Université de Bourgogne

Systèmes Intelligents Avancés TP

Auteurs :
Clément GHYS
Benjamin MILHET

Professeur :
M. ABDOU

clement_ghys@etu.u-bourgogne.fr
benjamin_milhet@etu.u-bourgogne.fr

2023 - 2024

Table des matières

1 Introduction **1**

2 TP1 - Découverte du framework jMetal **2**
2.1 Introduction . . . 2
2.2 Exercice 1 : Manipulation de jMetal . . . 2
2.3 Exercice 2 : Résolution d'un problème d'optimisation multi-objectif par une approche scalaire . . . 2
2.3.1 Classe Scalaire . . . 3
2.4 Exercice 3 : Résolution d'un problème d'optimisation multi-objectif par une approche non-scalaire 4
2.4.1 Classe Pareto . . . 4
2.4.2 Front de Pareto . . . 4
2.4.3 Tracé du front de Pareto . . . 5
2.5 Conclusion . . . 6

3 TP2 - Optimisation multi-objectifs **7**
3.1 Introduction . . . 7
3.2 Conception . . . 7
3.2.1 Les fonctions objectifs . . . 7
3.3 Sensor Deployment . . . 7
3.3.1 Le constructeur . . . 7
3.3.2 Lecture d'un fichier texte . . . 8
3.3.3 La fonction evaluate . . . 9
3.3.4 Tracé des cibles et des capteurs . . . 10
3.3.5 Front de Pareto . . . 13
3.3.6 Tracé du front de Pareto . . . 13
3.4 Conclusion . . . 14

4 Conclusion **15**

Introduction

Ce rapport présente les travaux pratiques réalisés dans le cadre du cours de Systèmes Intelligents Avancés. Dans un premier temps, nous allons nous familiariser avec le framework jMetal qui permet de manipuler des algorithmes d'optimisation et plus particulièrement des algorithmes génétiques, nous allons utiliser jMetal pour résoudre un problème d'optimisation multi-objectif par une approche scalaire et une approche non-scalaire. Dans un second temps, nous allons utiliser les connaissances acquises lors du premier TP pour un cas concret. L'objectif est d'optimiser le placement de capteurs autour de cible représentant des sources de pollution. Pour cela, nous devons utiliser une optimisation multi-objectif qui place au mieux les capteurs et qui ne place aucun capteurs sur des zones ou il n'y pas de cible de pollution.

Le code des différents exercices sont disponible sur notre dépôt : Github

TP1 - Découverte du framework jMetal

2.1 Introduction

Ce premier TP vise dans un premier temps à se familiariser avec le framework jMetal qui permet de manipuler des algorithmes d'optimisation et plus particulièrement des algorithmes génétiques. Dans un second temps, nous allons utiliser jMetal pour résoudre un problème d'optimisation multi-objectif par une approche scalaire. Enfin nous allons utiliser une approche non-scalaire en utilisant la méthode NSGA-II pour calculer un ensemble de solutions optimales et ainsi tracer le front de Pareto.

2.2 Exercice 1 : Manipulation de jMetal

Cette partie consiste simplement à télécharcger jMetal, le mettre en place dans un IDE et de parcourir les packages. Nous avons pu en parcourant les packages, voir les différents algorithmes d'optimisation implémentés dans jMetal comme par exemple NSGA-II, SPEA2, MOCell, etc.

2.3 Exercice 2 : Résolution d'un problème d'optimisation multi-objectif par une approche scalaire

Cet exercice consiste à résoudre un problème d'optimisation multi-objectif par une approche scalaire défini par les fonctions suivantes :

$$\begin{cases} f_1(x) = x_1 \\ f_2(x) = g(x) \times (1 - \sqrt{\frac{x_1}{g(x)}}) \\ g(x) = 1 + 9 \times \frac{\sum_{i=2}^{n} x_i}{n-1} \end{cases} \tag{2.1}$$

2.3.1 Classe Scalaire

Création d'une classe qui hérite de la classe Problem de jMetal et qui permet de définir les fonctions objectifs et les contraintes du problème.

Le constructeur de la classe Scalaire prend en paramètre le type de solution (Real ou Binary) et le nombre de variables du problème. On définit ensuite le nombre de fonctions objectifs, le nombre de contraintes et le nom du problème. On définira par la suite le nombre de variables à 30 comme demandé dans l'énoncé.

Listing 2.1 – Exemple de code Java

```
public Scalaire(String solutionType, int nombres) {
        super.numberOfObjectives_ = 1;
        super.numberOfConstraints_ = 0;
        super.numberOfVariables_ = nombres;
        super.problemName_ = "jmetal.Scalaire";

        if (solutionType.compareTo("Real") == 0) {
            this.solutionType_ = new RealSolutionType(this);
        } else {
            System.out.println("Error");
            System.exit(-1);
        }

        super.lowerLimit_ = new double[numberOfVariables_];
        super.upperLimit_ = new double[numberOfVariables_];

        for (int i = 0 ; i < numberOfVariables_ ; i++) {
            super.lowerLimit_[i] = 0.0;
            super.upperLimit_[i] = 1.0;
        }
    }
```

On définit ensuite la fonction evaluate qui permet de calculer les valeurs des fonctions objectifs et des contraintes pour une solution donnée.

Listing 2.2 – constructeur de la classe Scalaire

```
    @Override
    public void evaluate(Solution solution) throws JMException {
        Variable[] decisionVariables = solution.getDecisionVariables();

        double [] x = new double[numberOfVariables_];

        for (int i = 1 ; i < numberOfVariables_ ; i++) {
            x[i] = decisionVariables[i].getValue();
        }

        double f1 = x[0], f2 = 0.0, g = 0.0;

        for (int i = 1 ; i < numberOfVariables_ ; i++) {
            g += x[i];
        }

        g = 1 + 9 * g / (numberOfVariables_ - 1);
        f2 = g * Math.floor(1 - Math.sqrt(f1/g));

        double alpha = 1.0, beta = 1.0;
        double F = alpha * f1 + beta * f2;

        solution.setObjective(0, F);
    }
```

Une fois la classe Scalaire créée et les classes gGa et gGaMain configurées, on peut lancer l'algorithme génétique et observer les résultats.

α	β	FUN
1	1	1.0000835073822625
1	0.5	0.500044811209363
0.5	0.5	0.5000378049561031
0.5	0.2	0.20001095296083943
1	0.5	0.20001055140647905
1	0.0	0.0

TABLE 2.1 – Évolution de la valeur de la fonction objectif suivant les valeurs de α et β

Nous avons pu observer en faisant varier α et β que ces deux valeurs permettent de pondérer la fonction objectif et ainsi de trouver des solutions différentes. On en déduit que la fonction 2 à un plus gros impact sur notre fonction objectif lorsque nous faisons varier β, comparer à α ou son impact est très faible.

2.4 Exercice 3 : Résolution d'un problème d'optimisation multi-objectif par une approche non-scalaire

Pour cet exercice, nous allons utiliser la méthode NSGA-II pour calculer un ensemble de solutions optimales et ainsi tracer le front de Pareto.

2.4.1 Classe Pareto

Nous allons simplement créer une classe Pareto et reprendre le code de la classe Scalaire en modifiant la fonction evaluate et le constructeur pour utiliser deux fonctions objectifs.

Listing 2.3 – constructeur de la classe Pareto

```
public Pareto(String solutionType, int nombres) {
    super.numberOfObjectives_ = 2;

    // ... le reste du constructeur est identique a celui de la classe Scalaire
}
public void evaluate(Solution solution) throws JMException {
    // ... Le debut de la fonction evaluate est identique a celui de la classe Scalaire

    g = 1 + 9 * g / (numberOfVariables_ - 1);
    f2 = g * (1 - Math.sqrt(f1/g));

    // ce qui change par rapport a la classe Scalaire
    solution.setObjective(0, f1);
    solution.setObjective(1, f2);
}
```

L'approche non-scalaire permet de trouver un ensemble de solutions optimales et ainsi de tracer le front de Pareto en utilisant 2 fonctions objectifs contrairement à l'approche scalaire qui ne permet de se ramener qu'à une seule fonction mono-objectif.

2.4.2 Front de Pareto

Le front de pareto nous permet de visualiser les solutions optimales et de choisir la solution qui nous convient le mieux en fonction de nos critères. Une fois que l'on a lancé l'algorithme NSGA-II et que l'on a obtenu les valeurs optimales dans le fichier FUN on peut tracer le front de Pareto.

2.4.3 Tracé du front de Pareto

Pour tracer le front de Pareto, nous créer un script python qui permet de lire le fichier FUN et de tracer le front de Pareto en utilisant la librairie matplotlib.

Listing 2.4 – Script python pour tracer le front de Pareto

```
import matplotlib.pyplot as plt

with open('FUN') as file:
    data = file.readlines()

f1_values = []
f2_values = []
for line in data:
    values = line.strip().split()
    f1_values.append(float(values[0]))
    f2_values.append(float(values[1]))

# Tracer le front de Pareto
plt.figure(figsize=(8, 6))
plt.scatter(f1_values, f2_values, color='blue', marker='o', label='Front de Pareto')
plt.title('Front de Pareto')
plt.xlabel('f1')
plt.ylabel('f2')
plt.legend()
plt.grid(True)
plt.show()
```

On obtient le front de Pareto suivant :

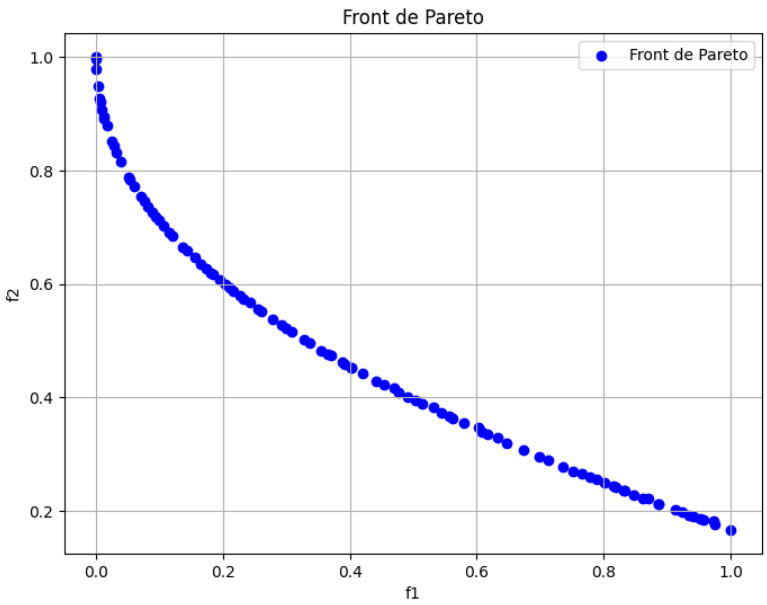

FIGURE 2.1 – Front de Pareto

On peut voir sur la figure 2.1 que le front de Pareto est convexe et que l'on peut donc trouver une solution optimale en fonction de nos critères.

Le front est assez régulier et ne présente pas de points aberrants. On peut donc en déduire que l'algorithme NSGA-II a bien convergé vers une solution optimale.

2.5 Conclusion

Ce TP nous a permis de nous familiariser avec le logiciel jMetal et de manipuler des algorithmes d'optimisation et plus particulièrement des algorithmes génétiques. Nous avons pu voir que l'approche scalaire permet de se ramener à un problème mono-objectif et que l'approche non-scalaire permet de trouver un ensemble de solutions optimales et ainsi de tracer le front de Pareto.

TP2 - Optimisation multi-objectifs

3.1 Introduction

Ce second TP nous demande d'utiliser les connaissances acquises lors du premier TP pour un cas concret. L'objectif est d'optimiser le placement de capteurs autour de cible représentant des sources de pollution. Pour cela, nous devons utiliser une optimisation multi-objectif qui place au mieux les capteurs et qui ne place aucun capteurs sur des zones ou il n'y pas de cible de pollution.

3.2 Conception

3.2.1 Les fonctions objectifs

La première fonction objective consiste à maximiser le nombre de cible ouvertes. Cela signifie que si c'est possible, l'ensemble des cibles doive être dans une zone d'un ou plusieurs capteurs. Pour cela, nous comptons le nombre de cible couverte par un capteur mais aussi le nombre de capteur qui ne contient aucune cible et nous retournons la différence entre ces deux valeurs pour cette première fonction.

La seconde fonction objective consiste à maximiser le nombre minimum de capteurs autour des cibles. Cette fonction a pour rôle de répartir de manière uniforme nos capteurs autour des différentes cibles de pollution. Pour celle-ci, nous comptons le nombre minimum de capteurs autour d'une cible ce qui permet un réajustement automatique et équitable de nos capteurs. Nous retournons pour notre deuxième fonction le nombre minimum de capteurs pour une cible.

3.3 Sensor Deployment

3.3.1 Le constructeur

Dans le constructeur de notre classe SensorDeployement, nous reprenons le code réaliser lors du premier TP en l'adaptant à notre problème. Dans le cas de ce TP, nous avons deux fonctions objectifs. Une autre différence par rapport au premier TP est la valeur de la limite supérieur de notre variable de décision qui n'est plus à 1 mais qui est égal à la dimension de la zone d'étude correspond à un carré de 100m par 100m. De plus, nous appelons une fonction que nous avons coder pour récupérer les coordonnées des cibles dans un fichier texte.

Listing 3.1 – Constructeur de notre classe SensorDeployment

```
public SensorDeployment(String solutionType, int nombreCapteurs, int rayonCapteurs, int
    dimmension) {
    this.nombreCapteurs = nombreCapteurs * 2; // 2 variables par capteur (x, y)
    super.problemName_ = "SensorDeployment"; // Nom du probleme
    super.numberOfObjectives_ = 2; // Nombre de fonctions objectifs
    super.numberOfConstraints_ = 0; // Nombre de contraintes
    super.numberOfVariables_ = this.nombreCapteurs; // Nombre de variables de decision
    this.dimmension = dimmension; // Dimension de l'espace de recherche

    // Type de solution
    if (solutionType.compareTo("Real") == 0) {
        this.solutionType_ = new RealSolutionType(this);
    } else {
        System.out.println("Error");
```

```
            System.exit(-1);
        }

        super.lowerLimit_ = new double[numberOfVariables_]; // Limite inferieure des variables de
            decision
        super.upperLimit_ = new double[numberOfVariables_]; // Limite superieure des variables de
            decision

        for (int i = 0 ; i < numberOfVariables_ ; i++) { // Initialiser les limites des variables de
            decision
            super.lowerLimit_[i] = 0.0; // Limite inferieure a 0
            super.upperLimit_[i] = this.dimmension; // Limite superieure a la dimension de l'espace de
                recherche
        }

        this.rayonCapteurs = rayonCapteurs; // Initialiser le rayon des capteurs

        this.lireFichierEtInitialiser("src/jmetal/cible.txt"); // Lire le fichier contenant les
            coordonnees des cibles

        for (int i = 0; i < this.coordonneesCibles.length; i++) { // Parcourir les cibles
            System.out.println(this.coordonneesCibles[i][0] + " " + this.coordonneesCibles[i][1]); //
                Afficher les coordonnees de la cible
        }
    }
```

3.3.2 Lecture d'un fichier texte

La fonction ci-dessous nous permet de lire les coordonnées d'un fichier texte contenant les coordonnées de nos cibles sous ce format : x, y. Notre fonction parcours chaque ligne de notre fichier et récupère de manière distinct les deux valeurs séparé par une virgule.

Listing 3.2 – Fonction de lecture d'un fichier texte en Java

```
    public void lireFichierEtInitialiser(String cheminFichier) {
        ArrayList<int[]> listeCible = new ArrayList<>(); // Liste contenant les coordonnees des cibles

        try {
            File fichier = new File(cheminFichier); // Creer un fichier a partir du chemin
            Scanner scanner = new Scanner(fichier); // Creer un scanner a partir du fichier
            while (scanner.hasNextLine()) { // Parcourir les lignes du fichier
                String[] parts = scanner.nextLine().split(","); // Recuperer les coordonnees de la
                    cible (x, y) separees par une virgule
                int x = Integer.parseInt(parts[0]); // Recuperer la coordonnee x
                int y = Integer.parseInt(parts[1]); // Recuperer la coordonnee y
                listeCible.add(new int[]{x, y}); // Ajouter les coordonnees de la cible a la liste
            }
            scanner.close(); // Fermer le scanner

            this.coordonneesCibles = new int[listeCible.size()][2]; // Initialiser le tableau
                contenant les coordonnees des cibles
            for (int i = 0; i < listeCible.size(); i++) { // Parcourir la liste contenant les
                coordonnees des cibles
                this.coordonneesCibles[i] = listeCible.get(i); // Ajouter les coordonnees de la cible
                    au tableau
            }
        } catch (FileNotFoundException e) {
            System.err.println("Fichier non trouve: " + e.getMessage());
        }
    }
```

3.3.3 La fonction evaluate

L'objectif de cette fonction est d'appliquer au mieux nos deux fonctions objectifs en plaçant nos capteurs en fonction de celle-ci. Pour cela, on compte le nombre de cible couverte par un capteur mais aussi le nombre de capteur qui ne contient aucune cible. Cela nous permet de suivre la première fonction objectif qui consiste à maximiser le nombre de cible couverte. Pour la seconde fonction objectif, nous cherchons à maximiser le nombre minimum de capteurs autour des cibles, pour cela, l'objectif est d'avoir une répartition équitable de nos capteurs autour de nos cibles. Pour cette deuxième fonction objectif, nous comptons le nombre minimum de capteurs autour d'une cible ce qui permet un réajustement automatique et équitable.

Listing 3.3 – Fonction evaluate de notre classe SensorDeployment

```
@Override
public void evaluate(Solution solution) throws JMException {
    Variable[] decisionVariables = solution.getDecisionVariables(); // Recuperer les variables de
        decision

    int[] listNombreCapteurParCible = new int[this.coordonneesCibles.length]; // Tableau contenant
        le nombre de capteurs autour de chaque cible
    Arrays.fill(listNombreCapteurParCible, 0); // Initialiser les valeurs du tableau a 0
    int nombreCapteurSansCible = 0; // Nombre de capteurs qui ne couvrent aucune cible

    for (int i = 0; i < this.nombreCapteurs; i += 2) { // Parcourir les capteurs 2 par 2 (x, y)
        double x = decisionVariables[i].getValue(); // Recuperer la valeur de la variable de
            decision
        double y = decisionVariables[i + 1].getValue(); // Recuperer la valeur de la variable de
            decision
        boolean couvreCible = false; // Indique si le capteur couvre une cible

        for (int j = 0; j < this.coordonneesCibles.length; j++) { // Parcourir les cibles pour
            verifier si le capteur couvre une cible
            double distance = Math.sqrt(Math.pow(x - this.coordonneesCibles[j][0], 2) + Math.pow(y
                - this.coordonneesCibles[j][1], 2)); // Calculer la distance entre le capteur et
                la cible
            if (distance <= this.rayonCapteurs) { // Si la distance est inferieure ou egale au
                rayon du capteur, la cible est couverte
                listNombreCapteurParCible[j]++; // Incrementer le nombre de capteurs autour de la
                    cible
                couvreCible = true; // Indiquer que le capteur couvre une cible
            }
        }

        if (!couvreCible) { // Si le capteur ne couvre aucune cible, incrementer le nombre de
            capteurs sans cible
            nombreCapteurSansCible++; // Incrementer le nombre de capteurs sans cible
        }
    }

    int nombreCiblesCouvertes = 0; // Nombre de cibles couvertes
    int minCapteursAutourCible = Integer.MAX_VALUE; // Nombre minimum de capteurs autour d'une
        cible
    for (int couverture : listNombreCapteurParCible) { // Parcourir le tableau contenant le nombre
        de capteurs autour de chaque cible
        if (couverture > 0) { // Si la cible est couverte
            nombreCiblesCouvertes++; // Incrementer le nombre de cibles couvertes
            if (couverture < minCapteursAutourCible) { // Si le nombre de capteurs autour de la
                cible est inferieur au minimum
                minCapteursAutourCible = couverture; // Mettre a jour le minimum
            }
        }
    }
    if (nombreCiblesCouvertes == 0) { // Si aucune cible n'est couverte
        minCapteursAutourCible = 0; // Definir le minimum a 0
    }
    // Mise a jour des objectifs avec une penalite pour les capteurs sans cible
```

```
    solution.setObjective(0, -(nombreCiblesCouvertes - nombreCapteurSansCible)); // Maximiser le
        nombre de cibles couvertes en tenant compte de la penalite
    solution.setObjective(1, -minCapteursAutourCible); // Maximiser le nombre minimum de capteurs
        autour des cibles
}
```

3.3.4 Tracé des cibles et des capteurs

Pour représenter nos cibles et capteurs au sein d'un affichage graphiques, nous avons décider de réaliser un script Python à l'aide de la librairie Matplotlib. Nous affichons un graphique de taille 100 par 100 représentant la taille de la zone d'étude avec les points bleus représentant les cibles et les croix rouges les capteurs. Les cercles rouges autour des croix sont de taille 10 et représentent les zones de nos capteurs.

Listing 3.4 – Script python pour tracer les cibles et les capteurs

```
import matplotlib.pyplot as plt # Importation de la bibliotheque matplotlib.pyplot

cibles = [] # Liste des cibles
with open("cibles.txt", "r") as file: # Ouverture du fichier cibles.txt en mode lecture
    for line in file: # Pour chaque ligne du fichier
        x, y = map(int, line.strip().split(',')) # On recupere les coordonnees de la cible
        cibles.append((x, y)) # On ajoute la cible a la liste des cibles

capteurs = [10.744964702168431, 3.356672123620143, 83.92700773184632, 97.46078570327255,
    19.792995491301923, 99.96635269754944, 42.12528145697245, 47.09181147646758 ] # Liste des
    capteurs

plt.figure(figsize=(8, 8)) # Creation d'une figure de taille 8x8 pouces

for x, y in cibles: # Pour chaque cible
    plt.plot(x, y, 'bo') # On trace un point bleu pour la cible

for i in range(0, len(capteurs), 2): # Pour chaque capteur
    plt.plot(capteurs[i], capteurs[i+1], 'rx', markersize=10) # On trace une croix rouge pour le
        capteur
    circle = plt.Circle((capteurs[i], capteurs[i+1]), 10, color='red', fill=False) # On trace un
        cercle rouge de rayon 10 autour du capteur
    plt.gca().add_artist(circle) # On ajoute le cercle a la figure

plt.xlim(0, 100) # On limite les abscisses entre 0 et 100
plt.ylim(0, 100) # On limite les ordonnees entre 0 et 100
plt.xlabel('X') # On nomme l'axe des abscisses
plt.ylabel('Y') # On nomme l'axe des ordonnees
plt.title('Cibles et Capteurs') # On donne un titre a la figure

plt.show() # On affiche la figure
```

Lors des premiers essais, nous avons fixer un total de 18 capteurs pour nos 6 cibles fixes. On remarque une répartition presque optimale de nos capteurs avec au moins 3 capteurs par cible et des capteurs qui peuvent avoir 2 cibles si possibles, comme nous l'indique nos résultats ci-dessous :

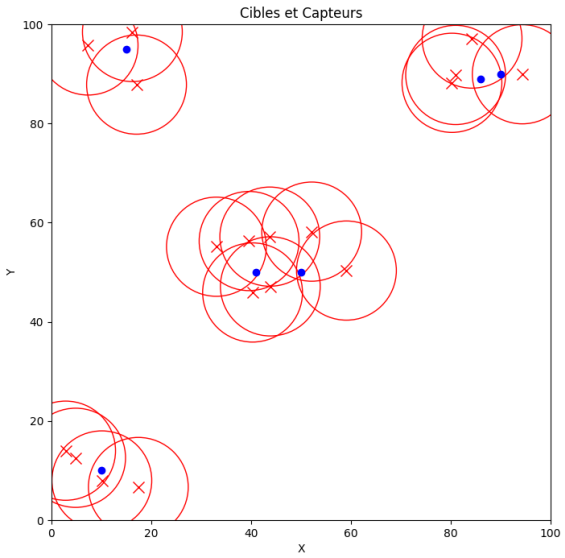

FIGURE 3.1 – Affichage des cibles et des capteurs pour 18 capteurs et 6 cibles

Ensuite, nous avons fixer un total de 6 capteurs pour nos 6 cibles fixes, soit un capteur par cible. On remarque une répartition optimale un avec un capteur par cible et certain qui ont même deux cibles, comme nous l'indique nos résultats ci-dessous :

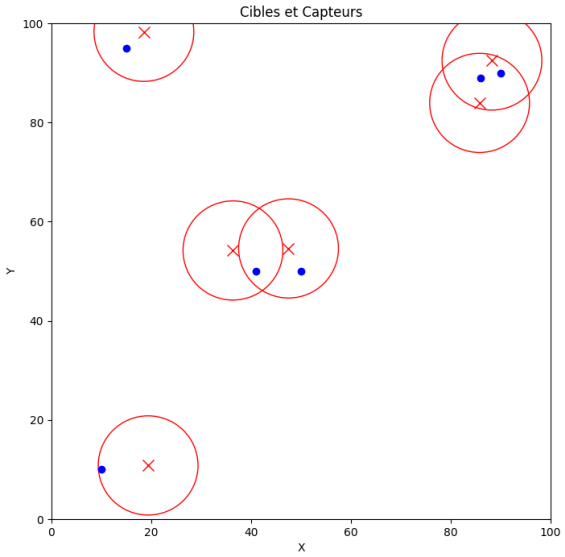

FIGURE 3.2 – Affichage des cibles et des capteurs pour 6 capteurs et 6 cibles

Notre dernier test consiste à utiliser seulement 4 capteurs pour nos 6 cibles parce que elles sont regroupé dans 4 zones distincts dans notre zone d'étude. Pour ce cas, nous rencontrons un peu plus de difficultés à placé correctement nos capteurs avec nombreux cas où nos cibles ne sont pas couvertes pas des capteurs.

Dans ce cas ci, nous n'avons pas rencontré de problèmes et nos 4 capteurs couvrent bien nos 6 cibles :

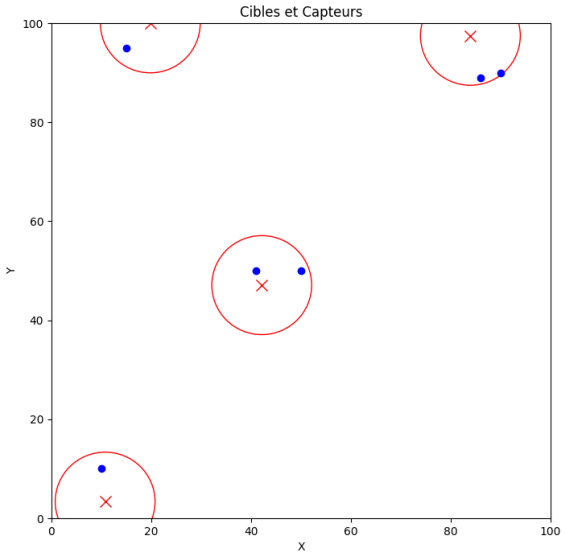

FIGURE 3.3 – Affichage des cibles et des capteurs pour 4 capteurs et 6 cibles

Cependant, dans certain cas, nos capteurs se place de manière anormal :

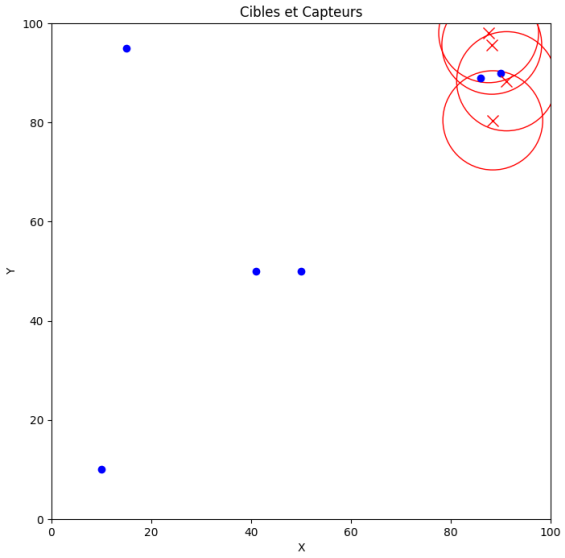

FIGURE 3.4 – Affichage des cibles et des capteurs pour 4 capteurs et 6 cibles avec erreurs

3.3.5 Front de Pareto

Le front de pareto nous permet de visualiser les solutions optimales et de choisir la solution qui nous convient le mieux en fonction de nos critères.

Une fois que l'on a lancé l'algorithme NSGA-II et que l'on a obtenu les valeurs optimales dans le fichier FUN on peut tracer le front de Pareto.

3.3.6 Tracé du front de Pareto

Pour tracer le front de Pareto, nous créer un script python qui permet de lire le fichier FUN et de tracer le front de Pareto en utilisant la librairie Matplotlib.

Listing 3.5 – Script python pour tracer le front de Pareto du TP2

```
import matplotlib.pyplot as plt

with open('FUN') as file:
    data = file.readlines()

f1_values = []
f2_values = []
for line in data:
    values = line.strip().split()
    f1_values.append(float(values[0]))
    f2_values.append(float(values[1]))

# Tracer le front de Pareto
plt.figure(figsize=(8, 6))
plt.scatter(f1_values, f2_values, color='blue', marker='o', label='Front de Pareto')
plt.title('Front de Pareto')
plt.xlabel('f1')
plt.ylabel('f2')
plt.legend()
plt.grid(True)
plt.show()
```

On obtient le front de Pareto suivant :

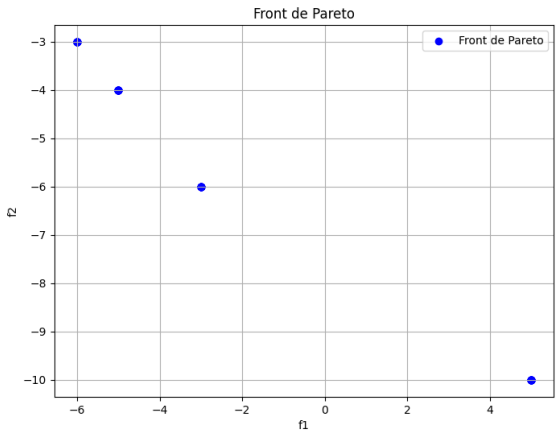

FIGURE 3.5 – Front de Pareto

Sur notre front de Pareto, nous pouvons observer seulement 4 solutions optimales en fonctions de nos 2 fonctions objectifs. Suivant si nous souhaitons privilégier la première ou la seconde fonctions objectifs, nous devons faire un choix de ce qui est le plus important ou le moins pire pour notre cas.

3.4 Conclusion

Ce second TP nous a permis d'appliquer l'aspect théorique des enseignements reçu et du premier TP dans un projet plus concret. Ce second TP nous montre aussi l'intérêt d'avoir plusieurs objectifs afin d'être plus précis dans notre recherche d'optimisation de placement de nos capteurs. Avec un seul objectif, on aurait du faire plus de sacrifice sur le placement des capteurs ou leur répartition sur les différentes cibles.

Conclusion

Lors de ces deux TP nous avons pu nous familiariser avec le framework jMetal et les algorithmes génétiques. Nous avons pu voir que l'approche scalaire permet de se ramener à un problème mono-objectif et que l'approche non-scalaire permet de trouver un ensemble de solutions optimales et ainsi de tracer le front de Pareto. Nous avons également pu mettre en pratique les connaissances acquises pour résoudre un problème d'optimisation multi-objectif pour un cas concret. L'objectif était d'optimiser le placement de capteurs autour de cible représentant des sources de pollution. Pour cela, nous avons utilisé une optimisation multi-objectif qui place au mieux les capteurs et qui ne place aucun capteurs sur des zones ou il n'y pas de cible de pollution.

Ces deux TP nous on permis de mettre en pratique les connaissances acquises lors du cours de Systèmes Intelligents Avancés et de nous familiariser avec le framework jMetal afin de les mettre en pratique dans un cas concret.