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

Rapport

LY Jean-Baptiste

Mai 2020

Table des matières

Chapitre 1

Introduction, objectifs et modalités

L'objectif 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*.

On définit qu'une tentative et un essai ont la même signification. Ces termes désignent un code qu'on a vraiment essayé, c'est-à-dire qu'on l'a comparé avec le code secret.

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.

Un code est dit "compatible" s'il respecte les contraintes induites par toutes les informations accumulées lors des essais précédents (on s'interdit de tester des combinaisons incompatibles avec l'information disponible, même si elles sont informatives).

Par exemple, si on a tenté à la première itération le code "0123" et que le codeur répond qu'un seul caractère est bien placé, le code "4567" n'est pas compatible. Le code "4167" est par contre compatible puisqu'un le caractère '1' est bien placé.

Tout au long du projet, pour le tout premier essai d'une partie, on génère un code compatible de manière aléatoire, afin d'acquérir des premières informations sur lesquelles on pourra s'appuyer pour les prochains essais. Afin de bien comparer les algorithmes et montrer des résultats représentatifs et cohérents, la première tentative (générée aléatoirement) et le code secret à deviner sont les mêmes pour tous les joueurs (représentant les différents algorithmes).

Ainsi tous les joueurs débutent la partie avec les mêmes premiers indices (donnés par la première tentative) et ont tous le même code secret à prédire. Cela permet une équité et évite des cas où par exemple, un joueur génère un premier code aléatoire et obtient par chance le code secret directement, ou alors génère un premier code aléatoire qui aurait tous les pièces mal placées, et auraient donc l'avantage sur les autres joueurs.

J'ai donc inclus cette notion dans le code permettant de tracer les courbes.

J'ai choisi d'implémenter les algorithmes en Python car c'est le langage recommandé et aussi pour sa simplicité dans le cadre de ce travail. Cela me permet de me concentrer uniquement sur la facette algorithmique du projet.

J'ai utilisé deux articles scientifiques qui sont indiquées dans la bibliographie. Le premier article *Efficient solutions for Mastermind using genetic algorithms* de Lotte Berghman, Dries Goossens, et Roel Leus, m'a permis de comprendre l'algorithme génétique appliqué au jeu du *master mind*. Le second article *Optimisation par algorithme génétique sous contraintes* de Nicolas Barnier et Pascal Brisse, m'a permis une ouverture sur l'algorithme hybride, une combinaison entre le CSP et l'algorithme génétique.

Les algorithmes peuvent être testés avec le fichier *mastermind_ demonstration.ipynb*. Les courbes sont tracées dans le fichier *mastermind_ courbes.ipynb*.

1.1 L'utilisation d'un tel programme permet de créer une séquence d'essais qui converge nécessairement vers la solution (le code caché).

L'utilisation d'un tel programme qui permet de créer une séquence d'essais converge nécessairement vers la solution. En effet, prenons un exemple : si une précédente tentative comporte aucune pièce bien placée et aucune pièce mal placée, alors la prochaine tentative ne comportera aucune pièce de cette précédente tentative. Elle comportera alors d'autres pièces, ce qui augmentera la probabilité d'avoir les bonnes pièces.

Dans le cas où des précédentes tentatives comportent des pièces bien placées et mal placées, la prochaine tentative comportera des pièces bien placées et mal placées situées dans ces précédentes tentatives. Au fur et à mesure de la partie, le nombre de pièces bien placées et mal placées des tentatives augmentera jusqu'à atteindre leur nombre maximal, et ainsi obtenir le code secret.

Chapitre 2

Modélisation et résolution par CSP

Dans cette partie, on décide de modéliser le problème de la recherche d'un code compatible avec l'information disponible par un CSP à n variables $X_1, ..., X_n$ de domaine D_p, avec $p = 2 \times n$.

2.1 Algorithme A1 : Engendrer et tester

L'algorithme *Engendrer et tester* consiste à tester toutes les solutions complètement instanciées. Dès qu'un code compatible est trouvé, on l'essaye.

Notations :

i : instanciation courante

V : liste de variables non-instanciées dans i

D : domaines des variables

n et $essaisPrecedents$: les contraintes

Algorithm 1: Engendrer et tester

```
Input: i, V, D, n, essaisPrecedents
Output: codeEssaiCoompatible
if nbreEssais = 0 then
    i = code.randomInitialize()
else
    if V = ∅ then
        return i
    else
        x_k = V[0]
        forall v ∈ D_{x_k}
         do
            if i ∪ (x_k → v).estCompatible() then
                return i ∪ (x_k → v)
            else
                i = EngendrerEtTester(i ∪ (x_k →
                  v), V \ {x_k}, D, n, essaisPrecedents)
                if i.estCompatible() then
                    break
return i
```

2.2 Algorithme A2 : Retour arrière chronologique

Cet algorithme reprend le même processus que l'algorithme *Engendrer et tester*, mais l'algorithme *Retour Arrière Chronologique (RAC)* consiste à vérifier à chaque instanciation, sa consistance locale, c'est-à-dire dans le cadre de ce projet, si les caractères de l'instanciation courante sont distincts deux à deux.

Si l'instanciation courante est consistante localement, alors on réitère de manière récursive l'algorithme sur cette instanciation, sinon on passe au prochain caractère et on reteste sa consistance locale. Dès qu'un code est compatible, on l'essaye.

Cela a pour effet d'éviter d'aller plus loin dans l'arbre d'exploration lorsqu'on remarque qu'il y a deux caractères identiques dans le code.

Notations :

i : instanciation courante

V : liste de variables non-instanciées dans i

D : domaines des variables

n et $essaisPrecedents$: les contraintes

Algorithm 2: RAC sans forward checking

```
Input: i, V, D, n, essaisPrecedents
Output: codeEssaiCoompatible
if nbreEssais = 0 then
    i = code.randomInitialize()
else
    if V = ∅ then
        return i
    else
        x_k = V[0]
        forall v ∈ D_{x_k}
         do
            if i ∪ (x_k → v) n'a que des caractères uniques then
                if i ∪ (x_k → v).estCompatible() then
                    return i ∪ (x_k → v)
                else
                    i = RAC(i ∪ (x_k →
                      v), V \ {x_k}, D, n, essaisPrecedents)
                    if i.estCompatible() then
                        break
return i
```

2.3 Algorithme A3 : Retour arrière chronologique avec forward checking sans doublon

Cet algorithme reprend le même processus que l'algorithme RAC précédent, mais il utilise en plus le foward checking n'utilisant que les contraintes all-diff : à chaque instanciation d'une nouvelle variable avec un nouveau caractère, on supprime ce caractère de tous les domaines des variables non instanciées.

Cela a pour effet d'éviter d'instancier des codes contenant deux caractères identiques et donc d'élaguer encore plus de branches dans l'arbre d'exploration.

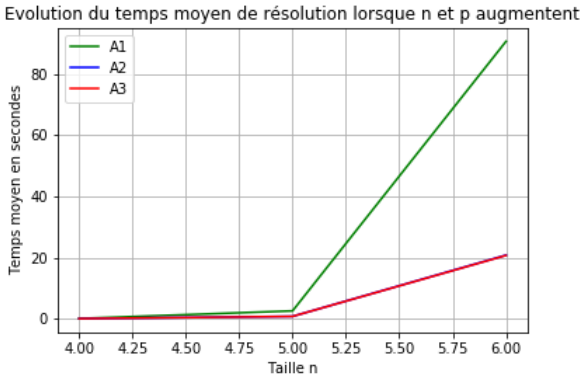
Notations :
i : instanciation courante
V : liste de variables non-instanciées dans i
D : domaines des variables
n et $essaisPrecedents$: les contraintes

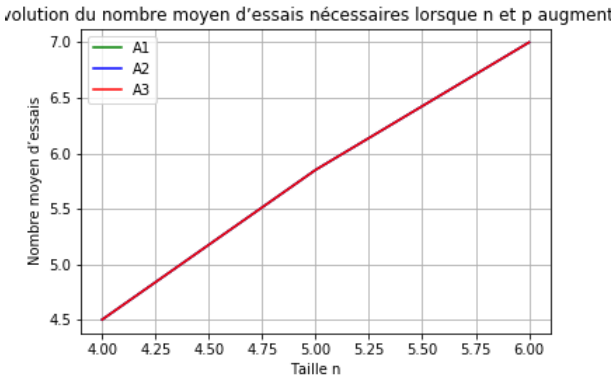
Algorithm 3: RAC avec forward checking sans doublon

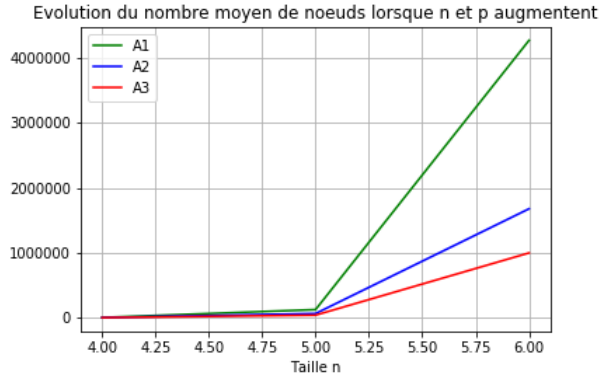
```
Input: i, V, D, n, essaisPrecedents
Output: codeEssaiCoompatible
if nbreEssais = 0 then
    i = code.random()
else
    if V = ∅ then
        return i
    else
        x_k = V[0]
        forall v ∈ D_{x_k}
         do
            D_bis = D \ {v}
            if i ∪ (x_k → v) n'a que des caractères uniques then
                if i ∪ (x_k → v).estCompatible() then
                    return i ∪ (x_k → v)
                else
                    i = RAC(i ∪ (x_k →
                      v), V \ {x_k}, D_bis, n, essaisPrecedents)
                    if i.estCompatible() then
                        break
return i
```

2.4 Comparaisons et analyses des algorithmes A1, A2 et A3

Sur 20 instances, pour $n = 4, 5$ et 6, les courbes obtenues sont :







On constate que les algorithmes $A2$ et $A3$ sont bien meilleurs en tout point que l'algorithme $A1$ grâce à la présence du retour arrière chronologique, avec un gain d'un facteur de 4 pour le temps moyen de résolution.

En revanche l'algorithme $A3$, malgré le forward checking qui permet un gain de 500 000 noeuds non explorés par l'algorithme $A2$, a un temps moyen de résolution similaire à ce dernier. Cela est dû sûrement à l'implémentation avec la présence de la fonction *copy*() en Python permettant de copier une liste de manière indépendante, qui a augmenté le temps de calcul.

De plus, on constate que ces trois algorithmes possèdent le même nombre de tentatives pour toutes les instances. En plus des conditions initiales identiques (même premier code aléatoire généré en tant que première tentative et même code secret à deviner) pour débuter la partie, cela s'explique logiquement par le fait que tous ces algorithmes suivent le même pattern. Seul le nombre de branches élaguées, et donc le nombre de noeuds visités change. Ainsi la performance de calcul change entre ces trois algorithmes.

2.5 Algorithme A4 : Retour arrière chronologique avec forward checking avec doublon

On considère une variante du problème où le code peut admettre jusqu'à deux occurences du même caractère (on relâche donc la contrainte stipulant que les caractères doivent être deux à deux distincts).

Cet algorithme reprend le même processus que l'algorithme précédent, mais on autorise jusqu'à deux occurences du même caractère. Ainsi on doit veiller que le code n'a plus de deux mêmes caractères instanciés. S'il possède un caractère instancié deux fois, alors il supprime celui-ci du domaine des variables.

Notations :

i : instanciation courante

V : liste de variables non-instanciées dans i

D : domaines des variables

n, $essaisPrecedents$ et $nbreOccurences$: les contraintes

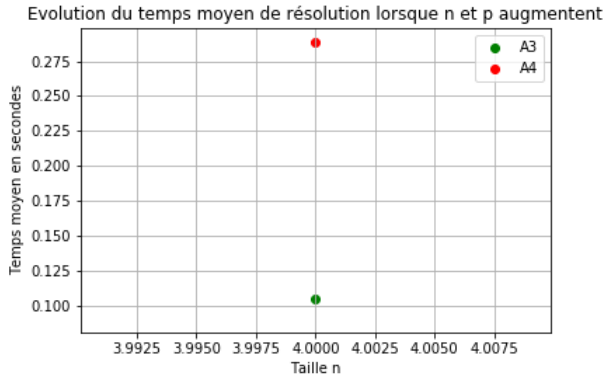
Algorithm 4: RAC avec forward checking avec doublon

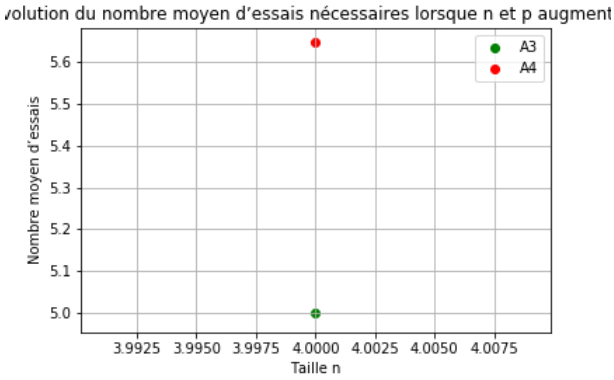
Input: $i, V, D, n, essaisPrecedents, nbreOccurences$
Output: $codeEssaiCoompatible$
if $nbreEssais = 0$ **then**
  $i = code.random()$
else
  if $V = \emptyset$ **then**
    return i
  else
    $x_k = V[0]$
    forall $v \in D_{x_k}$
    do
      $nbreOccurences[v] = nbreOccurences[v] + 1$
      if $nbreOccurences[v] = 2$ **then**
        $D_{bis} = D \setminus \{v\}$
      if $i \cup (x_k \to v)$ *n'a que des caractères uniques* **then**
        if $i \cup (x_k \to v).estCompatible()$ **then**
          return $i \cup (x_k \to v)$
        else
          $i = RAC(i \cup (x_k \to v), V \setminus \{x_k\}, D_{bis}, n, essaisPrecedents)$
          if $i.estCompatible()$ **then**
            break
return i

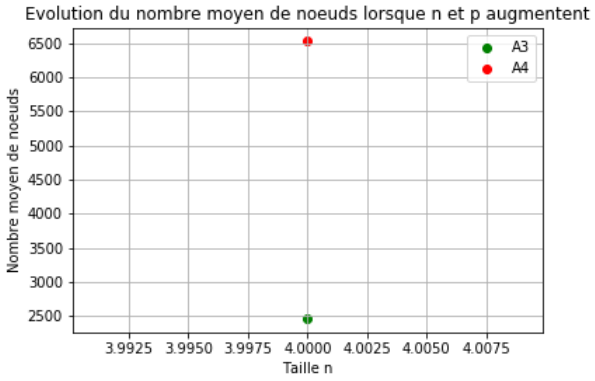
2.6 Comparaisons et analyses des algorithmes A3 et A4

À noter tout d'abord, que les résultats sont similaires aussi bien dans le cas initial du projet où on donnait une même première tentative et un même code secret à deviner pour les deux algorithmes (ce qui obligerait de donner des codes sans occurences doubles) et le cas où on donnerait une première tentative différente et un code secret différent (on accepte donc de donner des codes secrets à deviner possédant des doublons pour l'algorithme $A4$).

Sur 20 instances, pour $n = 4$ et $p = 8$, les graphes obtenus sont :







On constate que l'algorithme $A3$ est bien "meilleur" aussi bien en temps de calcul, qu'en nombre d'essais moyens et qu'en nombre moyen de noeuds explorés.

Cela est dû, dans le cadre de cette variante du problème, à la contrainte stipulant que les caractères doivent être deux à deux distincts. En effet l'algorithme $A4$ acceptera des doublons dans ses noeuds, et proposera des tentatives avec deux occurences du même caractère.

Cette annulation de contrainte provoque un nombre de noeuds davantage à explorer, 2,6 fois plus de noeuds pour l'algorithme $A4$ par rapport à l'algorithme $A3$, et donc paralèllement un temps moyen de résolution aussi 2 fois plus long et un nombre de tentatives accru pour l'algorithme $A4$.

Par exemple, pour une instance donnée, on a :

```
premiere tentative : ['1', '4', '5', '0']
code_secret sans doublon : ['5', '0', '6', '1']

Résumé de toutes les tentatives du joueur 3 :
Tentative 1 :
{0: '1', 1: '4', 2: '5', 3: '0' bien placés : 0 mal placés : 3
Tentative 2 :
{0: '0', 1: '1', 2: '2', 3: '4'} bien placés : 0 mal placés : 2
Tentative 3 :
{0: '3', 1: '0', 2: '1', 3: '5'} bien placés : 1 mal placés : 2
Tentative 4 :
{0: '3', 1: '5', 2: '4', 3: '1'} bien placés : 1 mal placés : 1
Tentative 5 :
{0: '5', 1: '0', 2: '6', 3: '1'} bien placés : 4 mal placés : 0
Victoire au bout de 5 tentative(s).
Joueur 3 :
Nombre de noeuds : 3070
Temps d'exécution : 0.1576709747314453 secondes.

Résumé de toutes les tentatives du joueur 4 :
Tentative 1 :
{0: '1', 1: '4', 2: '5', 3: '0'} bien placés : 0 mal placés : 3
Tentative 2 :
{0: '0', 1: '0', 2: '1', 3: '4'} bien placés : 1 mal placés : 1
Tentative 3 :
{0: '0', 1: '1', 2: '2', 3: '5'} bien placés : 0 mal placés : 3
Tentative 4 :
{0: '2', 1: '5', 2: '0', 3: '4'} bien placés : 0 mal placés : 2
Tentative 5 :
{0: '5', 1: '0', 2: '3', 3: '1'} bien placés : 3 mal placés : 0
Tentative 6 :
{0: '5', 1: '0', 2: '6', 3: '1'} bien placés : 4 mal placés : 0
Victoire au bout de 6 tentative(s).
Joueur 4 :
Nombre de noeuds : 7186
Temps d'exécution : 0.2768361568450928 secondes.
```

2.7 Algorithme A5 : Retour arrière chronologique avec amélioration du forward checking sans doublon

Cet algorithme reprend le même processus que l'algorithme A3, mais il possède une amélioration du forward checking qui permette de tenir compte d'autres contraintes que la contrainte all-diff. Cette amélioration se caractérise par la présence de la méthode *estPreCompatible*() qui est une amélioration de la méthode *estCompatible*(). Cette méthode *estPreCompatible*() peut vérifier la compatibilité de l'instanciation partielle courante alors qu'elle n'est pas encore complète. Si elle possède un nombre de pièces bien placées ou un nombre de pièces mal placées supérieure à celle d'un vrai essai au cours de la vérification de compatiblité, alors il est inutile de continuer l'exploration de la branche courante dans l'arbre d'exploration.

Par exemple prenons cet exemple ci-dessous :

```
n = 4
premiere tentative : ['2', '5', '4', '6']
code_secret sans doublon : ['2', '1', '7', '3']

Tentative 1 : {0: '2', 1: '5', 2: '4', 3: '6'}
bien placés : 1 mal placés : 0
noeud : ['0']
noeud : ['0', '1']
noeud : ['0', '1', '2']
noeud : ['0', '1', '3']
noeud : ['0', '1', '3', '2']
noeud : ['0', '1', '3', '4']
noeud : ['0', '1', '3', '5']
noeud : ['0', '1', '3', '6']

Tentative 2 : {0: '0', 1: '1', 2: '3', 3: '6'}
bien placés : 1 mal placés : 1
...
```

Au noeud `['0', '1', '2']`, on constate que l'algorithme élague une branche dans l'arbre d'exploration en passant directement au noeud `['0', '1', '3']` au lieu de continuer sur la feuille `['0', '1', '2','3]` comme l'algorithme $A3$ le ferait.

L'algorithme $A5$ a détecté que le nombre de pièces mal placées augmenté de 1 en comparant l'instanciation partielle `['0', '1', '2']` et la première tentative `['2', '5', '4', '6']` (`'2'` est mal placée).

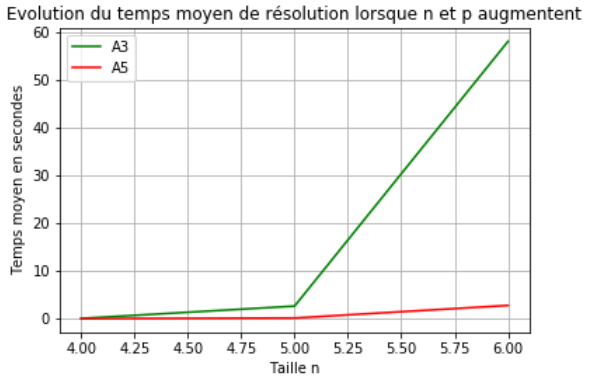
Ainsi on peut en déduire que le code n'aura aucune chance d'être compatible si on continue sur cette instanciation partielle. Il est donc préférable d'élaguer la branche courante.

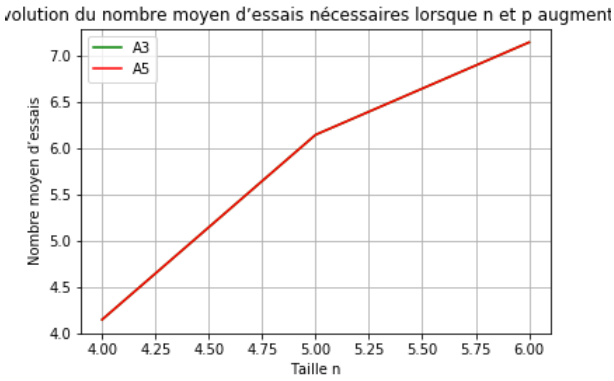
Algorithm 5: RAC avec forward checking sans doublon avec amélioration

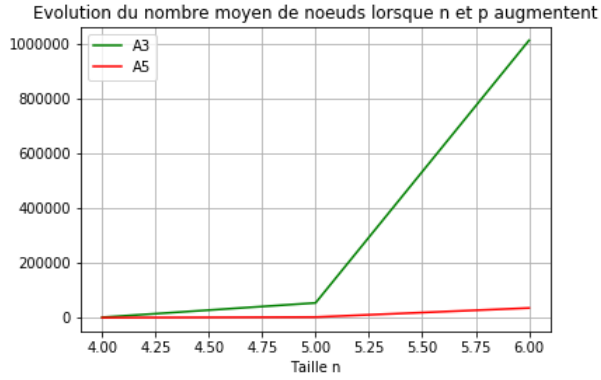
```
Input: i, V, D, n, essaisPrecedents
Output: codeEssaiCoompatible
if nbreEssais = 0 then
    i = code.random()
else
    if V = ∅ then
        return i
    else
        x_k = V[0]
        forall v ∈ D_{x_k}
        do
            D_bis = D \ {v}
            if i ∪ (x_k → v) n'a que des caractères uniques then
                if i ∪ (x_k → v).estCompatible() then
                    return i ∪ (x_k → v)
                else
                    if i ∪ (x_k → v).estPreCompatible() then
                        i = RAC(i ∪ (x_k →
                          v), V \ {x_k}, D_bis, n, essaisPrecedents)
                        if i.estCompatible() then
                            break
return i
```

2.8 Comparaisons et analyses des algorithmes A3 et A5

Sur 20 instances, pour $n = 4, 5$ et 6, les courbes obtenues sont :







On constate que grâce à son amélioration du forward checking, l'algorithme $A5$ est bien meilleur que l'algorithme $A3$. Le nombre de noeuds visités a drastiquement bien diminué avec l'algorithme $A5$, soit environ 20 fois moins qu'avec l'algorithme $A3$, et par conséquent, le temps de résolution a aussi diminué d'un facteur de 12 pour l'algorithme $A5$. Le nombre de tentatives reste bien évidemment identique pour les deux algorithmes puisqu'ils aboutissent toujours sur la même feuille dans l'arbre d'exploration.

Ainsi on peut se demander, si il est possible d'améliorer cet aspect algorithmique. Tous les algorithmes jusqu'à maintenant, suivent le même pattern, seule la performance de calcul change. Cela est dû à l'absence du hasard (omis le fait de générer un premier code aléatoire en première tentative).

Le choix du caractère à instancier pour la variable courante est le seul moment où il est possible d'introduire cette notion d'aléatoire présente dans les algorithmes dits de "*Las Vegas*".

2.9 Algorithmes A6 et A7 : Las Vegas

2.9.1 Algorithme A6 : Las Vegas naïf

L'algorithme *A6 Las Vegas naïf* consiste à générer un code aléatoire avec $code.random()$, s'il est compatible avec les essais précédents, alors on essaye ce nouveau code. Sinon, on regénère un nouveau code random et on reteste sa compatibilité.

Pour générer le code de manière aléatoire avec $code.random()$, on veille à éviter les doublons, en supprimant du domaine le caractère qui vient d'être instancié. À chaque itération, on instancie un caractère en veillant à être distinct deux à deux. Il est ainsi impossible de générer par exemple, un code comme "1241".

Il est très inspiré de l'algorithme *A3 RAC avec forward checking* car on utilise le foward checking. À la différence près que cet algorithme instancie de manière totalement aléatoire les variables (bien qu'il soit impossible d'avoir des doublons grâce au forward checking). Ainsi il explore l'arbre de recherche de façon aléatoire. En revanche, cet algorithme peut aussi théoriquement retourner un code dont sa compatibilité a déjà été testée négativement, ce qui se traduit par une exploration inutile de feuilles et donc de solutions, déjà visitées.

Algorithm 6: Las Vegas naïf

```
Input: D, n, essaisPrecedents
Output: codeEssaiCompatible
if nbreEssais = 0 then
    code = code.random()
else
    while not code.Compatible do
        code = code.random()
return code
```

2.9.2 Algorithme A7 : A5 version Las Vegas

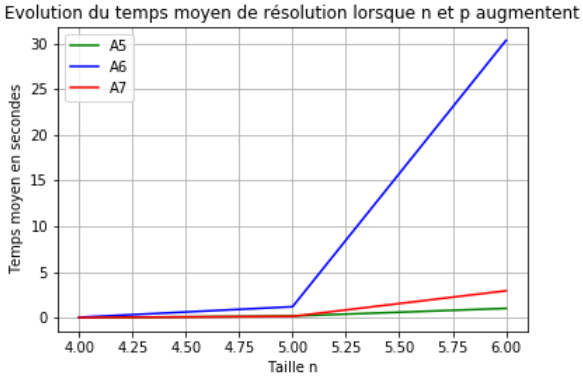
L'algorithme $A7$ (dit "$A5$ version *Las Vegas*") reprend l'algorithme $A5$ mais consiste à modifier le choix du caractère de la variable courante à instancier, qui est faite désormais de manière aléatoire.

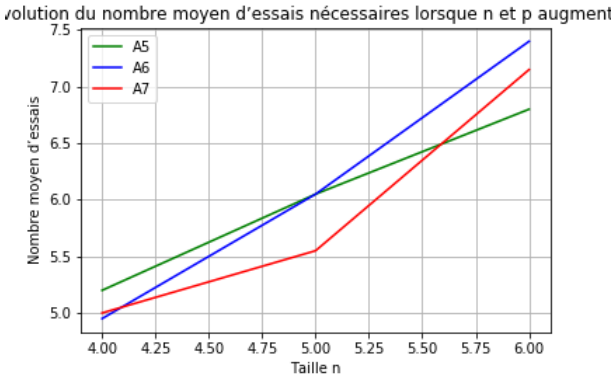
Algorithm 7: RAC avec forward checking sans doublon avec amélioration version Las Vegas

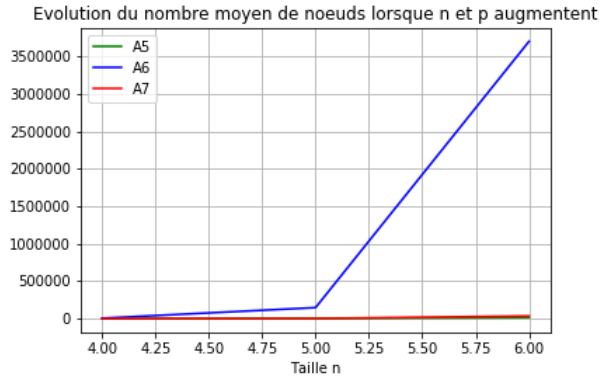
```
Input: i, V, D, n, essaisPrecedents
Output: codeEssaiCoompatible
if nbreEssais = 0 then
    i = code.random()
else
    if V = ∅ then
        return i
    else
        x_k = V[0]
        forall v ∈ D_{x_k}
         do
            v = random.choice(D_xk)
            D_bis = D \ {v}
            if i ∪ (x_k → v) n'a que des caractères uniques then
                if i ∪ (x_k → v).estCompatible() then
                    return i ∪ (x_k → v)
                else
                    if i ∪ (x_k → v).estPreCompatible() then
                        i = RAC(i ∪ (x_k →
                          v), V \ {x_k}, D_bis, n, essaisPrecedents)
                        if i.estCompatible() then
                            break
return i
```

2.9.3 Comparaisons et analyses

Sur 20 instances, pour $n = 4, 5$ et 6, les courbes obtenues sont :

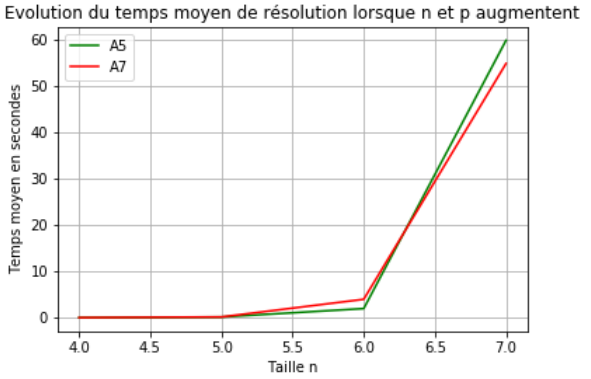


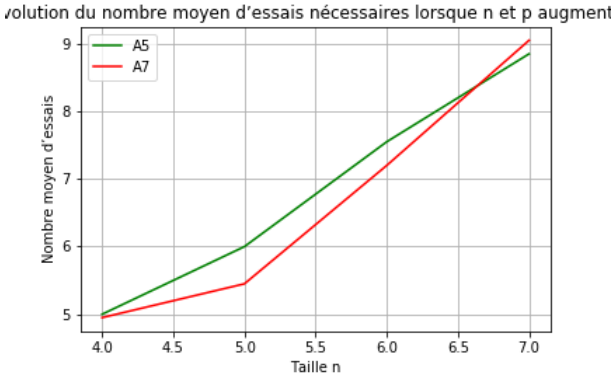


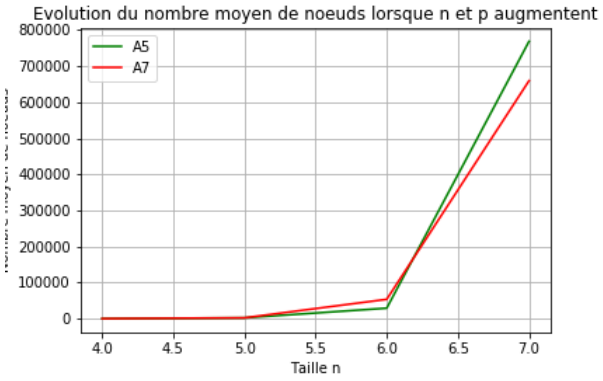


On constate que l'algorithme *Las Vegas naïf* est loin d'être le meilleur, en revanche concernant l'algorithme $A5$ et sa version dite de *Las Vegas* $A7$ ont des résultats très intéressants. Leur temps moyen de résolution, leur nombre moyen d'essais et leur nombre moyen de noeuds sont proches, bien que l'algorithme $A5$ ait un léger avantage sur l'algorithme $A7$ pour $n = 6$, mais pour $n = 4$ ou 5 le cas est inversé, cela est évidemment aléatoire.

J'ai donc décidé de retracer des graphes , toujours avec 20 instances, mais avec $n = 4, 5, 6$ et 7 et seulement pour les algorithmes les plus efficaces $A5$ et $A7$, qui sont présents ci-dessous :







En relançant ces deux algorithmes, on constate plus ou moins les mêmes résultats pour $A5$ et $A7$, mais l'avantage de $A7$ a l'air plus marqué.

Chapitre 3

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 (cas où les caractères du code sont deux à deux distincts).

3.1 A8 : L'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. L'ensemble E de codes compatibles présentera une taille maximale $maxsize$ et l'algorithme génétique s'arrêtera dès que la taille maximale est atteinte. L'avantage de l'utilisation d'un ensemble de codes compatibles est que l'on peut ensuite tenter de déterminer parmi cet ensemble la meilleure prochaine tentative.

Afin de limiter les temps de calcul, l'algorithme génétique peut s'arrêter après un nombre maximal de générations $maxgen$ (l'algorithme génétique pourra donc s'arrêter même si la taille de E est inférieure à $maxsize$).

Il se peut que l'algorithme ne retourne aucun code compatible après avoir atteint le nombre maximal de générations. Dans ce cas, on continue la recherche d'un code compatible jusqu'à ce qu'un timeout de 5 minutes soit atteint. Si à la fin de ce timeout aucun code compatible n'a été généré, on peut considérer que la méthode a échoué.

La population de la première génération est générée aléatoirement. La population évoluera grâce aux opérateurs de croisements et de mutations.

Le croisement, ou crossover, consiste à prendre deux parents, un indice aléatoire, et fusionner la partie gauche par rapport à cet indice d'un parent avec la partie droite de l'autre parent afin de générer un enfant.

Les différentes mutations comprennent le changement aléatoire d'un caractère, échange entre deux caractères, et l'inversion de la séquence de caractères entre deux positions aléatoire. Ces différentes mutations ont la même probabilité d'être utilisée.

La probabilité d'un code d'être sélectionné comme parent est proportionnelle à sa valeur adaptative ("fitness"). La valeur adaptative d'un code est liée aux nombres d'incompatibilités avec les essais précédents. Dès qu'un code compatible est trouvé, il sera ajouté à l'ensemble E.

Soit bp le nombre de pièces bien placées d'une tentative t (code qu'on a vraiment essayé), soit bp' et mp', respectivement le nombre de pièces bien placées et mal placées si un code t' d'un individu était une tentative avec comme code secret à deviner une ancienne vraie tentative t, et nt le nombre de tentatives qu'on a faites jusqu'à maintenant.
La diférence entre bp et bp', et entre mp et mp' indique la qualité du code t'. Si leurs différences sont égales à 0, alors cela signifie que le code t' est compatible.

$$fitness(t', nt) = \sum_{i=0}^{nt} \left(|bp'_i(t') - bp_i|\right) + \sum_{i=0}^{nt} \left(|mp'_i(t') - mp_i|\right)$$

Plus la fitness est faible, meilleure elle sera.

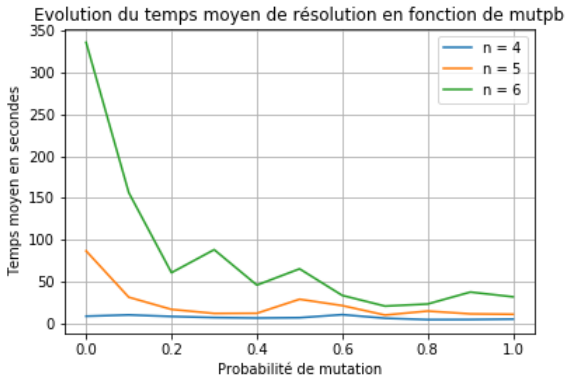
Dès qu'un code est compatible, on l'ajoute dans E à chaque fin de génération. La prochaine population est remplacée entièrement par la précédente, néanmoins les parents qui vont générer la prochaine génération sont sélectionnés aléatoirement parmi les 50% meilleurs individus de la dernière population.

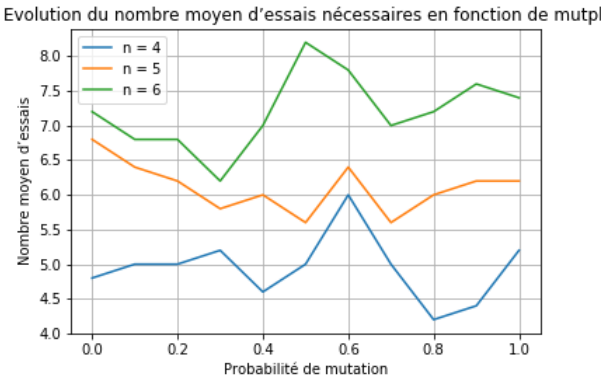
Algorithm 8: Algorithme génétique

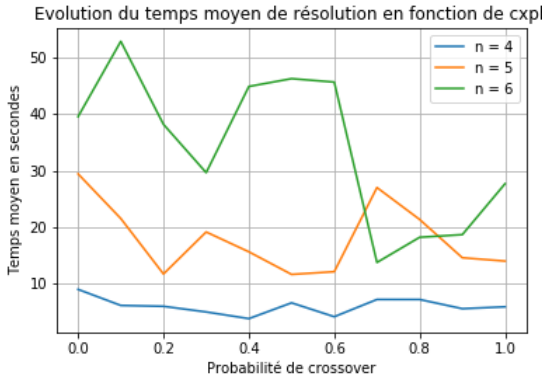
Input: $D, n, essaisPrecedents, maxsize, maxgen, popsize, cxpb, mutpb, timeout$
Output: $codeEssaiCompatible$
if $nbreEssais = 0$ **then**
    $code = code.random()$
else
    $E = \{\}$
    $g = 0$
    $historique[nbreEssais] = fitness$
    $time.initialize()$
    while $E = \emptyset$ *ou si le timeout n'est pas encore atteint* **do**
        while $g \leq maxgen$ *et* $|E| \leq maxsize$ **do**
            Généreration d'une nouvelle population en utilisant le crossover, la mutation, l'inversion, et la permutation
            Calcul de la fitness
            Ajouter les codes compatibles dans E s'ils ne sont pas dans E
            $g = g + 1$
    if $E \neq \emptyset$ **then**
        return $code \in E$
    else
        La méthode a échoué

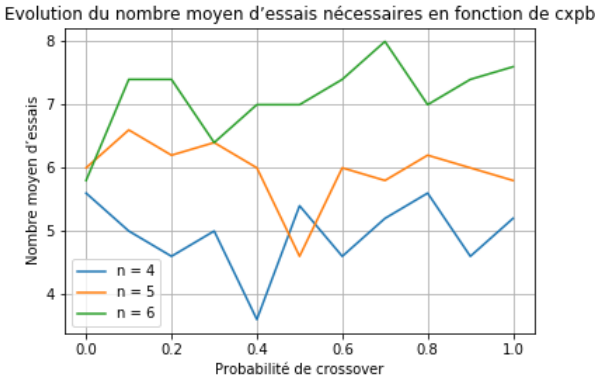
Il s'agit maintenant de choisir les bons paramètres afin d'obtenir des résultats de bonne qualité en un temps acceptable. Pour cela il faut fixer certains paramètres et en faire varier qu'un seul dans l'objectif de tester le nombre moyen de tentatives et le temps moyen de résolution.

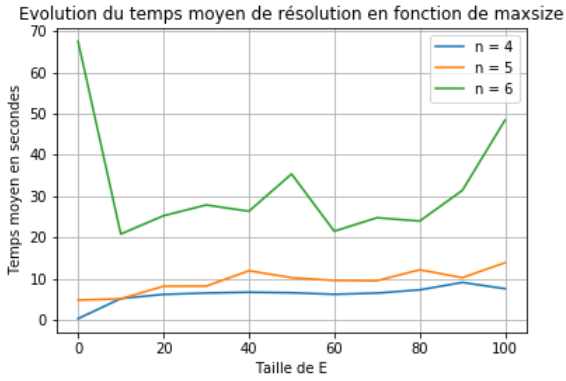
Sur 10 instances, on fixe par défaut $maxsize = 60$, $maxgen = 50$, $popsize = 50$, $cxpb = 0.8$ et $mutpb = 0.8$, on obtient les courbes ci-dessous :

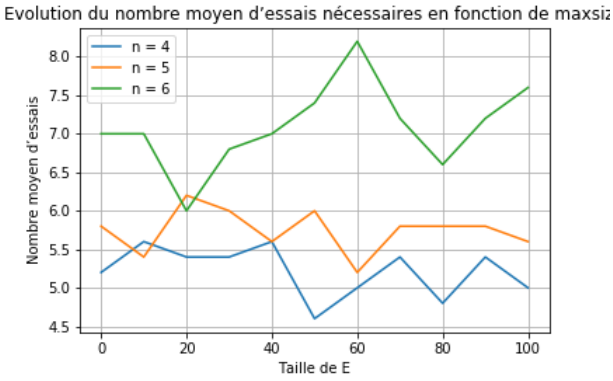


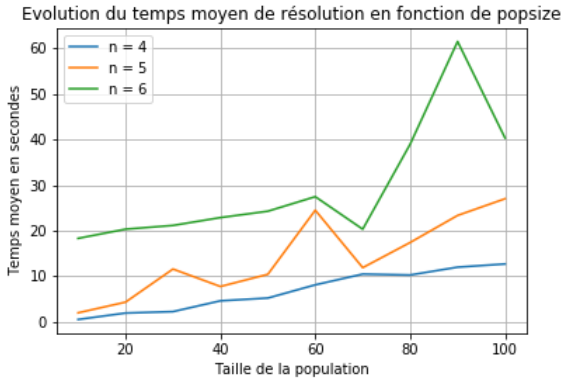


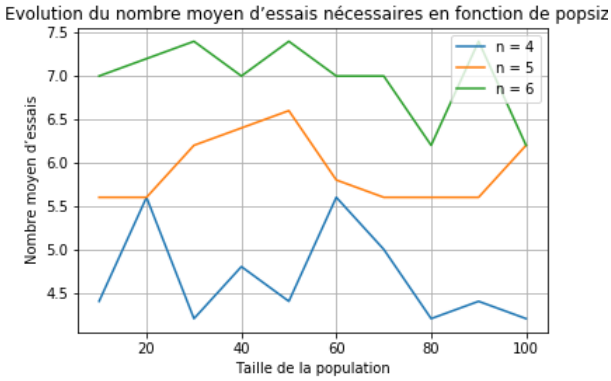

Evolution du temps moyen de résolution en fonction de cxpb
Temps moyen en secondes
Probabilité de crossover
n = 4
n = 5
n = 6

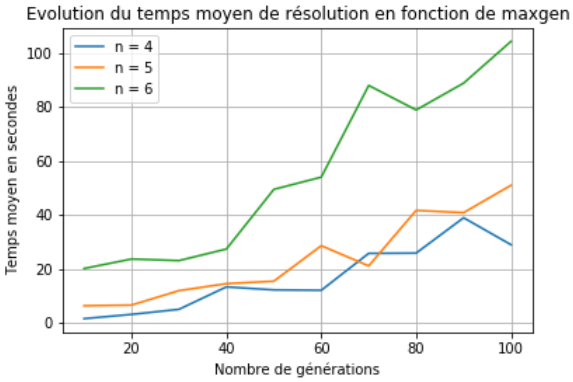

Evolution du nombre moyen d'essais nécessaires en fonction de cxpb
Nombre moyen d'essais
Probabilité de crossover
n = 4
n = 5
n = 6

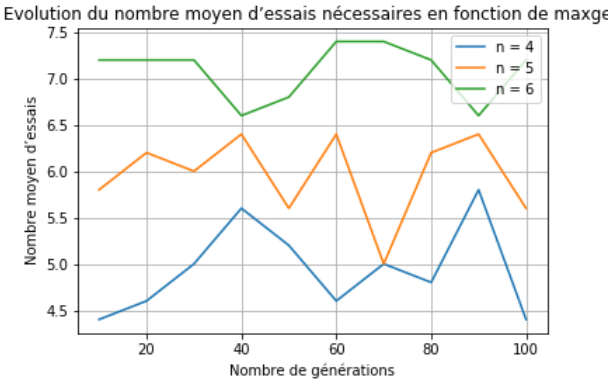

Evolution du temps moyen de résolution en fonction de maxsize
Temps moyen en secondes
Taille de E
n = 4
n = 5
n = 6


Evolution du nombre moyen d'essais nécessaires en fonction de maxsiz
n = 4
n = 5
n = 6
Nombre moyen d'essais
Taille de E


Evolution du temps moyen de résolution en fonction de popsize
n = 4
n = 5
n = 6
Temps moyen en secondes
Taille de la population


Evolution du nombre moyen d'essais nécessaires en fonction de popsiz
n = 4
n = 5
n = 6
Nombre moyen d'essais
Taille de la population





Grâce aux courbes obtenues, on peut constater des valeurs possédent un bon compromis entre le nombre de tentatives et le temps de résolution. Ainsi nn choisit alors de fixer pour la suite des analyses dans les prochaine parties :

Paramètres	*mutpb*	*cxpb*	*maxsize*	*popsize*	*maxgen*
Valeurs	0.8	0.8	60	70	70

3.2 Choix de la prochaine tentative de l'algorithme génétique

À la fin de l'algorithme génétique en cas de réussite, on a un ensemble E qui contient un ou des codes compatibles destinés à être proposer comme prochaine tentative.

Il existe plusieurs façons de choisir la prochaine tentative parmi E :
- Choix du code de façon aléatoire.
- 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.
- Choix du code présentant l'estimation la plus élevée du nombre de codes compatibles restants si un code était tenté.
- Choix du code présentant l'estimation la plus faible du nombre de codes compatibles restants si un code était tenté.

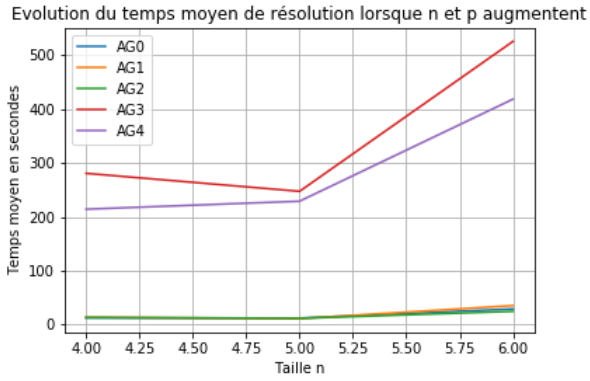
Le choix du code présentant le plus ou le moins de similarités avec les autres codes compatibles se fait de cette manière :
Soit $c \in E$, on détermine le nombre de pièces bien placées et mal placées si c était la tentatative et un autre $c^* \in E$ le code secret à deviner, pour chaque $c^* \in E\backslash\{c\}$. Ces valeurs sont sommées pour tous les $c^* \in E\backslash\{c\}$ et le nombre de pièces bien placées et le nombre de pièces mal placées ont le même poids. Le code qui possède le plus de similarités est le code ayant la meilleure qualité.

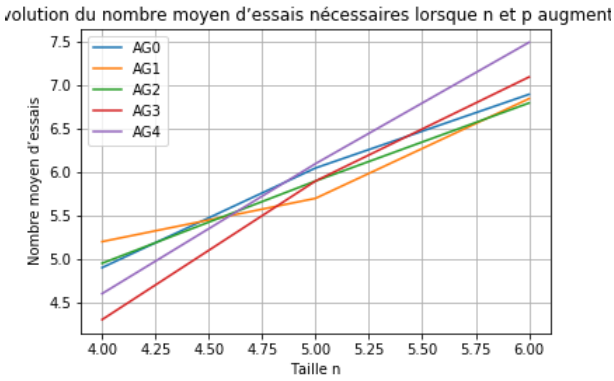
Le choix du code présentant l'estimation la plus élevée ou la plus faible du nombre de cas compatibles restants si un code était tenté se fait de cette façon :
On prend un sous-ensemble aléatoire $S \subseteq E$. Pour chaque $c \in E$ en tant que tentative et pour chaque $c^* \in S\backslash\{c\}$ en tant que possible code secret à deviner, on cherche combien d'autres codes dans $S\backslash\{c, c^*\}$ pourraient rester compatibles. Le code candidat c avec le minimum de codes compatibles restants est le code possédant la meilleure qualité. Plus la taille de $|S|$ est élevée, plus la qualité de la solution est meilleure, mais le temps de calcul prendra aussi parallèlement plus de temps. Dans la suite du projet, la taille de S est égale à la taille de l'ensemble E.

3.3 Comparaisons et analyses

Prenons le meilleur algorithme utilisant la modélisation et résolution par CSP qui est $A5$, et la meilleure façon de choisir la prochaine tentative parmi E pour l'algorithme génétique qui est ... .

Sur 20 instances, pour $n = 4, 5$ et 6, et soit l'algorithme génétique AG utilisant différentes façons de choisir la prochaine tentative :
$AG0$: Choix du code de façon aléatoire
$AG1$: Choix du code présentant le plus de similarités avec les autres codes compatibles
$AG2$: Choix du code présentant le moins de similarités avec les autres codes compatibles
$AG3$: Choix du code présentant l'estimation la plus faible du nombre de codes compatibles restants si un code était tenté
$AG4$: Choix du code présentant l'estimation la plus élevée du nombre de codes compatibles restants si un code était tenté
Les courbes obtenues sont :





On constate que la stratégie du choix du code présentant le moins de similarités avec les autres codes compatibles est meilleure que celle du choix du code présentant le plus de similarités avec les autres codes compatibles, et la stratégie du choix du code présentant l'estimation la plus faible du nombre de codes compatibles restants si un code était tenté est meilleure que celle du choix du code présentant l'estimation la plus élevée du nombre de codes compatibles restants si un code était tenté.

La meilleure stratégie semble être celle du choix du code présentant le moins de similarités avec les autres codes compatibles, bien que théoriquement d'après Lotte Berghman, Dries Goossens, et Roel Leus [1], la stratégie présentant l'estimation la plus faible du nombre de codes compatibles restants si un code était tenté devrait être la meilleure.

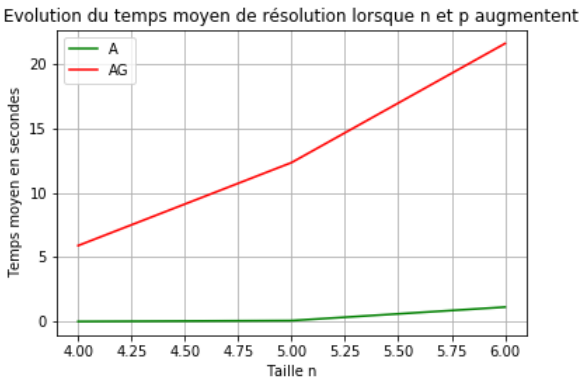
Cette différence de résultats peut s'expliquer par le nombre d'instances qui ne serait pas assez élevé (ou un défaut d'implémentation ?). Néanmoins considérons pour la suite, que le meilleur algorithme génétique serait avec le choix du code présentant le plus de similarités avec les autres codes compatibles.

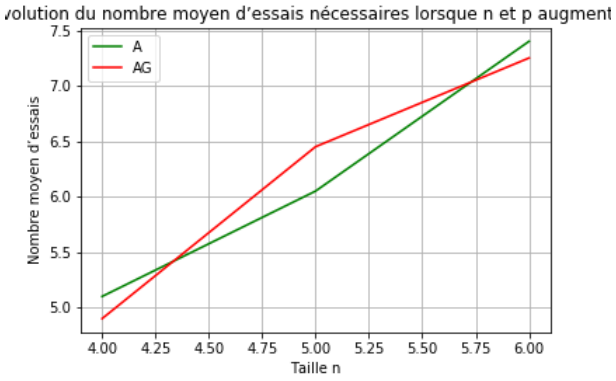
Chapitre 4

Comparaisons entre les modélisations et résolution par CSP et par algorithme génétique

Comparons les modélisations et résolution par CSP et par algorithme génétique. Prenons les meilleurs algorithmes de ces types de résolutions, les algorithmes $A5$ et $AG1$ (l'algorithme génétique avec le choix du code présentant le plus de similarités avec les autres codes compatibles).

Sur 20 instances, pour $n = 4, 5$ et 6, les courbes obtenues sont :





On constate donc que l'algorithme génétique a un nombre de tentatives assez similaire à celui du CSP, bien qu'il ait un léger avantage par rapport à ce dernier. En revanche le temps de résolution est plus grand évidemment pour l'algorithme génétique. Cette variation de qualité peut tout simplement se traduire par la présence de l'aléatoire avec l'algorithme génétique.

Chapitre 5

Algorithme hybride : mi-génétique mi-CSP

5.1 Algorithme A9 : L'algorithme hybride

Dans la partie précédente, on peut constater que l'algorithme génétique peut prendre beaucoup de temps à trouver un code compatible, voire échoue si on prend en compte un timeout. Dans ce cas, au lieu d'affecter un échec à la partie , on pourrait penser à changer d'algorithme en pleine partie, comme laisser la main à l'algorithme $A5$ qui est le meilleur parmi les algorithmes utilisant un CSP.

Néanmoins cela signifierait que l'algorithme $A5$ va devoir réitérer depuis le début, mais pourquoi ne pas exploiter les solutions non admissibles générées par l'algorithme génétique ?

Nicolas Barnier et Pascal Brisset [2] montrent qu'on pourrait combiner ces deux types d'algorithmes de cette façon là : l'algorithme génétique génère des individus qui représentent des sous-domaines des variables du CSP. Ce dernier va alors essayer de chercher des solutions admissibles sur chacun de ces sous-domaines, bien qu'il se puisse qu'il n'en trouve pas. Un algorithme de CSP s'appliquant sur deux individus distincts peuvent aussi donner la même solution admissible.

Il s'agit désormais d'essayer d'appliquer cela dans le cadre de ce projet.

Dans le cadre de ce projet, un individu sur lequel le CSP sera appliqué, est un code non compatible. Il représente bien un sous-domaine de variables du CSP. En effet, chaque caractère instancié supprime ce même caractère du domaine. Il s'agit donc d'exploiter cette facette.

On choisit arbitrairement de prendre les 10% meilleurs codes non compatibles (en fonction de la fitness) de la génération courante, et une taille de $n/2$ pour l'espace de recherche sur un individu. Cela signifie qu'on retire les $n/2$ derniers caractères instanciés du code non compatible, et qu'on reprend ensuite à ce stade le déroulement du CSP avec le domaine mis à jour. Il se peut bien sûr que ce code modifié, ait des occurences doubles dans sa première moitié, mais l'algorithme (avec une petite modification) utilisant le CSP va le détecter, et s'arrêtera. On applique ce procédé à la fin de chaque génération.

On pourrait aussi, si les moyens le permettaient, comme le disent Nicolas Barnier et Pascal Brisset [2], de paralléliser ces tâches. L'algorithme génétique serait appliqué par une machine "maître", et sous-traite l'évaluation des individus à des machines "esclaves" qui implémentent uniquement le CSP.

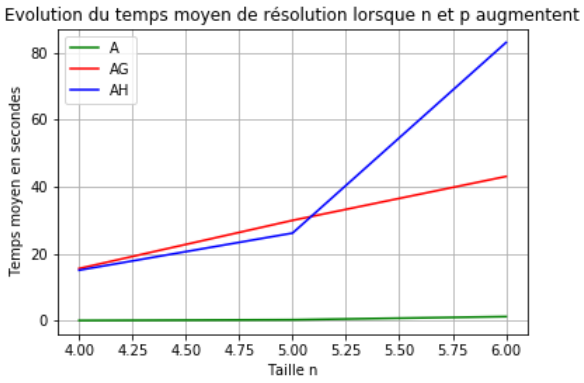
Algorithm 9: Algorithme hybride

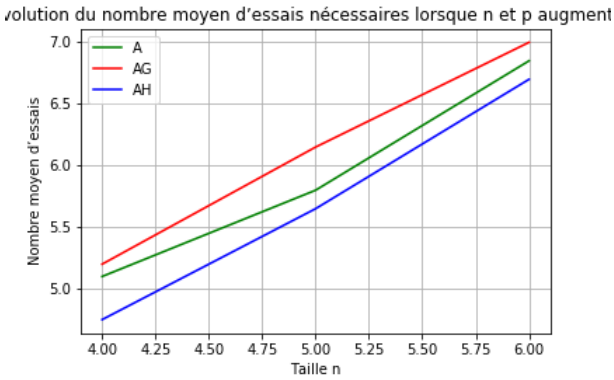
Input: $D, n, essaisPrecedents, maxsize, maxgen, popsize, cxpb, mutpb, timeout$
Output: $codeEssaiCompatible$
if $nbreEssais = 0$ **then**
    $code = code.random()$
else
    $E = \{\}$
    $g = 0$
    $historique[nbreEssais] = fitness$
    $time.initialize()$
    while $E = \emptyset$ *ou si le timeout n'est pas encore atteint* **do**
        while $g \leq maxgen$ *et* $|E| \leq maxsize$ **do**
            Généreration d'une nouvelle population en utilisant le crossover, la mutation, l'inversion, et la permutation
            Calcul de la fitness
            Ajouter les codes compatibles dans E s'ils ne sont pas dans E
            Appliquer le CSP sur les 10% meilleurs codes non compatibles et ajouter dans E les codes compatibles obtenus
            $g = g + 1$
    if $E \neq \emptyset$ **then**
        return $code \in E$
    else
        La méthode a échoué

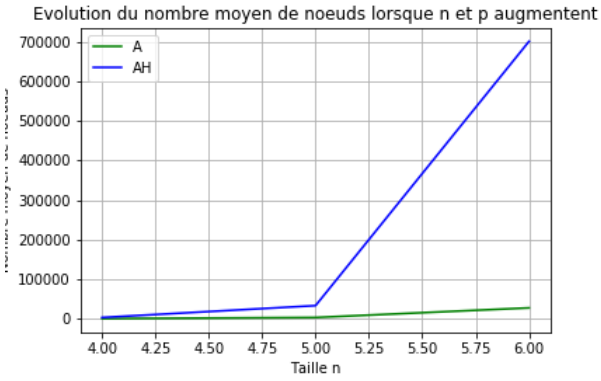
5.2 Comparaisons et analyses

Comparons alors cet algorithme hybride $A9$ avec le meilleur algorithme de CSP $A5$ et l'algorithme génétique précédent $AG1$.

Sur 20 instances, pour $n = 4, 5$ et 6, les courbes obtenues sont :







L'algorithme hybride est meilleur que les algorithmes précédents. En effet il garantit presque tout le temps une solution admissible par rapport à l'algorithme génétique, alors que ce dernier peut ne pas trouver une réponse à temps. En plus de presque garantir tout le temps une solution admissible, le nombre de codes compatibles augmente dans E, ce qui permet plus de codes à choisir dans E et par conséquent, la méthode de sélection aura alors un panel de choix plus large pour choisir le meilleur code compatible dans E. Ce qui provoque alors un nombre moyen d'essais encore plus bas que celui des deux autres algorithmes grâce à la meilleure qualité pour le code.

En revanche, le CSP inclus dans l'algorithme hybride parcourt évidemment plus de noeuds en contrepartie (puisqu'il s'exécute à chaque fin de génération), et par conséquent le temps de résolution augmente drastiquement lorsque n est supérieur à 5.

Chapitre 6

Conclusion

En conclusion, l'algorithme CSP garantit une solution compatible avec un temps acceptable de résolution à la fin de son exécution mais elle peut être loin d'être la solution optimale contrairement à l'algorithme génétique. Ce dernier serait légèrement plus apte à donner des meilleures solutions dans le cas où il réussirait à donner un code, en revanche il se peut bien qu'il n'en trouve pas. Pour palier à ce problème, il est alors possible de combiner ces deux méthodes afin de garantir une solution tout en proposant la meilleure possible. Le seul problème resterait celui du temps d'exécution qui est bien plus supérieur aux deux autres algorithmes, prix à payer pour obtenir la meilleure solution possible.

On pourrait avoir d'autres idées pour améliorer l'algorithme hybride en temps de calcul (tout en veillant au compromis temps/qualité), comme moduler les paramètres que j'ai choisi arbitrairement : la taille de l'espace de recherche et le nombre d'individus à prendre sur lesquels le CSP va s'appliquer. On pourrait aussi modifier le moment où le CSP va s'appliquer, par exemple seulement dans le cas où l'algorithme ne trouve aucun code compatible (dans le cas présent le CSP s'applique à chaque fin de génération tout le temps), toutes les k générations, seulement pour la dernière génération etc.

Bibliographie

[1] Lotte Berghman, Dries Goossens, and Roel Leus. 2009. *Efficient solutions for Mastermind using genetic algorithms.* Computers Operations Research 36, 6 (June 2009), 1880–1885. DOI :https ://doi.org/10.1016/j.cor.2008.06.004

[2] Nicolas Barnier and Pascal Brisset. *Optimisation par algorithme génétique sous contraintes.* (17 Apr 2014) https ://hal-enac.archives-ouvertes.fr/hal-00934534/file/284.pdf