TME 4 : Algorithmes génétiques sur les expressions arithmétiques

Objectifs pédagogiques : récupération de projets sous eclipse, fusion de projets (sensibilisation aux problèmes d'architecture que cela pose), application multi-package, héritage, utilisation de collections simples tirées de l'API, equals() et copie profonde, manipulation d'ArrayList, utilisation des interfaces

4.1 Récupération de projets

Lors des TME précédents, vous êtes partis «de rien». Cette semaine, vous allez réutiliser et fusionner les projets des deux semaines précédentes.

L'objectif est d'utiliser l'algorithme génétique du TME 2 pour chercher une expression arithmétique f (code développé au TME 3) qui passe par un point $(x_1, \cdots, x_n, y)$, donc telle que $f(x_1, \cdots, x_n) = y$ Par exemple, on cherche une fonction f telle que $f(x_1 = 0.3, x_2 = 0.8, \cdots x_n = 0.12) = 0.37$.

Pour cela, vous allez tirer un point cible aléatoirement puis construire et évaluer une population d'expressions arithmétiques, en mettant en place les mécanismes d'évolution permettant de rechercher une expression qui passe par le point cible.

⇒ Dans votre workspace habituel, créez un nouveau projet Java tme4 et copiez-collez les packages `pobj.algogen` et `pobj.arith` obtenus à la fin des tme2 et tme3 dedans.

4.2 Fusion des projets

L'objectif du TME est de faire en sorte que les individus évalués dans la population créée lors du TME 2 soient les expressions créées lors du TME 3. Un Individu portera donc une Expression comme valeur propre (au lieu d'un double). L'objectif et de trouver une expression dont l'évaluation (au sens du TME 3) se rapproche d'une valeur cible.

Pour réaliser cette fusion, vous avez deux problèmes à résoudre, que vous allez traiter successivement:

- faire en sorte que les expressions soient vues comme des individus par la population,
- faire en sorte que l'environnement d'évaluation des individus soit adapté à votre objectif : trouver une fonction qui passe par un point.

Il faut donc donner un sens aux concepts génétiques (muter, croiser) quand les individus constituant une population sont des expressions.

4.2.1 Refactoring de `algogen`

On souhaite manipuler les expressions au sein de l'algorithmique génétique du TME 2. Comme le code développé au tme2 n'utilise pas d'interfaces à part l'Environnement, sa réutilisation est problématique. On va procéder à une montée en abstraction à fonctionnalité constante (ou *refactoring*) du code `algogen` du tme2.

⇒ Introduisez une interface `IIndividu` qu'implémente la classe `Individu`. Remplacez toutes les références à `Individu` par des références à cette interface dans votre code (en particulier dans `Population`). Il ne doit persister qu'une seule référence à la classe `Individu` dans le corps d'une méthode de la classe `PopulationFactory`.

NB: On pourra utiliser le menu "Refactor->Extract interface" et cocher les méthodes adaptées de `Individu`.

NB2: Il faudra satisfaire les besoins des autres classes (en particulier `Population`) en remontant sur `IIndividu` toutes les méthodes utiles de `Individu`.

NB3: Comme en TD, il faut que les `IIndividu` soient `Comparable<IIndividu>` pour permettre le tri

de la population avec `Collections.sort`.

⇒ Introduisez une classe abstraite `AbstractIndividu` qui implémente `IIndividu` et dont dérive `Individu`. Cette classe porte le `fitness` et toutes les méthodes liées à cet attribut (la comparaison en particulier).

NB: On pourra utiliser le menu Eclipse "Refactor->Extract superclass..." puis éditer à la main.

NB2: pensez à déclarer `abstract` la méthode `IIndividu clone()` dans `AbstractIndividu`.

⇒ Renommez la classe `Individu` en `IndividuDouble`. Utilisez pour cela le menu "Refactor->Rename" d'eclipse.

Cette classe ne porte qu'un double et une fonction d'affichage à ce stade ainsi que les opérations critiques muter, croiser et clone avec leur sémantique pour un Individu qui est un double. Par analogie on construira un `IndividuExpression` portant une expression dans les questions suivantes.

⇒ Comme le seul ancêtre commun de `Double` et `Expression` est `Object`, modifiez la signature de `getValeurPropre` pour rendre un `Object` dans `IIndividu`. Propagez cette modification, on utilisera des tests `instanceof` pour résoudre le type de la valeur propre là où c'est utile.

NB: On peut taper `inst` suivi de Ctrl-espace pour avoir la complétion template *instanceof and typecast* sous Eclipse qui permet de faire de cast "propres" (sans risque de ClassCastException).

⇒ Déportez vos classes `IndividuDouble` et `ValeurCible` dans un nouveau package `pobj.algogen.doubles` (on peut directement glisser déposer les classes sous eclipse).

⇒ Assurez-vous que le `main` du tme2 fonctionne toujours et testez votre programme.

⇒ Faites un diagramme de classes UML décrivant la nouvelle structure du package `pobj.algogen`.

L'étape suivante consiste à construire un `IndividuExpression` portant une expression comme valeur propre et l'environnement d'évaluation correspondant.

4.2.2 Un adapter pour `arith`

Modifier `arith` pour les besoins d'`algogen` est intrusif; cela force à remettre en cause la base de code établie dans le TME 3, alors qu'elle remplit déjà pleinement son rôle : gérer des expressions arithmétiques.

On souhaite donc au maximum NE PAS MODIFIER le package gérant les expressions arithmétiques développé au TME 3. On s'appuiera donc uniquement sur l'API publique du package arithmétique, c'est-à-dire l'interface `Expression`, la classe `ExpressionFactory` pour construire des expressions et la classe `EnvEval` pour gérer les valeurs des variables et évaluer les fonctions.

⇒ Créez un nouveau package `pobj.algogen.arith`. Dans ce package, construisez la classe `IndividuExpression` qui étend `AbstractIndividu` et porte une `Expression` comme valeur propre.

Les points critiques qu'il faut implémenter sont les deux méthodes d'évolution: `muter` et `croiser`. Pour la mutation, vous pourrez simplement générer une nouvelle expression aléatoirement. Pour le croisement, par analogie avec l'approche du TME 2, vous produirez comme descendant de deux expressions `e1` et `e2` une nouvelle expression `e3` qui est leur moyenne: `e3 = (e1 + e2) / 2`. Cette expression sera construite explicitement à l'aide de `ExpressionFactory`, en utilisant à la racine l'opérateur binaire « / » avec, comme fils gauche un opérateur « + » dont les fils sont `e1` et `e2` et comme fils droit la constante 2. Notons que dans la méthode `clone`, il est inutile de copier les `Expression` car elles ont été développées comme des classes immutables.

⇒ Créez une classe `FonctionCible` qui implémente l'interface `algogen.Environnement`. Elle doit porter un `EnvEval` qui donne la valeur des variables au point $(x_1, \cdots, x_n)$ et une valeur (un `double`) qui donne la valeur cible y de la fonction en ce point.

L'objectif est de trouver des expressions qui passent par un point. Il faut donc implémenter la fonction `double eval(IIndividu i)`. On retypera cet `IIndividu` en `IndividuExpression` pour accéder à l'`Expression` qu'il stocke. Une expression est d'autant meilleure qu'elle propose une valeur proche de la valeur y recherchée au point $(x_1, \cdots, x_n)$ considéré. La valeur de la fitness d'une expression sera donc l'inverse du carré de la différence entre la valeur de la fonction au point considéré et la

valeur cible. Mathématiquement, on écrit :
$fitness(f) = \frac{1}{(y-f(x_1,\cdots,x_n))^2}$
⇒ Modifiez les méthodes de `PopulationFactory` pour créer vos nouveaux individus et environnements. Assurez-vous que la méthode `main()` fonctionne toujours ou adaptez-la au besoin pour tester vos classes.
Vous devez observer une certaine convergence et, si vous laissez trop de générations passer, une explosion exponentielle de la taille des expressions. On pourra tester sur des populations de 1000 individus mais sur peu de générations.

4.3 Copie profonde

La classe `Expression` est immutable. En d'autres termes, il n'existe pas de moyen de modifier une expression à travers son API publique. Si l'on n'avait pas fait ce choix, il faudrait ajouter une opération de copie profonde d'une expression, pour assurer la cohérence de la population. A quel moment faudrait-il copier les expressions si elles n'étaient pas immutables ? Comment déclarer et implémenter une telle opération ?

4.4 Remise du TME

Donnez la réponse aux questions ci-dessus. Par ailleurs, copiez-collez le code de votre classe principale et une trace d'exécution de votre programme pour le cas où vous évaluez une population de dix individus constitués d'expressions à deux variables.