



1ère Année Master Informatique, Semestre 2

**Option :** Systèmes informatiques Intelligents (SII)

**Module :** Méta-heuristique et Algorithmes  
Evolutionnaires

<p><b><u>PROJET :</u></b> Développement de Solveur SAT</p>
--

**Binôme :**

- OUHOCINE Sarah

**Professeur :**

- M<sup>me</sup> H. DRIAS

**Groupe :**

- 3

# Table des matières

## 1 Introduction

- 1.1 Problématique.....
- 1.2 Complexité SAT.....
- 1.3 Données.....
- 1.4 Modélisation du problème SAT (structure de données).....

## 2 Les Méthodes heuristiques pour SAT

- 2.1 Méthodes non informées (aveugles).....
  - 2.1.1 BFS et Algorithme.....
  - 2.1.2 DFS et Algorithme.....
- 2.2 Méthodes informées .....
  - 2.2.1 A\*.....

## 3 Algorithme génétique pour SAT

- 3.1 Espace de recherche.....
- 3.2 Croisement.....
- 3.3 Mutation.....
- 3.4 Algorithme.....
- 3.5 Expérimentation.....
  - 3.5.1 Réglage des paramètres.....
  - 3.5.2 Performance.....
    - 3.5.2.1 Qualité de la solution.....
    - 3.5.2.2 Temps d'exécution.....

## 4 PSO pour SAT

- 4.1 Espace de recherche.....
- 4.2 Algorithme.....
- 4.3 Expérimentation.....
  - 4.3.1 Réglage des paramètres.....
  - 4.3.2 Performance.....
    - 4.3.2.1 Qualité de la solution.....
    - 4.3.2.2 Temps d'exécution.....

## 5 Conclusion

- 5.1 Etude Comparative.....
- 5.2 Analyse des résultats.....
- 5.3 Interface.....
- 5.4 Récapitulatif et objectif du projet .....

# 1 INTRODUCTION

## 1.1 PROBLEMATIQUE :

Le problème de satisfiabilité ou en abrégé « **problème SAT** » est un **problème de décision** c.à.d. une question accompagnant la description de l'instance dont la réponse est soit « OUI » soit « NON ».

**>>INSTANCE** :  $X = \{x_1, x_2, \dots, x_n\}$  un ensemble de littéraux,  $C = \{c_1, c_2, \dots, c_n\}$  un ensemble de clauses telle que :

\*Un littéral  $l$  : est une variable propositionnelle  $x$  (littéral positif) ou la négation d'une variable propositionnelle  $\neg x$  (littéral négatif).

\*Une clause  $C$  : est une disjonction de littéraux.

$$C = \bigvee_{i=1}^n l_i = l_1 \vee l_2 \vee l_3 \vee \dots \vee l_n \text{ où les } l_i \text{ sont des littéraux.}$$

\*Une Formule de Logique Propositionnelle  $F$  : est construite à partir des littéraux et des connecteurs booléens ( $\wedge$  : **et**), ( $\vee$  : **ou**), ( $\neg$  : **non**).

\*Une Formule  $F$  est en CNF : est une conjonction de clauses

$$F = \bigwedge_{i=1}^n C_i = C_1 \wedge C_2 \wedge C_3 \wedge \dots \wedge C_n \text{ où les } C_i \text{ sont des clauses.}$$

**>>QUESTION** : Etant donné une **Formule de Logique Propositionnelle**  $F$ , existe-il une instantiation (un ensemble de valeurs booléennes associées aux littéraux) de l'ensemble des littéraux  $X$  telle que la conjonction des clauses de  $C$  est vraie ?  
Autrement dit : existe une affectation des **littéraux** qui rend cette formule  $F$  vraie (satisfiable)

**PS** : Dans ce projet, on va développer des solveurs pour le problème 3-SAT (i.e. chaque clause est une disjonction de 3 littéraux)

## 1.2 Complexité SAT :

Il s'agit de trouver la solution optimale (La solution qui satisfait toutes les clauses)  
Et comme un littéral peut prendre 2 valeurs possibles soit Vraie soit fausse, donc on se retrouve à parcourir un espace de recherche de  $2^n$  solution (**pire cas**) où  $n$  est le nombre variables d'où la complexité est  $O(2^n)$ .

## 1.3 Données :

Les données sont enregistrées dans des fichiers dont l'extension **.cnf**, il existe plusieurs types de fichiers classifiés selon le nombre de variables (**uf nbVariable-numInsatnce**).

Dans notre projet on s'est intéressé aux instances des 3 types (uf20, uf50, uf75). Dans chaque instance, le nombre de variables et le nombre de clauses est mentionné en commentaire (ainsi que d'autres informations...) en haut du fichier.

Juste après ce commentaire, On retrouve les informations de la formule tel que :

**Une ligne i** dans le fichier représente La clause  $C_i$  (disjonction de 3 littéraux) de la formule suivie d'un zéro Indiquant la fin de la clause i.

Les littéraux sont représentés par des entiers signés

Par exemple : -18 représente le littéral ( $\neg X_{18}$ ), 19 représente le littéral ( $X_{19}$ ) ...

Commentaire des infos

```
p cnf nbVariable nbClauses
C1 -18 19 7 0
   3 18 -5 0
   -5 -8 -15 0
   ....
   ....
Cn  4 -16 -5 0
```

#### 1.4 Modélisation du problème SAT (structures de données)

A >> **Structure de données utilisé pour récupérer les données du fichier .cnf** :  
C'est une Matrice (nx3) où n est le nombre de clauses.

FICHIER.cnf

MATRICE M

```
C1 -18 19 7 0
   3 18 -5 0
   -5 -8 -15 0
   ....
   ....
Cn  4 -16 -5 0
```



	M [i, 1]=Littéral1	M [i, 2]=Littéral2	M [i, 3]=Littéral 3
M [1, j]=C 1	-18	-19	7
	3	18	-5
	-5	-8	-15
M [n, j]=Cn	4	-16	-5

Chaque cellule [i, j] de la matrice M représente le littéral j de la clause i tel que  $1 \leq i \leq n$  et  $1 \leq j \leq 3$ .

On a vu déjà que le problème SAT est un problème NP (NP-complet) car il est structuré en deux phases :

>> **1<sup>ère</sup> phase** : Engendrer une solution quelconque à l'instance qui peut être bonne ou mauvaise.

>> **2<sup>ème</sup> phase** : Vérifier en temps polynomial si la solution est effective ou pas.

B >> **Structure de données utilisé pour engendrer une solution quelconque :**  
C'est un tableau binaire de taille égale à n (n étant le nombre de variables) ou chaque case aura la valeur soit vraie « 1 » pour dire que la variable est vraie, soit « 0 » pour dire que la variable est fausse.

S :

1/0	1/0	1/0															1/0
$X_1$	$X_2$	$X_3$															$X_n$

C >> **Vérification si une solution S est effective ou pas :**  
Chaque variable va être remplacée par sa valeur (0/1 à partir du tableau S) dans toutes ses occurrences dans la matrice M, pour évaluer ensuite les clauses  $C_i$  de la formule  $F$ .

>> **Un littéral  $l$  est vrai si :**

- Il est composé d'une variable et cette variable est vraie.
- Il est composé d'une négation d'une variable et cette variable est fausse.

>> **Une clause  $C$  est vraie si :** au moins un de ses littéraux est vrai.

>> **Une formule  $F$  est vraie si et seulement si :** toutes ses clauses sont vraies.

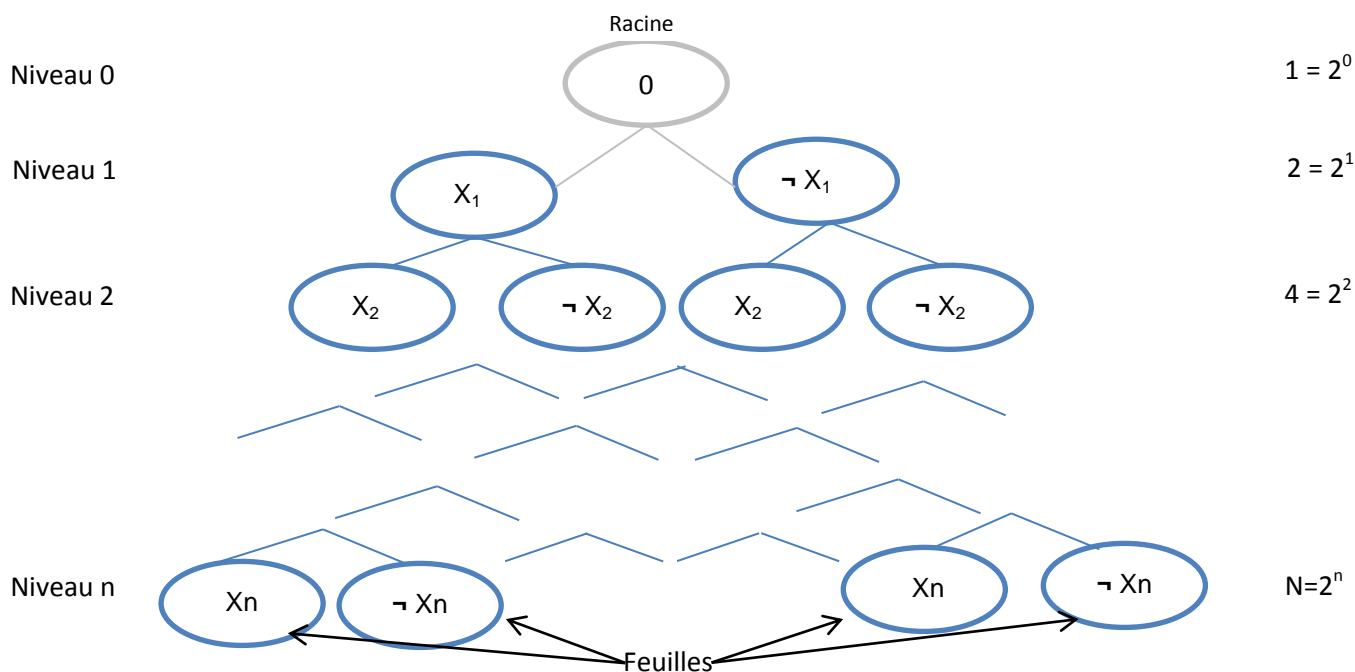
$$F = (\underbrace{l_i \vee l_i \vee l_i}_{C1}) \wedge (\underbrace{l_i \vee l_i \vee l_i}_{C2}) \wedge (\underbrace{l_i \vee l_i \vee l_i}_{C3}) \wedge \dots \wedge (\underbrace{l_i \vee l_i \vee l_i}_{Cn})$$

$$1 \leq i \leq \text{nbVar}$$

$$l_i \text{ est soit } X_i \text{ soit } \neg X_i$$

## 2 LES METHODES HEURISTIQUES POUR SAT

Dans ces méthodes l'espace de recherche est un arbre binaire (l'ensemble d'états) dont la profondeur est égale au nombre de variables+1, le 1<sup>er</sup> niveau de profondeur contient un seul état qui est la racine, elle est égale à 0 (valeur ignorée/neutre). Chaque branche de cet arbre (chemin de la racine à la feuille) est une solution potentielle pour le problème SAT.



Notons que l'ordre des variables peut être quelconque dans une solution (i.e. l'ordre de la racine aux feuilles)...

## 2.1 Méthodes non informées (aveugles) :

Les méthodes consistent à construire à partir de l'ensemble de configurations possibles une solution de manière exhaustive.

La démarche pour résoudre de tels problèmes consiste à représenter le problème par un espace d'états, chaque état est une configuration possible du problème

La résolution consiste à trouver un chemin menant de la configuration initiale à la configuration finale.

\* Un état (Configuration) : est soit variable la  $X_i$ , soit  $\neg X_i$  tel que  $1 \leq i \leq \text{nbVariables}$ .

### 2.1.1 Parcours en largeur d'abord (BFS)

Le parcours BFS consiste à visiter les états (nœuds) de manière horizontale, niveau par niveau : la racine puis les états du 1<sup>er</sup> niveau de profondeur, puis ceux du second niveau... et ainsi de suite jusqu'au n<sup>ième</sup> niveau de profondeur où  $n$  : est le nombre de variables.

Nous parcourons l'arbre ci-dessus, en vérifiant les solutions niveau par niveau, certains problèmes SAT n'incluant pas l'ensemble complet des variables, il est possible de trouver la solution optimale sans atteindre les états du dernier niveau, pour cela nous avons ajouté dans chaque état « un chemin » stockant le chemin emprunté de la racine jusqu'à cet état-là.

Pour chaque niveau, nous stockons tous ses états, nous vérifions par la suite chacun de ses chemins, puis nous calculons sa performance afin d'avoir le chemin de meilleure performance (solution qui satisfait le maximum de clauses).

```
ArrayList<Integer> breadthFirst() {
    ArrayList<Integer> c = new ArrayList<Integer>();
    ArrayList<Tree> pile = new ArrayList<Tree>();
    Tree gauche, droite, travers = T;
    int d = 0;
    pile.add(T);
    c.add(travers.getV());
    d = check(c);
    while(d != c_num){
        if(travers.getLeft() != null) {
            gauche = travers.getLeft();
            c.add(gauche.getV());
            gauche.setHeu_v(gauche.getHeu_v() + check(c));
            pile.add(gauche);
            c.remove(c.size()-1);
        }
        if(travers.getRight() != null) {
            droite = travers.getRight();
            c.add(droite.getV());
            droite.setHeu_v(droite.getHeu_v() + check(c));
            pile.add(droite);
            c.remove(c.size()-1);
        }
        travers = pile.get(max(pile));
        pile.remove(travers);
        while(c.size() != Math.abs(travers.getV())) {
            c.remove(c.size()-1);
        }
        c.add(travers.getV());
        d = check(c);
    }
    return c;
}
```

### 2.1.2 Parcours en profondeur d'abord(DFS)

Le parcours DFS consiste à visiter les états de manière verticale i.e. les successeurs des états les plus récemment visités en premier. Ceci conduit à aller jusqu'à l'état  $n$  où  $n$  est le nombre de variables puis remonter dans l'arbre au niveau qui se trouve jusqu'au-dessus pour explorer les autres branches de la même manière.

Les solutions possibles pour le problème SAT pour  $n$  variables :  $\rightarrow 2^n$

Solution 1	X1, X2, X3, .....Xn
	X1, X2, X3..... $\neg$ Xn
	.....
	.....
Solution 2 <sup>n</sup>	$\neg$ X1, $\neg$ X2, $\neg$ X3 .....Xn
	$\neg$ X1, $\neg$ X2, $\neg$ X3 ..... $\neg$ Xn

```

ArrayList<Integer> depthFirst (){
    int d = 0;
    ArrayList<Integer> c = new ArrayList<Integer>();
    Tree travers = T;
    ArrayList<Tree> pile = new ArrayList<Tree>();
    pile.add(T);
    c.add(travers.getV());
    while (!pile.isEmpty()){
        if(travers != T) c.add(travers.getV());
        if(travers.getRight() != null ) pile.add(travers.getRight());
        if(c.size() == var_num + 1){
            d = Pcheck(c);
            if (d == c_num){
                System.out.println(check(c));
                return c;
            }else {
                travers = pile.get(pile.size() - 1);
                pile.remove(pile.size() - 1);
                while(true) {
                    if(c.get(c.size()-1) > 0 ) c.remove(c.size()-1);
                    else break;
                }
                c.remove(c.size()-1);
            }
        }else {
            travers = travers.getLeft();
        }
    }
    return null;
}

```

## 2.2 Méthodes informés :

Consiste à faire des parcours informés basés sur des heuristiques, on fait appel à des heuristiques

\* Une Heuristique : a pour rôle de diriger l'expansion des états vers l'état cible en développant les nœuds les plus prometteurs.

### 2.2.1 A\* :

Il s'agit d'un algorithme généralement appliqué aux graphes (y compris arbres), cet algorithme utilise une fonction d'évaluation, qui correspond dans notre cas au nombre de clauses satisfaites sur chaque état, ce dernier nous aide à guider notre parcours de l'arbre.

Soit  $F(n)$  la fonction conçue pour un état  $n$  qui donne l'estimation du chemin optimal de la solution depuis la racine, une fois cette fonction définie, nous trions les états par ordre croissant en choisissant à chaque fois l'état de plus grande valeur....

La fonction d'estimation est définie comme suit :

$F(n) = G(n) + H(n)$  où

$G(n)$  : est la profondeur de l'état  $n$

$H(n)$  : est le nombre de clauses satisfaites

## 3 ALGORITHME GENETIQUE POUR SAT

Les algorithmes génétiques sont des algorithmes d'optimisation stochastique fondés sur les mécanismes de la sélection naturelle et de la génétique, dont on prend une population initiale et on applique une suite de croisements entre eux, de telles façons à obtenir une nouvelle génération (on peut obtenir quelques mutations)

### 3.1 Espace de recherche :

Il s'agit de la population qui est constituée d'un nombre  $n$  fixe de solutions, initialement ses solutions sont générées aléatoirement, ensuite dans chaque itération et après avoir fait le croisement, le nombre de la population change de telle façon qu'on ne garde que les meilleures solutions.

Du coup, l'espace de recherche devient les meilleures solutions entre les générations.

### 3.2 Croisement :

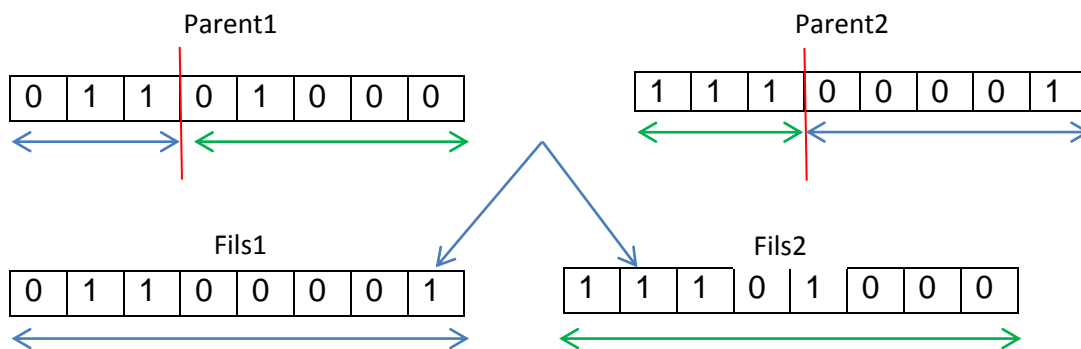
Le croisement est le fait de mélanger les chromosomes des parents (2 solution choisit aléatoirement de la population) afin de créer une nouvelle génération différente à partir de la génération précédente

Il existe plusieurs méthodes de croisements, dans notre projet on a fait de telle sorte qu'on choisit **un bit** inférieur au nombre des variables, les bits qui se trouvent avant le bit choisit dans le père numéro 1 vont être identiques à ceux du 1<sup>er</sup> fils et le reste



des bits du 1<sup>er</sup> fils vont être les bits qui se trouvent après le bit choisi dans le père numéro 2.

```
public static int[][] croisement (int[] a, int[] b, fichier f){
    int[][] ab = new int[2][a.length];
    Random rand = new Random();
    int k = rand.nextInt(f.nbrVar);
    for(int i=0; i<k; i++) {
        ab[0][i]=a[i];
        ab[1][i]=b[i];
    }
    for(int i=k; i<f.nbrVar; i++) {
        ab[1][i]=a[i];
        ab[0][i]=b[i];
    }
    return ab;
}
```



### 3.3 Mutation

La mutation se fait par un choix d'un nombre aléatoire puis inverser ce nombre (si c'est un 0, il devient un 1 et vice-versa).

```
public int[] mutation (int[] solution ){
    Random rand = new Random();
    int chromosome = rand.nextInt();
    solution[chromosome]=(solution[chromosome]+1)%2;
    return solution;
}
```



### 3.4 Algorithmme

```

public void Ag () {
    int iter=1;
    int sol1,sol2 ; //num solution conserve par le croisement
    int[][] fils;
    int Rc,Rm;
    int chromosome;
    int cptDeStagnation=1;
    Random rand = new Random();
    // genere la population
    salution sol = new salution(f,this.taillePopulation);
    // boucle pour les croisement (tant que )
    while(iter<maxIterm&&cptDeStagnation<teauxDeStagnation) {
        sol1 = rand.nextInt(this.taillePopulation);
        do {
            sol2 = rand.nextInt(this.taillePopulation/2);
        }while(sol1==sol2);
        //genere Rc
        Rc=rand.nextInt(101);
        // faire le croisement
        fils=salution.croisement(sol.population[sol1],sol.population[sol2],f);
        //genere Rm
        Rm=rand.nextInt(101);
        if(Rm<teauxDeMutation)
        {
            //changer un chromosome
            salution.mutation(fils[0]);
            salution.mutation(fils[1]);
        }
        //
        cptDeStagnation=sol.classificationSolution(fils[0],f.nbrClauseSatisfait(fils[0]), cptDeStagnation,f);
        cptDeStagnation=sol.classificationSolution(fils[1],f.nbrClauseSatisfait(fils[1]), cptDeStagnation,f);

        iter++;
    }
    stagnation = iter<maxIterm;
    this.bestSol=sol.population[0];
    this.bestF=sol.nbrStisfaction[0];
}

```

### 3.5 Expérimentation

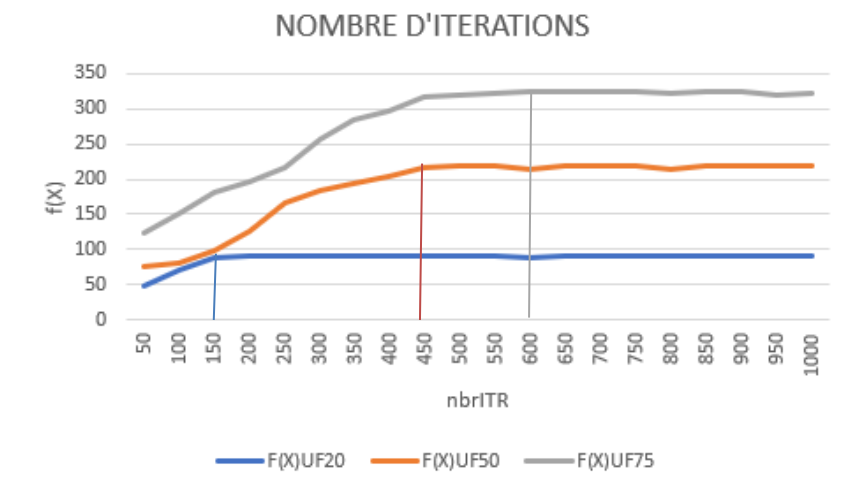
L'algorithme génétique dépend de 4 paramètres :

- Le nombre maximum d'itération (Max\_Iter): le nombre maximal de générations générées par cette solution.
- Taille de la population (Taille\_Pop).
- Taux de croisement (TC) : le nombre de fois qu'on procède au croisement dans une même itération ou génération.
- Taux de mutation (TM) : le nombre de fois qu'on procède à la mutation dans une même itération ou génération

#### 3.5.1 Réglage des paramètres :

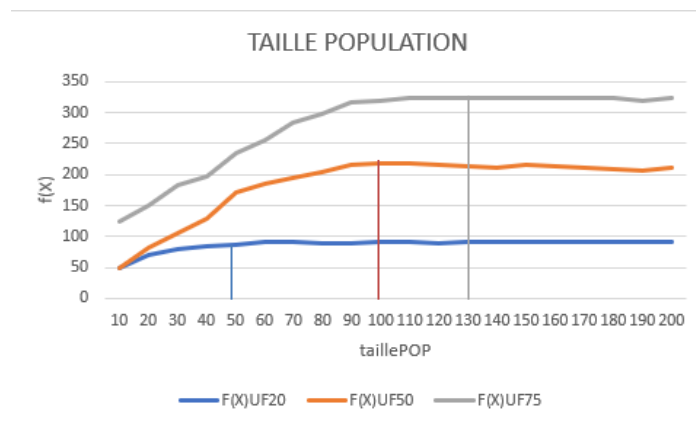
Après avoir exécuté les différents types des fichiers .cnf avec des différentes valeurs pour chaque paramètre, On a fait les statistiques, puis on a enregistré les résultats dans un fichier Excel afin de trouver les paramètres qui satisfaites le maximum nombre de clauses... on a obtenu les graphes suivants :

### I. Nombre max d'itération :



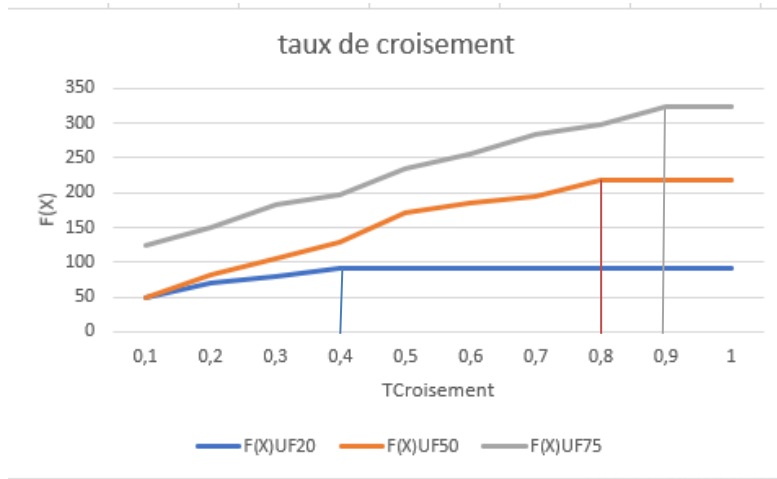
Les meilleures valeurs de max itérations obtenues pour les fichiers **uf20**, **uf50**, **uf75** sont : **150**, **450**, **600** respectivement.

### II. Taille de la population :



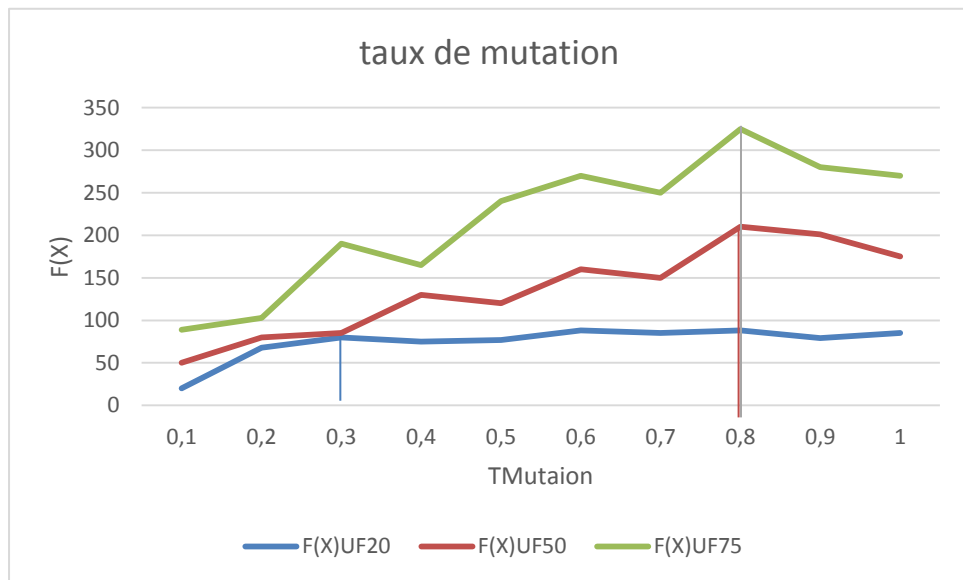
Les meilleures valeurs de la taille de population obtenues pour les fichiers **uf20**, **uf50**, **uf75** sont **50**, **100**, **130** respectivement.

### III. Taux de croisement :



Les meilleures valeurs du taux de croisement obtenues pour les fichiers  $uf20$ ,  $uf50$ ,  $uf75$  sont 0.4, 0.8, 0.9 respectivement.

## VI. Taux de mutation :



Les meilleures valeurs du taux de mutation pour les fichiers  $uf20$ ,  $uf50$ ,  $uf75$  sont 0.3, 0.8, 0.8 respectivement.

### Remarque :

« Les valeurs d'un paramètre se diffèrent d'un type de fichier à un autre (cela est dû à l'augmentation de la complexité qui est liée à l'augmentation des nombres de variables). »

uf20

Max_Iter	150
Taille_Pop	50
TC	0.4
TM	0.3

uf50

Max_Iter	450
Taille_Pop	100
TC	0.8
TM	0.8

uf75

Max_Iter	600
Taille_Pop	130
TC	0.9
TM	0.8

3.5.2 Performance : La performance inclut la qualité de la solution et le temps d'exécution.

#### 3.5.2.1 Qualité de la solution :

En fixant les 4 paramètres cités précédemment aux meilleures valeurs obtenues dans les réglages des paramètres, on parvient la solution optimale pour chacune des instances des fichiers (uf20, uf50, uf75) et cela par rapport au nombre de clauses satisfaites par cette dernière (nombre de clauses satisfaites par la solution est égale au nombre total des clauses des instances des fichiers)

Type du fichier	NbClauses du fichier	Qualité de solution	Pourcentage de Qualité
uf20	91	91	100%
uf50	218	218	100%
uf75	325	325	100%

#### 3.5.2.2 Temps d'exécution :

La solution optimale pour chacune des instances des fichiers (uf20, uf50, uf75) assure un temps d'exécution minimum et très réduit par rapport aux autres solutions (entre 5 et 10 secondes)

D'après « **le pourcentage de la qualité de la solution** » et « **le temps d'exécution** » de chacune des instances des fichiers (uf20, uf50, uf75), on peut conclure que la solution est performante autrement dit L'algorithme génétique a réussi à trouver la solution optimale en un temps réduit i.e. solution qui atteint la meilleure performance pour chaque instance.

## 4 PSO pour SAT

Cet algorithme s'inspire à l'origine du monde du vivant, il s'appuie notamment sur un modèle permettant de simuler le déplacement d'un groupe d'oiseaux.

Cette méthode d'optimisation se base sur la collaboration des individus entre eux.

Cette idée veut qu'un groupe d'individus peu intelligents puisse posséder une organisation globale complexe.

Ainsi grâce à des règles de déplacements, les particules peuvent converger progressivement vers un minimum local. Cette méta heuristique fonctionne mieux pour des espaces en variables continues (à l'origine, PSO a d'abord été conçu pour une optimisation continue, Il a ensuite été adapté aux problèmes combinatoires discrets).

### Codage de la solution

- une solution (particule), est un vecteur de  $n$  valeurs booléennes, chacune affectée à une variable il s'agit donc d'une chaîne de  $n$  bits.

Chaque particule est un point dans l'espace multidimensionnel, qui est l'espace de recherche.

#### 4.1 Espace de recherche :

L'ensemble de toutes les instanciations potentielles pour l'instance, il s'agit alors de l'ensemble des vecteurs booléens de longueur égale à  $n$  (l'ensemble des particules). La taille de l'espace de recherche est égale à  $2^n$ .

-Au départ de l'algorithme chaque particule est donc positionnée dans l'espace de recherche du problème. Chaque itération fait bouger les particules en fonction de 3 composantes :

- Sa vitesse actuelle  $V_k$
- Sa meilleure solution  $P_i$
- La meilleure solution obtenue dans son voisinage  $P_g$

Cela donne l'équation de mouvement suivante :

$$V_{k+1} = wV_k + c_1 (P_i - X_k) + c_2 (P_g - X_k)$$

$$X_{k+1} = X_k + V_{k+1}$$

Avec :

- $w$  : L'inertie
- $c_1, c_2$  : Nombres aléatoires.

-Le processus PSO se déroule en plusieurs itérations, faisant évoluer les valeurs de la fonction des particules vers de meilleures.

-Les nouvelles solutions sont évaluées à chaque itération et la meilleure solution globale est déterminée.

## 4.2 Algorithme PSO :

>> **Partie déclaration des variable :**

```
int iter=1;
int [] Gbest = new int[f.nbrVar];
int [][] Pbest = new int[taillePopulation][f.nbrVar];
int [] fPbest = new int [taillePopulation];
int fGbest,maxPbest;
int cptStagnation=0;
double r1,r2;
int r;
```

\***iter** : c'est le nombre d'itération déjà faites.

>>**Partie d'initialisation :**

```
/// initialize N positions (N=nbr population , nbr particule)
salution sol = new salution(f,this.taillePopulation);
/// initialize N velocities
int [] vel = new int [this.taillePopulation];
for (int i =0;i<taillePopulation;i++) {
    vel[i]=rand.nextInt(maxVelocity+1);
}
// the particules position is evaluated once initialized
/// Pbesti = Xi
for (int i=0;i<taillePopulation;i++) {
    for (int j=0;j<f.nbrVar;j++) { Pbest[i][j]=sol.population[i][j]; } fPbest[i]=sol.nbrStisfaction[i];
}
/// calculate Gbest
for(int i=0;i<f.nbrVar;i++) { Gbest[i]=sol.population[sol.maxSol()][i]; }
fGbest= f.nbrClauseSatisfait(Gbest);
```

>>**Partie qui se fait dans chaque itération :**

```
while(iter<maxIterm && cptStagnation<this.teauxDeStagnation) {
    /// for each particle
    for (int i=0; i <taillePopulation;i++) {
        /// update the velocity
        r1=Math.random();r2=Math.random();
        vel[i]=(int)(w*vel[i]+c1*r1*distance(Pbest[i],sol.population[i])+c2*r2*distance(Gbest,sol.population[i]))%maxVelocity;
        ///move the particle
        for(int j=0;j<vel[i];j++) { r=rand.nextInt(f.nbrVar); sol.population[i][r]=Pbest[i][r]; }
        for(int j=0;j<vel[i];j++) { r=rand.nextInt(f.nbrVar); sol.population[i][r]=Gbest[r]; }
        ///evaluate it's fitness
        sol.nbrStisfaction[i]=f.nbrClauseSatisfait(sol.population[i]);
        ///update Pbest
        if(sol.nbrStisfaction[i]>fPbest[i]) {
            for(int j=0;j<f.nbrVar;j++) {Pbest[i][j]=sol.population[i][j];}
        }
    }
    ///update Gbest
    ///calculer fmax(Pbesti)
    maxPbest=0;
    for(int i=1;i<Pbest.length;i++) {
        if(fPbest[maxPbest]<fPbest[i]) maxPbest=i;
    }
    ///update if necessary
    if(fPbest[maxPbest]>fGbest) {
        cptStagnation=0;
        for(int i=0;i<Gbest.length;i++) {
            Gbest[i]=Pbest[maxPbest][i];
        }
    }
    }else {cptStagnation++;}
    iter++;
}
```

>>**Partie qui enregistre le résultat final :**

```
this.stagnation=iter<maxIterm;
this.bestSol=Gbest;
this.bestF=fGbest;
```

#### 4.3 Expérimentation :

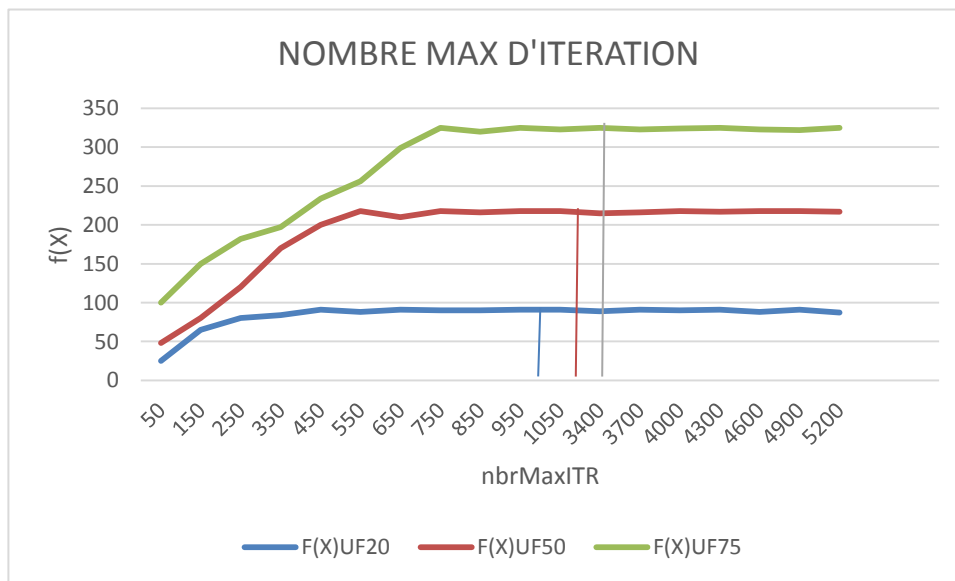
PSO dépend de 5 paramètres :

- Le nombre maximum d'itération(Max\_Iter).
- W : coefficient d'inertie.
- C1.
- C2.
- Nombre de particules(Nb\_Part).

##### 4.3.1 Réglage des paramètres :

Après avoir exécuté les différents types des fichiers .cnf avec des différentes valeurs pour chaque paramètre, On a fait les statistiques, puis on a enregistré les résultats dans un fichier Excel afin de trouver les paramètres qui satisfait le maximum nombre de clauses... on a obtenu les graphes suivants :

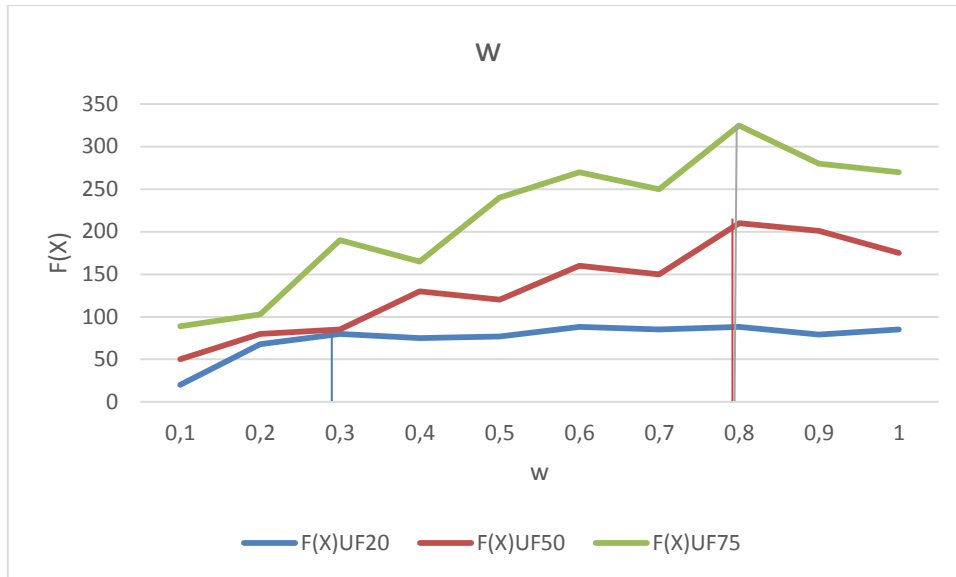
##### I. Nombre max d'itération :



Les meilleures valeurs du nombre maximum d'itération pour les fichiers  $uf20$ ,  $uf50$ ,  $uf75$  sont 1000, 2500, 3400 respectivement.

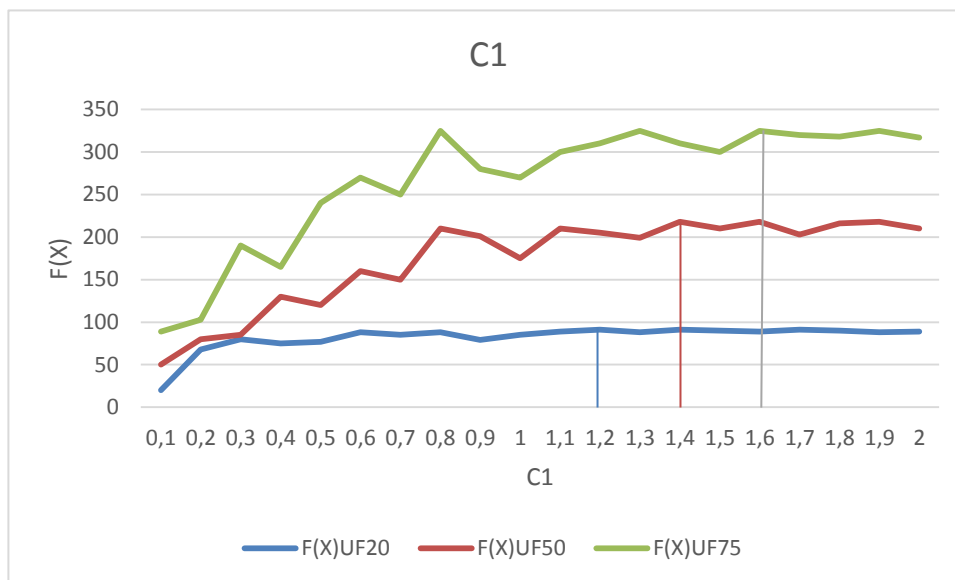


## II.W (coefficient d'inertie) :



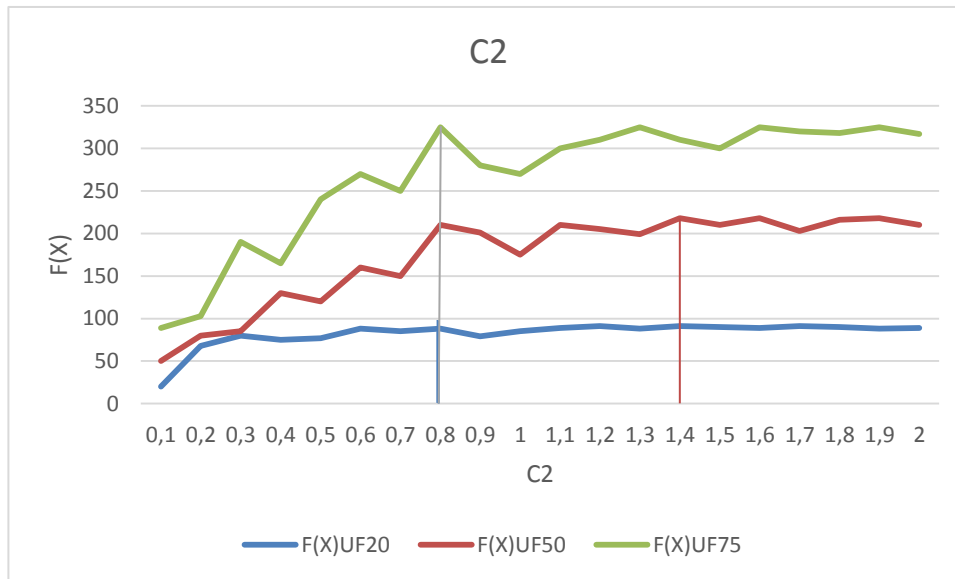
Les meilleures valeurs du coefficient d'inertie pour les fichiers **uf20**, **uf50**, **uf75** sont **0.3**, **0.8**, **0.8** respectivement.

## III.C1 :



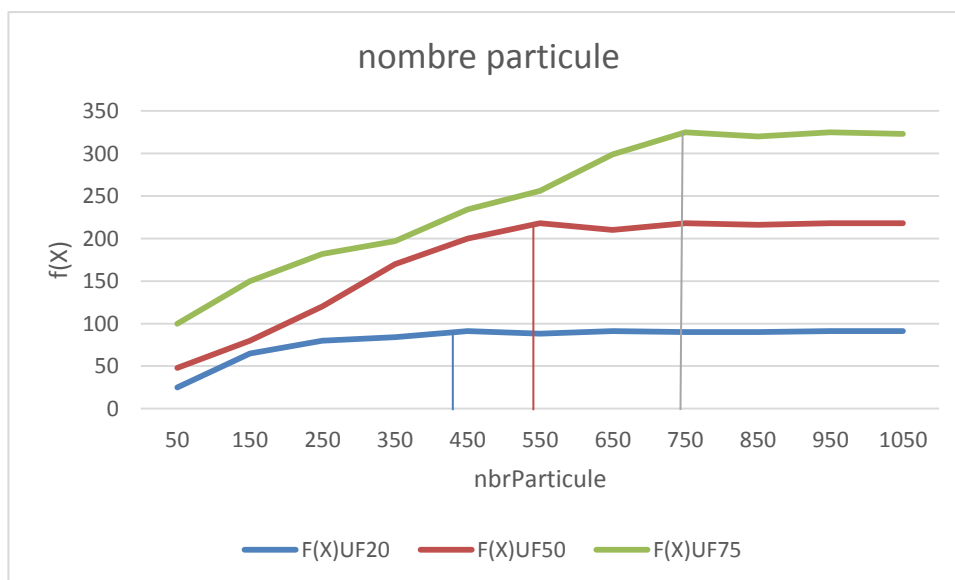
Les meilleures valeurs du  $c1$  pour les fichiers **uf20**, **uf50**, **uf75** sont **1.2**, **1.4**, **1.6** respectivement.

## VI.C2 :



Les meilleures valeurs du  $c2$  pour les fichiers  $uf20$ ,  $uf50$ ,  $uf75$  sont 0.8, 1.4, 0.8 respectivement.

## V. Nombre de particule :



Les meilleures valeurs du nombre de particules pour les fichiers  $uf20$ ,  $uf50$ ,  $uf75$  sont 450, 550, 750 respectivement.

uf20

Max_Iter	1000
W	0.3
C1	1.2
C2	0.8
Nb_Part	450

uf50

Max_Iter	2500
W	0.8
C1	1.4
C2	1.4
Nb_Part	550

uf75

Max_Iter	3400
W	0.8
C1	1.6
C2	0.8
Nb_Part	750

3.3.2 Performance : La performance inclut la qualité de la solution et le . . temps d'exécution.

#### 3.3.2.1 Qualité de la solution :

En fixant les 5 paramètres cités précédemment aux meilleures valeurs obtenues dans les réglages des paramètres, on parvient la solution optimale pour chacune des instances des fichiers (uf20, uf50, uf75) et cela par rapport au nombre de clauses satisfaites par cette dernière (nombre de clauses satisfaites par la solution est égale au nombre total des clauses des instances des fichiers)

Type du fichier	NbClauses du fichier	Qualité de solution	Pourcentage de Qualité
uf20	91	91	100%
uf50	218	218	100%
uf75	325	325	100%

#### 3.3.2.2 Temps d'exécution :

La solution optimale pour chacune des instances des fichiers (uf20, uf50, uf75) assure un temps d'exécution minimum et réduit (entre 9 et 15 secondes).

D'après « **le pourcentage de la qualité de la solution** » et « **le temps d'exécution** » de chacune des instances des fichiers (uf20, uf50, uf75), on peut conclure que la solution est performante autrement dit PSO a réussi à trouver la solution optimale en un temps réduit.

Mais elle est moins performante par rapport à l'algorithme génétique en termes de temps d'exécution

## 5 CONCLUSION

### 5.1 Etude Comparative :

La comparaison se fait généralement par rapport à la performance de la solution qui inclut la qualité de la solution et son temps d'exécution, ci-dessous le tableau comparatif contenant la performance de chaque solution :

	Qualité de la solution	Temps d'exécution
BFS	100%	200 à 300 sec
DFS	100%	50 à 100 sec
A*	100%	40 à 70 sec
L'AG	100%	5 à 10 sec
PSO	100%	9 à 15 sec

D'après le tableau comparatif on remarque que toutes les méthodes ont réussi à trouver la solution optimale (qualité de solution = 100%), maintenant pour faire une comparaison entre ces méthodes il faut analyser le temps d'exécution de chacune, « la méthode qui a trouvera la solution optimale en un temps minimum(le plus petit) donc c'est la meilleure méthode à utiliser pour résoudre notre problème SAT »

### 5.2 Analyse des résultats :

1-->> Après l'analyse des résultats sur les méthodes heuristiques, on constate que l'algorithme A\* est plus performant que DFS qui est à son tour plus performant que le BFS

**Conclusion 1** : les méthodes informées sont plus performantes que les méthodes non informées (aveugles) pour notre problème SAT.

2-->> Après l'analyse des résultats sur les méthodes méta heuristiques, on constate que l'algorithme génétique (AG) est plus performant que PSO

**Conclusion 2** : l'Algorithme génétique est plus performante que PSO pour notre problème SAT.

3-->> Les méthodes heuristiques (BFS, DFS, A\*) ont arrivé à trouver la solution optimale mais cela a pris beaucoup de temps lors de l'exécution (méthodes lentes entre 40 à 300 sec), ainsi que d'espace mémoire. Contrairement aux méta-heuristiques elles ont réussi à trouver la solution optimale en un temps très réduit par rapport aux méthodes heuristiques (entre 5 à 15 secondes).

**Conclusion 3** : les méthodes méta heuristiques sont plus performantes que les méthodes heuristiques pour notre problème SAT

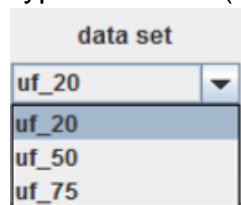
**PS :** (Parfois les méthodes méta heuristiques n'arrivent pas à trouver la solution optimale, mais elles résolurent quand même le temps d'un temps exponentiel à un temps polynomial).

### 5.3 Interface : L'interface inclut 3 parties

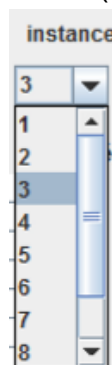
- >> Partie des inputs
- >> Partie Réglage des paramètres
- >> Partie des outputs

1- **Partie des inputs** : L'utilisateur de l'interface doit choisir :

A) Un dataset ou type du fichier (uf20, uf50, uf75).



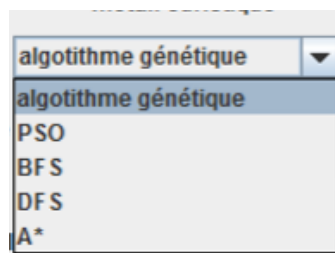
B) Une instance du dataset (1, 2, .....,1000).



Une fois ces deux derniers choisis (dataset, instance), les clauses de l'instances vont être affichés dans l'interface

clauses		
0	8	1
-13	16	-10
13	12	2
-20	-12	11
16	-17	11
10	19	-9
17	-1	9
7	-11	2
-8	-17	-19
-3	-12	-8
1	18	9
1	7	12
14	-16	3
10	9	-8

### C) La méthode (BFS, DFS, A\*, AG, PSO)



#### 2- Partie réglage des paramètres :

Pour l'interface de l'AG on a 5 champs qui se remplissent automatiquement, les 4 premiers champs correspondent aux paramètres de l'AG et le 5<sup>ème</sup> c'est le taux de stagnation.

Pour l'interface de PSO on a 6 champs qui se remplissent automatiquement, les 5 premiers champs correspondent aux paramètres de PSO et le 6<sup>ème</sup> c'est le taux de stagnation.

Pour l'interface des méthodes heuristiques, cette partie n'existe pas.

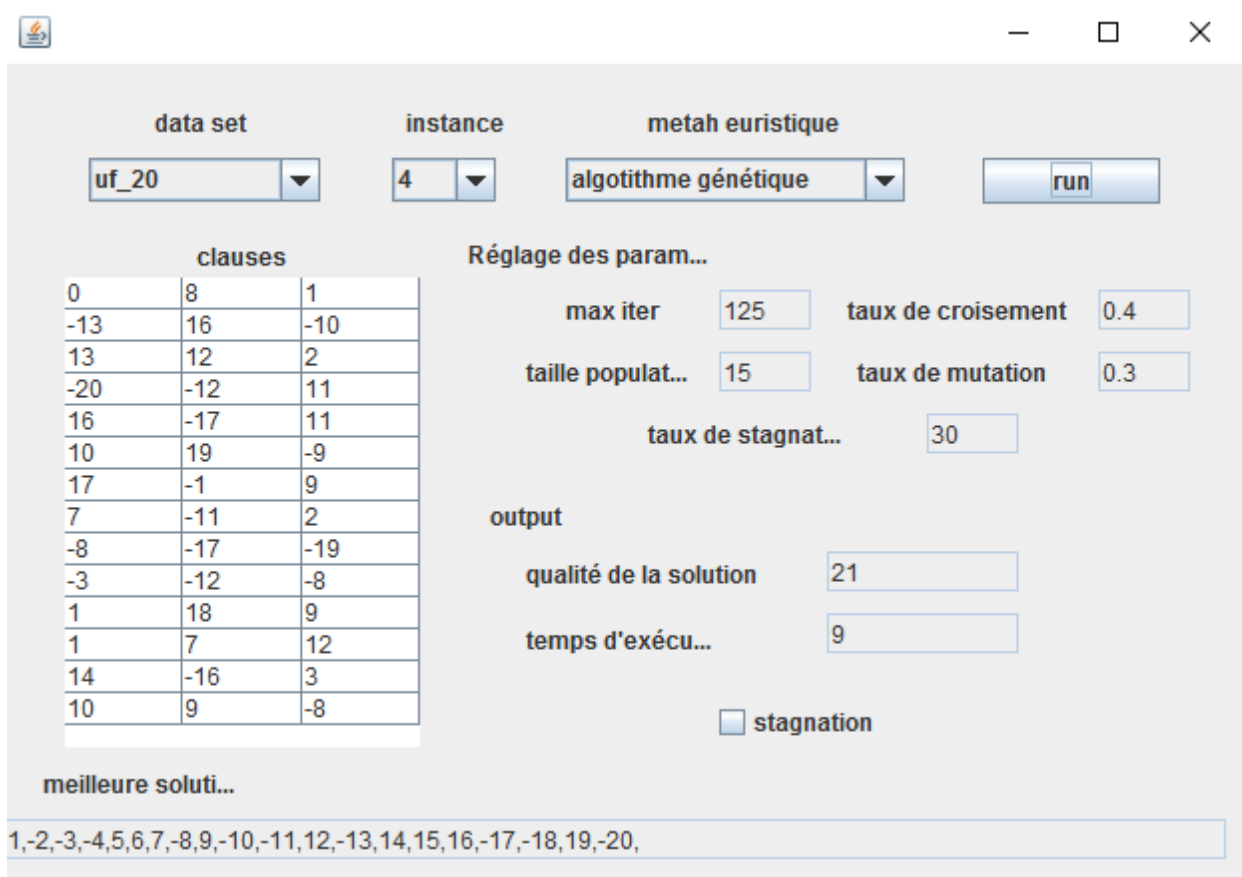
#### 3- Partie des outputs :

Cette partie qui donne les résultats, elle contient 4 champs :

- >> Un champ qui retourne la qualité de la solution (Nombre de clauses satisfaites).
- >> Un champ qui retourne le temps d'exécution.
- >> Une case à cocher qui informe s'il y'a une stagnation ou pas.

- >> Un champ qui affiche la meilleure solution.

Ci-dessous l'interface complète de l'AG :



data set: uf\_20

instance: 4

metah euristique: algorithme génétique

run

clauses

0	8	1
-13	16	-10
13	12	2
-20	-12	11
16	-17	11
10	19	-9
17	-1	9
7	-11	2
-8	-17	-19
-3	-12	-8
1	18	9
1	7	12
14	-16	3
10	9	-8

Réglage des param...

max iter: 125

taux de croisement: 0.4

taille populat...: 15

taux de mutation: 0.3

taux de stagnat...: 30

output

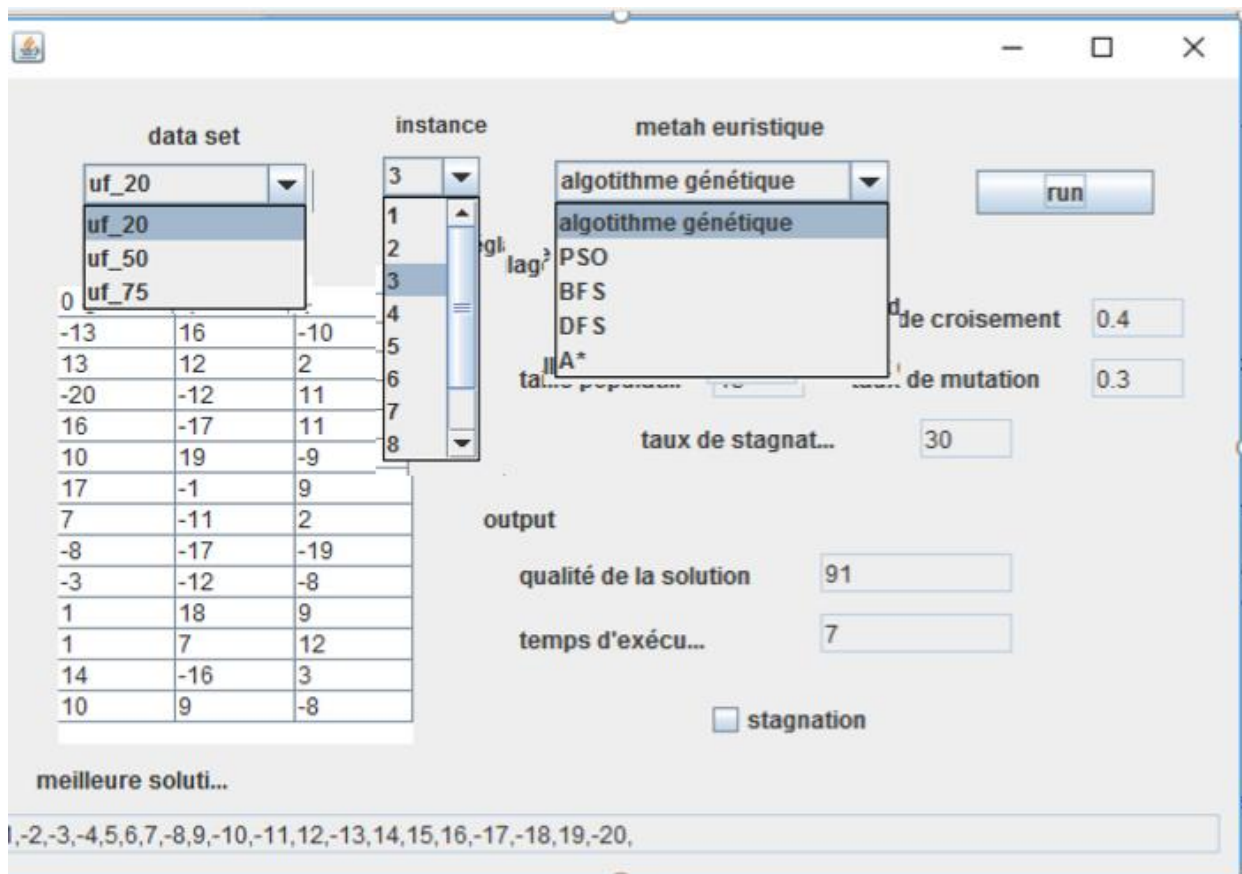
qualité de la solution: 21

temps d'exécu...: 9

☐ stagnation

meilleure soluti...

1,-2,-3,-4,5,6,7,-8,9,-10,-11,12,-13,14,15,16,-17,-18,19,-20,



data set: uf\_20

instance: 3

metah euristique: algorithme génétique

run

clauses

0	8	1
-13	16	-10
13	12	2
-20	-12	11
16	-17	11
10	19	-9
17	-1	9
7	-11	2
-8	-17	-19
-3	-12	-8
1	18	9
1	7	12
14	-16	3
10	9	-8

Réglage des param...

max iter: 125

taux de croisement: 0.4

taille populat...: 15

taux de mutation: 0.3

taux de stagnat...: 30

output

qualité de la solution: 91

temps d'exécu...: 7

☐ stagnation

meilleure soluti...

1,-2,-3,-4,5,6,7,-8,9,-10,-11,12,-13,14,15,16,-17,-18,19,-20,

#### 5.4 Récapitulatif et objectif du projet :

Dans notre travail, nous avons commencé par une présentation du problème SAT ainsi que sa modélisation et sa conception, puis nous avons détaillé l'algorithme que nous avons utilisé pour la mise en œuvre du BFS, DFS, A\*, l'AG et PSO. En vrai l'objectif de ce travail était d'adopter deux méta heuristiques qui sont le l'GA et PSO pour le problème SAT afin de comparer les résultats aux heuristique ainsi que la représentation de solutions, les politiques de générations, les notions d'opérateurs variés...

Pour valider notre approche, nous avons accompagné notre implémentation d'une série de jeux de tests de paramètres à l'aide des benchmark, les résultats obtenus ont été globalement satisfaisants en termes de qualité et du temps d'exécution.

---

*FON*





