

## Programation et Génie logiciel: Projet “MasterMind”

---

Environnement: openjdk version "1.8.0\_72-internal".  
IDE: Eclipse Platform version 3.8.1  
Les fichiers annexes sont disponibles sous les chemins relatifs (unix) au dossier  
"Vidal-Adrien\_MasterMind\_vs\_GA".  
  
Application: ./MasterMind-vs-GA.jar  
Sources: ./Mastermind/src  
Ressources: ./Images

---

## 1 Conception objet:

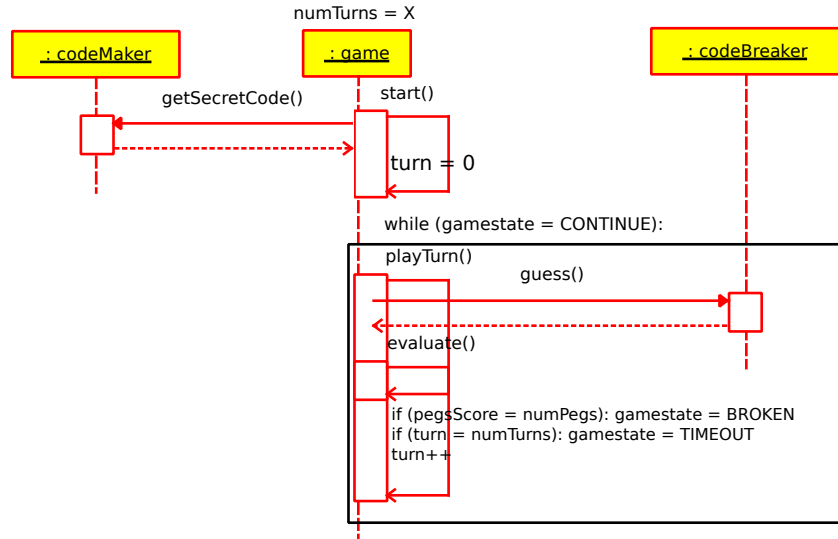
Les règles du jeu du MasterMind varient selon les versions mais les règles de base restent généralement les mêmes. Pour cette application, il sera possible de jouer avec entre 4 et 6 couleurs, avec des codes de 4 à 8 billes et sur un plateau permettant entre 8 et 16 essais. La section 3 détaille comment l'interface graphique adapte le plateau de jeu aux différentes règles possibles.

### 1.1 Interactions entre les joueurs:

MasterMind est un jeu pour deux joueurs. Le premier, que nous appelleront *code maker* compose un code secret de  $P$  couleurs  $s = (s_1, s_2, \dots, s_P) \in C \equiv \{R_{[1]}, B_{[2]}, Y_{[3]}, O_{[4]}, G_{[5]}, P_{[6]}\}[1 \dots N]^P$  choisies parmi les  $N$  premières couleurs avec répétitions permises que le second joueur, que nous appelleront *code breaker*, tentera de deviner par une série d'essais  $g_t = (g_{t1}, g_{t2}, \dots, g_{tP}), t \leq T$  consécutifs pendant  $T$  tours qui sont évalués par le *code maker*. Si le code est deviné dans la limite d'essais permis, le *code breaker* remporte la partie, sinon la victoire revient au *code maker*.

Le nombre de combinaisons possibles est de  $P^N$ .

Dans cette application, l'évaluation des codes sera automatisée plutôt que manuelle comme dans le jeu classique. Une classe `Game` (voir `./Mastermind/src/Game.java`) gèrera le déroulement des tours, les échanges d'information entre les joueurs et les conditions de victoire/défaite du jeu. Les trois états possibles du jeu (code deviné, à cours d'essais ou continuer) correspondent à une énumération (voir `./Mastermind/src/GameState.java`) (La **Figure 1** représente le diagramme de séquence simple d'une partie.



**Figure 1.** Diagramme de séquence simple d'une partie de MasterMind.  
 numTurns: Nombre d'essais permis.  
 numPegs: Nombre de billes composant les codes.  
 pegsScore: Score du nombre de billes exactes (voir 1.2).

### 1.2 Evaluation des codes:

Un code donné par le *code breaker* est évalué par rapport au code secret à l'aide de deux scores qui servent d'indices pour le *code breaker*:

- Le score de position  $X_t$  (posScore): Un indice orange est attribué pour chaque bille de couleur correctement positionnée ( $s_p = g_{tp}$ ). Lorsque le score de position  $X_t = P$ , le code  $g_t$  correspond exactement au code secret  $s$ .
- Le score couleur  $Y_t$  (colorScore): Un indice blanc est attribué pour chaque couleur donnée présente dans le code secret. Un seul indice orange est donné par couleur. Les autres billes de la même couleur n'attribuent pas d'indice supplémentaire. Une bille qui a déjà attribué un indice orange ainsi que toutes les autres billes de la même couleur n'attribuent pas d'indice blanc.

Les codes sont représentés par une classe `Code` (voir `./Mastermind/src/Code.java`) qui comprend une liste de couleurs. Les couleurs sont quant à elles représentées par une énumération paramétrée (par ordre dans lequel il devient possible de les jouer ; voir `./Mastermind/src/PegColor.java`). Une classe `GuessCode` (voir `./Mastermind/src/GuessCode.java`) héritant de `Code` représente les codes d'essai et contient en plus de cela les scores d'évaluation. L'évaluation d'un `GuessCode` se est implémentée dans la classe `Code` (voir `evaluate()`). Ainsi, tout `Code` est capable de fournir une évaluation de tout `GuessCode` par rapport à sa propre suite de couleurs en y attachant les scores calculés. Cette évaluation peut se traduire par l'algorithme suivant:

---

**Algorithme:**

---

```

EVALUATE(c, g):
  ▷Input: Code c
  ▷Input: GuessCode g without scores
  ►Output: GuessCode g with scores
  list checked = [ ]
  for each color i ∈ colors(c)
    do if i = colors(g)[i]
      then X(g)++
  for each color i ∈ colors(c) and each color j ∈ colors(g)
    do if i = j and j ∉ checked
      then append(checked, j)
      Y(g)++
  Y(g) -= X(g)
  if Y(g) < 0 then Y(g) ← 0
  return g

```

---

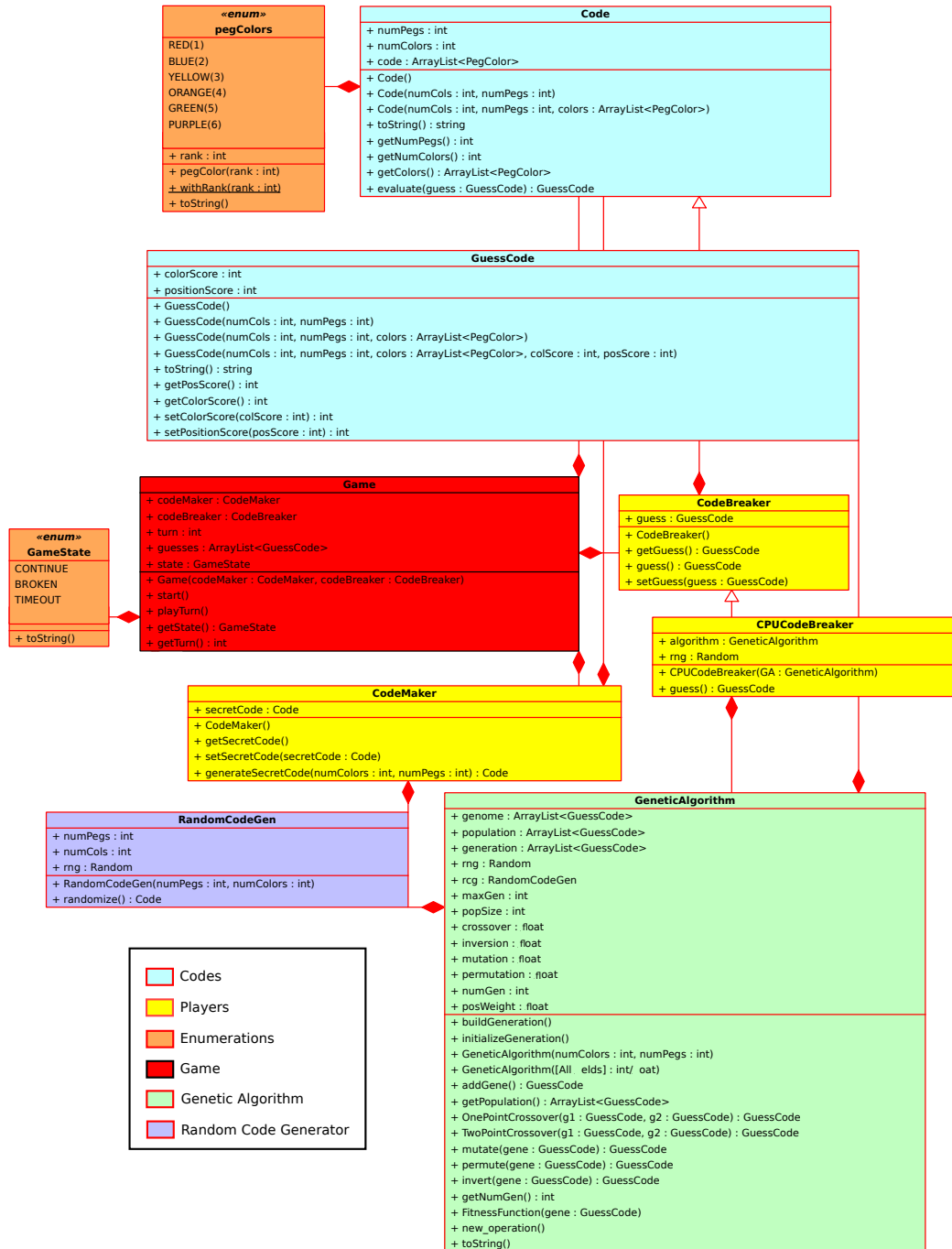
### 1.3 Les joueurs “CPU”:

Les classes présentées ci-dessus permettent de créer une partie entre deux utilisateurs. Des joueurs “CPU”, gérés par l’application doivent être implémentés afin de permettre des parties entre l’utilisateur et le programme ou pour simuler des parties entièrement jouées par le programme. La version CPU du *code maker* ne nécessite que la création de codes secrets aléatoires à deviner. Cette simple méthode peut être implémentée dans la classe `CodeMaker` mais le générateur de codes lui-même est implémenté dans sa propre classe (voir `./Mastermind/src/RandomCodeGen.class`) afin de pouvoir être utilisé pour la génération de codes dans les méthodes de l’algorithme génétique (voir 2 ).

En revanche, une nouvelle classe est créée pour le *code breaker* version CPU (voir `./Mastermind/src/CPUCodeBreaker.class` dans laquelle la méthode `guess` qui ne faisait que retourner le `GuessCode` actuel est recouverte par une nouvelle implémentation qui fait appel aux méthodes d’une instance de l’algorithme génétique.

Cette méthode fait appel à chaque fois qu’elle est appelée à la méthode `populate()` de l’algorithme et récupère une population de codes viables parmi lesquels l’un est choisi au hasard pour être joué. Elle relaie ensuite ce même code scoré depuis l’instance de `Game` d’où la classe est instanciée vers son algorithme génétique.

Les méthodes de l’algorithme génétiques sont détaillées dans la partie 2. La **Figure 2** à la page suivante représente la diagramme des classes du jeu avant implémentation de l’interface graphique. Ces classes telles quelles sont utilisables pour tester le jeu et l’algorithme dans une méthode `main()`.



**Figure 2.** Diagramme de classe du projet sans interface graphique. Une fonction `main()` doit être implémentée afin de pouvoir utiliser ces classes pour former une partie.

## 2 L'algorithme génétique:

Voir `./mastermind/src/GeneticAlgorithm.java`.

L'algorithme génétique utilisé et sa fonction objective sont inspirés d'une publication de Berghman *et al.*<sup>[1]</sup> avec tout de même d'importantes différences. Il s'agit d'une méthode linéaire basée sur la création de populations à partir de codes recombinés aléatoirement et mutés sur plusieurs générations puis sélectionnés par une fonction objective à partir des scores des codes précédemment joués. A chaque tour, des générations sont constituées à partir de la précédente et ses codes éligibles sont ajoutés à la population  $\hat{E}_t$  de codes parmi laquelle le CPU en jouera un aléatoirement.

Un "génome"  $\Omega$  est constitué à partir des essais précédemment joués. Ces codes ne prennent pas part dans les recombinaisons mais sont utilisés pour l'évaluation des codes générés par la fonction objective `fitnessFunction()`.

La génération 1 de chaque tour est réinitialisée à partir de la population du tour précédent par la méthode `buildGeneration()` afin de ne recombiner que des codes qui furent éligibles. Les générations comptent pour ce faire autant d'individus que la population. Au tour 1, la population est simplement constituée de codes générés aléatoirement par la méthode `initializeGeneration()`. Les générations comptent toujours un nombre maximal de codes. Les codes sont recombinés par tirage aléatoire sans remise de paires de deux. En cas de total impair, un code est généré aléatoirement afin de compléter la génération.

L'algorithme suivant résume le processus sur l'étendue d'une partie. Celui-ci est en réalité partagé entre plusieurs classes. La complexité maximale de cet algorithme est de  $O(popsiz e \cdot maxgen \cdot T)$  lorsque les paramètres ne permettent pas la création de populations vides (voir 2.1).

---

### Algorithme:

---

```

GENETIC-ALGORITHM( $s, co, m, p, v, a, b, e, popsize, maxgen$ ):
  ▷Input: Code secret:  $s$ 
  ▷Input: crossing-over preference:  $co$ 
  ▷Input: mutation rate:  $m$ 
  ▷Input: permutation rate:  $p$ 
  ▷Input: inversion rate:  $v$ 
  ▷Input: position score weight:  $a$ 
  ▷Input: color score weight:  $b$ 
  ▷Input: eligibility:  $e$ 
  ▷Input: maximum population size:  $popsiz e < P^N$ 
  ▷Input: maximum number of generations:  $maxgen$ 
  genome  $\Omega \leftarrow [ ]$ 
  turn  $t \leftarrow 1$ , numgen  $h \leftarrow 1$ 
   $G_1 \leftarrow \text{INITIALIZE-GENERATION}()$  {
    while  $|G_1| < popsize$ 
      do append(RANDOMIZE-CODE()) {
        return  $\xi(c_1, c_1, \dots, c_P) \in C \equiv \{R_{[1]}, B_{[2]}, Y_{[3]}, O_{[4]}, G_{[5]}, P_{[6]}\}[1 \dots N]^P$ 
      }  $G_1$ 
  }
  while  $X_t < P$  and  $t < T$ 
    do while  $|\hat{E}_t| < 1$ 
      do while  $|\hat{E}_t| < popsize$  and  $h < maxgen$ 
        do  $G_{h-1} = \hat{E}_{t-1}$ 
         $G_h = \text{BUILD-GENERATION}()$  {
          for each gene  $\gamma \in G_{h-1}$ 
            do if  $\xi\mathbb{R}[0; 1] \leq co$  then  $\gamma = \text{ONE-POINT-CROSSOVER}(\gamma)$ 
              else  $\gamma = \text{TWO-POINT-CROSSOVER}(\gamma)$ 
            if  $\xi\mathbb{R}[0; 1] \leq m$  then  $\gamma = \text{MUTATE}(\gamma)$ 
            if  $\xi\mathbb{R}[0; 1] \leq p$  then  $\gamma = \text{PERMUTE}(\gamma)$ 
            if  $\xi\mathbb{R}[0; 1] \leq v$  then  $\gamma = \text{INVERT}(\gamma)$ 
            if  $\gamma \notin G_h$  then append( $\gamma, G_h$ )
            else append(RANDOMIZE-CODE(),  $G_h$ )
          }
        for each gene  $\gamma \in G_h$ 
          do if  $\text{FITNESS}(\Omega, \gamma, a, b) \leq e$ 
            then append( $\hat{E}_t, \gamma$ )
         $h++$ 
      PLAY( $g_t \leftarrow \xi c \in \hat{E}_t$ ) {  $g_t \leftarrow \text{EVALUATE}(s, g_t)$ 
         $t++$  }
    append( $g_t, \Omega$ )

```

---

## 2.1 La fonction objective:

Le calcul de la fonction objective (*fitness*) se fait en évaluant un code générés à l'ensemble des codes précédemment joués. Plus la différence entre ces scores ainsi obtenus  $X'_t(c)$ ,  $Y'_t(c)$  et le score des codes dans le jeu  $X_t$ ,  $Y_t$ , est basse, plus le code généré est proche du code secret  $s$ . La fonction devient plus restrictive a chaque tour en comparant tout les codes précédemment joués et en faisant la somme des différences lorsque son résultant est comparé au seuil *eligibility* constant.

$$fitness = a \cdot \left( \sum_{t=1}^{|\Omega| < T} |X'_t(c) - X_t| \right) + b \cdot \left( \sum_{t=1}^{|\Omega| < T} |Y'_t(c) - Y_t| \right)$$

Par défaut la valeur d'*eligibility* est de 0 et donc seuls les codes strictement éligibles sont ajoutés à la population. Néanmoins, une valeur restrictive d'*eligibility* peut être néfaste si combinée avec des valeurs de *popsiz* et *maxgen* basses. Par défaut, la valeur de *maxgen* est paramétrée par l'interface à  $25 \times P \times N$  et celle de *popsiz* à

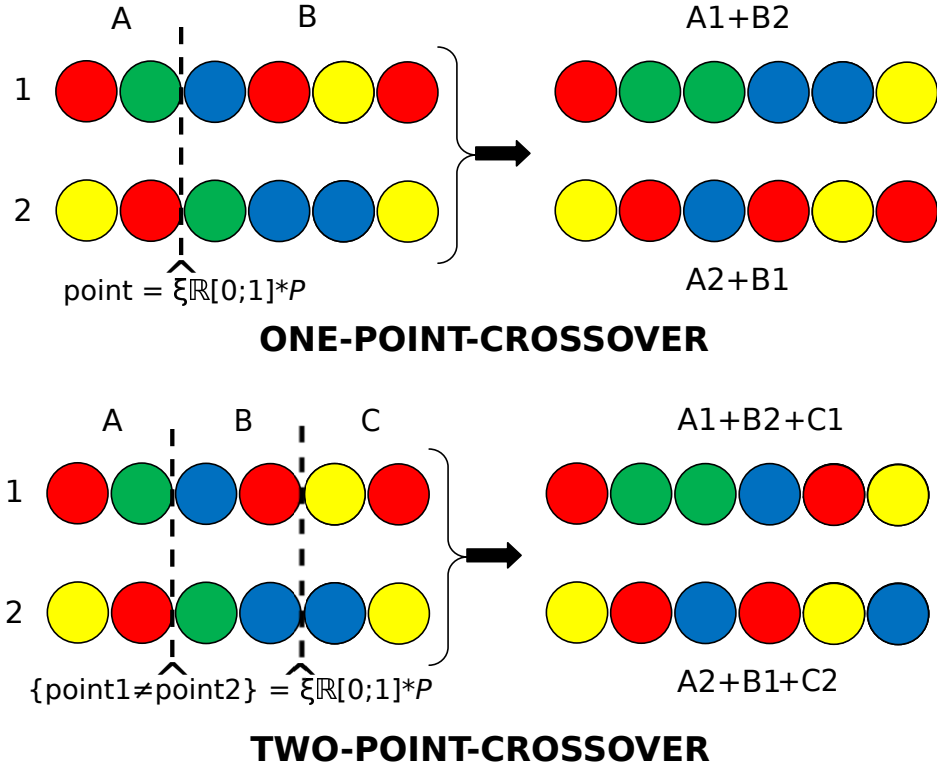
$\max(300, P^N)/2$  afin que l'algorithme ait toujours des paramètres acceptables s'il n'est pas reconfiguré par l'utilisateur. La valeur de *popsiz* ne peut jamais dépasser  $P^N$  car elle ne peut contenir que des codes uniques et ne peut donc être plus grande que le nombre total de codes possibles.

Afin de contrecarrer les paramétrages trop restrictifs de l'algorithme qui résulteraient par une population vide lorsque *maxgen* est atteint, le joueur CPU fera plusieurs fois appel à `populate()` jusqu'à avoir une population d'au moins 1 individu mais de telles configurations sont de préférence à éviter car plus coûteuses en temps que de créer un grand nombre de générations.

Les sommes des différences de scores de position et de couleur ne sont par défaut pas pondérés mais une pondération est paramétrable par les paramètres *posWeight(a)* et *colWeight(b)*.

## 2.2 Méthodes de recombinaison et mutation des codes:

Deux types de méthodes sont utilisées pour la recombinaison de codes avec par défaut 50% de chances de réaliser l'une ou l'autre. Une préférence peut être donnée à l'algorithme par le paramètre `crossoverBalance`. Ces deux méthodes s'apparentent aux processus biologiques du *crossover* à 1 ou deux points. La **Figure 3** représente le fonctionnement de ces deux méthodes.



**Figure 3.** Représentation des méthodes de crossover. La méthode `onePointCrossover()` coupe les séquences en un même point aléatoire et recombine les deux moitiés faites. La méthode `twoPointCrossover()` coupe les séquences en deux mêmes points aléatoires et échange la partie du milieu.

Après recombinaison, des méthodes de mutation de la séquence sont aléatoirement appliquées de manière indépendante.

- **mutate()** change la couleur d'une bille à une position  $\xi \mathbb{R}[0; 1] * P$  aléatoire en une autre couleur  
 $c \leftarrow \xi(c_1, c_1, \dots, c_P) \in C \equiv \{R_{[1]}, B_{[2]}, Y_{[3]}, O_{[4]}, G_{[5]}, P_{[6]}\} [1 \dots N]^P$ .
- **permute()** inverse les couleurs de deux billes de positions  $\xi \mathbb{R}[0; 1] * P$  aléatoires différentes et en échange les couleurs.
- **invert()** coupe la séquence en deux points aléatoires  $\xi \mathbb{R}[0; 1] * P$  différents, inverse la séquence du milieu et la replace dans la séquence entre ces mêmes points.

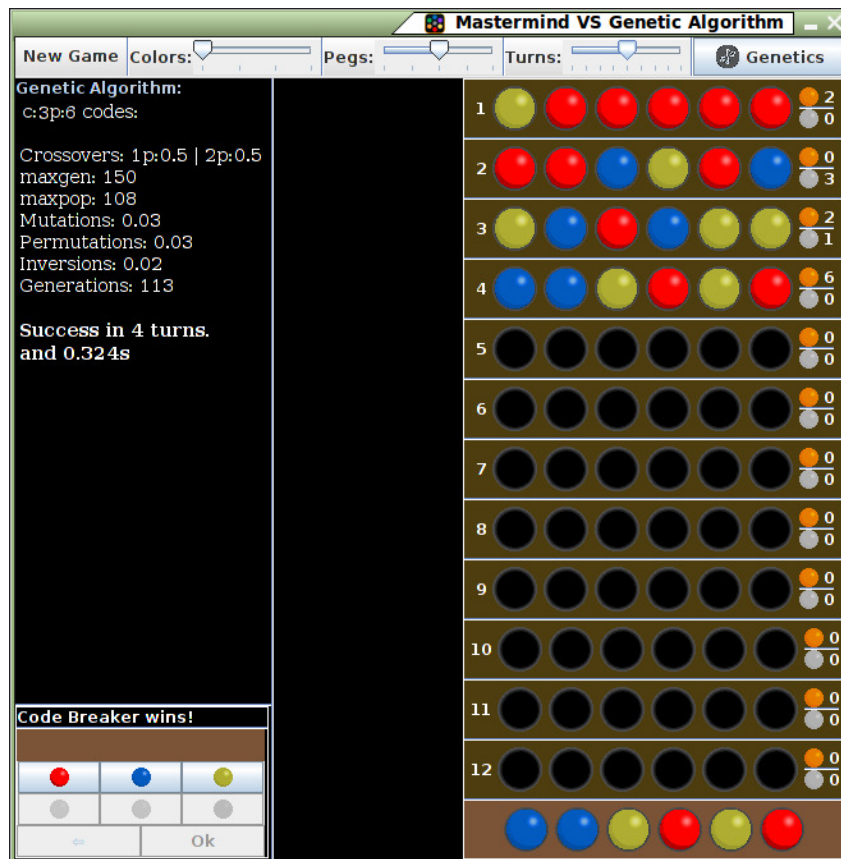
### 3 Interface Graphique:

Voir `./mastermind/src/MasterMindGui.java`.

#### 3.1 Fenêtre principale:

L'interface graphique ( **Figure 4** ) se veut de ressembler au plateau du jeu avec une grille dont les lignes numérotées représentent les essais et les colonnes les positions des billes. Les billes ou emplacements vides sont représentés par des icones ressemblant à des sphères de couleur. Le plateau s'adapte au changement des règles du jeu par l'intermédiaire de composants `JSlider` et 'Pegs' et 'Turns' disposés dans la barre de menu de l'interface. À droite de chaque ligne sont indiqués les scores d'indices obtenus après avoir joué un essai. Un panel supplémentaire contient le code secret qui est décoloré pendant que le *code breaker* joue et se révèle à la fin de la partie.

Un menu déroulant "New Game" permet de lancer des parties entre deux utilisateurs, de jouer contre un code aléatoirement généré, de donner un code à résoudre composé manuellement ou aléatoirement au *code breaker* CPU ou enfin d'interrompre à tout moment une partie en cours.



**Figure 4.** Capture d'écran de l'interface graphique (fenêtre principale).  
Bureau cinnamon sous linux mint.

La composition de codes se fait par l'intermédiaire d'un clavier visuel ( **Figure 5** ) au dessus duquel s'affiche la séquence en cours de composition. Il est possible de revenir en arrière grâce à un bouton '←' et le bouton 'Ok' valide le code et fait avancer les étapes jeu nécessitant une interaction de l'utilisateur. Ce dernier ne s'active que lorsque le code composé atteint la longueur  $P$  et se désactive si une couleur est retirée à la fin de la séquence. Les boutons des couleurs s'activent ou se désactivent lorsque la position de l'indicateur de 'Colors' change et que ces couleurs deviennent disponibles/indisponibles pour la composition de codes.

Un panneau affiche les paramètres et performances de l'algorithme génétique lorsque celui-ci est en action.





Figure 5. Clavier de composition de codes.

Les changements sur l'interface pour adapter le plateau, pour afficher les essais ou pour écrire les données sur l'algorithme sont gérés par les méthodes `resetBoard()`, `drawGuessTab(turn)`, `drawClues(turn)` et `printGASStats(algorithm)`.

### 3.2 Fenêtre de configuration de l'algorithme:

La barre de menu contient également un bouton 'Genetics' qui déclenche l'apparition d'une fenêtre de dialogue contenant tout les composants permettant le paramétrage de l'algorithme génétique (Figure 6). Cette fenêtre est définie dans la classe `GeneticsMenu` contenue dans `MasterMindGui`.

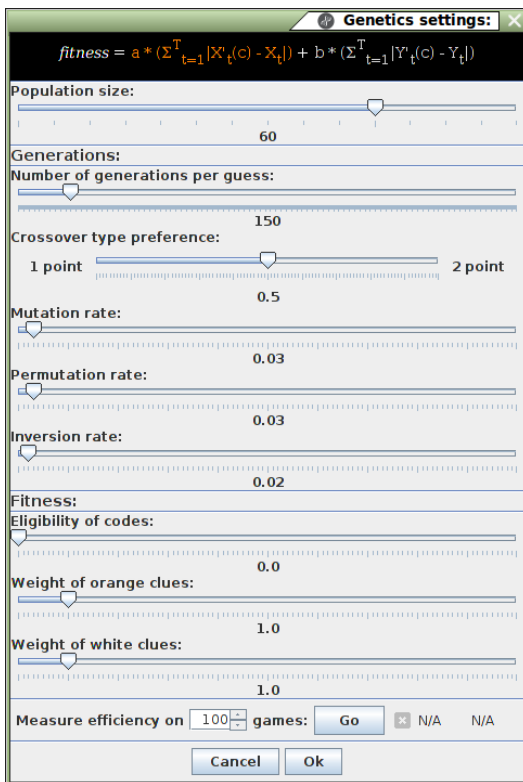



Figure 6. Fenêtre de configuration de l'algorithme génétique.

Afin d'échapper à une telle situation ou écourter une mesure qui peut être longue pour les codes complexes, un bouton  apparaît lorsqu'une mesure est lancée. Le résultat de la mesure, même si interrompue reste affiché jusqu'à ce qu'une nouvelle soit lancée ou que l'un des paramètres de l'algorithme soit changé.

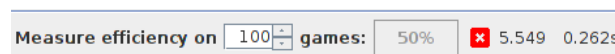


Figure 7. Outil de mesure d'efficacité de l'algorithme génétique de l'interface graphique en cours d'exécution.

Les mesures ne sont pas interrompues par la fermeture de la fenêtre de configuration de l'algorithme ou les changements apportés aux règles du jeu et aux paramètres de l'algorithme.

Une ligne de texte donne des indications simples sur les étapes de la partie ou sur les processus en cours.

Un plateau vide est retracé dès lors qu'une nouvelle partie est lancée ou que les règles du jeu sont changées mais reste tel quel après la fin d'une partie tant que l'une de ces actions n'est pas comise.

Les essais sont tracés sur la ligne correspondante à chaque tour et les scores sont indiqués à leur droite à côté d'icônes représentant les jetons indice.

La fenêtre est composée de composants `JSlider` pour chaque paramètre de l'algorithme génétique.

Tout en haut, un cadre affiche du texte informatif lorsque l'utilisateur passe le curseur sur les éléments de la fenêtre.

En bas, le bouton 'Ok' cache la fenêtre qui conserve les positions des indicateurs tandis que le bouton 'Cancel' fait de même en replaçant les indicateurs aux positions qu'ils occupaient avant l'ouverture de la fenêtre.

Les positions et/ou valeurs maximales des deux premiers *sliders* sont modifiées lorsque les règles du jeu sont changées dans la fenêtre principale pour donner un paramétrage par défaut acceptable pour les valeurs de *popsize* et *margin*.

Le bouton 'Go' permet de faire des mesures de la vitesse de résolution moyenne de codes de l'algorithme en nombre de coups et en temps sur un nombre de parties configurable.

Dans ce mode, de mesure, la condition de défaite du *code breaker* n'est pas considérée et le nombre d'essais permis est donc illimité. Certaines configurations de l'algorithme sont donc à éviter afin que les mesures ne restent pas éternellement bloquées sur une partie devenue irrésoluble. Voir Figure 7.

### 3.3 Contrôle du jeu:

La classe `MasterMindGui` possède des champs pouvant accueillir des instances des classes `Game`, `CodeMaker`, `CodeBreaker` et `GeneticAlgorithm`. Lors du lancement d'une nouvelle partie via le menu 'New Game', des instances sont créées pour le jeu, les joueurs et si besoin l'algorithme génétique à partir des paramètres donnés par la position des `JSlders`.

Le déroulement du jeu est contrôlé par l'`actionListener` du bouton 'Ok' lors des interactions humaines et par les classes `CPUSolve` ou `CPUSolveMany` lors de la résolution par le *code maker* CPU.

Le bouton 'Ok' récupère la séquence de couleur présente dans le champ `inputColors`. Ce champ contient les couleurs entrées lors de l'activation des boutons des billes par l'utilisateur. Cette séquence est affiliée en tant que code aux instances du jeu ou des joueurs et les méthodes nécessaires sont appelées afin de simuler le tour du jeu. Les couleurs entrées ainsi que les scores évalués sont affichés sur le plateau de jeu. Le bouton 'Ok' agit également sur l'interface lorsque le jeu renvoie une condition de fin du jeu.

La classe `CPUSolve` fait de même que le bouton 'Ok' mais simule tous les essais faits par le *code breaker* CPU. Elle est instanciée et utilisée par le bouton 'Ok' lorsque le mode "Challenge the CPU" est choisi, après la composition du code secret ou directement lorsque l'option "CPU guesses random code" est choisie, après génération d'un code secret aléatoire.

La classe `CPUSolveMany` simule un nombre de jeux donnés de la même manière `CPUSolve` mais n'affiche rien au niveau de l'interface durant un jeu. Entre chaque simulation de jeu, le nombre de tours moyen et le temps qui furent nécessaires au *code breaker* CPU pour résoudre le code. La simulation des jeux faites par cette classe ne tient pas en compte de la condition de défaite du *code breaker* afin de calculer correctement le nombre de tours.

Les classes `CPUSolve` ou `CPUSolveMany` contenues dans `MasterMindGui` implémentent `SwingWorker<T,V>` afin de réaliser, dans un *thread* parallèle, les itérations de l'algorithme ou des parties en affichant synchroniquement les résultats sans rendre l'interface indisponible le temps des calculs.

Ainsi il est toujours possible de jouer tout type de jeu dans la fenêtre principale lorsqu'une mesure est en cours ou de lancer une mesure durant une partie. La **Figure 8** à la page suivante représente le diagramme de classe complet après implémentation de l'interface graphique.

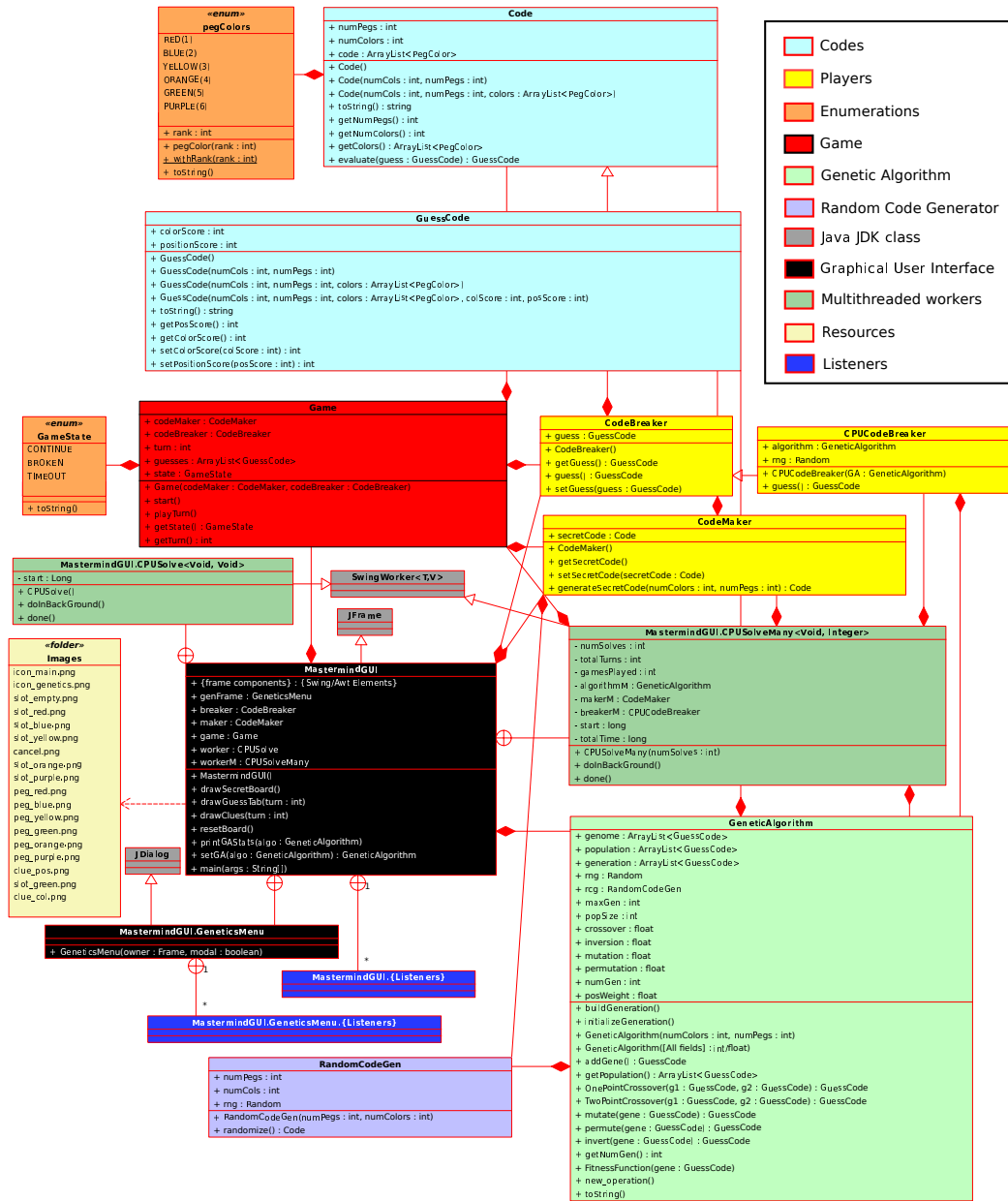


Figure 8. Diagramme de classe complet de l'application.

## 4 Étude des performances de l'algorithme:

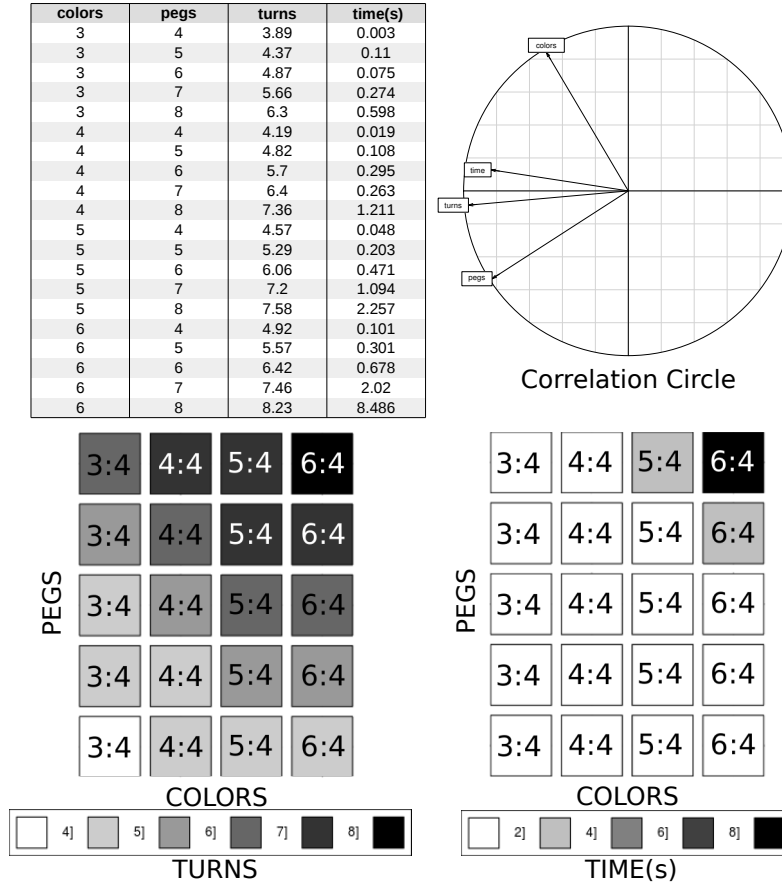
Plusieurs mesures de l'efficacité de l'algorithme peuvent être faites dans différentes conditions à partir de la fenêtre de configuration de l'algorithme. Toutes ces mesures sont collectées manuellement en utilisant l'application. Toutes les mesures sont faites pour 100 jeux.

Les mesures vues dans la partie 4.1 sont disponibles dans le fichier `./measures.txt`.

Les mesures vues dans la partie 4.2 sont disponibles dans le dossier `./measures_5:5`.

### 4.1 Paramètres par défaut:

Une première série de mesure est faite sur l'algorithme génétique avec les paramètres par défaut implémentés dans l'interface. Voir **Figure 9** ci-dessous.



**Figure 9.** Mesures du nombre de tours et temps moyen de résolution des différents types de codes ( $P$  = “pegs”,  $N$  = “colors”) par l'algorithme génétique avec les paramètres par défaut:  $popsize = \min(P^N, 300)/2$ ,  $25 \times P \times N$ ,  $co = 0.5$ ,  $m = 0.03$ ,  $p = 0.03$ ,  $v = 0.02$ ,  $e = 0$ ,  $a = 1$ ,  $b = 1$ . Figures réalisées avec la librairie ade4 de R. Le cercle des corrélations provient d'une analyse des composantes principales (ACP). Script disponible dans le fichier `./measures.R`.

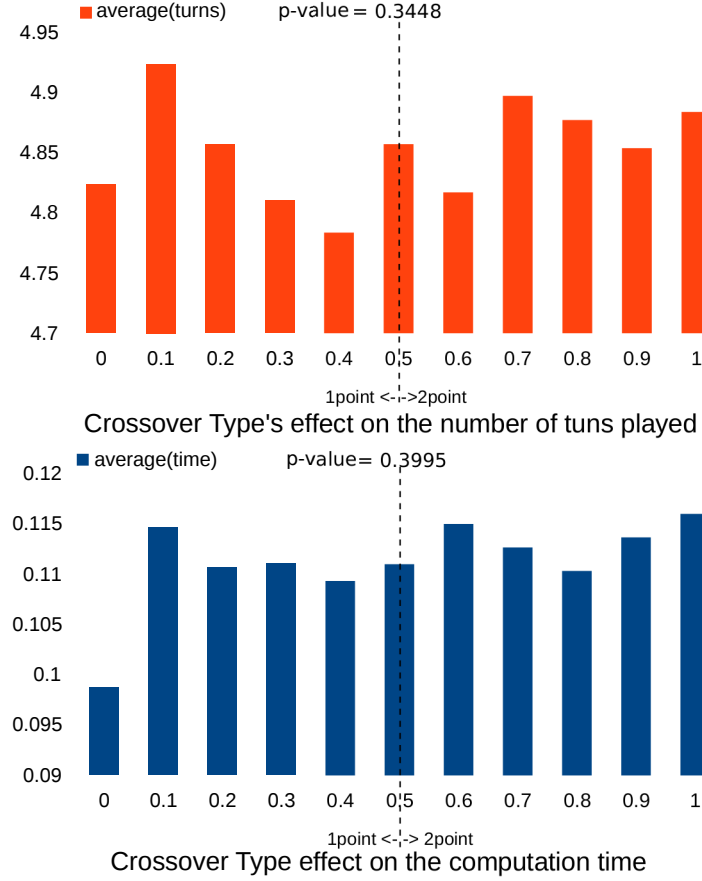
On peut observer que la longueur du code à trouver a une plus forte influence sur l'efficacité de l'algorithme génétique que le nombre de couleurs.

Le temps de calcul est moins influencé par la complexité du code. Il n'atteint de fortes valeurs que lorsque le `code maker` CPU doit faire plusieurs itérations de la méthode `populate()` sur les derniers tours, là où la fonction objective devient très stricte et il devient difficile de trouver des codes éligibles parmi ceux générés.

## 4.2 Effet du type de crossing-over:

Par défaut, les deux types de crossing-over ont chacun des chances égales de se produire pour la recombinaison de codes. Mais il est possible d'induire une préférence par le biais de la variable *co*. Le crossover à 1 point a une probabilité  $P(1p) = co$  de se produire tandis que le crossover 2 points à une probabilité  $P(2p) = 1 - co$ .

La **Figure 10** représente les moyennes de mesures en triplicat ( `./5:5_measures/crossover.tab` ) du nombre de tours et du temps de calcul faites en triplicat sur les codes de type  $P=5, N=5$ .



**Figure 10.** Mesures moyennes ( `./5:5_measures/crossoverAv.tab` ) du nombre de tours joués et du temps de calcul pour des codes de type  $P=5, N=5$  avec des valeurs de préférence pour le type de crossover variables. Les autres paramètres correspondent aux défauts indiqués **Figure 9**. Les tests statistiques sont réalisés sous R. Script disponible dans le fichier `./measures.R`.

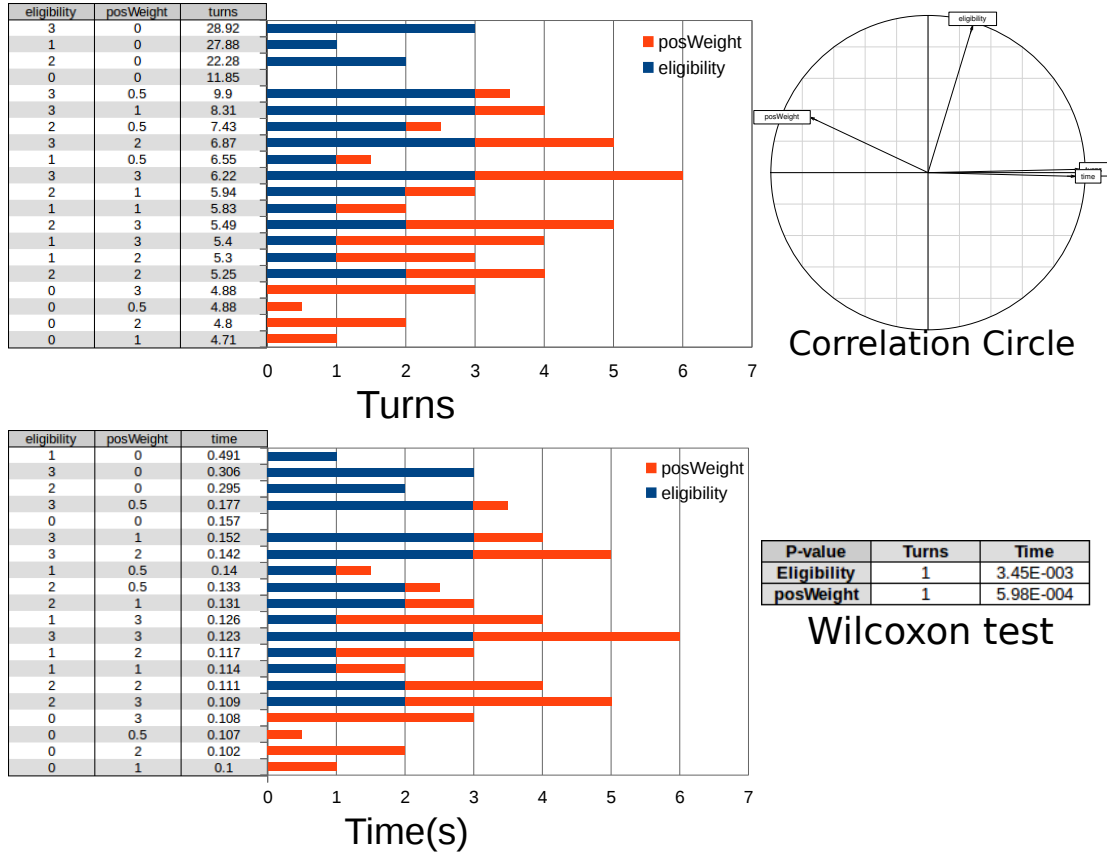
Le type de crossover favorisé ne semble pas avoir d'effet notable sur l'efficacité de l'algorithme. Les variations pour le nombre de tours comme pour le temps de calcul sont faibles et aucune tendance semble se dessiner. Un test statistique (*t-test*) confirme l'hypothèse  $H_0$  d'indépendance ( $p\text{-value} > 0.05$ ) entre ces mesures et la variable *co*.

Les paramètres des probabilités d'exécution des fonctions mutagènes ne semblant pas avoir d'effet mesurable avec la précision donnée, celles-ci ne seront pas traitées dans ce rapport.

### 4.3 Effet de l'éligibilité et du poids des scores:

L'éligibilité  $e$  est de 0 par défaut. Ce qui signifie que seuls les codes viables (*i.e* des codes qui peuvent correspondre au code secret étant donné les scores des précédents essais) peuvent être joués. Lorsque  $e > 0$ , des codes non-viables peuvent être joués rendant la sélection de codes jouables moins difficile. Lorsque cela est le cas, il peut être désirable de donner plus de poids au score de position (indices oranges) étant donné que ce score est le plus déterminant pour la viabilité d'un code.

La **Figure 11** représente les mesures de nombre de tours et de temps de calcul ordonnés par ordre décroissant. Il n'est pas nécessaire d'ajuster également le poids du score de couleur étant donné que l'on obtiendra les mêmes rapports entre les deux éléments de la fonction.-



**Figure 11.** Mesures du nombre de tours joués et du temps de calcul pour des codes de type  $P=5, N=5$  ordonnés par ordre descendant avec des valeurs d'éligibilité et de poids du score de position variables. Les autres paramètres correspondent aux défauts indiqués **Figure 9**. Les tests statistiques sont réalisés sous R. Script disponible dans le fichier `./measures.R`.

On peut observer que l'augmentation de l'éligibilité a une influence néfaste sur l'efficacité de l'algorithme qui fait alors une sélection trop généreuse des codes. Mais celui-ci est contrebalancé si l'on augmente le poids du score de position. Diminuer ce score en dessous de 1 en faveur du score de couleurs réduit légèrement l'efficacité de l'algorithme car les positions sont plus déterminantes dans la viabilité d'un code. Un trop fort poids du score de positions montre également un effet néfaste en écrasant le score des couleurs.

Les distributions du temps et du nombre de tours ne suivant pas une loi normale, le test statistique *t-test* est remplacé par un test de *Wilcoxon*. Le test statistique montre que contrairement au nombre de tours, le temps de calcul est indépendant des paramètres. Les variations du temps de calcul sont bien moindre que celles du nombre de tours et suivent simplement la distribution de ce dernier comme le montre le cercle des corrélations. Cela est probablement dû au fait que les codes  $P=5, N=5$  ne sont pas d'une grande complexité et que l'on ne se trouve jamais dans le cas où plusieurs itérations de `populate()` doivent être réalisées.

Les paramètres par défaut pour l'éligibilité comme pour le poids des scores de position semblent présenter la configuration optimale. Ce résultat est conforme à celui obtenu par Berghman *et al.*<sup>[1]</sup>.

## 5 Références:

[1]: **Efficient solutions for Mastermind using genetic algorithms** ; KBI 0806 ; Lotte Berghman, Dries Goosens et Roel Leus.