Introduction

Architecture générale

Le projet est implémenté en Python. Le code est réparti en trois fichiers :

`mastermind.py`	Fonctions permettant de simuler le jeu de mastermind (génération d'une combinaison aléatoire, comparaison de deux combinaisons, vérification de la compatibilité d'une combinaison avec les informations précédemment reçues).
`solver.py`	Implémentation des algorithmes de la partie 1 (engendrer et tester, retour arrière chronologique "simple", et forward checking) en classes génériques (indépendantes du problème du mastermind).
`projet.py`	Test des algorithmes sur le mastermind, dessin des courbes demandées, etc.

Choix d'implémentation

- Chaque couleur est représentée par un nombre
- Une combinaison est représentée par une liste de couleurs.
- Le "retour" d'une combinaison est représenté par un `tuple` de la forme `(nombre de pions bien places, nombre de pions mal places)`.
- A un instant donné du jeu, les informations obtenues dans les essais précédents sont appelées *contraintes*, et représentées par des `tuple` de la forme `(combinaison, retour)`.

Approche générale

Implémentation de la fonction de comparaison

Notre implémentation de la fonction de comparaison de deux combinaisons, `compare_combinations`, se fait en deux étapes :

1. Déterminer les indices qui ont la même couleur
2. En oubliant ces indices, pour compter les "mal placés", on souhaite un conteneur qui n'ait pas d'ordre mais tienne compte des multiplicités. C'est exactement ce que fait la classe `Counter` du module `collections`. Nous n'avons plus alors qu'à compter les couleurs en commun.

L'implémentation permet entre autre de comparer une combinaison complète avec une incomplète, ce qui facilite le code ensuite pour les algorithmes type retour arrière.

Question 0

A chaque étape, la proposition d'une combinaison compatible avec les informations précédemment reçues entraîne la fin de la partie si la combinaison est la bonne, ou élimine au moins cette combinaison.
Or l'ensemble $S_{n,p}$ des combinaisons possibles à n caractères parmi p est une partie de D_p^n, qui est un ensemble fini, donc est lui-même fini.
L'algorithme termine donc (en moins de $\#S_{n,p}$ étapes).

Modélisation et résolution par CSP

Abstraitisation

Les algorithmes de cette section, réunis dans `solver.py`, se veulent génériques, en le sens que leur implémentation ne suppose aucune connaissance préalable du problème à résoudre. En particulier, aucune hypothèse n'est faite sur la forme des domaines des variables ou des contraintes.

Par ailleurs, nous remarquons que tous les algorithmes demandés peuvent en fait être rangés seulement en deux "catégories" :

- *Engendrer et Tester*.
- *Retour arrière chronologique*, et *Forward checking* avec différentes méthode de simplification des domaines.

En effet, on peut voir l'algorithme de retour arrière chronologique comme un forward checking dans lequel on n'élimine aucune variable des domaines restants, et l'algorithme de forward checking lui-même ne dépend pas de la méthode d'élimination des variables, donc A2, A3, A4 et A5 peuvent être considérés comme le même algorithme.

Hiérarchie des solveurs

Pour ne pas répéter du code, nous définissons une classe abstraite `Solver`, de laquelle héritera une classe par algorithme.

Nous définissons ensuite les classes `GenerateRandomAndTest` et `EnumerateAndTest` qui implémentent l'algorithme *Engendrer et Tester* (cf. question 1), ainsi qu'une classe abstraite `ForwardChecking`, de laquelle hériteront `Backtracking`, `ForwardAllDiff`, `Forward2Diff` et `ImprovedForward` (respectivement A2, A3, A4 et A5 dans le sujet).

L'architecture peut donc être résumée par la figure suivante :

```
Solver
|
|--- GenerateRandomAndTest
|
|--- EnumerateAndTest
|
|--- ForwardChecking
     |
     |--- Backtracking
     |
     |--- ForwardAllDiff
     |
     |--- Forward2Diff
     |
     |--- ImprovedForward
```

`Solver`

De part la volonté de généralité que nous nous sommes imposée, afin de résoudre le problème, il est nécessaire de fournir une fonction déterminant si une assignation (potentiellement partielle) satisfait les contraintes.

Un Solver doit donc recevoir les paramètres suivants :

`domains`	La liste des domaines des variables (sous forme de `set`).
`check_satisfaction_func`	Fonction prenant en paramètre une liste de contraintes et une combinaison. Détermine si la combinaison satisfait toutes les contraintes .

Remarque : La combinaison à trouver n'est évidemment pas donnée à la classe solver, même si cela pourrait potentiellement alléger certaines syntaxe, afin d'écarter tout "soupçon".

`ForwardChecking`

Comme l'algorithme de forward checking ne dépend pas de la manière d'éliminer en amont des valeurs dans les domaines, la fonction `solve` est commune. Une classe fille n'aura qu'à implémenter une fonction membre `forward_func`.

Choix d'implémentation

- La fonction membre `solve` de la classe `Solver` ne renvoie pas une combinaison mais un *générateur* (au sens pythonique) de combinaisons. Cela nous permet d'éviter de tester à nouveau des combinaisons déjà évaluées après avoir fait une proposition (si une combinaison n'était pas valide pour un certain ensemble de contraintes, elle ne sera *a fortiori* pas valide avec une contrainte en plus).
- Nous avons fait l'hypothèse que les algorithmes, à l'exception du forward checking, ne tenaient pas compte de la contrainte alldiff. A défaut, l'algorithme de backtracking et celui de forward checking ne présentaient plus qu'une différence essentiellement langagière.
- Le domaine d'une variable sera représentée par `set` contenant les valeurs que peut prendre cette variable.

Question 1.1

Nous proposons deux versions pour l'algorithme *engendrer et tester*.
La première, `GenerateRandomAndTest`, génère des combinaisons aléatoirement jusqu'à en trouver une satisfaisant les contraintes. La seconde, `EnumerateAndTest`, énumère toutes les combinaisons jusqu'à en trouver une satisfaisante.

Pour $n = 4$ et $p = 8$

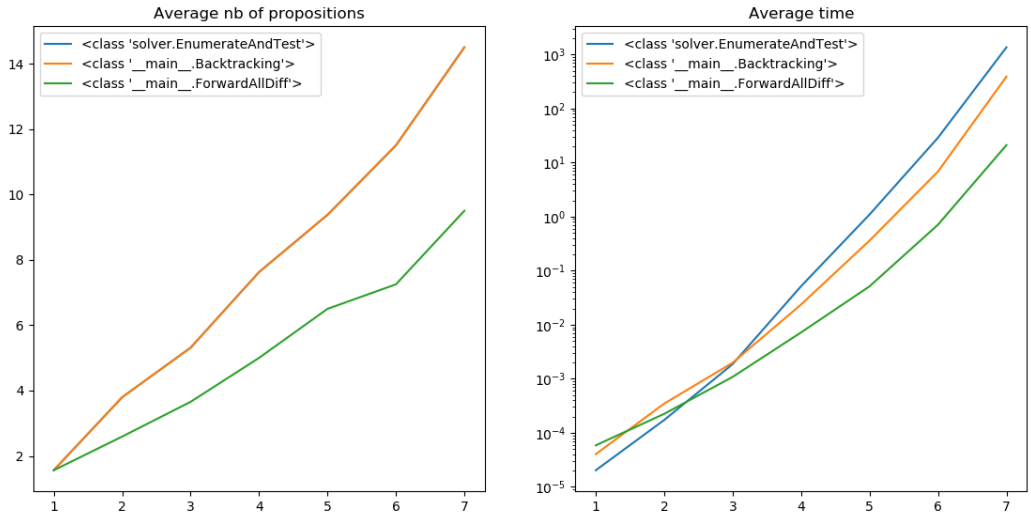
Pour gagner un peu en précision, on effectue les tests sur un total de 1000 instances (évidemment, les mêmes pour tous les algorithmes, afin de comparer ce qui est comparable). Les résultats pour les différents algorithmes sont les suivants :

```
Testing <class 'solver.GenerateRandomAndTest'> (N = 1000)
nb of propositions
avg : 5.5
std : 1.04
average time : 0.081

Testing <class 'solver.EnumerateAndTest'> (N = 1000)
nb of propositions
avg : 7.5
std : 1.02
average time : 0.040

Testing <class '__main__.Backtracking'> (N = 1000)
nb of propositions
avg : 7.5
std : 1.02
average time : 0.019

Testing <class '__main__.ForwardAllDiff'> (N = 1000)
nb of propositions
avg : 4.9
std : 0.95
average time : 0.006
```

En faisant varier n et p

Nous nous sommes restreints comme proposé à $p = 2n$. De plus, comme `GenrateRandomAndTest` était clairement la plus longue, nous l'avons exclue de ce graphique.

Le graphe des temps est en ordonnée logarithmique.



Analyse des résultats

Nombre de propositions On remarque tout d'abord que la génération aléatoire semble faire moins de propositions que l'énumération. Cette mesure est surprenante, mais ne semble pas totalement dénuée de sens ; On peut en effet supposer que faire des propositions aléatoires (mais respectant toujours les contraintes) introduira plus de "diversité" dans les combinaisons proposées, permettant ainsi de réduire plus rapidement l'espace des solutions.

On remarque ensuite que *Enumerer et tester* et le *retour arrière chronologiques* proposent exactement le même nombre de combinaisons. Cela n'est pas étonnant, puisque le retour arrière chronologique permet seulement d'éliminer plus vite des portions de l'espace des solutions, mais n'ajoute aucune "logique".

On peut enfin être surpris de constater que l'algorithme de retour arrière chronologique fasse en moyenne plus de propositions que les algorithmes de forward checking, qui ne semblent permettre que d'éliminer (encore) plus vite des solutions. La subtilité ici est que nous avons "ajouté" au forward checking la connaissance de la contrainte alldiff, qui n'est pas dans les deux autres. L'algorithme n'explore donc qu'un espace plus petit.

Question 1.2

On crée une nouvelle classe fille de `ForwardChecking`, qu'on nomme `Forward2Diff` qui fait sensiblement la même chose que `ForwardAllDiff` : dès qu'une valeur apparaît deux fois dans une combinaison, on l'enlève de tous les domaines restants.
On modifie la fonction `generate_random_combination` en une fonction `generate_random_2_combination` qui génère une combinaison pouvant contenir (au plus) deux fois le même caractère.
On n'effectuera cependant pas de comparaison avec `ForwardAllDiff` comme demandé dans le sujet car cela ne nous semble pas avoir de sens :

- Soit on compare sur le même ensemble de combinaisons, qui satisfont la contrainte AllDiff ; `Forward2Diff` sera plus lent et proposera plus de combinaisons, puisqu'il

parcourt un ensemble des solutions plus grand.

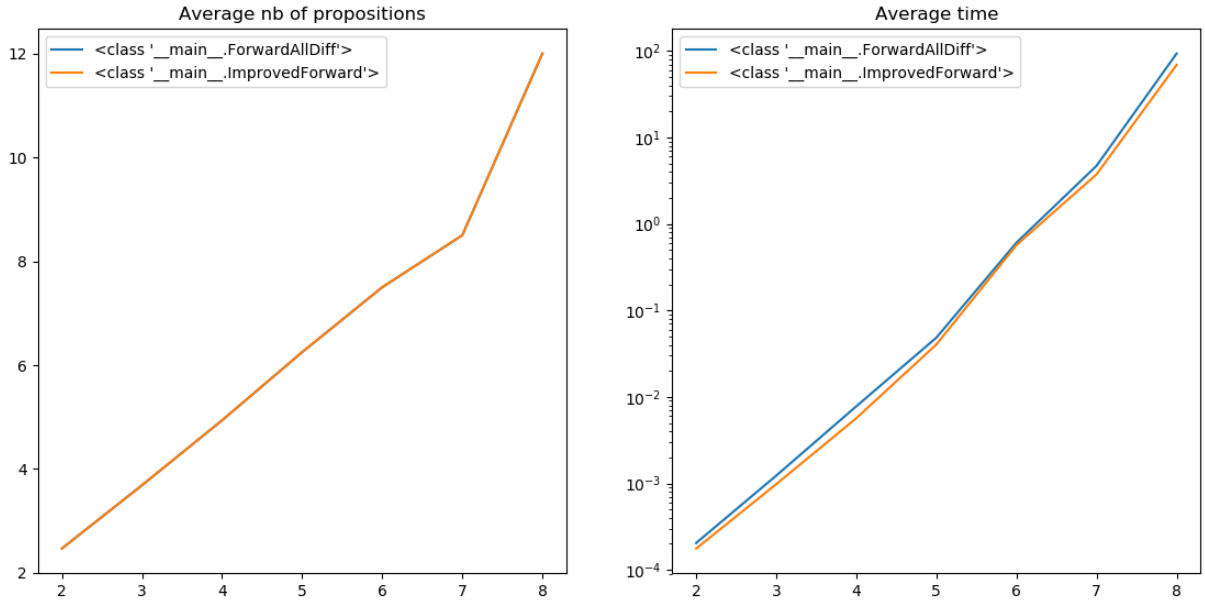
- Soit on compare sur des solutions qui n'ont pas plus de deux fois la même valeur, mais alors `ForwardChecking` ne trouvera pas forcément la solution.
- Soit on compare sur des jeux de données différents ; il va de soi que `Forward2Diff` proposera plus de combinaisons et sera moins rapide, mais à cause du fait qu'il répond à un problème plus "difficile" (il y a plus de possibilités à explorer).

Question 1.3

Notre méthode d'élimination repose sur la remarque suivante :

Le nombre de couleurs en commun (bien ou mal placées) entre la bonne combinaison et un essai est égal au nombre de couleurs bien placées plus le nombre de couleurs mal placées.

Par exemple, si pour $n = 4$ et $p = 8$ on a testé `1234` et obtenu `(1, 1)`, alors si on cherche une solution commençant par `12`, les couleurs suivantes ne pourront être ni `1`, ni `2`, ni `3` ni `4`.



On constate que le nombre de propositions est le même, pour la même raison qu'avec `Backtracking`.
Les temps d'exécution sont aussi remarquablement similaires, ce qui est plus surprenant. Après avoir vérifié que les fonctions étaient correctement appelées, les domaines correctement vidés, nous concluons que le temps de calcul supplémentaire nécessaire à `ImprovedForward` (qui réalise bcp de comparaisons en plus) compense le temps d'exploration de branches qu'il permet d'économiser.

Modélisation et résolution par algorithme génétique

Abstraitisation

Comme dans la partie 1, il nous a semblé que les algorithmes demandés partageaient une forte similarité. Nous avons donc décidé d'abstraire autant que possible le code.

Nous implémentons une classe abstraite `Genetic` qui hérite de `Solver`. Les solveurs correspondant aux algorithmes demandés n'ont qu'à hériter de cette classe et implémenter une fonction membre `genetic_choice_func`, qui représente la méthode de sélection

Question 2.1

L'algorithme génétique a été implémenté avec trois opérateurs de mutation qui sont les suivants:

- changement aléatoire d'un caractère
- échange entre deux caractères
- inversion de la séquence de caractères entre deux positions aléatoires.

Pour les choix de paramètres, nous avons fixé l'ensemble E à une taille de 50, la taille de la population et le nombre de générations est 100 et la probabilité de mutation à 0.8.

Question 2.2

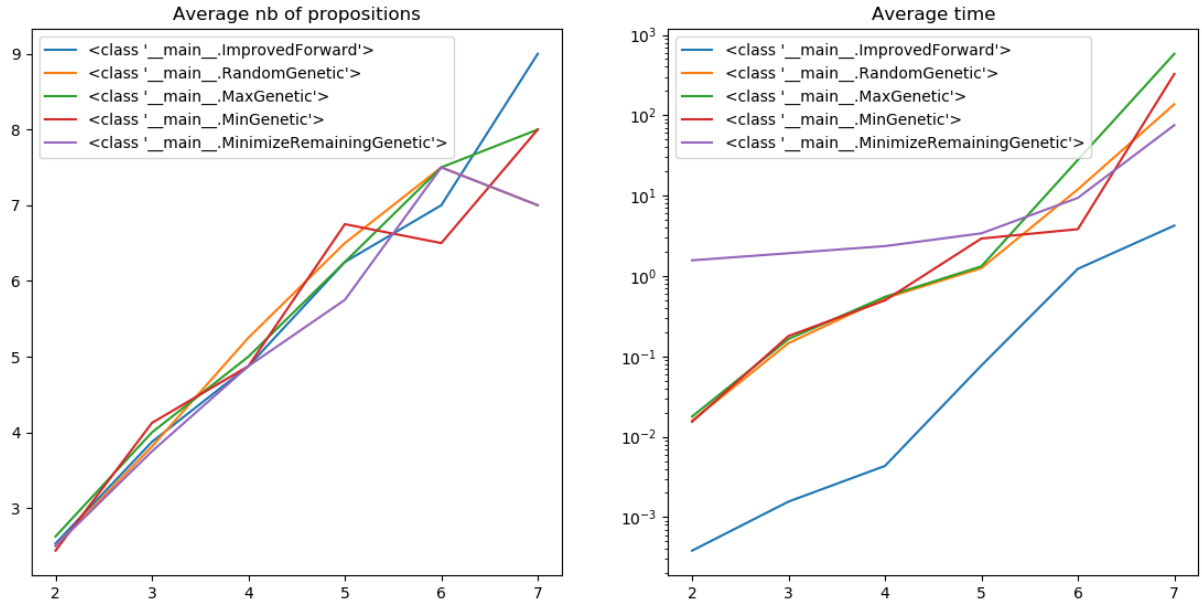
Les variantes AG2 et AG3 de l'algorithme génétique AG choisissent le prochain code en fonction de sa similarité avec les codes compatibles précédents, AG2 choisit donc le plus similaire et AG3 le moins similaire.

La similarité est calculée en comparant chaque code avec tout les codes compatibles précédents, augmentant ainsi son score pour chaque caractère identique et chaque caractère présent dans le code à la mauvaise place.

On peut voir sur le graphe de la prochaine question que les trois algorithmes sont sensiblement équivalents en nombre de tentative et en temps d'exécution.

Question 2.3

Comparaison avec la meilleure méthode la partie 1.



On peut voir que le nombre de tentatives est sensiblement le même entre les algorithmes génétiques et le `ForwardAllDiff`. En revanche les algorithmes génétiques sont plus lents à trouver une solution que le `ForwardAllDiff`.

L'algorithme `MinimizeRemainingGenetic` semble avoir un coût statique important (même pour des combinaisons très petites, on fera bcp de comparaisons), mais ne pas croitre aussi vite en temps que les autres. Malheureusement, les temps de calcul nous ont empêchés de tester cette conjecture sur de plus grandes instances.