TME 10 : Fichier de configuration et stratégie d'évolution

Objectifs pédagogiques : pattern *Singleton*, chargement de fichiers, implantation du pattern *Strategy*

Créez un projet « tme10 ». Importez l'archive de votre TME 9.

10.1 Générateur aléatoire unique

Quand on utilise des méthodes adaptatives, on est souvent content de pouvoir rejouer exactement le même run que précédemment, en maîtrisant la séquence des nombres aléatoires qui ont été générés par le programme. A l'heure actuelle, si on relance le programme deux fois de suite, on obtient deux résultats différents. Pour maîtriser la séquence des nombres aléatoires, il est possible de passer à un objet de la classe `Random` (avec la méthode `setSeed(long graine)`) une « graine » à partir de laquelle sera engendrée la séquence des nombres suivants. Si on lui passe deux fois la même graine, on obtient deux fois la même séquence de nombres. Mais, dans l'état actuel du programme, nous avons de nombreuses instances distinctes de générateurs de nombres aléatoires répartis dans diverses classes, ce qui complique la génération d'une séquence unique à l'aide d'une graine passée en paramètre au programme. Pour résoudre ce problème, vous allez faire appel au pattern *Singleton*.

⇒ Dans le package `pobj.util`, créez une classe `Generateur` qui permet la de génération de nombres aléatoires. L'unique instance statique de cette classe peut être récupérée par une méthode statique `getInstance()`. Cette instance utilise un objet de type `Random` déclaré `static final`.

⇒ Utilisez la délégation pour que votre générateur dispose des méthodes `nextInt()` et `nextDouble()` usuelles.

⇒ Passez la graine à votre générateur via une méthode `setSeed(long seed)` que vous appellerez dans la méthode `main()` de la classe principale de votre logiciel.

⇒ Eliminez toutes les instances de `Random` de votre programme au profit d'invocations statiques de la forme `Generateur.getInstance().nextXXX()`. Pensez également à remplacer les invocations à `Math.random()` par des appels à la méthode `nextDouble()` du générateur. Utilisez une graine et vérifiez que votre programme donne deux fois de suite le même résultat.

10.2 Fichier de configuration

Le jeu de paramètres qu'accepte la méthode `main()` de votre logiciel commence à être conséquent. Vous allez créer un mécanisme de configuration flexible, s'appuyant sur des fichiers. Votre fichier de configuration contiendra les informations suivantes :

- TypeIndividu : Agent
- Labyrinthe : big.txt
- taillePop : 100
- nbGens : 70
- etc.

10.2.1 Définition de la configuration

⇒ Implémentez une classe `Configuration` munie des opérations :

- `String getParameterValue(String param)`,
- `void setParameterValue(String param, String value)`.

Vous la munirez d'une `Map<String,String>` pour assurer le stockage des valeurs de paramètres.

⇒ Déclarez une interface `AlgoGenParameter` et ajoutez-y des constantes de type `String`, pour loger les noms des paramètres légaux dans votre application (e.g. `public static final String TAILLE_POP = « TaillePopulation »;`).

⇒ Modifiez votre méthode `main()` pour qu'elle lise ses arguments depuis une instance de `Configuration`. Vous créerez cette instance à la main.

10.2.2 Sauvegarde/Chargement

⇒ Ajoutez des opérations permettant la sauvegarde et le chargement d'une configuration. Vous vous appuierez sur la sérialisation, en vous inspirant des fonctions dans `agent.maze.MazeLoader.java`

⇒ Pour éviter d'avoir à spécifier ce jeu de paramètres manuellement en ligne de commande à chaque fois, créez un fichier de configuration dont le nom sera l'unique paramètre passé en ligne de commande à votre logiciel.

⇒ Déportez le code de votre méthode `main()` dans une méthode qui prend une configuration en paramètre. La méthode `main()` crée la configuration et la passe à cette méthode.

10.3 La configuration comme Singleton

On souhaite à présent pouvoir accéder à la configuration de tout point du code de l'application. On suppose qu'à un instant donné il n'existe qu'une seule configuration active.

⇒ Implémentez le Design Pattern *Singleton*, pour offrir l'unique instance de `Configuration` dans l'application via une méthode static de la classe `Configuration` : `Configuration getInstance();` Vous pourrez à ce stade ajouter à votre classe Configuration divers éléments : stratégie de contrôle des agents, algorithme élitiste (on garde les 20% meilleurs de chaque génération) ou pas, ... Ces points de variation/configuration de votre application peuvent s'appuyer sur la classe `Configuration` via le Singleton, sans remettre en cause l'architecture de votre application (signatures actuelles préservées...). Mais on a ainsi une façon de positionner finement des paramètres de configuration en profondeur dans l'application, sans nécessairement en informer les couches hautes. En règle générale, chaque package peut définir sa propre interface `MesConstantesParameter` avec des constantes désignant les noms des paramètres de configuration qu'il utilise. Vous pourrez éventuellement développer une classe `Swing` pour afficher ces paramètres et les mettre à jour.

10.4 Stratégies d'évolution

Votre projet contient les opérations élémentaires qui permettent de créer un individu à partir de deux individus sélectionnés, puis de l'évaluer avant d'être inséré dans la population. Sur cette base, nous avons mis en place un algorithme qui permet de modifier une population de génération en génération. Vous allez à présent mettre en place deux mécanismes d'évolution distincts :

- le premier, de type générationnel, est celui que nous avons utilisé jusqu'ici.
- le second, plus progressif, consiste à sélectionner deux parents en fonction de leur fitness, à engendrer un rejeton à partir de ces parents, et à remplacer un individu choisi en fonction de sa fitness par ce rejeton. De même que dans le cas générationnel, on effectue cette opération un nombre de fois fixé a priori ou bien jusqu'à ce que l'un des individus engendrés soit totalement satisfaisant.

10.4.1 Stratégies de sélection d'un parent

Vous allez faire appel au pattern *Strategy* pour permettre le choix entre la sélection uniforme existante et une sélection dépendante de la fitness. Toutes les opérations réalisées dans le cadre de ce TME se situent dans le package `algogen`.

⇒ Créez une interface `IndivSelecteur` qui définit une méthode `getRandom(Population pop)` qui renvoie un `Individu` de la population passée en paramètres.

⇒ Créez une classe `SelecteurUniforme` qui implémente `IndivSelecteur`. Sa méthode `getRandom(Population pop)` renvoie un individu tiré au hasard dans la population à laquelle ce sélecteur est associé.

⇒ Dans la classe `Population`, ajoutez une méthode `getSommeFitnesses()` qui renvoie la somme des fitnesses de tous les individus présents dans la population.

Créez une classe `SelecteurParFitness` qui implémente `IndivSelecteur`. La méthode `getRandom(Population pop)` de cette classe associe à chaque individu de la population passée en paramètres une probabilité de sélection proportionelle à sa fitness. selon une méthode de sélection appelée « roue de la fortune » décrite par la figure 1. L'idée est que chaque individu se voit attribuer une proportion de la roue proportionnelle à sa fitness, on lance la roue et on choisit l'individu sur lequel est le pointeur quand elle s'arrête.

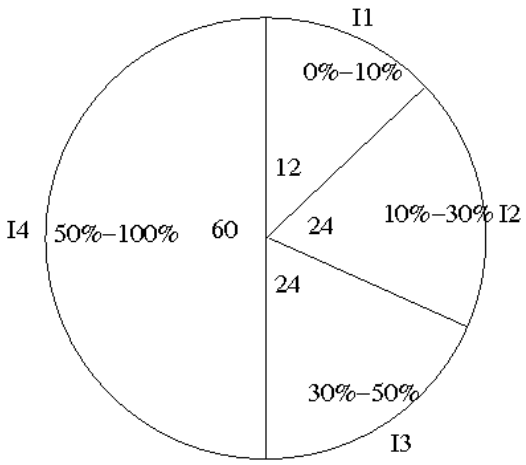

Figure 1: Illustration du principe de la roue de la fortune. La population contient les individus I1 à I4 de fitness respectives 12, 24, 24 et 60. On calcule leur fitness relative en divisant par la somme des fitness (ici, 120). Si on tire 0.2814, c'est I2 qui sera choisi (car le nombre tiré est entre 10% et 30%).

Pour réaliser la sélection par la roue de la fortune, on découpe l'intervalle $[0, 1[$ en sous-intervalles dont la taille est proportionnelle à la fitness de chacun des individus, puis on tire un nombre au hasard dans $[0, 1[$ et on renvoie l'individu dont l'intervalle est pointé par ce nombre. En pratique, on appelle x le produit du nombre tiré aléatoirement par la somme des fitnesses de tous les individus. On se dote d'un compteur de fitness initialisé à 0 et on lui ajoute la fitness de chaque individu au fur et à mesure qu'on les parcourt. Quand ce compteur dépasse la valeur x, l'individu pointé est renvoyé.

10.4.2 Implantation des stratégies d'évolution

Pour permettre le choix entre les deux mécanismes d'évolution décrits en introduction, vous allez faire appel au pattern *Strategy*. Au sein de la classe `Population`, assurez-vous que la méthode de production d'une nouvelle génération `Population reproduire(Population pop)` intègre aussi l'opération de mutation des individus nouvellement créés. La nouvelle signature de la méthode sera `Population reproduire(Population pop, double ratio)`.

⇒ Créez une interface `IEvolution` qui définit la méthode `Population reproduire(Population pop, double ratio)`.

⇒ Insérez cette interface en tant qu'attribut de la classe `Population` et déléguez-lui l'appel de la méthode `Population reproduire(Population pop, double ratio)` qui doit se trouver déjà au sein de la classe `Population`.

Associez à l'interface `IEvolution` deux implémentations : les classes `EvolutionGenerationnelle` et `EvolutionParNiche`. Ces classes contiennent un attribut de type `IndivSelecteur` qui leur est passé lors de la construction et qui permet, via un pattern *Strategy*, de choisir les individus soit de façon uniforme, soit en fonction de leur fitness.

La méthode `Population reproduire(Population pop, double ratio)` de la classe `EvolutionGenerationnelle` se comporte comme celle que vous aviez définie précédemment dans la classe `Population`. Celle de la classe `EvolutionParNiche` choisit les parents de façon proportionnelle à leur fitness et installe leur rejeton à la place de l'individu qui a la plus mauvaise fitness au sein de la population.

⇒ Pour implémenter ce dernier point, ajoutez une méthode `void removeLast()` au sein de la classe `Population`.

⇒ Pour décider si vous faites appel à une évolution générationnelle ou à une évolution par niche, ajoutez au fichier de configuration créé ci-dessus une nouvelle entrée `evolutionGenerationelle` qui vaut `true` par défaut. De même, pour le choix uniforme ou par fitness des individus, vous ajouterez une entrée `selectionUniforme` qui vaut `true` par défaut. Vous ferez en sorte que la classe `Population` crée les mécanismes d'évolution et de sélection du bon type en fonction de la valeur donnée à ces paramètres.

Faites le nécessaire pour tester le comportement de ces deux stratégies sur les deux types d'individus sur lesquels nous avons travaillé jusqu'à présent (les expressions arithmétiques et les agents).

Pour tester les différentes configurations, vous noterez qu'il faut appeler l'évolution par niche sur un bien plus grand nombre de génération, chaque génération de remplaçant qu'un individu à la fois.

10.5 Remise du TME

Mesurez le temps d'exécution avant et après utilisation de la classe `Generateur`. Constatez-vous une perte d'efficacité significative ? Copiez-collez dans votre mail le code de votre classe `Generateur`. Copiez-collez un fichier de configuration rempli et la trace d'exécution correspondante.

Comparez les performances des deux stratégies d'évolution codées en précisant les paramètres que vous avez utilisés dans chaque cas.