TP 4-6 - Programmation objet et Langage Java

ETN4

Thème : Héritage
Objectif : Optimisation et Intelligence Collective.

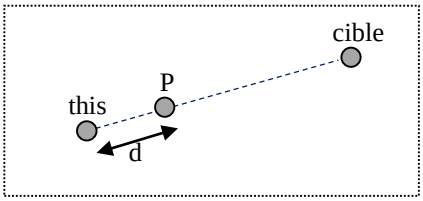
Programme demandé :
Le programme modélise une colonie de robots autonomes répartis sur une zone minière d'une planète éloignée. Les robots explorent le sol grâce à un capteur leur permettant de connaître la teneur en minerai à la verticale de leur position. Le but du programme est de trouver la stratégie de recherche optimale conduisant à la découverte de la position la plus fructueuse. Le projet se déroule en 3 séances de TP, réparties sur trois parties. En annexe, un diagramme UML décrit l'ensemble des classes utilisées dans ce projet.

Partie A. Modélisation de la zone minière

1. Classe Point

La zone de recherche est définie par la région du plan délimitée par des valeurs (x,y) comprises entre 0 et 1. Ecrire la classe Point avec les méthodes qui vous sembleront utiles, en incluant les méthodes suivantes :

- Constructeur vide
  Le constructeur vide attribue des coordonnées arbitraires. On définira pour cela un attribut statique :
  `private static Random random=new Random(0);`
  On utilisera cet attribut en écrivant par exemple : `x=random.nextDouble();`
- Constructeur avec deux arguments (abscisse et ordonnée). Le point est initialisé avec ces coordonnées.
- distance(Point p)
  Le point courant (*this*) détermine sa distance Euclidienne à un point p donné en argument.
- move(Point cible, double *d*) *// cible : direction ; d : distance de déplacement autorisée par unité de temps en tenant compte de la puissance du mobile (ex : robot).*
  La méthode renvoie un point P situé sur le segment entre *this* et la cible. La méthode renvoie la valeur cible si les deux points sont égaux ou si la distance à la cible est inférieure à *d*. Dans les autres cas, le point renvoyé est à la distance *d* de *this* (on utilisera le théorème de Thalès pour calculer les coordonnées). On veillera à ce que le nouveau point reste dans la zone (ex : si x<0 alors x=0).



2. Cartes des teneurs en minerai

A une position P donnée correspond une teneur en minerai sur la carte. Cette dernière est modélisée par une interface *AbstractProblem* contenant une seule méthode :

```
double teneur(Point position) ;     // respecter la syntaxe exacte
```

Cette interface nous permet de décliner plusieurs classes de problème.

On demande au minimum d'écrire les deux classes suivantes, qui sont inspirées de fonctions utilisées par les chercheurs en optimisation (voir Wikipedia pour plus de détails et d'exemples : https://en.wikipedia.org/wiki/Test_functions_for_optimization).

- Classe Sphere : $T=1-(a^2+b^2)/2$
  - minimum de T est 0 en position (1,1),
  - maximum de T est 1.0 en position (0,0).

- Classe Eggholder :
  - minimum de T est -1049.13 en position (0,1),
  - maximum de T est 959.64 en position (1.0,0.895).

  Soit k=1024 ; x=k.(a-0.5) ; y=k.(b-0.5).

  T=-f(x,y) avec :

$$f(x,y) = -(y+47)\sin\sqrt{\left|\frac{x}{2}+(y+47)\right|} - x\sin\sqrt{|x-(y+47)|}$$

Pour tous les problèmes, il faut impérativement normaliser la teneur entre 0.0 et 255.0
Sur *Madoc* se trouve un fichier *TP4_Java.zip* contenant la classe *Viewer*. Celle-ci doit être placée dans le package source où se trouvent les autres classes du projet. (Note : la classe Viewer présuppose l'existence de l'interface *AbstractProblem* et de la classe *Point*.)
La classe Viewer permet de visualiser la carte du minerai en fausses couleurs grâce à une palette allant du bleu (teneur proche de 0) au rouge (teneur proche de 255) en passant par le vert et l'orange. Voici un exemple d'utilisation de cette classe :

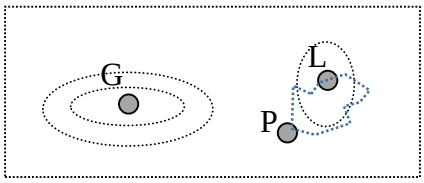
```
Sphere pb=new Sphere();
Viewer.display(pb);
```

Remarque : le problème *Sphere* est très simple et il sera facile de trouver l'optimum. Le problème *Eggholder* est beaucoup plus difficile à traiter et vous demandera de déterminer la bonne stratégie pour déjouer ses nombreux pièges !

Partie B. Modélisation de la colonie de robots

3. Classe Robot

Le robot possède une position courante P ainsi que la dernière position locale L à laquelle le robot a trouvé le maximum de teneur en minerai. De même, il connaît la teneur courante et la meilleure teneur obtenue durant ses déplacements. Enfin, deux attributs statiques mémorisent la meilleure position G globale et la meilleure teneur globale (pour toute la colonie).



Pour optimiser la recherche, on doit pouvoir modifier le comportement du robot à tout moment. Comme il existe une grande variété d'actions possibles, on préconise de définir des classes spécifiques pour chacune des stratégies envisagées. Le robot possède un attribut *behavior* (de type *Behavior*) qui détermine comment se déplace le robot. Cet attribut est initialisé à null. Dans ce cas, le robot est inerte et son comportement consiste à rester sur place. Un accesseur permet d'attribuer à *behavior* une instance de classe lui permettant de définir sa stratégie, déclinée sous différentes implémentations (exploration locale, regroupement global, …).

Le mode de base du déplacement du robot est un pur mouvement Brownien ; une marche aléatoire à la recherche de minerais. Ce mode est défini par une classe de base, la classe *Behavior* décrite dans la section suivante. Cette classe, ainsi que ses classes dérivées, possède une seule méthode : `void move(Robot robot)`.
Lorsqu'on demande au robot de se déplacer (ex : `robot.walk();`), le robot fait appel au sein de sa méthode `walk` à la méthode `behavior.move`. L'avantage de cette technique de délégation est que le comportement du robot peut varier selon l'état (l'instance associée) dans lequel se trouve l'attribut `behavior`.
On remarquera que la méthode `move` de Behavior a pour paramètre le robot concerné. Comme il s'agit de l'objet même qui fait appel à cette méthode, il faut écrire la méthode `walk` de la classe Robot de la façon suivante :

```
public void walk(){
      if (behavior!=null)
            behavior.move(this);
}
```

4. Classe Behavior

Cette classe de base visite aléatoirement la zone : elle prend un point A au hasard (constructeur vide de la classe Point) et calcule le vecteur MA, où M est le point central (0.5, 0.5). Soit P la position courante du robot, la nouvelle orientation P' du robot est donnée par : P'=P+MA. Le robot est limité dans son déplacement : il ne peut s'y rendre qu'à la vitesse équivalente au paramètre *d*, comme indiqué dans la méthode `move` de la classe Point (*d* est de l'ordre de 0.05). Notez bien que le point A est défini à chaque nouvel appel de la méthode `move`. Ecrire une classe de test contenant une méthode main avec les instructions suivantes :

```
Point p=new Point(0.5,0.5); // position initiale pour le robot
Robot robot=new Robot(p);
Behavior explore=new Behavior();
robot.setBehavior(explore); // le robot possède le comportement explore
for (int i=0;i<10;i++){
      robot.walk();
      System.out.println(i+" : "+robot.getPosition());
}
```

5. Classe BasicMission

Cette classe gère la colonie de robots. Elle est associée à un gisement (attribut de type *AbstractProblem*). Le constructeur possède deux arguments, le gisement à considérer et le nombre de robots à déployer. Les robots sont mémorisés par un attribut de type tableau. On écrira les méthodes *set* et *get* pour gérer le robot ayant un index donné. Le constructeur fait appel à une méthode d'initialisation qui créé les robots, les positionne au centre (0.5,0.5) de la zone et leur attribue le comportement *Behavior*. La méthode principale de la classe est la méthode *run* possédant les instructions suivantes :

```
      for (int iter=0;iter<100;iter++){
            collect();
            System.out.println("iter="+iter+" "+Robot.getAllBestGain());
            walk();
      }
```

La méthode *collect* met à jour la valeur en teneur récoltée par les robots (voir la méthode *teneur* précédente pour récupérer cette valeur) et la méthode *walk* déplace tous les robots en utilisant leur propre méthode *walk*. La méthode `getAllBestGain` renvoie la valeur de teneur maximale obtenue par la colonie de robots. Ecrire une méthode de test pour le problème *Sphere* avec 20

robots et afficher la meilleure teneur obtenue ; on remarquera que la colonie ne trouve pas forcément la valeur optimale.

6. Classe Follow et LocalBest

Ces deux classes étendent *Behavior* en utilisant une stratégie de suivi : *Follow* se déplace vers la meilleure solution globale obtenue jusqu'alors ; *LocalBest* retourne à la meilleure position explorée individuellement par le robot. Dans les deux cas, on considère une orientation P' et on applique la méthode move de la classe Point (même valeur de *d* que précédemment).

7. Visualisation

Sur *Madoc* se trouve un fichier *TP4_Java.zip* contenant les classes *Viewer*, *Reader* et *Printer*, ainsi qu'un fichier tabulé *robots.txt* donné à titre d'illustration.

Reader et *Printer* doivent être placés dans le package *utilitaires* ; *robots.txt* dans le dossier *data*. La classe *Viewer* permet de visualiser le déplacement des robots à partir d'un fichier tabulé. L'animation tourne en une boucle sans fin. L'exemple suivant montre les conventions utilisées (séparateur \t) :

```
-1
0.5    0.5
0.4    0.32
-1
0.48   0.52
0.36   0.39
```

```
Temps t=0 de simulation
x, y du robot 0
x, y du robot 1
Temps t=1 de simulation
x, y du robot 0
x, y du robot 1
```

Sur cet exemple, nous avons deux robots considérés à deux temps successifs. Tous les *n* millisecondes, le simulateur lit les positions relatives au temps courant de simulation et actualise l'affichage. L'instruction suivante lance la visualisation avec la carte de minerai définie par le problème *pb* (voir question A.2) : `Viewer.display(pb) ;`

La spirale en mouvement qui apparait alors correspond au fichier de points *robots.txt* donné en exemple. Pour visualiser la colonie de robots, vous devez utiliser la classe *Printer* qui permet d'écrire les positions dans le fichier texte. Voici un exemple d'utilisation pour les deux premières lignes de l'exemple précédent :

```
Printer pr=new Printer("data/robots.txt");
pr.println("-1");
double x=0.5,y=0.5 ;
pr.println(x+"\t"+y);
pr.close(); // à n'utiliser qu'après avoir écrit toutes les lignes.
```

On créera un attribut *printer* dans la classe *BasicMission* et on complètera la méthode *run* en ajoutant une méthode *store* après l'appel à la méthode *collect*. La méthode *store* enregistre le symbole -1 dans le fichier ainsi que toutes les positions courantes des robots (ne pas oublier d'appeler la méthode *close* à la fin de la méthode *run*). Lancer une simulation et visualiser ensuite le résultat avec *Viewer*.

Partie C. Introduction de politiques de recherche

8. Classe AbstractPolicy

Les stratégies précédentes montrent leurs limites : la colonie ne converge pas toujours vers un optimum global et les robots sont dispersés en des points d'intérêts divers. L'observation des animaux sociaux révèle que la bonne politique consiste à combiner exploration locale (*Behavior*, *LocalBest*, …) et globale (*Follow*). Le but de notre programme est de pouvoir estimer les performances de différentes politiques. Pour ce faire, nous allons créer une classe abstraite *AbstractPolicy* servant de modèle à de futures classes dérivées.

L'approche envisagée consiste à découper le temps de recherche global en différentes phases, chacune de ses phases possédant une durée déterminé et un mode de comportement spécifique. Par exemple, la première phase peut durer un temps de 80 itérations pendant laquelle les robots se comportent en mode *Behavior*, puis en mode *Follow* lors d'une deuxième phase s'étendant sur 20 itérations. Le constructeur de la classe possède un argument entier *n* représentant le nombre de phases. Les phases *p* sont indexées de 0 à *n*-1.
Deux méthodes abstraites sont à écrire : l'une renvoyant une instance de *Behavior* pour la phase *p*, l'autre renvoyant la durée de la phase *p*.

9. Classe SimplePolicy

Cette classe dérive de la précédente et utilise en attribut un tableau d'objets de type *Behavior* et un tableau mémorisant les temps respectifs. Voici un exemple de création d'une politique de recherche, basée sur 3 modes de durées variables :

```
SimplePolicy policy=new SimplePolicy(3);
policy.add(new Behavior(),60);  //   phase 0 de durée 60 itérations
policy.add(new LocalBest(),20); // phase 1
policy.add(new Follow(),20);    // phase 2
```

10. Classe SmartMission

Cette classe dérive de la classe *BasicMission*. Elle possède un attribut *policy* supplémentaire de type *AbstractPolicy*, qui sera initialisé par un argument du constructeur. La classe utilise les mêmes méthodes que sa classe parente, tout en redéfinissant certaines méthodes :

- Initialisation : les robots ne reçoivent plus de comportement par défaut
- *run* : la méthode respecte l'algorithme suivant (tous les paramètres indiqués (n, t, mode) sont donnés par les méthodes de l'attribut *policy*) :

```
Soit n le nombre de phases
Pour la phase p allant de 0 à n-1 faire
      Mettre à jour les modes de comportement des robots en fonction
      de la phase p
      soit t la durée de la phase p
      Pour t itérations faire :
            collect()
            store()
            walk()
```

Remarque : les méthodes *collect*, *store* et *walk* sont celles de la classe parente.
Ecrire une méthode de test et comparer les performances obtenues en modifiant la politique utilisée (choisir quelques exemples diversifiés de configuration de *SimplePolicy*). Proposer des critères de comparaison appropriés, comme par exemple la teneur moyenne obtenue par la colonie lors de la dernière itération. Pour ne pas biaiser les résultats, il faut que le nombre total d'itérations soit le même pour chaque configuration.

Programmes optionnels :

- Classe *Hybride*. Cette classe implémente un nouveau comportement des robots (*Behavior*). La classe possède un attribut *proba* qui indique la probabilité avec laquelle le comportement est de type *Follow* et non *Explore*. On tire un nombre aléatoire entre 0 et 1 (ex : la coordonnée x d'un point aléatoire). Si cette valeur est inférieure à *proba*, alors suivre le comportement *Follow*, sinon utiliser *Explore*. On recommande de faire appel aux méthodes de ces dernières classes plutôt que d'en recopier les instructions. Tester la méthode sur les 2 problèmes avec différentes valeurs de probabilité.

- Classe *EvolutiveHybride*. Cette classe dérive de *AbstractPolicy* et se définit comme une séquence de comportements hybrides avec une probabilité évoluant au cours des phases. On proposera une méthode donnant la valeur de *proba* pour une phase donnée, définie à partir d'une fonction paramétrée (ex : fonction logistique). Estimer le jeu de paramètres donnant les performances optimales.

- Définir la stratégie optimale utilisant un type *SimplePolicy* basé uniquement sur des séquences de comportement de type B (*Behavior*) ou F(*Follow*). On prendra par exemple 5 phases de 20 itérations et on calculera les performances de toutes les combinaisons possibles (2^5 au total ; exemple de combinaison : séquence BBFBF). Comparer le meilleur résultat avec ceux obtenus avec les stratégies hybrides précédentes.

Annexe

Diagramme de classes UML