impacter le temps d'exécution.

Évolution du nombre d'essais

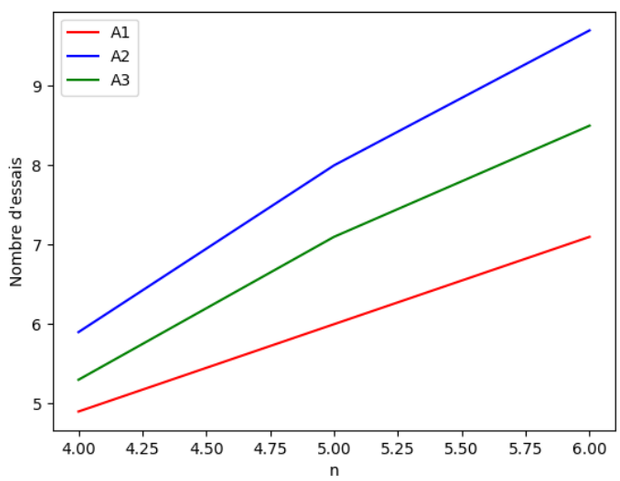

Figure 4: Évolution du nombre moyen d'essais en fonction de n

Pour les trois algorithmes, le nombre d'essais nécessaire est linéairement dépendant à la taille du code. L'algorithme Engendrer et tester semble avoir besoin de moins d'essais que les deux autres. A2 trouve le code avec légèrement plus de tentatives que A3.

Dans la figure suivante, On analyse aussi l'impact de la taille de l'alphabet p sur le nombre d'essais. On fixe n à 5 en faisant varier p de 5 à 15.

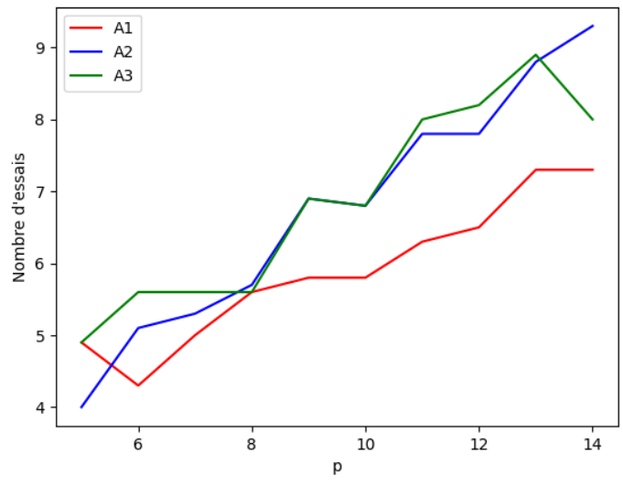

Figure 5: Évolution du nombre moyen d'essais en fonction de p

Ici aussi, les nombres moyens d'exécutions augmentent avec p et l'algorithme Engendrer et tester a besoin de moins d'essais. Pour A3 et A2, il n'y a pas de différences claires entre les deux méthodes.

Question 1.2

Nous considérons ici une version du jeu où le code peut admettre deux occurrences du même caractère. Pour adapter l'algorithme A3 au relâchement de la contrainte, on modifie la fonction check forward. Quand le nombre d'occurrences d'une valeur atteint 2, cette valeur est supprimée du domaine des autres variables. On compare les performances des de A3 et A4 sur un problème de taille $n = 4$ et $p = 8$ en exécutant chaque algorithme 40 fois.

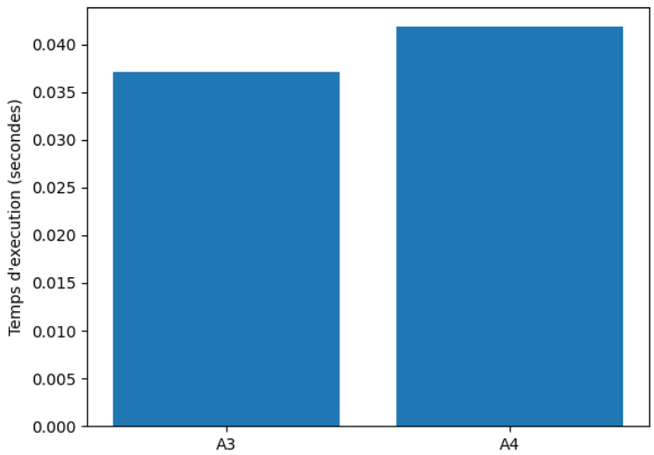

Figure 6: Temps d'exécutions de A3 et A4

L'algorithme $A4$ est un peu plus lent car la fonction check forward de $A4$ est plus lente. Dans la figure 7, on représente le nombre moyen d'essais nécessaires au deux méthodes.

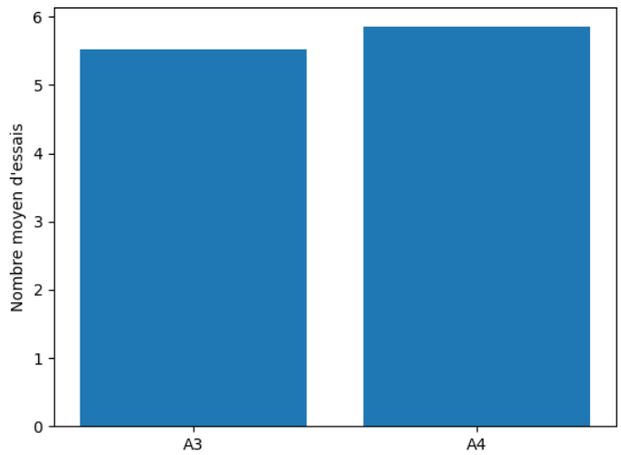

Figure 7: Nombre moyen d'essais nécessaires à A3 et A4 pour trouver le code

Les nombres d'essais des deux méthodes sont très proches, A4 a besoin de légèrement plus d'essais que A3.

Question 1.3

Nous revenons à la version où le code ne contient que des caractères distincts. Pour améliorer l'algorithme A3, nous adaptons le forward checking pour tenir compte des autres contraintes. Pour cela nous allons rajouter ces modifications.
Contrainte des caractères bien placés : Par rapport à chaque ancienne proposition, si le nombre de variables qui sont restées inchangées dans la nouvelle instanciation est égale aux pions rouges obtenus à cette étape, alors le nombre de cases qui doivent rester inchangées pour respecter la contrainte est déjà atteint. Il faut donc empêcher les autres variables de garder la même valeur que précédemment. Nous enlevons donc les valeurs correspondantes des domaines des variables restantes. Par exemple, si on a obtenu 2 pions rouges pour le code **ABCD** et qu'on instancie **AB–**, on ne pourra plus mettre **CD** pour les deux variables restantes car on en a déjà 2 qui sont restés à la mémé place.
Contrainte des caractères corrects mais mal placés : Par rapport à chaque ancienne proposition, si le nombre de caractères qui ont changé de place dans la nouvelle instanciation est égale aux pions blancs obtenus à cette étape, cela veut dire que le nombre de caractères qui doivent changer de place pour respecter la contrainte des pièces correctes mais mal placées est déjà atteint. Il faut alors empêcher les variables restantes de prendre une valeur qui été présente dans l'ancienne proposition mais à un endroit différent. On réduit les domaines des variables restantes pour respecter cette condition.
Contrainte all-diff : On enlève des domaines des variables la valeur de la variable nouvellement instanciée.
Si une des variables a un domaine vide alors l'instanciation n'est pas consistante.
Dans la figure suivante, on représente les temps d'exécutions moyens des deux algorithmes sur des instances de tailles différentes. Pour chaque valeur de n, on génère 20 jeux, et on lance les deux algorithmes sur chaque jeu.

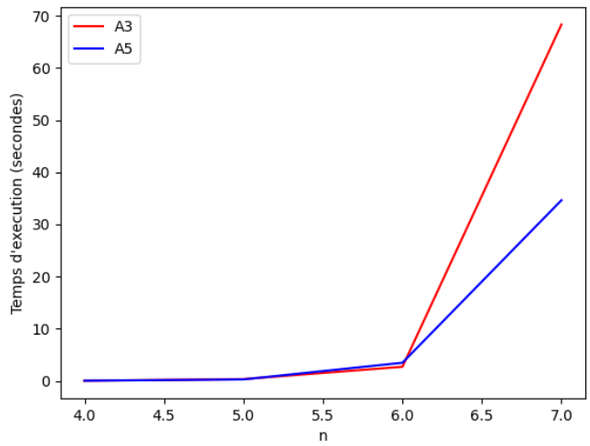

Figure 8: Évolution du temps moyen d'exécutions en fonction de n

Les temps d'exécutions sont exponentielles dans les deux cas, La différence n'est apparente que quand le problème devient assez compliqué. A5 est plus rapide que A3 à ce moment là. A5 cherche moins longtemps que A3, car les domaines des variables y sont plus réduits que dans A3.

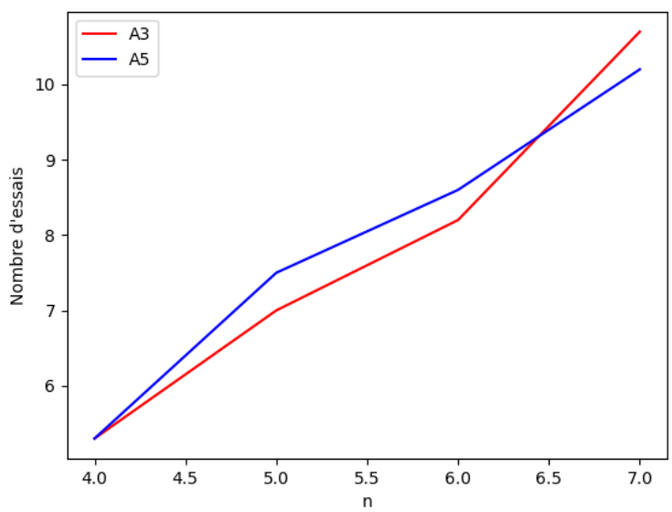

Figure 9: Comparaison des nombres d'essais nécessaires a A3 et A5 en fonction de n avec $p = 2 * n$

Le nombre d'essais est linéairement dépendant à la taille du code mais il n y a pas de différences entre le nombre d'essais des deux algorithmes.

Partie 2 : modélisation et résolution par algorithme génétique

Dans cette partie, on décide de résoudre le problème de la recherche d'un code compatible avec un algorithme génétique. Le but de l'algorithme génétique est de générer après chaque nouvel essai un ensemble E de codes compatibles avec l'ensemble des essais précédents. Pour ce faire, on commence avec un ensemble de code généré aléatoirement. On choisi des individus dans la population qu'on mutera pour obtenir de nouveaux codes. On défini une mesure ou "fitness" qui est liée aux nombres d'incompatibilités avec les essais précédents et la probabilité de choisir un code est liée à cette mesure. Dès qu'un code compatible est trouvé, il sera ajouté à l'ensemble E. L'algorithme agit comme suit :

- **Générer une population :** On génère un ensemble de code aléatoirement,
- **Évaluation de la fitness :** Pour chaque code, on calcul son score en pénalisant chaque enfreinte aux informations précédentes.
- **Sélection des codes à faire évoluer :** On procède par plusieurs tournois de 10 codes, c'est à dire que nous faisons plusieurs échantillons de 10 codes pris parmi la population et on sélectionne le meilleur candidat parmi chaque échantillon, on obtient à la fin de cette partie un ensemble de code à muter.
- **Mutation :** On défini un probabilité qu'a chaque variable de muter, c'est a dire de changer de caractère.
- **Consistance :** Ajout des codes consistants à l'ensemble E.

On répète ces étapes jusqu'à atteindre la taille maximale de l'ensemble E, $sizemax$ ou le nombre de générations maximales et dans le cas ou E est vide même après l'atteinte de ces valeurs, on continu jusqu'à dépasser 5 minutes de temps d'exécution. Si même après, aucune combinaison n'est trouvée, l'algorithme aura échoué.
Il y a plusieurs paramètres sur lesquels jouer, la taille de l'ensemble E, le type de la mutation, la taille de la population et la probabilité que chaque caractère a de muter.

Question 2.1

Nous avons implémenté une première version de l'algorithme génétique où le choix de la prochaine tentative se fera aléatoirement parmi l'ensemble E. Nous avons tenter plusieurs combinaisons de paramètres pour avoir les meilleurs temps d'exécutions. Pour ce faire une idée, on a fixé certains paramètres en faisant évoluer d'autres. La figure suivante représente un test réalisé sur la taille de la population, nous avons fixé le nombre maximale de génération à 50, $maxsize$ à 6, la probabilité de muter pour chaque caractère à 0.25

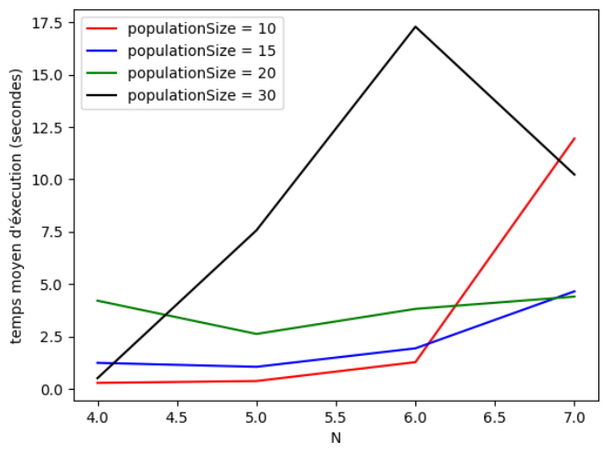

Figure 10: Comparaison des temps d'exécution par taille de population en fonction de la taille du problème

On remarque d'abord qu'il n'y a pas de valeur qui soit la meilleure pour toutes les tailles de problèmes. Avoir une plus grande population n'est pas toujours avantageux, on augmente le nombre

de calculs à faire à chaque génération et pour les codes de petites tailles, ce n'est pas nécessaire. Pour les grandes tailles, une plus grande population permet de chercher plus efficacement. On a donc décidé que pour chaque taille de code N, une différente taille de population sera utilisée. La meilleure combinaison de paramètres trouvée est la suivante :

N	4	5	6	7	8
taille de la population optimale	10	10	10	20	50

Nous avons aussi tenter plusieurs valeur de probabilité différentes comme $\frac{1}{n}$ ou bien des probabilités fixes pour toutes les tailles. Pour chaque paramètres, nous avons fait des tests et les meilleurs paramètres trouvés sont détaillés dans le tableau suivant.

paramètre	maxgen	maxsize	probabilité	timeout	taille du tournoi
valeur	100	6	0.25	5 min	10

La figure suivante montre les temps d'exécutions avec ces paramètres.

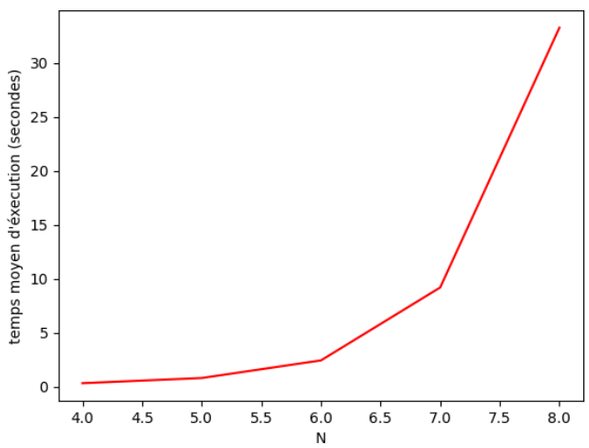

Figure 11: Meilleurs temps d'exécution en fonction de la taille du problème

Il est à noter que cet algorithme est le seul qui permet de résoudre des problèmes de taille $n = 8$ avec $p = 16$ en un temps acceptable. Pour cette valeur de n et p, il y a 16^8 possibilités soit plus de 4 milliards de combinaisons possibles.

Question 2.2

Dans cette partie, on examine comment la sélection du code à jouer parmi tout les codes compatibles de l'ensemble E peut influencer le jeu. Nous avons comparer ces 4 méthodes de sélection:

- Choix du code présentant le plus de similarités avec les autres codes compatibles.
- Choix du code présentant le moins de similarités avec les autres codes compatibles.
- Estimation du nombre de codes compatibles restants si un code était tenté : on sélectionne le code qui diminuerait le plus la taille de E s'il était le prochain coup joué dans le jeu.
- Choix aléatoire.

Pour tester ces méthodes, nous avons fixé la taille de l'ensemble E à 10, le nombre de générations maximales à 100, la taille de la population optimale déterminée précédemment a été utilisé. Pour chaque valeur de N, nous avons généré 20 instances du jeu, et pour chaque instance, nous avons testé les 4 méthodes. Les temps moyens sont représentés sur la figure suivante.

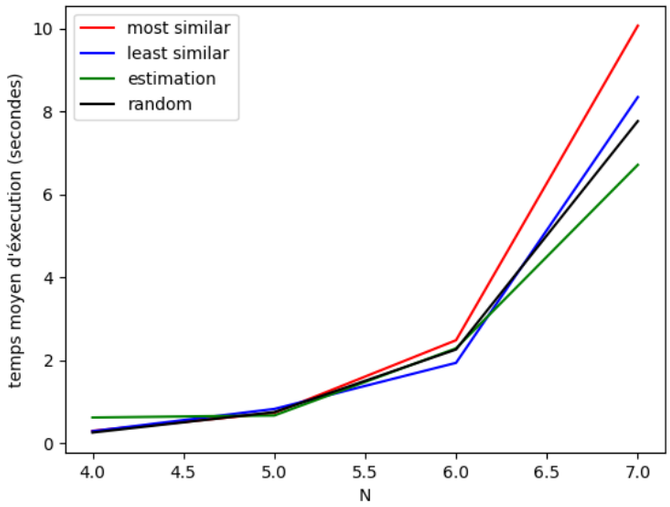

Figure 12: Temps d'exécution des méthodes de sélections de codes en fonction de la taille du problème

La première remarque qu'on peut faire est que ces méthodes ne peuvent être appliqué que si E contient plusieurs codes. Durant les premiers coups du jeu, on arrive à trouver plusieurs codes compatibles mais plus le jeu avance plus il est difficile de trouver une multitude de codes sans atteindre le temps de génération maximales.
Il n'y a pas de différences apparentes pour les codes de petites taille. les performances divergent quand on arrive a $n = 7$, la méthode de sélection avec estimation est plus rapide mais ce n'est pas assez significatif pour pouvoir être sur de son efficacité.

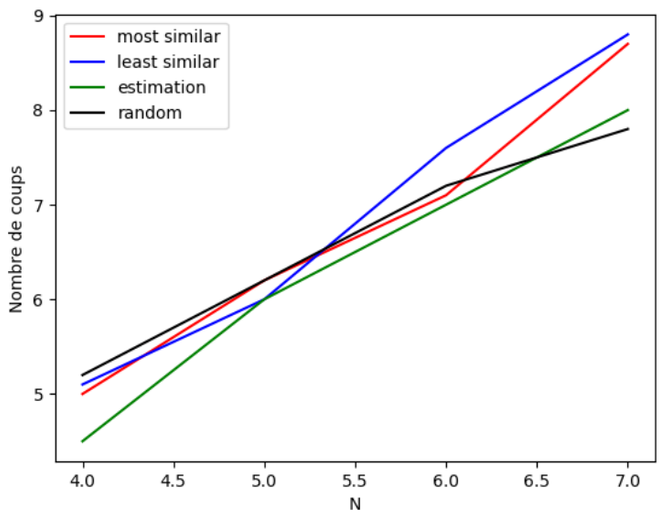

Figure 13: Nombre de coups nécessaires aux méthodes de sélections de codes en fonction de la taille du problème

Pour ce qui est du nombre de coups nécessaires pour trouver le code secret, il est linéairement dépendant de n mais il n'y a pas de méthodes qui y arrive avec clairement moins de coups qu'une autre.

Question 2.3

Dans cette dernière question, nous allons comparer le meilleur algorithme de satisfaction de contraintes contre l'algorithme génétique avec les meilleurs performances. Le meilleur algorithme en temps de calculs qui utilise la satisfaction de contraintes est le retour arrière chronologique "RAC". Nous comparons d'abord les temps de résolutions par rapport à n de 4 à 8 avec $p = 2 * n$. Pour chaque valeur de N, on génère 10 tests qu'on résout avec les deux méthodes. On affiche ensuite les résultats moyennés.

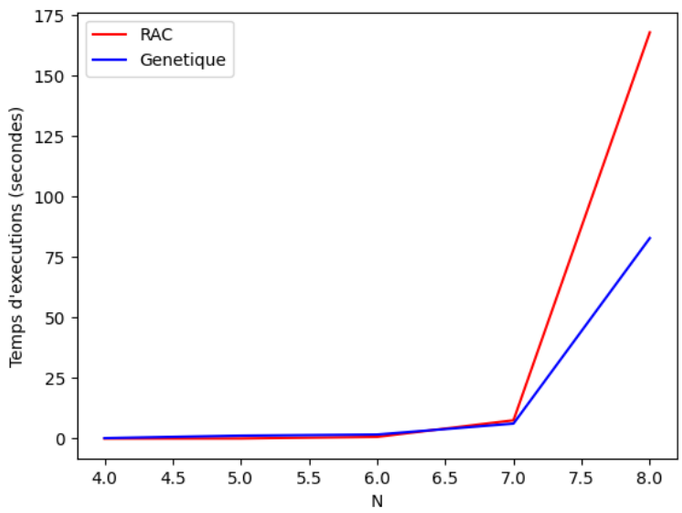

Figure 14: Comparaison des temps d'exécutions en fonction de n

Le retour arrière chronologique est légèrement plus rapide pour les premières valeurs de n mais dès que $n = 7$, les temps du RAC explose et l'algorithme génétique est deux fois plus rapide.
Pour ce qui est du nombre de tentatives nécessaires à chaque méthode pour trouver le code, nous avons affiché les résultats dans la figure ci dessous.

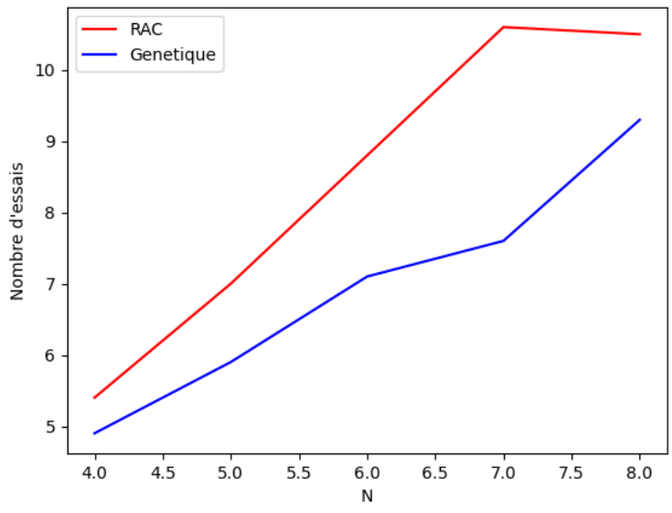

Figure 15: Comparaison des nombres d'essais nécessaires en fonction de n

Comme pour les fois précédentes, le nombre de coups joués est linéairement dépendant de la longueur du code secret. En utilisant l'algorithme génétique, le jeu est résolu avec moins de coups

qu'avec le retour arrière chronologique. L'algorithme génétique est donc meilleur sous les deux aspects.

Conclusion

Pour résoudre le problème du master mind, nous avons d'abord exploré des approches simples et naïves comme la méthode engendrer et tester. Nous avons ensuite tenté une approche plus intelligente avec le retour arrière chronologique puis en le renfonçant avec le forward checking. Nous avons aussi utilisé un algorithme génétique en les comparant aux meilleures méthodes qui le précède. Tout au long de ce projet, nous avons joué avec les différents paramètres des algorithmes et nous avons regardé comment ils réagissent à différents niveaux de complexité dans le problème. Le master mind s'est révélé être un problème dur à résoudre quand il atteint une grande taille, mais nous sommes parvenu à résoudre des problèmes de taille conséquente en un temps acceptable. Il y a bien sur une multitude de pistes à explorer pour améliorer notre travail, nous pourrions tenter des implémentations plus rapides ou bien définir de meilleures fitness ou faire des mutations et des croisements plus complexes. Ce problème reste donc un sujet ouvert.