République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Benyoucef BENKHEDDA -Alger1-

Faculté des Sciences

Département Informatique

Rapport

Projet Méthodes Bio Inspirées

TP numéro 2

Algorithme génétique

En utilisant un langage de programmation Java

Réalisé par

AOUNE Ramzi

GOUIZI Mounia

2023/2024

I. Introduction :

Les algorithmes génétiques offrent une méthode d'optimisation basée sur le principe de l'évolution naturelle. Notre programme se déploie en utilisant un algorithme génétique avec pour objectif la maximisation d'une fonction spécifiée dans la plage [1 , 30].

Cette approche heuristique promet une exploration robuste et efficiente de l'espace des solutions, s'inspirants des mécanismes évolutifs pour trouver des solutions optimales.

En simulant des processus tels que la sélection naturelle, la recombinaison génétique, et la mutation, notre algorithme vise à produire des solutions de haute qualité dans un domaine donné.

Cette approche évolutive présente l'avantage d'explorer d'une manière adaptative l'espace de recherche, offrant ainsi une méthode puissante pour résoudre des problèmes d'optimisations complexes.

II. Structure du programme :

Le programme est structuré en plusieurs classes pour promouvoir la modularité et la lisibilité du code

II.1. *Génétique (La classe principale) :*

C'est la classe principale du projet.

```java
package Contexte;

import java.util.Arrays;

public class Génétique {
static Scanner scanner = new Scanner(System.in);    //Scanner declaration


    public static void main(String[] args) {

        int TaillePopulation = 4;

        System.out.print("Generations : ");
        int MaximumGeneration    =  scanner.nextInt();          //int
        System.out.print("Pc : ");
        double Pc   =  scanner.nextFloat();          //float
        System.out.print("Pm : ");
        double Pm =  scanner.nextDouble();          //double


//      int MaxGen = 30;
//      double Pc = 0.9;
//      double Pm = 0.01;

        int NombreCroisementFinale = 0;
        int NombreMutationFinale = 0;
        boolean[][] PopulationInitiale = {  { true, false, false, true, false },
                                            { false, false, true, true, false },
                                            { false, true, false, true, true },
                                            { true, true, false, true, true }   };

        IndividuClass[] PremierePopulation = initPop(PopulationInitiale);



        double[] TableDeStatistiques = statistique(PremierePopulation);
        double Minimum = TableDeStatistiques[0];
        double Maximum = TableDeStatistiques[1];
```

```
43          double SommeIndividus = TableDeStatistiques[2];
44          double Moyenne = TableDeStatistiques[3];
45          int Generation = 0;
46
47
48          System.out.println(Minimum);
49          System.out.println(Maximum);
50          System.out.println(SommeIndividus);
51          System.out.println(Moyenne);
52
53          while (Generation < MaximumGeneration) {
54              Generation++;
55
56              Génération Gen = generate(PremierePopulation, TaillePopulation, SommeIndividus , Pc
57              IndividuClass[] nouvellePopulation = Gen.nouvellePopulation;
58
59
60              int nombreDuCroisementPourPopulationInitiale = Gen.nombreDuCroisementPourPopulationI
61              int nombreDuMutationPourPopulationInitiale = Gen.nombreDuMutationPourPopulationIniti
62
63              double[] results = statistique(PremierePopulation);
64              Minimum = results[0];
65              Maximum = results[1];
66              SommeIndividus = results[2];
67              Moyenne = results[3];
68
69              NombreCroisementFinale += nombreDuCroisementPourPopulationInitiale;
70              NombreMutationFinale += nombreDuMutationPourPopulationInitiale;
71
72              System.out.println("Generation" + Gen);
73
74              for (IndividuClass ind : nouvellePopulation) {
75                  System.out.println("Genome: " + getGenomeString(ind.getChromosome()) + ", x: " +
76                          + ind.getFitnessFunc());
77              }
78  //           // Display average fitness
79              System.out.println("\nAverage Fitness------------------------------:" + Moyenne);
80  //
81  //           // Display maximum fitness
```

```
81  //           // Display maximum fitness
82              System.out.println("Maximum Fitness-------------------------------:" + Maximum);
83
84  //           // Display number of crossovers and mutations
85              System.out.println("Number of Crossovers:" + nombreDuCroisementPourPopulationInitial
86              System.out.println("Number of Mutations:" + nombreDuMutationPourPopulationInitiale);
87
88              PremierePopulation = nouvellePopulation;
89
90          }
91          System.out.println("Number of Crossovers total :" + NombreCroisementFinale);
92          System.out.println("Number of mutations  total :" + NombreMutationFinale);
93
94      }
```

II.2. *IndividuClass :*

La classe **IndividuClasss** représente un individu avec ses attributs tels que le chromosome, la valeur x, la fitness, et les liens vers ses parents dans le processus évolutif.

```java
package Contexte;

public class IndividuClass {
    boolean[] Chromosome;
    int X;
    double FitnessFunc;
    double Correction;
    IndividuClass Parent1;
    IndividuClass Parent2;

    IndividuClass(boolean[] Chromosome, int X, double FitnessFunc, double Correction, IndividuClass Parent1, IndividuClass Parent2) {
        this.Chromosome = Chromosome;
        this.X = X;
        this.FitnessFunc = FitnessFunc;
        this.Correction=Correction;
        this.Parent1 = Parent1;
        this.Parent2 = Parent2;

    }
```

II.3. *Croisement (La classe qui calcule et affiche l'opération de croisement) :*

La classe **Croisement** permet de calculer et afficher le résultat du croisement, contenant les deux nouveaux chromosomes résultants et le nombre de croisements effectués.

```java
package Contexte;

public class Croisement {
    public static final int Crosses = 0;
    public boolean[] Enfant1;
    public boolean[] Enfant2;

    public int croisement;

    public Croisement(boolean[] Enfant1, boolean[] Enfant2, int croisement) {
        this.Enfant1 = Enfant1;
        this.Enfant2 = Enfant2;

        this.croisement = croisement;
    }
}
```

II.4. *Mutation (La classe qui calcule et affiche l'opération de mutation) :*

La classe **Mutation** permet de calculer et afficher le résultat de la mutation, contenant le nouveau chromosome résultant et le nombre de mutations effectuées.

```java
package Contexte;

public class Mutation {
    public boolean[] Enfant;
    public int nombreMutation;

    Mutation(boolean[] Enfant, int nombreMutation) {
        // TODO Auto-generated constructor stub

        this.Enfant = Enfant;
        this.nombreMutation = nombreMutation;

    }

}
```

II.5. *Génération :*

La classe **Génération** affiche le résultat d'une génération, contenant la nouvelle population à définir, le nombre total de croisements et de mutations effectués après l'application des opérations de croisement et mutation

```java
package Contexte;

public class Génération {
    public IndividuClass[] nouvellePopulation;
    public int nombreDuCroisementPourPopulationInitiale;
    public int nombreDuMutationPourPopulationInitiale;

    public Génération(IndividuClass[] nouvellePopulation, int nombreDuCroisementPourPopulationInitiale, int nombreDuMutationPourPopulationInitiale) {
        this.nouvellePopulation = nouvellePopulation;
        this.nombreDuCroisementPourPopulationInitiale = nombreDuCroisementPourPopulationInitiale;
        this.nombreDuMutationPourPopulationInitiale = nombreDuMutationPourPopulationInitiale;
    }
}
```

III. Structure du programme :

III.1. *Définir la population initiale :*

La fonction **initPop** génère la population initiale en attribuant des caractéristiques génétiques de départ. Elle établit ainsi le point de départ pour l'algorithme génétique, définissant les individus initiaux avec des gènes qui seront soumis à des opérations évolutives telles que la reproduction, la recombinaison et la mutation. Cette étape est cruciale pour l'exploration subséquente de l'espace des solutions, formant la base sur laquelle l'algorithme évoluera pour trouver des solutions optimales.

```java
static IndividuClass[] initPop(boolean[][] PopulationInitiale) {
    IndividuClass[] Population = new IndividuClass[4];
    boolean[] Chromosome ;
    int X = 0;
    double Correction = 0;
    IndividuClass Parent1 = null;
    IndividuClass Parent2 = null;
    for (int i = 0; i < PopulationInitiale.length; i++) {
        Chromosome = PopulationInitiale[i];

        X = decode(Chromosome);
        float FitnessFunc = f(X);
        IndividuClass individu = new IndividuClass(Chromosome, X, FitnessFunc, Correction, Parent1, Parent2);
        Population[i] = individu;

    }
    return Population;
}
```

III.2. *Phase de Codage et de Décodage :*

Les fonctions **getGenomeString** et **decode** s'occupent respectivement de la représentation et de la traduction des chromosomes. La première fonction génère une chaîne représentant les caractéristiques génétiques d'un individu, tandis que la seconde traduit cette représentation en une forme interprétable, facilitant l'analyse des solutions générées par l'algorithme génétique. Ces deux fonctions sont essentielles pour assurer la cohérence entre la représentation génétique des individus et leur interprétation dans le contexte de l'optimisation de la fonction spécifiée.

```java
static String getGenomeString(boolean[] bs) {
    StringBuilder result = new StringBuilder();
    for (boolean bit : bs) {
        result.append(bit == true ? 'I' : 'O');
    }
    return result.toString();
}

// decode
static int decode(boolean[] Chromosome) {


    int S = 0;
    int D = 1;


        for (int i = Chromosome.length - 1; i >= 0; i--) {
            if (Chromosome[i] == true ) {

                S += D;
            }
            D *= 2;
        }

    return S;
}
```

III.3. *Déclaration de la fonction de fitness (FitnessFunc) :*

La fonction **f** évalue la fitness d'un individu en fonction de sa valeur x.

```
145     static float f(int X) {
146
147         return -X * X + 4 * X;
148
149     }
```

III.4. *L'opération de croisement :*

La fonction **crossover** injecte de la diversité génétique en combinant les chromosomes de deux individus. Cette opération de croisement favorise l'échange et la recombinaison de traits génétiques entre parents, contribuant ainsi à la création de progénitures possédant une variété de caractéristiques héritées. L'objectif principal de la fonction crossover est de promouvoir la diversification de la population, ce qui renforce la capacité de l'algorithme génétique à explorer efficacement l'espace des solutions pour atteindre des résultats optimaux.

```
152     public static Croisement crossover(boolean[] Parent1, boolean[] Parent2, double Pc) {
153
154         int Crosses = 0;
155
156
157
158         Random aléatoire = new Random();
159
160         boolean[] NouveauParent1 = Parent1;
161         boolean[] NouveauParent2 = Parent2;
162
163
164         for (int i = 0; i < 5; i++) {
165
166             if (aléatoire.nextDouble() <= Pc) {
167                 boolean temp = NouveauParent1[i];
168                 NouveauParent1[i] = NouveauParent2[i];
169                 NouveauParent2[i] = temp;
170                 Crosses++;
171             }
172         }
173
174         boolean[] Enfant1, Enfant2;
175
176         if (decode(NouveauParent1) > 0 && decode(NouveauParent2) > 0 && decode(NouveauParent1) < 31 && decode(NouveauParent2) < 31) {
177             Enfant1 = NouveauParent1;
178             Enfant2 = NouveauParent2;
179         } else {
180             // no crosses done
181             Enfant1 = Parent1;
182             Enfant2 = Parent2;
183             Crosses--;
184         }
185
186         return new Croisement(Enfant1, Enfant2, Crosses);
187     }
```

III.5. *L'opération de mutation :*

La fonction **mutate** apporte de la variabilité en introduisant des mutations dans les chromosomes. Cette étape cruciale simule des changements aléatoires au niveau génétique, altérant certains traits des individus. Ces mutations peuvent introduire de nouvelles caractéristiques et favoriser l'exploration de l'espace des solutions, renforçant ainsi la capacité de l'algorithme génétique à trouver des solutions plus diversifiées et potentiellement plus optimales. En ajoutant cette composante stochastique, la fonction mutate contribue à la robustesse et à l'efficacité de l'algorithme en évitant la convergence prématurée vers des solutions sous-optimales.

```
190    static Mutation mutate(boolean[] Individu, double Pm) {
191        boolean[] IndividuOriginal = Individu;
192        int nombreMutation = 0;
193        Random aléatoire = new Random();
194
195        for (int i = 0; i < 5; i++) {
196            if (aléatoire.nextDouble() <= Pm) {
197                nombreMutation++;
198                Individu[i] = !Individu[i];
199            }
200        }
201
202        boolean[] Enfant = Individu;
203
204        int x = decode(Individu);
205
206        if (x < 1 && x > 30) {
207            Enfant = IndividuOriginal;
208            nombreMutation--;
209        }
210
211        return new Mutation(Enfant, nombreMutation);
212    }
```

III.6. *Statistiques :*

La fonction **statistique** que nous avons mise en place joue un rôle essentiel en calculant des paramètres statistiques cruciaux pour évaluer la performance de la population en cours d'optimisation. Ces statistiques fournissent des indicateurs clés sur la convergence et la diversité de la population, permettant ainsi une analyse approfondie de la dynamique du processus d'optimisation. Parmi les mesures évaluées, on retrouve généralement la moyenne, l'écart-type, voire la médiane des valeurs de la population. Ces informations fournissent des insights précieux sur la tendance centrale, la dispersion, et la distribution des solutions, facilitant ainsi la prise de décision quant aux prochaines étapes de l'algorithme génétique. En intégrant cette fonction statistique dans notre approche, nous renforçons la capacité de l'algorithme à ajuster ses paramètres de manière adaptative pour converger vers des solutions optimales de manière efficace et robuste.

```java
static double[] statistique(IndividuClass[] population) {
    double Minimum = population[0].FitnessFunc;
    double Maximum = population[0].FitnessFunc;
    double SommeIndividus = population[0].FitnessFunc;
    double Moyenne = 0;

    for (int i = 1; i < population.length; i++) {
        IndividuClass individu = population[i];
        SommeIndividus += individu.FitnessFunc;
        if (individu.FitnessFunc > Maximum) {
            Maximum = individu.FitnessFunc;
        }
        if (individu.FitnessFunc < Minimum) {
            Minimum = individu.FitnessFunc;
        }
    }
    Moyenne = SommeIndividus / population.length;
    return new double[] { Minimum, Maximum, SommeIndividus, Moyenne };
}
```

III.7. *Déduire et généralisation de nouvelle population :*

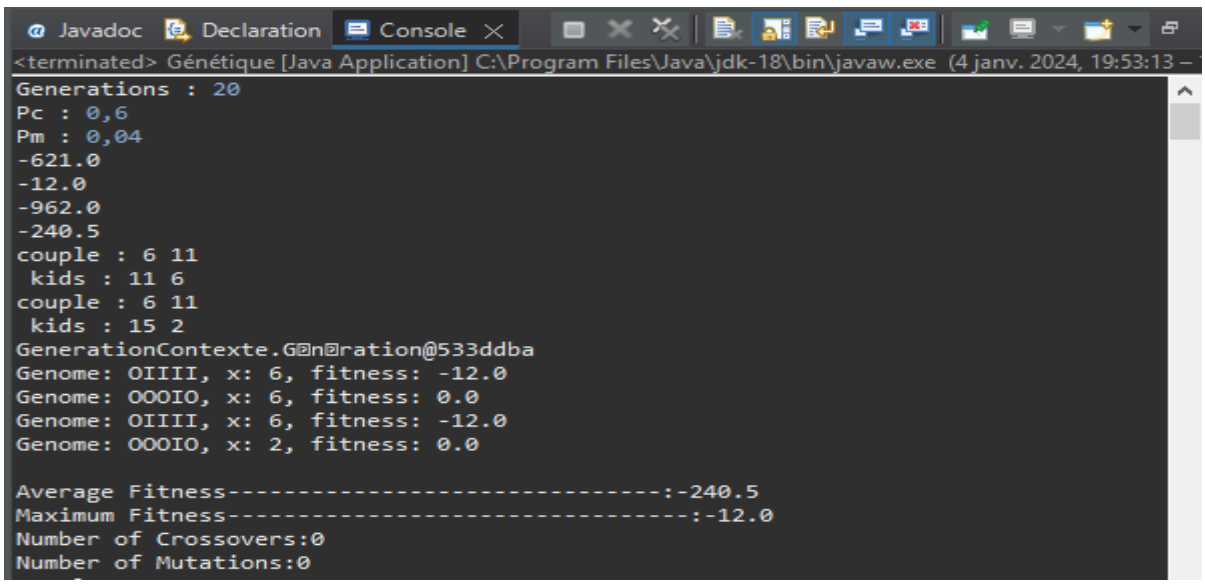
La fonction **generate** assume un rôle central dans la création de nouvelles générations dans notre algorithme génétique. Elle sélectionne les meilleurs individus de la génération actuelle en fonction de la performance par rapport à la fonction objectif, puis applique des opérations de croisement et de mutation. Cette approche mimique la sélection naturelle, favorisant la reproduction des individus les plus performants et introduisant simultanément de la diversité par le biais de mutations. Ainsi, generate orchestre un processus itératif visant à améliorer la population au fil des générations, conduisant à une exploration plus efficace de l'espace des solutions en vue d'atteindre des optima globaux.

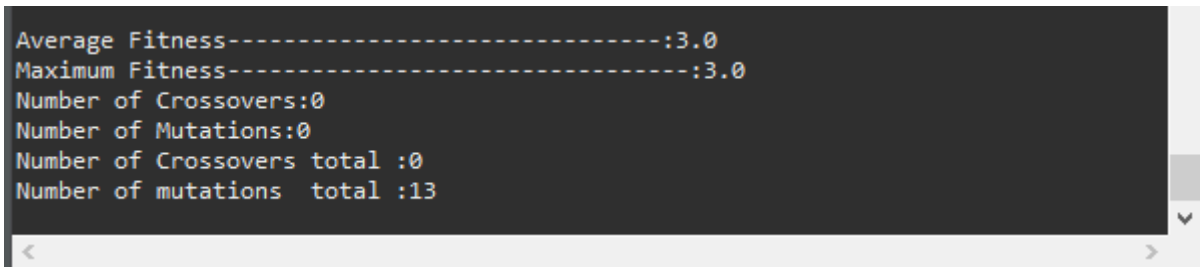
```java
public static Génération generate(IndividuClass[] PremierePopulation, int TaillePopulation, double SommeIndividus, double pc, double pm) {

    IndividuClass[] nouvPopulation = new IndividuClass[4];
    int taille = TaillePopulation;

    int numbCrossPopInit = 0, numbMutPopInit = 0;

    for (int i = 0; i < taille; i += 2) {

        Arrays.sort(PremierePopulation, new Comparator<IndividuClass>() {
            @Override
            public int compare(IndividuClass c1, IndividuClass c2) {
                return Double.compare(c2.getFitnessFunc(), c1.getFitnessFunc());
            }
        });



        IndividuClass gars1 = PremierePopulation[0];
        IndividuClass gars2 = PremierePopulation[1];
        double minFitness = PremierePopulation[1].getFitnessFunc();

        System.out.println("couple : " + gars1.getX() + " " + gars2.getX());



        Croisement X = crossover(gars1.getChromosome(), gars2.getChromosome(), pc);


        int Crosses = Croisement.Crosses;
        numbCrossPopInit += Crosses;


        boolean[] fils_1_beforeMutation = X.Enfant1;
        boolean[] fils_2_beforeMutation = X.Enfant2;

        Mutation mutationResult1 = mutate(fils_1_beforeMutation, pm);
        int nombreMutation1 = mutationResult1.nombreMutation;
        numbMutPopInit += nombreMutation1;
```

```java
            boolean[] Enfant1 = mutationResult1.Enfant;

            Mutation mutationResult2 = mutate(fils_2_beforeMutation, pm);
            int nombreMutation2 = mutationResult2.nombreMutation;
            numbMutPopInit += nombreMutation2;
            boolean[] Enfant2 = mutationResult2.Enfant;


            int X1 = decode(Enfant1);
            int X2 = decode(Enfant2);

            System.out.println(" kids : " + X1 + " " + X2);

            double FitnessFunc1 = f(X1);
            double FitnessFunc2 = f(X2);

            // accept best fitness
            IndividuClass individu1, individu2;

            if (minFitness < FitnessFunc1) {
                individu1 = new IndividuClass(Enfant1, X1, FitnessFunc1, FitnessFunc2 = 0, gars1, gars2);
            } else {
                individu1 = new IndividuClass(gars1.getChromosome(), gars1.getX(), gars1.getFitnessFunc(), FitnessFunc2, gars1.getParent1(),
                        gars1.getParent2());
            }

            if (minFitness < FitnessFunc2) {
                individu2 = new IndividuClass(Enfant2, X2, FitnessFunc1 = 0, FitnessFunc2, gars1, gars2);
            } else {
                individu2 = new IndividuClass(gars2.getChromosome(), gars2.getX(), gars2.getFitnessFunc(), FitnessFunc2, gars2.getParent1(),
                        gars2.getParent2());
            }

            nouvPopulation[i] = individu1;
            nouvPopulation[i + 1] = individu2;
        }


        return new Génération(nouvPopulation, numbCrossPopInit, numbMutPopInit);
```

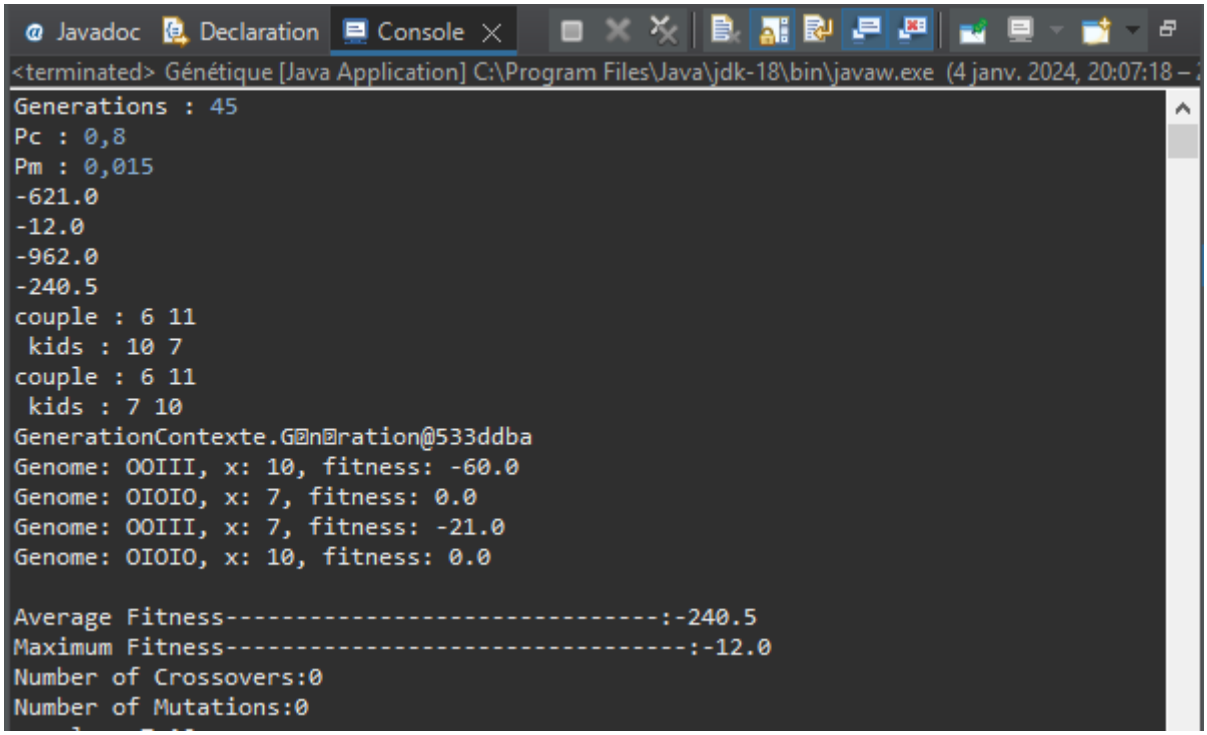
IV. Affichage et résultats pour chaque sélection :

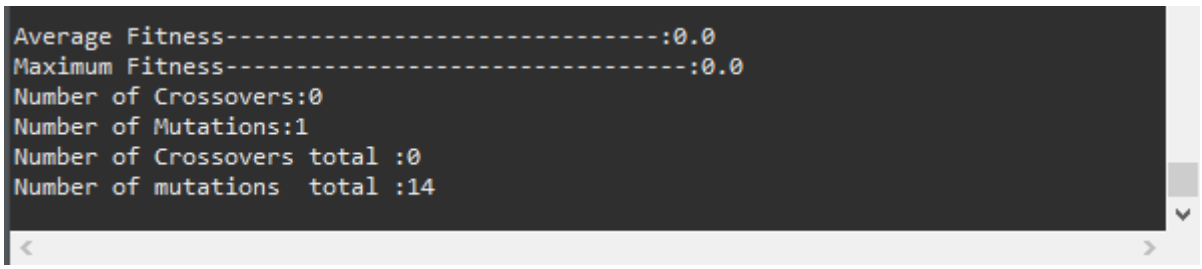
IV.1. *Sélection 1 :*



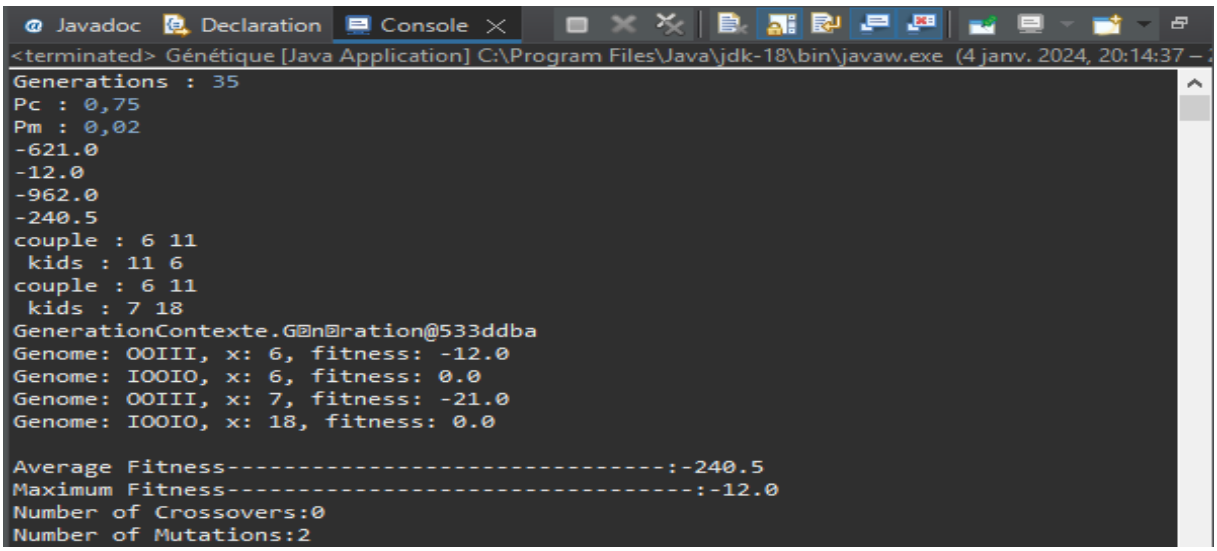

Average Fitness----------:3.0
Maximum Fitness----------:3.0
Number of Crossovers:0
Number of Mutations:0
Number of Crossovers total :0
Number of mutations total :13

IV.2. *Sélection 2 :*





IV.3. *Sélection 3 :*



```
Average Fitness--------------------------------:3.0
Maximum Fitness--------------------------------:3.0
Number of Crossovers:0
Number of Mutations:0
Number of Crossovers total :0
Number of mutations  total :15
```

IV.4. *Sélection 4 :*

```
Javadoc  Declaration  Console
<terminated> Génétique [Java Application] C:\Program Files\Java\jdk-18\bin\javaw.exe  (4 janv. 2024, 20:18:58 –
Generations : 75
Pc : 0,25
Pm : 0,035
-621.0
-12.0
-962.0
-240.5
couple : 6 11
 kids : 6 3
couple : 6 11
 kids : 6 3
GenerationContexte.G@n@ration@533ddba
Genome: OOIIO, x: 6, fitness: -12.0
Genome: OOOII, x: 3, fitness: 0.0
Genome: OOIIO, x: 6, fitness: -12.0
Genome: OOOII, x: 3, fitness: 0.0

Average Fitness--------------------------------:-240.5
Maximum Fitness--------------------------------:-12.0
Number of Crossovers:0
Number of Mutations:1
```

```
Average Fitness--------------------------------:4.0
Maximum Fitness--------------------------------:4.0
Number of Crossovers:0
Number of Mutations:0
Number of Crossovers total :0
Number of mutations  total :42
```

Alors finalement on aura ce tableau d'analyse final qui résumera ces résultats capturés comme suit.

Sélection	Génération	Pc	Pm	Average fitness	Maximum fitness	Crossover Total	Mutation Total	Temps (ms)
1	20	0.6	0.04	0	13	3	3	15
2	45	0.8	0.015	0	14	0	0	29
3	35	0.75	0.02	3	3	0	15	47
4	75	0.25	0.035	4	4	0	42	36