

Programmation objet et Langage Java - 29/01/2024

TP 4-6

Optimisation et Intelligence Collective

Table des matières :

1. Introduction.....	3
2. Modélisation de la zone minière.....	4
2.1. Classe Point.....	4
2.2. Classe des cartes des teneurs en minerai.....	9
3. Modélisation de la colonie de robots.....	12
3.1. Classe Robot et Behaviour.....	12
3.2. Classe BasicMission.....	15
3.3. Classe Follow et LocalBest.....	17
3.4. Visualisation.....	18
4. Introduction de politique de recherche.....	19
4.1. Classe AbstractPolicy.....	19
4.2. Classe SimplePolicy.....	19
4.3. Classe SmartMission.....	21
4.4. Validation des performances.....	22
5. Conclusion.....	26
6. Annexes.....	27
6.1. Modélisation de la zone minière.....	27
6.2. Modélisation de la colonie de robots.....	32
6.3. Introduction de politique de recherche.....	38

1. Introduction

Dans le cadre de notre quatrième année du cycle ingénieur de la filière Électronique et Technologies du Numérique (ETN4), nous avons entrepris la programmation d'une colonie de robots autonomes en vue de réaliser une exploration optimale d'une zone minière sur une planète lointaine.

Le programme élabore la modélisation d'une colonie de robots autonomes déployés sur une zone minière d'une planète éloignée. Ces robots explorent le sol à l'aide d'un capteur qui leur permet d'évaluer la teneur en minerai à la verticale de leur position. L'objectif du programme est de déterminer la stratégie de recherche optimale conduisant à la découverte de la position la plus lucrative.

Cette programmation s'effectue en langage Java à l'aide du logiciel Eclipse mis à notre disposition. Le projet se déroule en trois séances de travaux pratiques, réparties sur trois parties distinctes et se divise en différentes classes.

Ce rapport présente une étude approfondie ainsi que les résultats obtenus au cours des différentes étapes de ce projet.

2. Modélisation de la zone minière

Cette section vise à modéliser et à comprendre les caractéristiques individuelles de la zone minière, y compris les caractéristiques ponctuelles et cartographiques. Nous définirons quatre classes concrètes : Point, AbstractProblem, Sphere et Eggholder. La première classe nous permet de créer un point et d'accéder au point après l'avoir déplacé d'une distance spécifiée. Les trois dernières classes nous permettent de modéliser des cartes de teneurs en minerai, y compris des cartes de type Sphere et Eggholder.

2.1. Classe Point

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

- **Constructeur vide ou avec arguments**

Le rôle le plus important du constructeur est de terminer l'initialisation lors de la création d'un objet. Lorsque nous créons (new) un nouvel objet, le constructeur correspondant sera appelé et l'initialisation des paramètres sera terminée selon que les paramètres sont transmis ou non.

Dans cette partie, le constructeur vide vise à attribuer des coordonnées arbitraires. On définit donc pour cela un attribut statique :

```
private static Random random;
```

On utilisera cet attribut dans le constructeur vide en écrivant :

```
this.x=random.nextDouble();this.y=random.nextDouble();
```

de sorte qu'on peut initialiser un point aléatoire chaque fois qu'on entre dans le constructeur vide.

Voici la figure des codes :

```

public class Point {
    private static Random random;
    private double x,y;
    public double x_contain,y_contain;
    public Point() {
        random=new Random();
        this.x=random.nextDouble();
        this.y=random.nextDouble();
        x_contain=0;
        y_contain=0;
    }
    public Point(double x,double y) {
        this.x=x;
        this.y=y;
    }
}

```

Figure 1: Codes des deux types de constructeur

D'après cette figure 1, dans ce cas, on crée un objet avec deux arguments (abscisse et ordonnée). Le point est initialisé avec ces coordonnées comme cette figure.

`this.x=x;this.y=y;`

- Distance(Point p)

On détermine la distance entre le point courant et le point donné (Point p) à l'aide de l'équation Euclidienne :

$$d = \sqrt{(x - x_p)^2 + (y - y_p)^2}$$

L'argument p est le point donne

- Move(Point cible, double d)

Le point courant doit se déplacer vers le point cible en vitesse d qui est la distance de déplacement autorisée par unité de temps en tenant compte de la puissance du mobile.

Cette méthode renvoie un point P situé sur le segment entre le point de départ et la cible. Il est nécessaire de prendre en compte que si le point dépasse le point cible ou non. Si les deux points (départ, cible) sont égaux ou le point dépasse le point cible, le point renvoyé est la cible. Dans les autres cas, le point renvoyé est à la distance d du point départ.

On réalise cette méthode grâce au Théorème de Thalès:

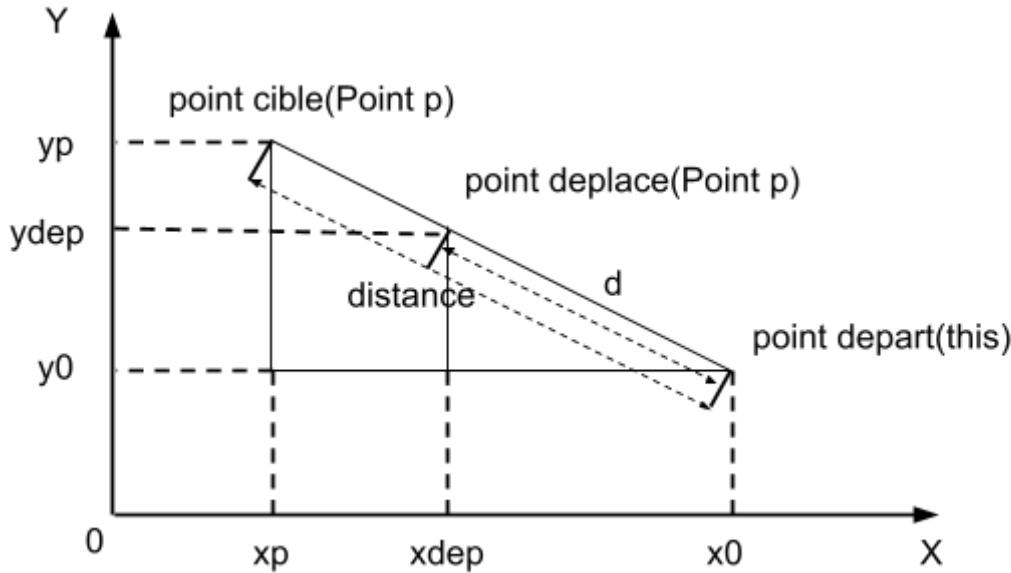


Figure 2 : Théorème de Thalès

L'équation de Théorème de Thalès est ci-dessous :

$$\frac{x_0 - x_{\text{dep}}}{x_0 - x_p} = \frac{d}{\text{distance}} = \frac{y_0 - y_{\text{dep}}}{y_0 - y_p}$$

Donc, on est capable de déduire l'abscisse et l'ordonnée du point renvoyé.

```
public Point move(Point cible, double d) {
    System.out.println("le point avant est "+this.x+ " , "+this.y);
    double dis=distance(cible);

    double xp=(d/dis)*(cible.x-this.x)+this.x;
    double yp=(d/dis)*(cible.y-this.y)+this.y;
    if(Math.abs(dis) < 1e-15) {
        xp=this.x;yp=this.y;
        System.out.println("il arrive le cible");
    }

    Point p=new Point(xp,yp);
    if(dis<d||(xp==cible.x&&yp==cible.y)) {
        p=cible;
    }
    System.out.println("le point actuel est "+xp+" , "+yp);
    System.out.println();
}
```

Figure 3: Codes de la fonction mouvement

Il nous faut aussi veiller à ce que le nouveau point doit rester dans la zone($0 \leq x \leq 1, 0 \leq y \leq 1$). Dans le sujet de TP, un exemple nous est donné : si $x < 0$ alors $x = 0$.

Cependant, cette méthode de réglage n'est pas assez rigoureuse : comme mentionné précédemment, le point de départ est déplacé selon la direction vectorielle du point cible, et le point renvoyé est également situé dans cette direction. Lorsque le point cible est en dehors de la zone($0 \leq x \leq 1, 0 \leq y \leq 1$) et que le point déplacé est par ailleurs en dehors de la zone, le point réglage doit revenir dans la direction opposée selon le vecteur d'origine jusqu'au point d'intersection de la limite de zone. Donc, on va se débrouiller dans le cas où le point cible et le point de déplacés sont en dehors de la zone. Il y a quatre situations comme le tableau ci-dessous :

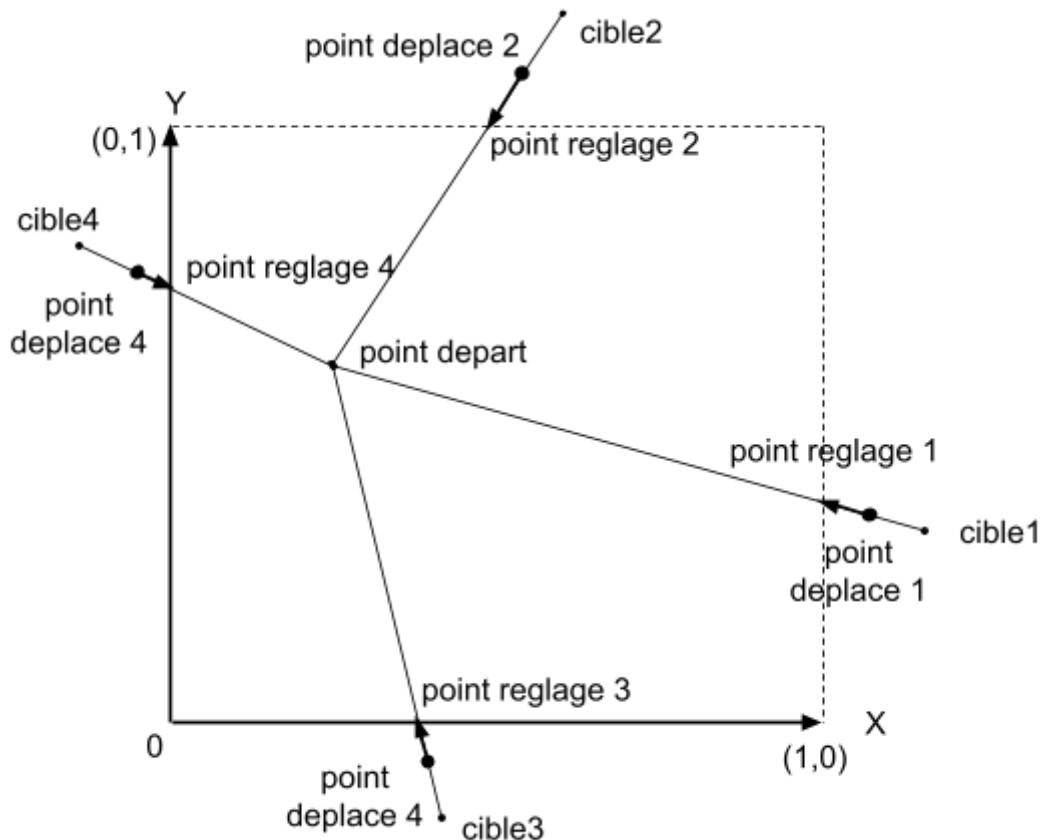


Figure 4 : Quatre situations de réglage

Pour calculer le point d'intersection, il faut premièrement analyser s'il y a un point déplacé en dehors de la zone :

```

if(! (p.x>=0&&p.x<=1&&p.y>=0&&p.y<=1)) {
    if(analyse_contain_segment(this.x,this.y,cible.x,cible.y,0,0,1,1)) {
        System.out.println("le point croise est "+xContain+", "+yContain);
        p=new Point(xContain,yContain);
    }
}

```

Figure 5: Codes d'analyse du point déplacé

Maintenant, on a l'hypothèse, il y a un point d'intersection entre le point départ et le point cible, on peut le déduire en considérant la forme point-pente d'une ligne droite :

$$y = m(x - x_0) + y_0,$$

Dont m est la pente, x_0 est l'abscisse de départ, y_0 est l'ordonnée de départ.

$$\text{double } m = (\text{ycible} - \text{ydepart}) / (\text{xcible} - \text{xdepart});$$

D'après la figure 4, on prend le cas 1, donc les coordonnées de point d'intersection et la forme d'une ligne droite sont:

$$(\text{maxX}, y), y = m * (\text{maxX} - \text{x1}) + \text{y1};$$

Cependant, il convient de noter que la situation où il y a deux points d'intersections pour la ligne droite, donc on va analyser quel point est approprié :

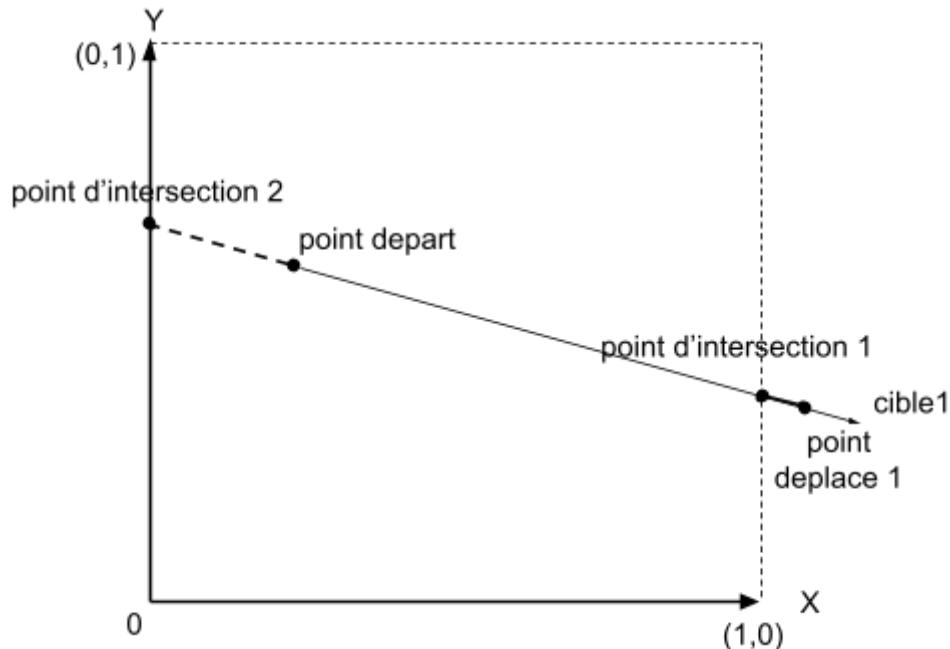


Figure 6: Situation où il y a deux points d'intersection

Dans ce cas en figure 6, la direction vectorielle est du point départ au point d'intersection 1, donc il est possible de le déterminer en comparant la distance entre le point départ et le point cible avec celle du point d'intersection 2 et le point cible.

```

if(index==2) {
    if(temp_dis[0]>temp_dis[1]) {
        x_contain=temp_x[1];y_contain=temp_y[1];
    }
    else {
        x_contain=temp_x[0];y_contain=temp_y[0];
    }
}
    
```

Figure 7: Analyse le point d'intersection approprié

Voici la figure de la fonction d'analyse du point d'intersection :

```
public boolean analyseContainSegment (double x1, double y1, double x2, double y2, double minX, double minY, double maxX, double maxY) {
    double m = (y2 - y1) / (x2 - x1);
    double y = m * (minX - x1) + y1;
    boolean flag=false;
    double []temp_dis=new double[2];
    double []temp_x=new double[2];
    double []temp_y=new double[2];
    int index=0;
    if (y > minY && y < maxY) {
        x_contain=minX;
        y_contain=y;
        temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contain-y2));
        temp_x[index]=x_contain;temp_y[index]=y_contain;
        index+=1;
        System.out.println("le point croisiel calcule est "+x_contain+" , "+y_contain);
        flag=true;
    }
    y = m * (maxX - x1) + y1;
    if (y > minY && y < maxY) {
        x_contain=maxX;
        y_contain=y;
        temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contain-y2));
        temp_x[index]=x_contain;temp_y[index]=y_contain;
        index+=1;
        System.out.println("le point croisse2 calcule est "+x_contain+" , "+y_contain);
        flag=true;
    }
    double x = (minY - y1) / m + x1;
    if (x > minX && x < maxX) {
        y_contain=minY;
        x_contain=x;
        temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contain-y2));
        Line breakpoint:Point [line: 129] - main(String[])
    }
}
```

Figure 8 : Codes 1 de la fonction analyseContainSegment

```
double x = (minY - y1) / m + x1;
if (x > minX && x < maxX) {
    y_contain=minY;
    x_contain=x;
    temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contain-y2));
    temp_x[index]=x_contain;temp_y[index]=y_contain;
    index+=1;
    System.out.println("le point croisse3 calcule est "+x_contain+" , "+y_contain);
    flag=true;
}

x = (maxY - y1) / m + x1;
if (x > minX && x < maxX) {
    y_contain=maxY;
    x_contain=x;
    temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contain-y2));
    temp_x[index]=x_contain;temp_y[index]=y_contain;
    index+=1;
    System.out.println("le point croisse4 calcule est "+x_contain+" , "+y_contain);
    flag=true;
}

if(index==2) {
    if(temp_dis[0]>temp_dis[1]) {
        x_contain=temp_x[1];y_contain=temp_y[1];
    }
    else {
        x_contain=temp_x[0];y_contain=temp_y[0];
    }
}
return flag;
}
```

Figure 9 : Codes 2 de la fonction analyseContainSegment

D'après la figure 8 et 9 au-dessus, chaque fois qu'on détermine le point croisé, il faut ensuite sauvegarder les coordonnées du point d'intersection et la distance de sorte qu'on peut choisir le point correct après.

2.2. Classe des cartes des teneurs en minerai

Dans cette section, nous allons utiliser la méthode d'interface Java pour modéliser la carte du minerai afin d'afficher et de calculer le contenu du minerai à un certain point P.

En langage Java, l'interface est un type abstrait et une collection de méthodes abstraites. Une classe hérite et implémente les méthodes abstraites dans l'interface en héritant de l'interface.

Dans la séance de TP, il est impératif d'écrire une classe d'interface `AbstractProblem` contenant la fonction abstrait teneur pour calculer la valeur de teneur, laquelle sera héritée et implémentée par la classe `Sphere` et `Eggholder`.

```
5 public interface AbstractProblem {
6     public abstract double teneur(Point position);
7 }
```

Figure 10 : Codes de la classe abstraite AbstractProblem

- **Double teneur(Point position) ;**

Selon le cahier des charges, on demande au minimum d'écrire les deux classes suivantes, qui sont inspirées de fonctions utilisées par les chercheurs en optimisation, Les équations de calcul de la teneur en minéraux sont suivantes:

Classe Sphere :

$$T = 1 - \frac{(a^2+b^2)}{2}$$

minimum de T est 0 en position (1,1),
maximum de T est 1.0 en position (0,0).

Classe Eggholder :

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

$$T = -f(x, y)$$

minimum de T est -1049.13 en position (0,1),
maximum de T est 959.64 en position (1.0,0.895).
k=1024, x=k*(a-0.5), y=k*(b-0.5).

Ensuite, nous pouvons utiliser la classe `Viewer` donnée sur madoc pour visualiser la carte du mineraï. Pour toutes les questions, le contenu doit être normalisé entre 0,0 et 255,0 afin qu'une palette de couleurs soit utilisée pour visualiser la carte du mineraï du bleu (contenu proche de 0) au rouge (contenu proche de 255).

Pour `Sphere`, la valeur de T est comprise entre 0 et 1, on va donc multiplier 255 pour decaler la gamme de couleur entre 0 et 255:

double couleur=T*255;// T in(0,255)

Pour `Eggholder`, la valeur de T est comprise -1049.13 et 959.64, on va donc la normaliser en gamme entre 0 et 1 puis multiplier 255:

T=(T-Min)/(Max-Min); //T in (0,1)
double couleur= T*255 ; //T in(0,255)

Voici la carte `Sphere` et de `Eggholder`:

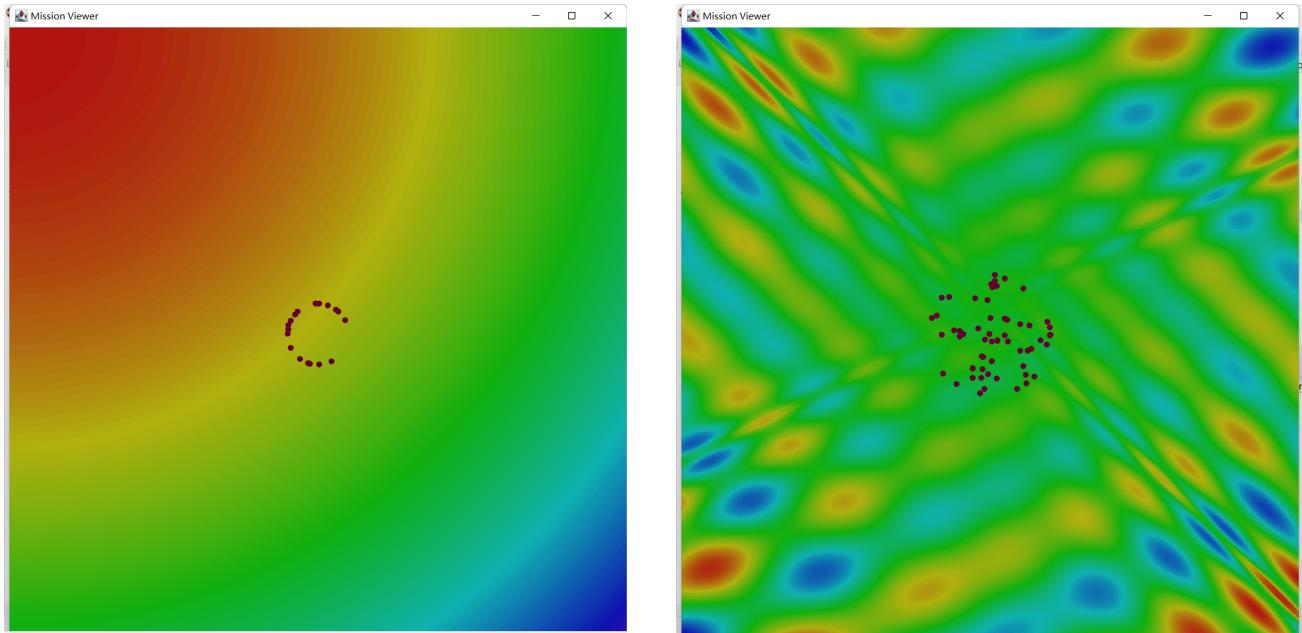


Figure 11 : Visualisation la carte de façon Sphere (à gauche) et Eggholder (à droite)

D'après cette image, la couleur signifie la teneur minière dans la zone. Plus le point rouge est proche, plus la teneur en minerai est élevée. Sur la figure 11, le contenu est le plus grand en haut à gauche et le plus petit en bas à droite.

3. Modélisation de la colonie de robots

Ce chapitre vise à modéliser le comportement d'un seul robot ainsi que de tous les robots miniers. Nous définissons cinq classes : Robot, Behaviour, BasicMission, Follow et LocalBest. La première catégorie vise à contrôler à tout moment le comportement dynamique du robot. La classe de comportement définit la méthode d'exploration aléatoire du robot. La classe BasicMission gère un essaim de robots menant des explorations aléatoires. Au fur et à mesure que le robot explore, il est possible de trouver le meilleur emplacement et la quantité optimale de minerai. Ensuite, on introduit deux nouvelles stratégies : Follow et LocalBest pour que le robot se déplace vers les positions optimales locales et globales.

3.1. Classe Robot et Behaviour

- **Attributs**

Chaque robot détient une position courante (P) ainsi que la dernière position locale (L) où il a identifié la plus grande concentration de minerai. Le robot est également informé de la teneur en minerai actuelle et de la meilleure teneur obtenue au cours de ses déplacements, des informations spécifiques à chaque robot. De plus, il existe deux attributs statiques qui enregistrent la meilleure position globale (G) et la teneur globale maximale. Ces attributs diffusent ces informations à l'ensemble de la colonie de robots.

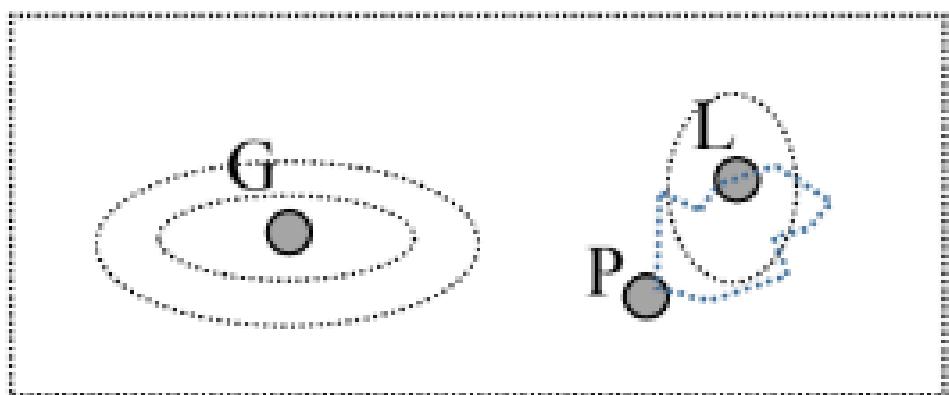


Figure 12 : Positions que le robot possède

Voici les définitions des attributs et fonctions de la classe robot.

```

1 public class Robot {
2     private static Point G;
3     private static double teneur_meilleur;
4     private double teneur_par_robot;
5     private Behaviour behaviour;
6     private Point courant;
7     private Point locale;
8     public Robot(Point p) {
9         behaviour=null;
10
11         courant=p;
12         locale=p;
13         teneur_meilleur=Double.NEGATIVE_INFINITY;
14         teneur_par_robot=Double.NEGATIVE_INFINITY;
15         G=new Point(0.5,0.5);
16     }
17 }
```

Figure 13: Attributs que le robot possède

D'après cette figure 13, nous définissons trois attributs de position du robot : le point actuel, le point de concentration maximum local et le point de concentration maximum global. Il convient de noter que nous utilisons le mot "Static" lors de la définition de la position globale et de la concentration globale des variables membres, de sorte que la variable membre n'appartienne pas à un objet, mais n'appartient qu'à une seule classe. Même s'il existe plusieurs instances d'objet de cette classe, il n'existe qu'une seule variable membre statique. Dans notre cas, il est avantageux que tous les robots soient capables de partager la meilleure position globale.

Afin d'optimiser les recherches, il faut pouvoir modifier le comportement du robot à tout moment. Comme il existe une grande variété d'actions possibles, il est nécessaire de définir des classes spécifiques pour chacune des stratégies envisagées. Donc, on introduit l'attribut Behavior.

L'attribut behavior dont le type de classe est Behavior 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, qui inclut l'utilisation de la fonction Move pour régir le déplacement de robot.

- **Void walk()**

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

Selon le cahier des charges, il faut demander au robot de se déplacer en faisant appel au sein de sa méthode Walk à la méthode Behavior.Move. L'avantage pour cette technique est qu'il va permettre au robot de choisir le mode de mouvement selon l'instance associée tels que Behavior, Follow et Localbest.

Pour la méthode Behavior.Move, elle sert à la mouvement Brownien des robots:

- **Void move(Robot robot)**

Selon le cahier des charges, on remarque que la méthode Move de classe Behavior à pour paramètre le robot concerné. C'est la raison pour laquelle la classe Behavior

utilise les attributs et fonctions du robot au courant en réalisant la stratégie d'exploration.

Pour effectuer le mouvement aléatoire de la stratégie Behavior, la mise en jeu de la classe Point est nécessaire, dont le constructeur vide et la fonction Point.Move servent à cela.

Le principe de mouvement est donné par :

$$P' = P + MA$$

dont P' est le point déplacé, P est le point courant, MA est le vecteur du point de central(0.5,0.5) au point aléatoire. Par ailleurs, la vitesse de déplacement du point(d) doit être 0.05, ce qui est un facteur qui influence les résultats d'exploration des robots.

```
void move(Robot robot) {
    System.out.println("enter behaviour");
    Point A=new Point();
    Point O=new Point(0.5,0.5);
    Point courant=robot.getPosition();
    double xp=courant.obtenir_abscisse();double yp=courant.obtenir_ordonne();
    double xa=A.obtenir_abscisse();double ya=A.obtenir_ordonne();
    double xo=O.obtenir_abscisse();double yo=O.obtenir_ordonne();
    double xoa=xa-xo;double yoa=ya-yo;
    Point cible=new Point(xp+xoa,yp+yoa);
    Point deplacement=courant.move(cible, 0.05);
    robot.setPosition(deplacement);
}
```

Figure 14: Codes de la fonction move de la classe Behavior

D'après cette figure 14 au-dessus, on a utilisé la fonction Obtenir_Abscisse et Obtenir_Ordonne pour obtenir les coordonnées du point. Ce sont ces fonctions de membres qui nous permettent d'acquérir et de changer les attributs dans une instance.

```
public Point getPosition() {
    return this.courant;
}
public void setPosition(Point p) {
    this.courant=p;
}
public double getBestGain() {
    return this.teneur_par_robot;
}
public static double getAllBestGain() {
    return Robot.teneur_meilleur;
}
public static Point getbestlocation() {
    return Robot.G;
}
public Point getlocation_par_robot() {
    return this.locale;
}
public void changeteneur(double g) {
    this.teneur_par_robot=g;
}
public void changeteneur_meilleur(double g) {
    Robot.teneur_meilleur=g;
}
public void changelocation_meilleur(Point p) {
    Robot.G=p;
}
public void changelocation(Point p) {
    this.locale=p;
}
```

Figure 15 : Codes des fonctions qui régissent les propriétés de la classe

3.2. Classe BasicMission

Dans cette section, après avoir créé la zone de recherche, le robot et le comportement qui doit être adopté, il faut maintenant modéliser une colonie des robots. Au fur et à mesure que les robots explorent, ils sont capables de découvrir les meilleures zones locales ainsi que les zones mondiales.

- **Attributs**

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.

```

1 public class BasicMission {
2     protected AbstractProblem gisement;
3     protected int nombre_robots;
4     protected Robot [] robot_liste;
5     protected Printer printer;
6     protected String source="data/robots.txt";
7
8     public BasicMission(AbstractProblem gise,int nb) {
9         this.gisement=gise;this.nombre_robots=nb;
10        this.robot_liste=new Robot[nb];
11        robots_initialisation();
12
13        printer=new Printer(source);
14    }
15 }
```

Figure 16 : Codes des attributs et le constructeur de la classe BasicMission

Comme les fonctions définies avant, il faut aussi écrire les méthodes Set et Get pour gérer le robot ayant un index donné.

```

public void set(int index,Robot r) {
    robot_liste[index]=r;
}
public Robot get(int index) {
    return this.robot_liste[index];
}
```

Figure 17 : Codes des fonctions gérant le robot ayant un index donné

- **Public void robots_initialisation()**

Le constructeur fait appel à une méthode d'initialisation qui crée les robots, les positionne au centre (0.5,0.5) de la zone et leur attribue le comportement Behavior.

```

public void robots_initialisation() {
    for(int i=0;i<this.nombre_robots;i++) {
        Point p=new Point(0.5,0.5);
        Robot r=new Robot(p);
        r.setBehavior(new Behaviour(this.gisement));
        set(i,r);
    }
}

```

Figure 18 : Codes de la fonction d'initialisation des robots

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();
}

```

Figure 19 : Codes de la fonction run de la classe BasicMission

- **Public void collect()**

La méthode Collect met à jour la valeur en teneur récoltée par les robots. Comme on a défini l'attribut de gisement dont type est AbstractProblem précédemment, on peut récupérer les valeurs des teneurs par la fonction Teneur.

En plus, cette méthode vise à renouveler la meilleure valeur locale détectée par un robot ainsi que celle globale détectée par tous les robots.

```

public void collect() {
    System.out.println("dans la fonction collect:");
    for(int i=0;i<this.nombre_robots;i++) {
        System.out.println("robot "+i+": ");
        Robot robot_courant=robot_liste[i];
        Point p_courant=robot_courant.getPosition();
        double gain=gisement.Teneur(p_courant);
        System.out.println("teneur maintenant "+gain);
        if(gain>robot_courant.getBestGain()) {
            robot_courant.changeteneur(gain);
            robot_courant.changelocation(p_courant);
        }
        if (robot_courant.getBestGain()>Robot.getAllBestGain()) {
            robot_courant.changeteneur_meilleur(robot_courant.getBestGain());
            robot_courant.changelocation_meilleur(p_courant);
            System.out.println("la location meilleure globale maintenant est "+p_courant.obtenir_abscisse()+"," +p_courant.obtenir_ordonne());
        }
        System.out.println("le teneur meilleur cherche par robot"+i+" est "+robot_courant.getBestGain());
        System.out.println();
    }
}

```

Figure 20 : Codes de la fonction collect

- **Public static double getAllBestGain()**

C'est une méthode qui renvoie la valeur de teneur maximale obtenue par la colonie de robots. Chaque fois qu'on obtient la valeur de teneur courant, on la compare avec celle globale pour décider si la renouveler ou non dans la fonction Collect.

```

public static double getAllBestGain() {
    return Robot.teneur_meilleur;
}

```

Figure 21 : Codes de la fonction getAllBestGain

Il convient de noter qu'on a défini cette méthode comme un type static pour partager la valeur entre tous les robots.

- **Public void walk()**

La méthode Walk déplace tous les robots en utilisant leur propre méthode walk.

```
public void walk() {
    System.out.println("dans la fonction walk:");
    for(int i=0;i<this.nombre_robots; i++) {
        Robot robot_courant=robot_liste[i];
        System.out.println("robot "+i+": ");
        robot_courant.walk();
        System.out.println("le robot "+i+" deplace au point "+robot_courant.getPosition().obtenir_abscisse()+" , "+robot_courant.getPosition().obtenir_ordonne());
        System.out.println();
    }
}
```

Figure 22 : Codes de la fonction Walk

3.3. Classe Follow et LocalBest

Comme nous l'avons dit, dans notre cas, nous souhaitons modifier le comportement du robot à tout moment. Afin de diversifier la stratégie d'exploration, deux nouvelles classes qui étendent le comportement doivent être ajoutées, telles que Follow et LocalBest.

- **Classe Follow**

Cette stratégie a pour le but de déplacer tous les robots vers la meilleure position globale obtenue après l'exploration aléatoire empruntant la stratégie Behavior.

La structure différente par celle de Behavior est la méthode Move. Dans ce cas, ce qu'on s'intéresse est le meilleur point global qui contient la meilleure valeur minière. Donc, on va déplacer les robots vers le point global directement.

```
public void move(Robot robot) {
    Point courant=robot.getPosition();
    Point cible =Robot.getbestlocation();
    Point deplacement=courant.move(cible, 0.05);
    robot.setPosition(deplacement);
}
```

Figure 23 : Codes de la fonction move de la classe Follow

On peut remarquer qu'il existe également une fonction appelée move dans la classe Behavior, et Follow étend Behaviour. C'est ce que nous appelons l'héritage de classe et le remplacement de fonctions dans le langage Java. L'avantage de la substitution de fonctions est que les sous-classes peuvent définir leur propre comportement selon leurs besoins. Dans la section précédente, nous avons défini des propriétés comportementales afin de pouvoir sélectionner une stratégie basée sur l'instance, ce qui est également conforme aux exigences ici.

- **Classe LocalBest**

Elle retourne à la meilleure position explorée individuellement par un robot.

```
public void move(Robot robot) {
    System.out.println("enter localbest");
    Point courant=robot.getPosition();
    Point deplacement=courant.move(robot.getLocation_par_robot(), 0.05);
    robot.setPosition(deplacement);
}
```

Figure 24: Codes de la fonction move de la classe LocalBest

3.4. Visualisation

Dans cette étape, on va implémenter les classes MissionViewer, Reader et Printer pour visualiser l'exploration des robots. À l'aide des fichiers qui nous sont fournis, nous copions les classes MissionViewer, Reader et Printer dans notre projet. Il est aussi nécessaire de comprendre la structure des données dans le texte robots.txt.

<pre> -1 0.5 0.5 0.4 0.32 -1 0.48 0.52 0.36 0.39 </pre>	<pre> 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 </pre>
---	--

Figure 25 : L'exemple des conventions utilisées sur le texte robots

Sur cet exemple, nous avons deux robots considérés à deux temps successifs. Toutes les millisecondes, le simulateur lit les positions relatives au temps courant de simulation et actualise l'affichage. Le chiffre -1 signifie des différents epochs pour tous les robots. Les deux lignes après le chiffre -1 signifient les deux positions maintenant des deux robots.

Donc pour visualiser la colonie de robots, on doit utiliser la classe Printer qui permet d'écrire les positions dans le fichier texte comme le format au-dessus. Ensuite, on crée un attribut printer dans la classe BasicMission et on complète la méthode run en ajoutant une méthode store après l'appel à la méthode collect. Dans cette méthode, on utilise la fonction println pour écrire le chiffre -1 séparant des robots par différentes epochs ainsi que les points courants des robots.

```

public void store() {
    printer.println("-1");
    for(int i=0;i<this.nombre_robots;i++) {
        Robot robot_courant=robot_liste[i];
        printer.println(robot_courant.getPosition().obtenir_abscisse()+"\t"+robot_courant.getPosition().obtenir_ordonne());
    }
}

```

Figure 26 : La fonction store de visualisation

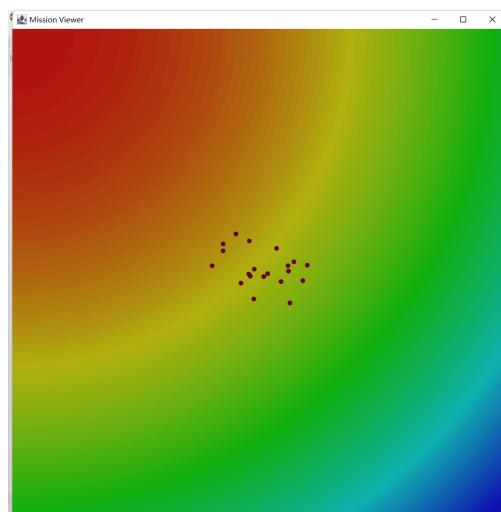


Figure 27 : Visualisation des robots sur la carte minière

4. Introduction de politique de recherche

Grâce aux chapitres précédents, nous avons réussi à établir une colonie de robots, à les faire se déplacer dans la zone de recherche, à recueillir des informations sur la teneur en minerai de la zone dans laquelle ils se trouvent, et ainsi à évoluer vers une zone plus riche en minerai.

En examinant les différentes méthodes de recherche précédentes, nous avons constaté qu'elles présentent quelques défauts et ne permettent pas une optimisation efficace de la recherche. En effet, la colonie ne converge pas toujours vers un optimum global, et les robots sont dispersés en divers points d'intérêt. Dans cette section, nous allons donc développer une politique générale pour la recherche de minerai.

4.1. Classe AbstractPolicy

Nous allons créer une classe abstraite `AbstractPolicy` servant de modèle à de futures classes dérivées. L'avantage de la définir de façon abstraite est de généraliser les attributs et les comportements communs des différentes politiques de recherche.

- **Constructeur avec arguments**

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`.

```
public abstract class AbstractPolicy {
    protected int nombre_phase;
    public AbstractPolicy(int n) {
        this.nombre_phase=n;
    }
}
```

Figure 28: Codes du constructeur avec arguments de la classe `AbstractPolicy`

- **Méthodes abstraites**

Selon le cahier des charges, il faut écrire deux méthodes abstraites dont l'une renvoie une instance de `Behavior` pour la phase `p`, l'autre renvoie la durée de la phase `p`. Les méthodes abstraites n'ont pas besoin d'être implémentées dans celle-ci.

```
protected abstract int obtenir_duree(int index);
protected abstract Behaviour obtenirBehaviour(int index);
```

Figure 29: Codes des méthodes abstraites de la classe `AbstractPolicy`

4.2. Classe SimplePolicy

Cette classe dérivée de la classe `AbstractPolicy` consiste à découper le temps de recherche global en différentes phases, chacune de ses phases possédant une durée déterminée et un mode de comportement spécifique. Donc, il nous faut définir

les attributs de façon tableau par rapport à la durée ainsi que le mode de comportement.

```
public class SimplePolicy extends AbstractPolicy{
    private Behaviour []behaviour_liste;
    private int []temps_iteration;
    private int index;
    public SimplePolicy(int n) {
        super(n);
        // TODO Auto-generated constructor stub
        this.behaviour_liste=new Behaviour[n];
        this.temps_iteration=new int[n];
        this.index=0;
    }
}
```

Figure 30: Codes des attributs et le constructeur de la classe SimplePolicy

D'après cette figure 30, on a ajouté un autre attribut Index pour la récupération et la modification des attributs.

Selon le cahier des charges, il nous faut écrire les codes correspondant aux ceux dans la fonction Main:

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

Figure 31: Codes de la fonction Main pour la classe SimplePolicy

La fonction Add a pour le but d'ajouter les différentes phases en différentes durées et stratégies. En plus, il faut également ajouter deux fonctions pour accéder aux propriétés des membres par rapport à la durée et à la stratégie de test.

```
@Override
public void add(Behaviour bh,int iter) {
    behaviour_liste[index]=bh;
    temps_iteration[index]=iter;
    index+=1;
}

@Override
protected int obtener_duree(int index) {
    // TODO Auto-generated method stub
    return this.temps_iteration[index];
}

@Override
protected Behaviour obtener_behaviour(int index) {
    // TODO Auto-generated method stub
    return this.behaviour_liste[index];
}

public static void main(String[] args) {
    // TODO Auto-generated method stub
    SimplePolicy policy=new SimplePolicy(3);
    policy.add(new Behaviour(),60); // phase 0 de durée 60 itérations
    policy.add(new LocalBest(),20); // phase 1
    policy.add(new Follow(),20); // phase 2
    System.out.println(" le nombre de phase est "+policy.nombre_phase);
    for(int i=0;i<3;i++) {
        System.out.println(" le policy maintenant est "+policy.obtenir_behaviour(i));
        System.out.println(" la duree pour ce policy est "+policy.obtenir_duree(i));
    }
}
```

Figure 32: Codes des fonctions de la classe SimplePolicy

4.3. Classe SmartMission

Cette classe dérive de la classe BasicMission. Il possède un attribut de politique supplémentaire de type SimplePolicy qui sera initialisé par les paramètres du constructeur pour attribuer tous les paramètres indiqués (index, durée, mode de comportement) à la classe SmartMission,

```
public class SmartMission extends BasicMission {
    private SimplePolicy policy;
    private int iteration;
    public SmartMission(AbstractProblem gise, int nb,int n_policy) {
        super(gise, nb);
        // TODO Auto-generated constructor stub
        this.policy=new SimplePolicy(n_policy);
        this.iteration=0;
    }
}
```

Figure 33: Codes d'initialisation de la classe SmartMission

La classe utilise les mêmes méthodes que sa classe parente, tout en redéfinissant la méthode Run:

- Run:

Les robots ne reçoivent plus de comportement par défaut, le mode de comportement du robot est affecté par la fonction Obtenir_Behaviour(i) dans la politique générale, dont i est l'indice du nombre de la phase. En plus, on obtient la durée par la fonction Obtenir_duree(i) définie dans la classe SimplePolicy.

```
@Override
public void run() {
    for(int i=0;i<this.policy.nombre_phase;i++) {
        System.out.println("strategie "+(i+1)+":");
        Behaviour behaviour=this.policy.obtenir_behaviour(i);
        this.iteration=this.policy.obtenir_duree(i);
        for(int j=0;j<nombre_robots;j++) {
            get(j).setBehavior(behaviour);
        }
    }
}
```

Figure 34: Codes d'initialisation des paramètres indiqués dans la classe SmartMission

On remarque que dans cette méthode, les fonctions Collect, Store et Walk utilisées sont des fonctions de la classe parent BasicMission.

```
public void run() {
    for(int i=0;i<this.policy.nombre_phase;i++) {
        System.out.println("strategie "+(i+1)+":");
        Behaviour behaviour=this.policy.obtenir_behaviour(i);
        this.iteration=this.policy.obtenir_duree(i);
        for(int j=0;j<nombre_robots;j++) {
            get(j).setBehavior(behaviour);
        }
        System.out.println("le point meilleur au debut est "+Robot.getbestlocation().obtenir_abscisse()+","+
                           +Robot.getbestlocation().obtenirordonne());
        for(int j=0;j<this.iteration;j++) {
            System.out.println("epoch "+(j));
            collect();
            store();
            walk();
        }
    }
    printer.close();
}
```

Figure 35 : Codes de la fonction Run dans la classe SmartMission

4.4. Validation des performances

Après avoir terminé le cours SmartMission, nous devons rédiger une méthode de test et comparer les performances obtenues en modifiant les stratégies utilisées pour qu'elles tendent à être les meilleures. Voici une politique utilisée:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Sphere pb=new Sphere();
    SmartMission smartmission=new SmartMission(pb,60,3);
    smartmission.policy.add(new Behaviour(pb),50);
    smartmission.policy.add(new LocalBest(pb),30);
    smartmission.policy.add(new Follow(pb),30);
    smartmission.run();
    Viewer.display(pb);
```

Figure 36: Codes d'une politique pour la validation des performances

Dans ce cas, on emprunte la vitesse d'exploration égale 0.05 et la carte de problème Sphere:

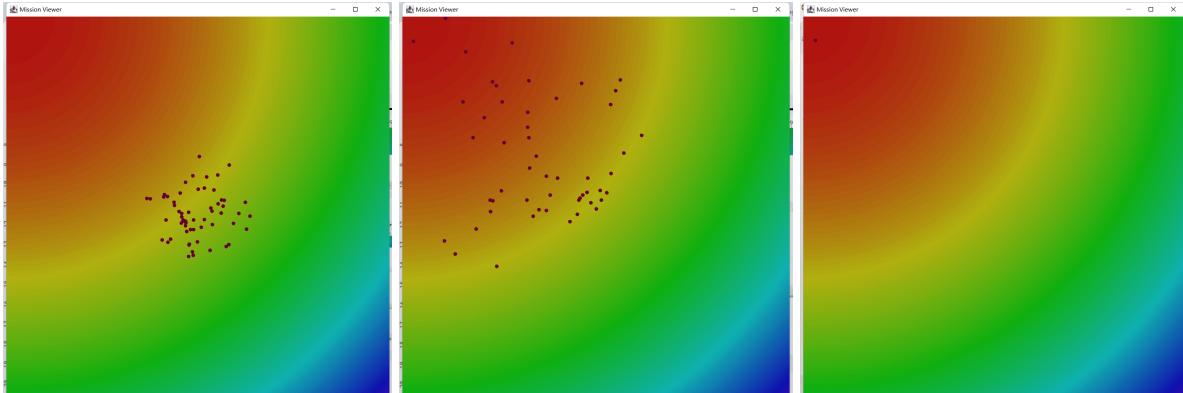


Figure 37: Performances des 60 robots en vitesse 0.05 sur la carte Sphere(Behavior, LocalBest, Follow)

Ensuite on modifie la vitesse de 0.5:

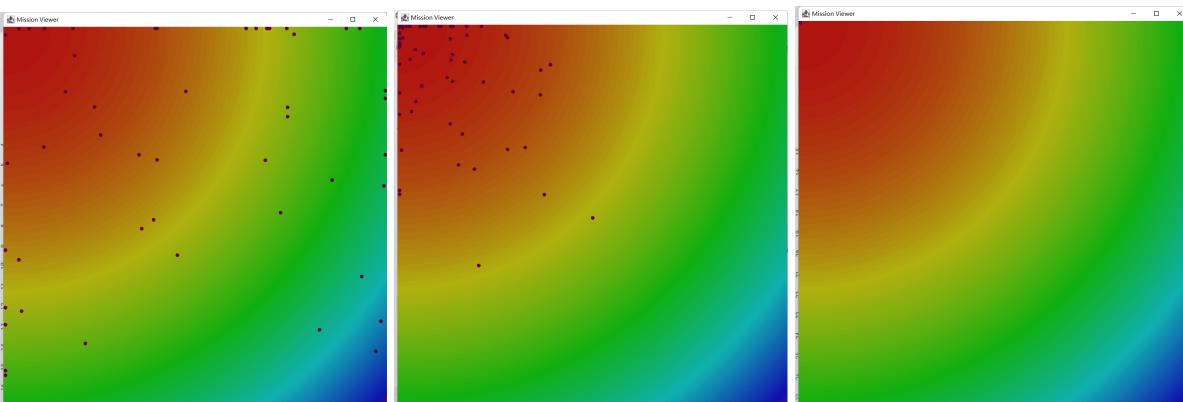


Figure 38 : Performances des 60 robots en vitesse 0.5 sur la carte Sphere(Behavior, LocalBest, Follow)

En plus, on emprunte l'autre carte Eggholder et effectue la même chose:

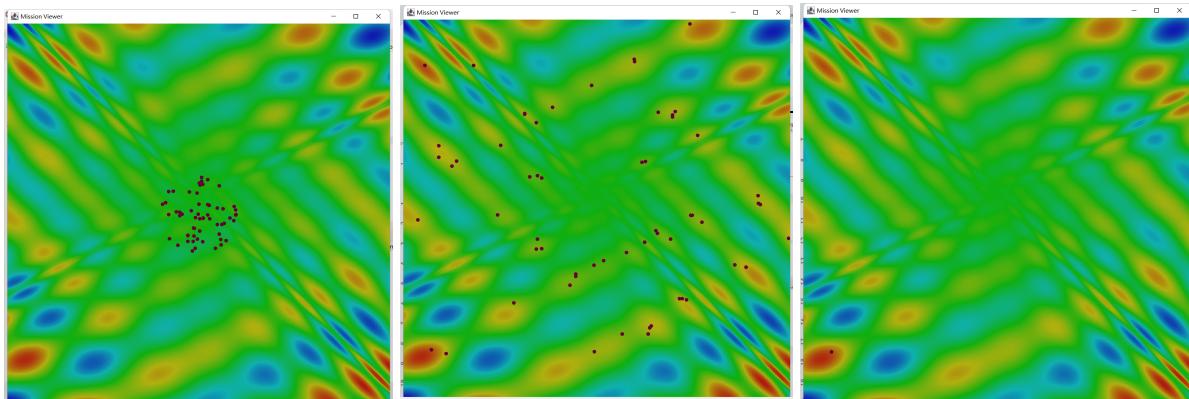


Figure 39: Performances des 60 robots en vitesse 0.05 sur la carte Eggholder(Behavior, LocalBest, Follow)

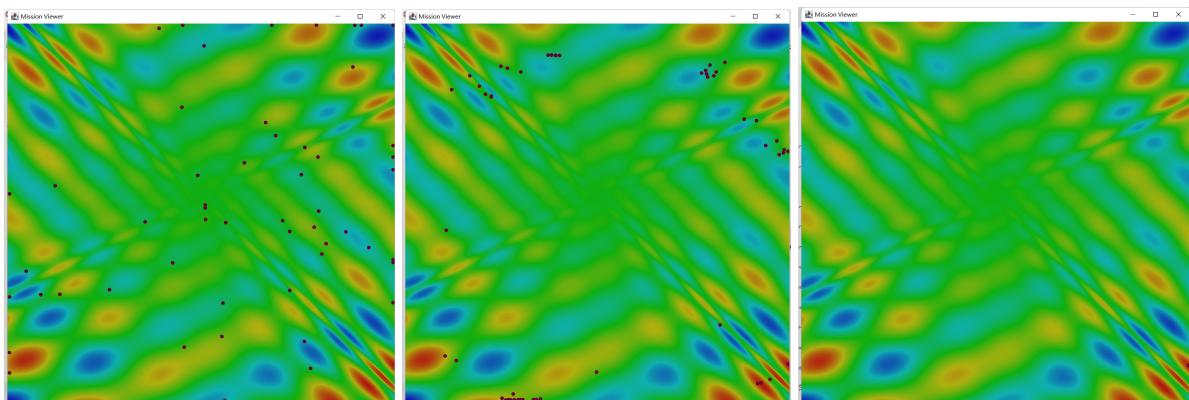


Figure 40: Performances des 60 robots en vitesse 0.5 sur la carte Eggholder(Behavior, LocalBest, Follow)

Ensuite, on modifie l'itération de la stratégie Behavior pour vérifier si la durée influence la performance.

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Eggholder pb=new Eggholder();
    SmartMission smartmission=new SmartMission(pb,60,9);
    smartmission.policy.add(new Behaviour(pb),80);
    smartmission.policy.add(new LocalBest(pb),30);
    smartmission.policy.add(new Follow(pb),30);
    smartmission.run();
    Viewer.display(pb);

}
```

Figure 41: Codes d'une politique pour vérifier la relation entre l'itération de la phase et la performance

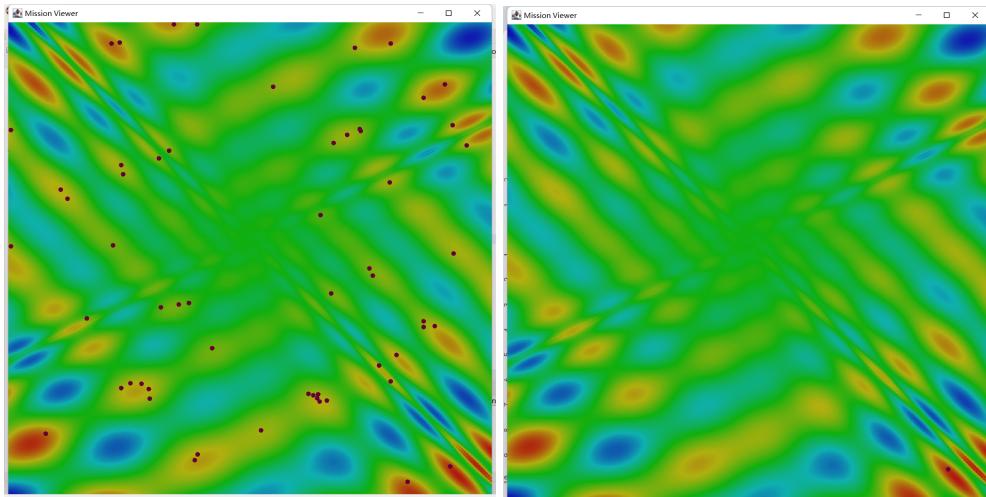


Figure 42 : Performances des 60 robots en vitesse 0.05 en durée 80 de Behavior sur la carte Eggholder(LocalBest, Follow)

Enfin, on utilise deux politiques pour voir si la performance fonctionne mieux qu'avant :

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Eggholder pb=new Eggholder();
    SmartMission smartmission=new SmartMission(pb,60,6);
    smartmission.policy.add(new Behaviour(pb),50);
    smartmission.policy.add(new LocalBest(pb),30);
    smartmission.policy.add(new Follow(pb),30);
    smartmission.policy.add(new Behaviour(pb),50);
    smartmission.policy.add(new LocalBest(pb),30);
    smartmission.policy.add(new Follow(pb),30);
    smartmission.run();
    Viewer.display(pb);
}
```

Figure 43: Codes des deux politiques pour la validation des performances

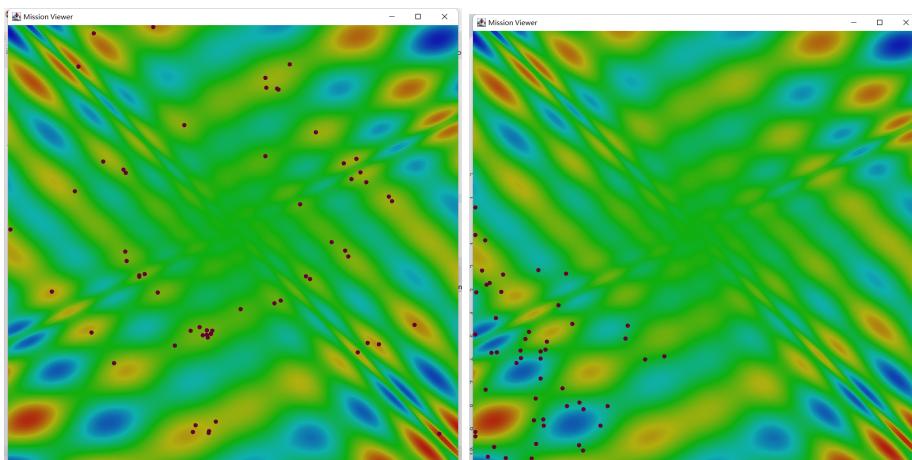


Figure 44: Performances des 60 robots en vitesse 0.05 sur la carte Eggholder(Behavior de la première politique, LocalBest de la première politique)

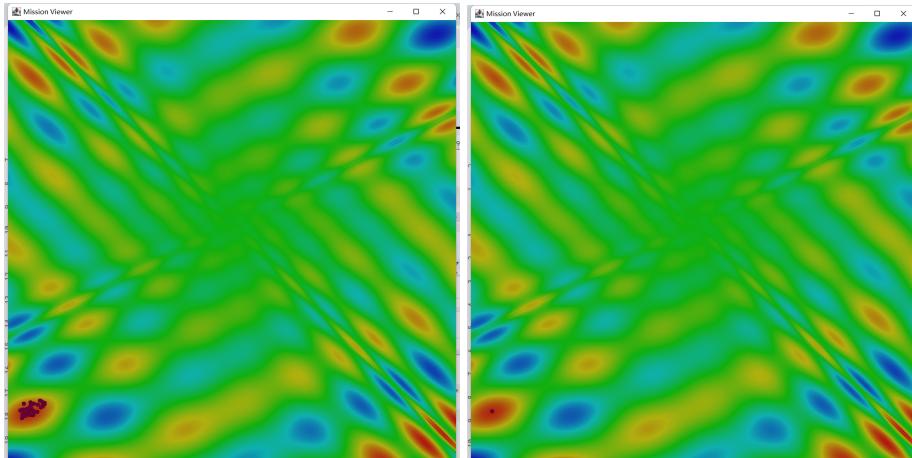


Figure 45: Performances des 60 robots en vitesse 0.05 sur la carte Eggholder(LocalBest de la deuxième politique, Follow de la deuxième politique)

En résumé, d'après la figure 37-40, on peut constater que la vitesse(d) régit le résultat d'exploration. Si la vitesse est plus élevée, les robots sont plus susceptibles d'aller loin en utilisant la stratégie Behavior et trouver la valeur maximale(255). La valeur maximale dans la carte minière Sphère ou la vitesse est 0.5 est 222.53 et celle en vitesse 0.5 est 254.68.

D'après la figure 39 et 41-42, on peut constater que la durée de la stratégie change un peu au résultat. Le temps a beaucoup moins d'impact sur les résultats de recherche que la vitesse. Il est plus difficile pour les robots de trouver la valeur maximale en modifiant la durée au lieu de la vitesse. La valeur maximale dans ce cas est 237.62 qui est plus grande qu'à celle de 226.06 de la figure 39.

D'après la figure 44-45, on peut constater que la performance change légèrement avec le nombre de politiques. Étant donné que l'utilisation de la deuxième stratégie Behavior est basée sur des points après la première stratégie Follow, donc il est possible pour le robot d'explorer en se concentrant toujours sur certaines zones, ce qui rendra plus difficile la recherche du point optimal.

5. Conclusion

Dans ce projet, nous avons utilisé le langage Java pour implémenter une simulation d'un groupe de robots recherchant l'emplacement optimal dans une zone minière. Au cours du processus d'implémentation, nous nous sommes inspirés des idées de la programmation orientée objet, à savoir l'encapsulation, l'héritage et le polymorphisme.

L'idée de l'encapsulation consiste à masquer les propriétés et les détails d'implémentation d'un objet et à exposer uniquement l'interface qui contrôle les propriétés de l'objet au monde extérieur. Par exemple, dans les classes écrites dans chaque chapitre, nous définissons les propriétés membres de la classe en private, protected, public et nous définissons les fonctions d'interface pour renvoyer la valeur d'un attribut membre ou modifier sa valeur.

L'idée de l'héritage est que les objets de sous-classe ont les propriétés et les méthodes de la classe parente, donc la sous-classe a le même comportement que la classe parent. Et les sous-classes peuvent remplacer les méthodes de la classe parente pour implémenter différentes fonctions. Par exemple, la classe SmartMission écrite dans le chapitre précédent hérite de BasicMission. Elle obtient l'effet d'attribuer différentes stratégies en redéfinissant la fonction Run de la classe parent. Et pour chaque stratégie spécifique, les fonctions Collect Walk Store de la classe parent sont utilisées. En plus, la classe abstraite AbstractPolicy utilisée dans le chapitre précédent est également une manifestation de la relation d'héritage. Les sous-classes qui héritent des classes abstraites peuvent implémenter des fonctions abstraites en fonction des besoins réels.

L'idée du polymorphisme reflète qu'un même comportement peut prendre plusieurs formes différentes. Elle est une abstraction du comportement qui présente la relation entre les implementations et peuvent être implémentées individuellement ou de plusieurs manières. Dans le chapitre précédent, nous avons utilisé l'interface AbstractProblem pour créer la carte de la zone minière, juste implémenter la fonction Teneur.

Grâce à ce projet et l'aide des enseignants, nous avons eu une bonne compréhension et application des idées de programmation orientée objet Java, ce qui sera bénéfique pour notre développement futur en entreprise.

6. Annexes

6.1. Modélisation de la zone minière

Annexe 1 : Classe Point

```
package Zone_miniere;
import java.math.BigDecimal;
import java.util.Random;
public class Point {
    private static Random random;
    private double x,y;
    public double x_contain,y_contain;
    public Point() {
        random=new Random();
        this.x=random.nextDouble();
        this.y=random.nextDouble();
        x_contain=0;
        y_contain=0;
    }
    public Point(double x,double y) {
        this.x=x;
        this.y=y;
    }
    public double distance(Point p) {
        double r = Math.sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
        return r;
    }
    public double obtener_abscisse() {
        return x;
    }
    public double obtener_ordonnee() {
        return y;
    }
    public boolean analyse_contain_segment (double x1, double y1, double x2,
double y2, double minX, double minY, double maxX, double maxY) {
        double m = (y2 - y1) / (x2 - x1);
        double y = m * (minX - x1) + y1;
        boolean flag=false;
        double []temp_dis=new double[2];
        double []temp_x=new double[2];
        double []temp_y=new double[2];
        int index=0;
        if (y > minY && y < maxY) {
            x_contain=minX;
            y_contain=y;

temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contai
n-y2));
```

```
temp_x[index]=x_contain;temp_y[index]=y_contain;
index+=1;
System.out.println("le point croisse1 calcule est "+x_contain+" ,
"+y_contain);
flag=true;

}

y = m * (maxX - x1) + y1;
if (y > minY && y < maxY) {
    x_contain=maxX;
    y_contain=y;

temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contai
n-y2));
    temp_x[index]=x_contain;temp_y[index]=y_contain;
    index+=1;
    System.out.println("le point croisse2 calcule est "+x_contain+" ,
"+y_contain);
    flag=true;

}

double x = (minY - y1) / m + x1;
if (x > minX && x < maxX) {
    y_contain=minY;
    x_contain=x;

temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contai
n-y2));
    temp_x[index]=x_contain;temp_y[index]=y_contain;
    index+=1;
    System.out.println("le point croisse3 calcule est "+x_contain+" ,
"+y_contain);
    flag=true;

}

x = (maxY - y1) / m + x1;
if (x > minX && x < maxX) {
    y_contain=maxY;
    x_contain=x;

temp_dis[index]=Math.sqrt((x_contain-x2)*(x_contain-x2)+(y_contain-y2)*(y_contai
n-y2));
    temp_x[index]=x_contain;temp_y[index]=y_contain;
    index+=1;
    System.out.println("le point croisse4 calcule est "+x_contain+" ,
"+y_contain);
    flag=true;
```

```
        }
        if(index==2) {
            if(temp_dis[0]>temp_dis[1]) {
                x_contain=temp_x[1];y_contain=temp_y[1];
            }
            else {
                x_contain=temp_x[0];y_contain=temp_y[0];
            }
        }
        return flag;
    }

    public Point move(Point cible,double d) {
        System.out.println("le point avant est "+this.x+ " , "+this.y);
        double dis=distance(cible);

        double xp=(d/dis)*(cible.x-this.x)+this.x;
        double yp=(d/dis)*(cible.y-this.y)+this.y;
        if(Math.abs(dis) < 1e-15) {
            xp=this.x;yp=this.y;
            System.out.println("il arrive le cible");
        }

        Point p=new Point(xp,yp);
        if(dis<d||(xp==cible.x&&yp==cible.y)) {
            p=cible;
        }
        System.out.println("le point actuel est "+p.x+ " , "+p.y);
        System.out.println();
        if(!(p.x>=0&&p.x<=1&&p.y>=0&&p.y<=1)) {

if(analyse_contain_segment(this.x,this.y,cible.x,cible.y,0,0,1,1)) {
                System.out.println("le point croissee est
"+x_contain+ " , "+y_contain);
                p=new Point(x_contain,y_contain);
            }
        }
        return p;
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Point p=new Point(0.2,0.8);
        Point a=new Point(0.8,1.2);
        Point b=new Point(1.2,0.6);
        Point c=new Point(0.4,-0.2);
        Point pp=p.move(a,0.1);
        System.out.println("le point change est "+pp.obtenir_abscisse()+" ,
"+pp.obtenir_ordonne());
    }
}
```

```

        Point pp1=p.move(b,1);
        System.out.println("le point change est "+pp1.obtenir_abscisse()+""
, "+pp1.obtenir_ordonne());
        Point pp2=p.move(c,0.4);
        System.out.println("le point change est "+pp2.obtenir_abscisse()+""
, "+pp2.obtenir_ordonne());
    }

}

```

Annexe 2 : Classe d'interface AbstractProblem

```

package Zone_miniere;

import java.awt.Dimension;

public interface AbstractProblem {
    public abstract double teneur(Point position);

}

```

Annexe 3 : Classe Sphere

```

package Zone_miniere;

import java.awt.Dimension;

import robots.Viewer;

public class Sphere implements AbstractProblem{
    public Sphere() {
    }

    @Override
    public double teneur(Point position) {
        // TODO Auto-generated method stub
        // minimum de T est 0 en position (1,1),
        // o maximum de T est 1.0 en position (0,0).
        double
T=1-(Math.pow(position.obtenir_abscisse(),2)+Math.pow(position.obtenir_ordonne()
, 2))/2; // T in(0,1)
        double couleur=T*255;// T in(0,255)
        return couleur;
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Sphere pb=new Sphere();
        Viewer.display(pb);
    }
}

```

```
}
```

Annexe 4 : Classe Eggholder

```
package Zone_miniere;

import robots.Viewer;

public class Eggholder implements AbstractProblem {
    public double f(double a,double b) {
        int k=1024;
        double x=k*(a-0.5);
        double y=k*(b-0.5);
        double
res=-(y+47)*Math.sin(Math.sqrt(Math.abs(x/2+(y+47))))-x*Math.sin(Math.sqrt(Math.
abs(x-(y+47))));
        return res;
    }
    @Override
    public double teneur(Point position) {
        // TODO Auto-generated method stub
        double
T=f(position.obtenir_abscisse(),position.obtenir_ordonne())*(-1); // T
in(-1048.13,959)
        double Min=-1049.13;double Max=959.64;
        T=(T-Min)/(Max-Min); //T in (0,1)
        double couleur= T*255 ; //T in(0,255)
        return couleur;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Eggholder pb=new Eggholder();
        Viewer.display(pb);
    }
}
```

6.2. Modélisation de la colonie de robots

Annexe 5 : Classe Robot

```
package robots;
import Zone_miniere.Point;

public class Robot {
    private static Point G;
    private static double teneur_meilleur;
    private double teneur_par_robot;
    private Behaviour behaviour;
    private Point courant;
    private Point locale;
    public Robot(Point p) {
        behaviour=null;

        courant=p;
        locale=p;
        teneur_meilleur=Double.NEGATIVE_INFINITY;
        teneur_par_robot=Double.NEGATIVE_INFINITY;
        G=new Point(0.5,0.5);
    }
    public void walk(){
        if (behaviour!=null)
            behaviour.move(this);
    }
    public void setBehavior(Behaviour behav) {
        behaviour=behav;
    }
    public Point getPosition() {
        return this.courant;
    }
    public void setPosition(Point p) {
        this.courant=p;
    }
    public double getBestGain() {
        return this.teneur_par_robot;
    }
    public static double getAllBestGain() {
        return Robot.teneur_meilleur;
    }
    public static Point getbestlocation() {
        return Robot.G;
    }
    public Point getlocation_par_robot() {
        return this.locale;
    }
    public void changeteneur(double g) {
        this.teneur_par_robot=g;
    }
}
```

```

public void changeteneur_meilleur(double g) {
    Robot.teneur_meilleur=g;
}
public void changelocation_meilleur(Point p) {
    Robot.G=p;
}
public void changelocation(Point p) {
    this.locale=p;
}

public static void main(String[] args) {
    // TODO Auto-generated method stub
}

}

```

Annexe 6 : Classe Behaviour

```

package robots;

import Zone_miniere.AbstractProblem;
import Zone_miniere.Point;
import Zone_miniere.Sphere;

public class Behaviour {
    protected AbstractProblem gisement;
    public Behaviour(AbstractProblem g) {
        this.gisement=g;
    }
    public Behaviour() {
        // TODO Auto-generated constructor stub
    }
    void move(Robot robot) {
        System.out.println("enter behaviour");
        Point A=new Point();
        Point O=new Point(0.5,0.5);
        Point courant=robot.getPosition();
        double xp=courant.obtenir_abscisse();double
        yp=courant.obtenir_ordonne();
        double xa=A.obtenir_abscisse();double ya=A.obtenir_ordonne();
        double xo=O.obtenir_abscisse();double yo=O.obtenir_ordonne();
        double xoa=xa-xo;double yoa=ya-yo;
        Point cible=new Point(xp+xoa,yp+yoa);
        Point deplacement=courant.move(cible, 0.05);
        robot.setPosition(deplacement);
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}

```

```

        Point p=new Point(0.5,0.5); // position initiale pour le robot
        Robot robot=new Robot(p);
        Sphere pb=new Sphere();
        Behaviour explore=new Behaviour(pb);
        robot.setBehavior(explore); // Le robot possède le comportement
explore
        for (int i=0;i<20;i++){
            robot.walk();
            System.out.println(i+" :
"+robot.getPosition().obtenir_abscisse()+" ,
"+robot.getPosition().obtenir_ordonne());
        }
    }

}

```

Annexe 7 : Classe BasicMission

```

package robots;

import Zone_miniere.AbstractProblem;
import Zone_miniere.Point;
import Zone_miniere.Sphere;
import utilitaires.Printer;

public class BasicMission {
    protected AbstractProblem gisement;
    protected int nombre_robots;
    protected Robot [] robot_liste;
    protected Printer printer;
    protected String source="data/robots.txt";

    public BasicMission(AbstractProblem gise,int nb) {
        this.gisement=gise;this.nombre_robots=nb;
        this.robot_liste=new Robot[nb];
        robots_initialisation();

        printer=new Printer(source);

    }
    public void robots_initialisation() {
        for(int i=0;i<this.nombre_robots;i++) {
            Point p=new Point(0.5,0.5);
            Robot r=new Robot(p);
            r.setBehavior(new Behaviour(this.gisement));
            set(i,r);
        }
    }
    public void set(int index,Robot r) {

```

```
        robot_liste[index]=r;
    }
    public Robot get(int index) {
        return this.robot_liste[index];
    }

    public void run() {
        // 3轮调试, 100轮看图
        for(int i=0;i<3;i++) {
            System.out.println("epoch "+(i+1));
            System.out.println("la situation de teneur: ");
            collect();
            store();
            System.out.println("le teneur meilleur dans ce epoch cherche
par tous les robots est "+Robot.getAllBestGain());
            System.out.println();
            System.out.println("la situation de deplacement de point:
");
            walk();
        }
        System.out.println("le teneur meilleur dans tous les epochs cherche
par tous les robots est "+Robot.getAllBestGain());
        printer.close();
    }
    public void walk() {
        System.out.println("dans la fonction walk:");
        for(int i=0;i<this.nombre_robots;i++) {
            Robot robot_courant=robot_liste[i];
            System.out.println("robot "+i+": ");
            robot_courant.walk();
            System.out.println("le robot "+i+" deplace au point
"+robot_courant.getPosition().obtenir_abscisse()+" ,
"+robot_courant.getPosition().obtenir_ordonne());
            System.out.println();
        }
    }
    public void collect() {
        System.out.println("dans la fonction collect:");
        for(int i=0;i<this.nombre_robots;i++) {
            System.out.println("robot "+i+": ");
            Robot robot_courant=robot_liste[i];
            Point p_courant=robot_courant.getPosition();
            double gain=gisement.teneur(p_courant);
            System.out.println("teneur maintenant "+gain);
            if(gain>robot_courant.getBestGain()) {
                robot_courant.changeteneur(gain);
                robot_courant.changelocation(p_courant);
            }
            if (robot_courant.getBestGain()>Robot.getAllBestGain()) {

robot_courant.changeteneur_meilleur(robot_courant.getBestGain());
```

```

        robot_courant.changelocation_meilleur(p_courant);
        System.out.println("la location meilleure globale
maintenant est "+p_courant.obtenir_abscisse()+" ,"+p_courant.obtenir_ordonne());
    }
    System.out.println("le teneur meilleure cherche par
robot"+i+" est "+robot_courant.getBestGain());
    System.out.println();
}
public void store() {
    printer.println("-1");
    for(int i=0;i<this.nombre_robots;i++) {
        Robot robot_courant=robot_liste[i];

printer.println(robot_courant.getPosition().obtenir_abscisse()+"\t"+robot_couran
t.getPosition().obtenir_ordonne());
    }

}
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Sphere pb=new Sphere();
    BasicMission bm=new BasicMission(pb,20);
    bm.run();
    Viewer.display(pb);
}

}

```

Annexe 8 : Classe Follow

```

package robots;

import Zone_miniere.AbstractProblem;
import Zone_miniere.Point;
import Zone_miniere.Sphere;

public class Follow extends Behaviour{
    public Follow(AbstractProblem g) {
        super(g);
    }
    public Follow() {
        // TODO Auto-generated constructor stub
    }

    public void move(Robot robot) {
        Point courant=robot.getPosition();

```

```
        Point cible =Robot.getbestlocation();
        Point deplacement=courant.move(cible, 0.05);
        robot.setPosition(deplacement);
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

Annexe 9 : Classe LocalBest

```
package robots;

import Zone_miniere.Point;
import Zone_miniere.Sphere;
import Zone_miniere.AbstractProblem;

public class LocalBest extends Behaviour{
    public LocalBest(AbstractProblem g) {
        super(g);
    }
    public LocalBest() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public void move(Robot robot) {
        System.out.println("enter localbest");
        Point courant=robot.getPosition();
        Point deplacement=courant.move(robot.getlocation_par_robot(),
0.05);
        robot.setPosition(deplacement);
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

6.3. Introduction de politique de recherche

Annexe 10 : Classe AbstractPolicy

```
package politiques_recherche;
import java.util.Date;
import robots.Behaviour;

public abstract class AbstractPolicy {
    protected int nombre_phase;
    public AbstractPolicy(int n) {
        this.nombre_phase=n;
    }
    protected abstract int obtener_duree(int index);
    protected abstract Behaviour obtener_behaviour(int index);
    protected abstract void add(Behaviour bh,int inter);

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Annexe 12 : Classe SimplePolicy

```
package politiques_recherche;

import robots.Behaviour;
import robots.Follow;
import robots.LocalBest;

public class SimplePolicy extends AbstractPolicy{
    private Behaviour []behaviour_liste;
    private int []temps_iteration;
    private int index;
    public SimplePolicy(int n) {
        super(n);
        // TODO Auto-generated constructor stub
        this.behaviour_liste=new Behaviour[n];
        this.temps_iteration=new int[n];
        this.index=0;
    }
    @Override
    public void add(Behaviour bh,int iter) {
        behaviour_liste[index]=bh;
        temps_iteration[index]=iter;
        index+=1;
    }
    @Override
```

```

protected int obtener_duree(int index) {
    // TODO Auto-generated method stub
    return this.temps_iteration[index];
}
@Override
protected Behaviour obtenerBehaviour(int index) {
    // TODO Auto-generated method stub
    return this.behaviour_liste[index];
}
public static void main(String[] args) {
    // TODO Auto-generated method stub
    SimplePolicy policy=new SimplePolicy(3);
    policy.add(new Behaviour(),60); // phase 0 de durée 60 itérations
    policy.add(new LocalBest(),20); // phase 1
    policy.add(new Follow(),20); // phase 2
    System.out.println(" le nombre de phase est "+policy.nombre_phase);
    for(int i=0;i<3;i++) {
        System.out.println(" le policy maintenant est
"+policy.obtenirBehaviour(i));
        System.out.println(" la duree pour ce policy est
"+policy.obtenir_duree(i));
    }
}

}

```

Annexe 13 : Classe SmartMission :

```

package politiques_recherche;

import Zone_miniere.AbstractProblem;
import Zone_miniere.Eggholder;
import Zone_miniere.Sphere;
import robots.BasicMission;
import robots.Behaviour;
import robots.Follow;
import robots.LocalBest;
import robots.Robot;
import robots.Viewer;

public class SmartMission extends BasicMission {
    private SimplePolicy policy;
    private int iteration;
    public SmartMission(AbstractProblem gise, int nb,int n_policy) {
        super(gise, nb);
        // TODO Auto-generated constructor stub
        this.policy=new SimplePolicy(n_policy);
        this.iteration=0;
    }
}

```

```
@Override
public void run() {
    for(int i=0;i<this.policy.nombre_phase;i++) {
        System.out.println("strategie "+(i+1)+":");
        Behaviour behaviour=this.policy.obtenirBehaviour(i);
        this.iteration=this.policy.obtenir_duree(i);
        for(int j=0;j<nombre_robots;j++) {
            get(j).setBehavior(behaviour);;
        }

        for(int j=0;j<this.iteration;j++) {
            System.out.println("epoch "+(j));

            collect();
            store();
            walk();
        }
    }
    printer.close();
}

public static void main(String[] args) {
    // TODO Auto-generated method stub
    Eggholder pb=new Eggholder();
    SmartMission smartmission=new SmartMission(pb,60,3);
    smartmission.policy.add(new Behaviour(pb),50);
    smartmission.policy.add(new LocalBest(pb),30);
    smartmission.policy.add(new Follow(pb),30);
    smartmission.run();
    Viewer.display(pb);

}
}
```