

République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

**Université Benyoucef BENKHEDDA- Alger1**

Faculté des Sciences

Département **Mathématiques et Informatique**



**Méthodes Bio-Inspirées**

---

# Implémentation d'un solveur SAT

---

**Réalisé par**

Benbaba Rym Amina

Dahdouh Ahmed

2019/2020

## Table des matières

<b>Introduction .....</b>	<b>5</b>
<b>Protocole.....</b>	<b>5</b>
<b>Entrée .....</b>	<b>5</b>
<b>Sortie.....</b>	<b>6</b>
<b>Mise en place.....</b>	<b>7</b>
<b>L'environnement de réalisation .....</b>	<b>7</b>
Les outils matériels.....	7
Choix de langage de programmation.....	7
Les outils logiciels.....	7
<b>Structure de données.....</b>	<b>8</b>
<b>Littéral.....</b>	<b>8</b>
<b>Clause .....</b>	<b>8</b>
<b>Satisfaite .....</b>	<b>8</b>
<b>Les algorithmes de résolution.....</b>	<b>10</b>
Partie 01- Les méthodes aveugles et heuristiques .....	10
Partie 02 – Les algorithmes génétiques .....	29
Partie 03-L'algorithme ACS.....	41
Comparaison des résultats de trois parties.....	49
<b>Conclusion.....</b>	<b>49</b>

## Liste des figures

Figure 1-format DIMACS CNF .....	6
Figure 2-class Litteral.....	8
Figure 3-class Clause.....	8
Figure 4-class Satisfaite.....	9
Figure 5-les lignes ignorées.....	9
Figure 6-Vector clause .....	9
Figure 7-fonction main.....	9
Figure 8-Algo en largeur d'abord .....	11
Figure 9-nombre de clauses satisfaites .....	15
Figure 10-fonction de l'heuristique.....	15
Figure 11- Out Of Memory Error.....	20
Figure 12- Histogramme montrant les résultats du test BFS pour le problème SAT sur les fichiers de Benchmarks UF-75.....	21
Figure 13- Histogramme montrant les résultats du test BFS pour le problème SAT sur les fichiers de Benchmarks UUF-75.....	21
Figure 14-Comparaison avec BFS.....	22
Figure 15-Nombre de Clauses Satisfiables avec l'algorithme DFS .....	23
Figure 16-Figure 14-Histogramme montrant les résultats du test DFS pour le problème SAT sur l'instance de Benchmarks UF75-01 .....	24
Figure 17-Nombre de Clauses Satisfiables avec l'algorithme A* avec la 1ere heuristique .....	25
Figure 18-Nombre de Clauses Satisfiables avec l'algorithme A* avec la 2eme heuristique .....	26
Figure 19-Histogramme montrant les résultats du test A* avec la 1ere heuristique pour le problème SAT sur l'instance de Benchmarks UF75-01 .....	27
Figure 20-Histogramme montrant les résultats du test A* avec la 2eme heuristique pour le problème SAT sur l'instance de Benchmarks UF75-01 .....	27
Figure 21-Comparaison avec secteur entre les algorithmes de partie 01 dans les premiers 5minutes ..	28
Figure 22-Comparaison avec secteur entre les algorithmes de partie 01 après 30minutes d'exécutions .....	29
Figure 23-Principe général des algorithmes génétiques .....	30
Figure 24-Fonction d'initialisation d'une population.....	31
Figure 25- La fonction fitness1 .....	32
Figure 26-La fonction fitness2 .....	32
Figure 27-trier la matrice de population selon les valeurs de fitness .....	33
Figure 28- croisement à 1 point.....	33
Figure 29-croisement à 2 points .....	34
Figure 30-fonction de croisement .....	35
Figure 31Principe de l'opérateur de mutation.....	35
Figure 32-Appel à la fonction mutation x fois .....	36
Figure 33-fonction de mutation .....	36
Figure 34-Interface graphique de GA.....	39
Figure 35-GA tous les benchmarks résultats.....	40
Figure 36-ACS initialisation depuis l'interface graphique.....	41
Figure 37-ACS nombre de fourmis .....	41
Figure 38-ACS initialiser les fourmis.....	42
Figure 39-ACS nombre des itérations .....	42
Figure 40-ACS critère d'arrêt .....	42

Figure 41-ACS pour chaque fourmi.....	42
Figure 42-ACS Construction et évaluation de solution e chaque fourmi.....	42
Figure 43-ACS online delayed update .....	43
Figure 44-ACS obtenir meilleure solution des fourmis.....	43
Figure 45-ACS meilleure fourmi.....	43
Figure 46-ACS Offline delayed update .....	44
Figure 47-ACS Algorithme de construction de solutions .....	44
Figure 48-L'interface graphique de ACS.....	45
Figure 49-ACS Nombre de clauses satisfaites sur l'instance UF75.....	48
Figure 50-ACS Nombre de clauses satisfaites sur l'instance UUF75.....	48
Figure 51-Comparaison.....	49

Tableau 1- : Les outils matériels .....	7
Tableau 2-complexité des algorithmes.....	20
Tableau 3-Statistiques BFS .....	20
Tableau 4-Résumé de statistiques de BFS.....	21
Tableau 5-Statistiques DFS pour UF75-01 .....	23
Tableau 6-Statistiques A* pour l'instance UF75-01 .....	26
Tableau 7-Comparaison entre les algorithmes de partie 01 .....	28
Tableau 8-AG modifier taille de population .....	37
Tableau 9-AG modifier nombre d'itérations.....	37
Tableau 10- AG modifier taux de mutation .....	38
Tableau 11-GA tous les Benchlarks .....	39
Tableau 12-ACS résultats avec 10 fourmis.....	45
Tableau 13-ACS résultats avec 30 fourmis .....	45
Tableau 14-ACS résultats avec 40 fourmis .....	46
Tableau 15-ACS résultats avec 50 fourmis .....	46
Tableau 16-ACS résultats avec 60 fourmis .....	46
Tableau 17-ACS résultats avec 70 fourmis .....	47
Tableau 18-ACS tous les Benchlarks .....	47

# Introduction

Le problème SAT, problème de décision qui consiste à déterminer si une formule propositionnelle mise sous forme normale conjonctive (CNF), possède une évaluation vraie (satisfiable ou non), est le premier problème à avoir été montré NP-complet pour la classe NP (Non déterministe Polynomial) (Cook (1971)), il occupe un rôle central en théorie de la complexité. Depuis plusieurs années, de nombreux algorithmes ont été proposés pour résoudre ce problème, et avec l'avènement des solveurs SAT modernes des instances industrielles de centaines de milliers de variables et de millions de clauses peuvent être résolues en un temps très réduit (quelques minutes).

Cependant, bien que tous les algorithmes connus soient équivalents asymptotiquement dans le pire cas, certains algorithmes sont en pratique beaucoup plus efficaces que d'autres. Le problème SAT revêt une grande importance industrielle : il est utilisé, par exemple, pour vérifier la correction de circuits électroniques vis-à-vis de leur spécification. De ce fait, des algorithmes très efficaces ont été développés, qui sont parfois capables de traiter des problèmes de très grande taille.

Ce sujet est réellement difficile. Certes, le problème est simple, mais les techniques de programmation utilisées pour obtenir un algorithme efficace ne le sont pas. Nous allons donc au cours de ce projet relever le défi de résoudre le problème SAT.

On cherche ici à comprendre les principales idées qui ont permis ces gains d'efficacité. On commencera donc par implémenter un algorithme naïf, puis on introduira différentes optimisations.

**L'objectif de ce projet est d'implémenter un solveur SAT relativement efficace.**

## Protocole

On commence par décrire la spécification de l'algorithme, ou, en d'autres termes, la mission que le programme SAT devra remplir.

### Entrée

Le solveur devra accepter, sur le canal d'entrée standard, la description du problème, c'est-à-dire une formule en forme normale conjonctive, au format DIMACS CNF.

La figure ci-dessous représente un extrait de fichier benchmark.

```

1 c This Formular is generated by mcnf
2 c
3 c     horn? no
4 c     forced? no
5 c     mixed sat? no
6 c     clause length = 3
7 c
8 p cnf 75 325
9 42 22 15 0
10 73 -22 -24 0
11 50 -20 53 0
12 -66 49 -15 0

```

Figure 1-format DIMACS CNF

Un fichier au format DIMACS CNF (est pris en charge par presque tous les solveurs SAT) est une disjonction de littéraux ; et qu'un littéral est soit une variable  $x$  qui est représentée par un entier compris entre 1 et  $n$ . La négation  $\neg$  est représentée par le signe  $-$ .

Une clause est représentée comme une liste de littéraux, séparés par des espaces, et terminée par un 0. Un problème est représenté comme une succession de clauses.

Les commentaires, sont signalés par un caractère  $c$ .

## Sortie

Maintenant, nous pouvons voir ce que fait réellement un solveur SAT. Il prend une expression booléenne détermine si elle est satisfaisante ou insatisfaisante.

- **Satisfiable** : Si les variables booléennes peuvent être affectées à des valeurs telles que la formule se révèle être VRAIE, alors nous disons que la formule est satisfaisable.
- **Non satisfiable** : s'il n'est pas possible d'attribuer de telles valeurs, alors nous disons que la formule n'est pas satisfaisante.

# Mise en place

Comme dit précédemment, on s'intéresse à la satisfiabilité de formules écrites sous une forme normalisée (CNF), qui est une conjonction de clauses, c'est à dire une conjonction de disjonctions de littéraux. Dans un premier temps, nous allons définir la structure de données permettant de représenter ces formules, et écrire un premier programme qui lit une telle formule dans un fichier et la stocke dans cette structure de donnée.

## L'environnement de réalisation



Pour la réalisation de notre solveur, nous avons eu recours à plusieurs moyens matériels et logiciels :

### Les outils matériels

Le développement de notre solveur est réalisé via un ordinateur portable ayant les caractéristiques présenter dans le tableau ci-dessous :

<b>Marque</b>	<b>Ordinateur portable ASUS</b>
<b>Processeur</b>	Intel® Core™ i5-5200U
<b>RAM</b>	4,00 Go
<b>Disque dur</b>	200 Go
<b>Système d'exploitation</b>	Windows 10 professionnel

Tableau 1- : Les outils matériels

Choix de langage de programmation	Les outils logiciels
	
Java est un langage de programmation orienté objet, créé par James Gosling et Patrick Naughton, présenté officiellement le 23 mai 1995 au <i>SunWorld</i> .	Eclipse IDE est un environnement de développement intégré libre, qui facilite la création, la compilation, les tests et l'exécution de projets java pouvant contenir de nombreux fichiers de code, repartis dans

<p>[2] Java permet de mieux structurer une application, offre une meilleure réutilisation du code, ainsi c'est un langage qu'on le maîtrise bien.</p>	<p>différents paquetages et référençant de nombreuses librairies.</p>
---	---

## Structure de données

Dans un premier temps, il faut construire la structure de données permettant de représenter les formules logiques.

### Littéral

Un littéral est défini par deux champs entiers indiquant le numéro de variable et une spécifiant si on considère la variable ou sa négation. Nous numérotions les variables de 1 à N (dans notre cas c'est 75), comme le montre la figure ci-dessous.

```
public class Litteral {

    //déclaration
    private int Numvar;
    private int valeur;
}
```

Figure 2-class Litteral

### Clause

Une clause est définie par un tableau de littéraux. Elle représente la disjonction des littéraux contenus dans le tableau, (*littéral [0]  $\vee$  littéral [1]  $\vee$  ...  $\vee$  littéral [littéral. Length-1]*).

```
//clause est un ensemble des littéraux
public class Clause implements Cloneable {
    //déclaration
    private Vector <Litteral> Litteraux; /
```

Figure 3-class Clause

### Satisfaite

Une formule est définie par un tableau de clauses. Pour des raisons de simplicités nous avons ajoutés un tableau de littéraux



```

public class Satisfaite implements Cloneable {

    //déclarations
    Vector<Clause>   Clauses;
    Vector<Litteral> Litteraux;

```

Figure 4-class Satisfaite

Après avoir défini la structure de base de notre solution, passant maintenant à la fonction main ou nous allons lire le contenu de fichier, nous pouvons simplement ignorer les lignes commençant par (p, c, % et 0) avec tous les paramètres et analyser simplement tous les nombres.

```

if(ligne.startsWith("c")||ligne.startsWith("p")||ligne.startsWith("%")||ligne.startsWith("0")){
    //System.out.println("a sauter");
    //on fait rien duuuuuu rien
}

```

Figure 5-les lignes ignorées

Une clause commence par un littéral, qui sont séparés par des espaces, et elle se termine par un 0.

Le fichier dans son ensemble représente une série de clauses, donc après avoir récupéré les littéraux nous allons les stocker dans le vecteur clause.

```

clauses.add(new Clause(litteraux));

```

Figure 6-Vector clause

Ensuite nous allons mélanger les valeurs de vecteur des littéraux avec la fonction Random, et l'ajouter avec les clauses dans la formule satisfaite comme suite :

```

//tq la liste nest pas vide
while(ListeLitteral.size()>0)
{
    randomVar = Litteral.random(ListeLitteral); // une valeur random de ListeLitteral
    contraire = Litteral.Negation(ListeLitteral, randomVar); //si il onr des valeurs contaires sinn null
    Litteral.SuppriemLitteral(ListeLitteral, randomVar); //supp randomVar de ListeLitteral
    RandomLitteraux.add(0,randomVar); //a l'index 0 ajouter la valeur de randomLitteral
    RandomLitteraux.add(RandomLitteraux.size(), contraire);
}

ListeLitteral = RandomLitteraux;

//declarer objet Sat en utilisant les deux vcteurs precidents
Satisfaite sat = new Satisfaite(clauses, ListeLitteral);

```

Figure 7-fonction main

## Les algorithmes de résolution

Dans cette partie, nous allons implémenter les différents algorithmes de résolution du problème, par exploration de l'ensemble des choix possibles. Une formule logique est satisfiable s'il est possible d'associer une valeur logique booléenne à chacune de ses variables, de manière à ce que cette formule soit vraie.

Notre solveur doit vérifier si une formule est satisfiable. Si elle l'est, il doit afficher sur une ligne *SAT*. Si la formule n'est pas satisfiable, il doit afficher le mot *UNSAT*.

On distingue deux classes d'algorithmes de recherche :

## Partie 01- Les méthodes aveugles et heuristiques

### Méthodes aveugles (non-informés)

Algorithmes qui réalisent une recherche exhaustive (brute force), sans utiliser aucune information concernant la structure de l'espace d'états pour optimiser la recherche tel que :

- Recherche en largeur
- Recherche en coût uniforme
- Recherche en profondeur
- Recherche en profondeur limitée
- Recherche par approfondissement itératif
- Recherche bidirectionnelle

### Recherche en largeur d'abord ou BFS (Bredth First Search)

Cet algorithme consiste à Explorer tous les fils d'un nœud d'abord

- Étendre le nœud le moins profond

L'algorithme donc :

- Parcourt l'arbre de recherche couche par couche
- S'arrête dès la rencontre d'un **état but**
- Au cas où la solution n'est pas trouvée, le processus est itéré au niveau de profondeur suivant.

Cette stratégie garantie de trouver une solution si elle existe, mais elle exige beaucoup de mémoire pour stocker toutes les alternatives à toutes les couches.

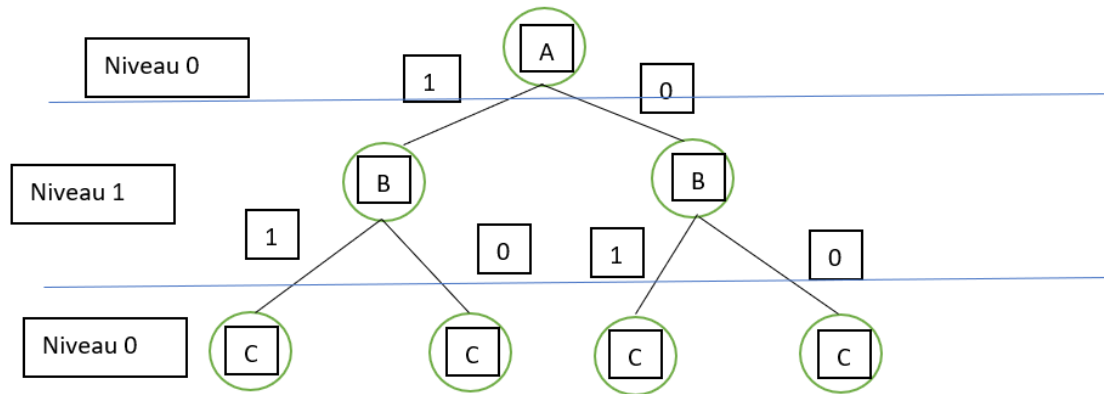


Figure 8-Algo en largeur d'abord

L'Algorithme de résolution est le suivant :

**BFS ( FORMULE sat, FILE Ouvert, FILE Ferme)**

**Entrée :** Liste non triée des littéraux initialisés avec (-1)

**Variables :**

- Littéral (Nom, Valeur)
- Clause vecteur de littéraux [3]
- Sat vecteur de clauses [325]

### **DEBUT**

Ajouter Sat a la liste Ouvert ;

Si (Ouvert = vide) alors échec ;

**Tant que** (Ouvert <> vide) **faire**

    Père= dernier élément d'Ouvert ;

    Insérer Père dans Fermé ;

**Si** (Père est satisfiable) **alors** Ecrire « YES SAT » et EXIT ;

**Sinon** (si Père.profondeur <75) **alors**

        Fils1 =Clone (Père)

        Fils2 =Clone (Père)

        Fils1.profondeur +1 ;

        Fils2.profondeur +1 ;

        Fils1.Littéral[Fils1.profondeur]=1 ; //pour la valeur contraire c'est 0

        Fils2.Littéral[Fils2.profondeur]=0 ; // pour la valeur contraire c'est 1

**Si** Fils1 existe pas dans Fermé **alors**

Ajouter Fils1 à Ouvert ;

**FSi ;**

**Si** Fils2 existe pas dans Fermé **alors**

Ajouter Fils2 à Ouvert ;

**FSi ;**

**FSi ;**

**Faite ;**

**FIN**

#### Recherche en prfondeur d'abord ou DFS (Depth First Search)

Cet algorithme consiste à explorer le premier nœud trouvé jusqu'à ce qu'il n'y ait plus de successeurs avec retour en arrière (backtrack) à un niveau supérieur et essai de la prochaine possibilité

L'algorithme donc :

- Étendre le nœud le plus profond
- Dans l'exploration, l'algorithme cherche à aller très vite "profondément" dans le graph, en s'éloignant du sommet s de départ.
- La recherche sélectionne à chaque étape un sommet voisin du sommet marqué à l'étape précédente.
- En cas d'échec (branche conduisant à un cas d'échec), on revient en arrière au niveau du père, et si possible, on recherche un autre de ses successeurs.

Cette stratégie consomme peu de mémoire par rapport BFS : liste des nœuds "ouverts" et de leurs successeurs encore non explorés.

L'Algorithme de résolution est le suivant :

#### **DFS ( FORMULE sat, FILE Ouvert, FILE Ferme)**

**Entrée :** Liste non triée des littéraux initialisés avec (-1)

**Variables :**

- Littéral (Nom, Valeur)
- Clause vecteur de littéraux [3]
- Sat vecteur de clauses [325]

**DEBUT**

Ajouter Sat a la liste Ouvert ;

Si (Ouvert = vide) alors échec ;

**Tant que** (Ouvert  $\neq$  vide) **faire**

    Père= premier élément d'Ouvert ;

    Insérer Père dans Fermé ;

**Si** (Père est satisfiable) **alors** Ecrire « YES SAT » et EXIT ;

**Sinon** (si Père.profondeur  $< 75$ ) **alors**

        Fils1 =Clone (Père)

        Fils2 =Clone (Père)

        Fils1.profondeur +1 ;

        Fils2.profondeur +1 ;

        Fils1.Litteral[Fils1.profondeur]=1 ; //pour la valeur contraire c'est 0

        Fils2.Litteral[Fils2.profondeur]=0 ; // pour la valeur contraire c'est 0

**Si** Fils1 existe pas dans Fermé **alors**

        Ajouter Fils1 à Ouvert ;

**FSi** ;

**Si** Fils2 existe pas dans Fermé **alors**

        Ajouter Fils2 à Ouvert ;

**FSi** ;

**FSi** ;

**Faite** ;

**FIN**

## Méthodes heuristiques (informés)

Une heuristique est une technique dépendante du problème à traiter qui améliore l'efficacité d'un processus de recherche, en sacrifiant éventuellement la prétention à être complet.

- Pour des problèmes d'optimisation où la recherche d'une solution exacte (optimale) est difficile (coût exponentiel), on peut se contenter d'une solution satisfaisante donnée par une heuristique avec un coût plus faible.

Les algorithmes de recherche utilisant les heuristiques :

- Hill Climbing
- Beam Search
- Best First search
- Branch And Bound
- Branch and Bound avec sous estimations
- Branch and Bound avec programmation dynamique
- A\*

L'utilisation d'une heuristique diminue l'espace des états, accélère donc la recherche et permet d'aboutir plus rapidement à une solution, mais ne garantit pas que cette dernière est la plus optimale.

### L'algorithme A\*

C'est un algorithme de recherche ordonnée basé sur une fonction d'évaluation

**$F(n) = G(n) + H(n)$ .**

- $G(n)$  est le coût de la chaîne allant de  $s_0$  à  $n$
- $H(n)$  appelée heuristique est une estimation du coût de la chaîne reliant  $n$  à un nœud final.

La différence avec les autres stratégies réside dans le fait que la liste de littéraux n'est pas triée aléatoirement mais selon la fréquence des littéraux dans les clauses. A chaque développement d'un nœud, la variable ayant la plus grande fréquence est choisie afin de satisfaire le plus grand nombre de clauses.

C'est un algorithme simple qui ne consommant que peu de mémoire.

La fonction de coût qui calcul le nombre de clauses satisfaites est la suivante :

```

//fonction qui retourne nombre des clauses satisfaites
public int NombreClauseSat()
{
    int nombreClausesSat=0;
    for(int i =0; i<Clauses.size();i++) //normalmnt c 325
    {
        if(this.Clauses.get(i).EstSat()==true) nombreClausesSat++;
    }
    return nombreClausesSat;
}

```

Figure 9-nombre de clauses satisfaites

La fonction de l'heuristique qui calcul le nombre de clauses contenant la variable Xi (littéral) est :

```

//fonction qui retourne nombre de clause contenant un littéral L
public int NombreClauseOntLitteral (int k)
{
    int cpt = 0;
    for(int i=0; i<this.getClauses().size();i++)
    {
        for(int j=0;j<this.getClauses().get(i).getLitteraux().size();j++)
        {
            if(((this.Clauses.get(i).getLitteraux().get(j).getNumvar()==k) || (- (this.Clauses.get(i).getLitteraux().get(j).getNumvar()==k))
                cpt++;
            }
        }
    }
    return cpt;
}

```

Figure 10-fonction de l'heuristique

## Heuristique 01

Pour la première Heuristique nous avons choisi les paramètres suivants :

G(n) : Nombre de clauses satisfaites.

H(n) : Nombre clause contenant la variable Xi.

Par exemple si nous avons une instance I définit comme suit :

X= {x1, x2, x3}, un ensemble de variables booléennes.

C= {c1, c2, c3, c4}, un ensemble de clauses.

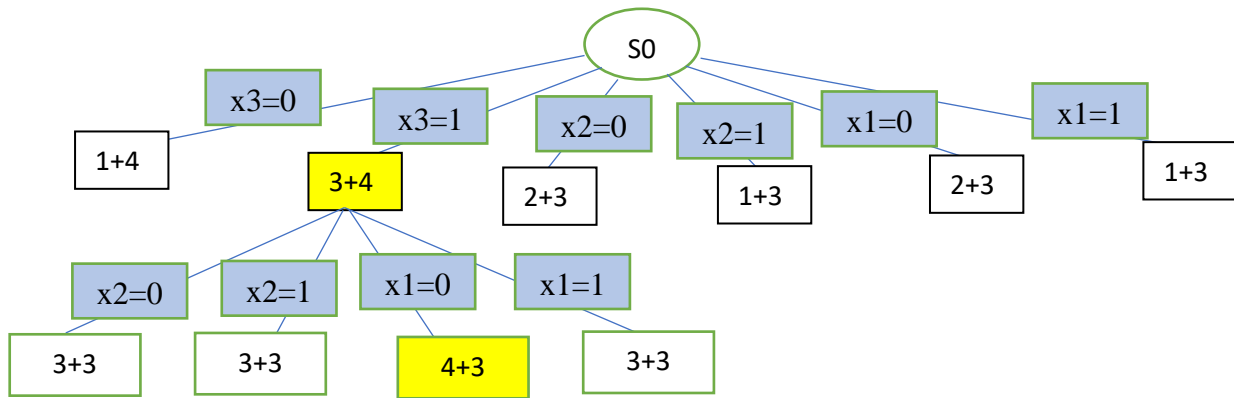
$$C1= x1 + (-x2) + x3$$

$$C2= x2 + x3$$

$$C3= (-x1) + (-x2) + x3$$

$$C4= (-x1) +(- x3)$$

Une solution peut-être :



Donc les instanciations de X pour que la conjonction des clauses de C est vraie est :

$\{s1=\{0,0,1\} \text{ et } s2=\{0,1,1\}\}$

Donc c'est un problème de maximisation dont L'Algorithme de résolution est le suivant :

### A\_ETOILE ( )

#### Entrée :

- Littéral (Nom, Valeur)
- Clause vecteur de littéraux [3]
- Sat vecteur de clauses [325]
- $G(x_i, x_j)$  : coût entre le noud  $x_i$  et  $x_j$ .
- $H(x)$  : heuristique.

**Sortie :** « YES SAT » si satisfaite, « UNSAT » sinon.

#### Variables :

- Ouvert : liste triée selon  $F(x)$  initialisée vide.
- Fermé : liste vide.

### DEBUT

Tab[] = tableau contenant le nombre d'occurrence de chaque littéral selon l'indice+1

Ajouter Sat a la liste Ouvert ;

Si (Ouvert = vide) alors échec ;

**Tant que** (Ouvert  $\neq$  vide) **faire**

Père= (max nombre de clause SAT) élément d'Ouvert ;



Insérer Père dans Fermé ;

**Si** (Père est satisfiable) **alors** Ecrire « YES SAT » et EXIT ;

**Sinon** (si Père.profondeur <75) **alors**

Fils1 =Clone (Père)

Fils2 =Clone (Père)

Fils1.profondeur +1 ;

Fils2.profondeur +1 ;

Fils1.Litteral[Fils1.profondeur]=1 ; //pour la valeur contraire c'est 0

Fils2.Litteral[Fils2.profondeur]=0 ; // pour la valeur contraire c'est 0

**Si** Fils1 existe pas dans Fermé **alors**

Ajouter Fils1 à Ouvert ;

**FSi** ;

**Si** Fils2 existe pas dans Fermé **alors**

Ajouter Fils2 à Ouvert ;

**FSi** ;

**FSi** ;

**Faite** ;

**FIN**

Heuristique 02 :

Pour la deuxième Heuristique nous avons choisi les paramètres suivants :

G(n) : Nombre de clauses non satisfaites.

H(n) : Nombre clause qui ne contient pas la variable Xi.

Par exemple si nous avons une instance I définit comme suit :

X= {x1, x2, x3}, un ensemble de variables booléennes.

C= {c1, c2, c3, c4}, un ensemble de clauses.

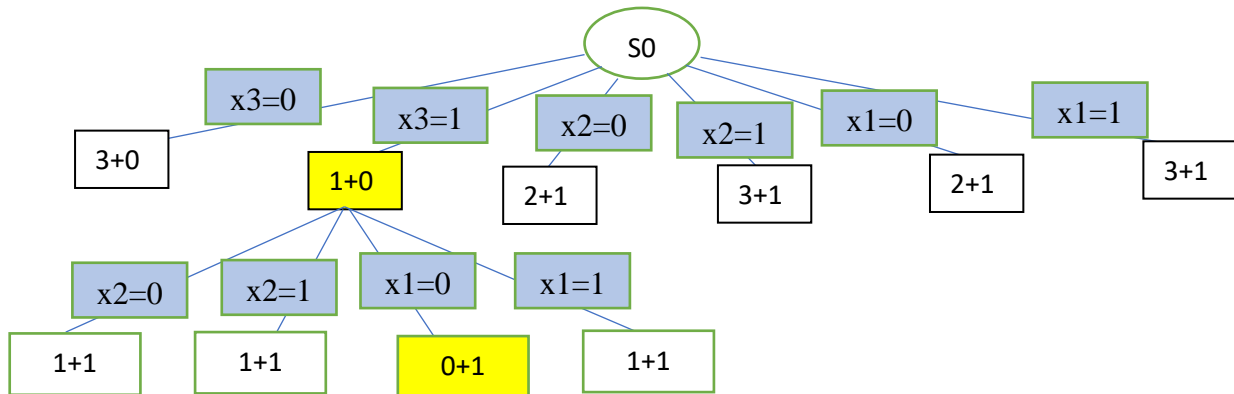
C1= x1 + (-x2) + x3

C2= x2 + x3

C3= (-x1) + (-x2) + x3

$$C4 = (-x1) + (-x3)$$

Une solution peut-être :



Donc les instanciations de X pour que la conjonction des clauses de C est vraie est :

$\{s1=\{0,0,1\} \text{ et } s2=\{0,1,1\}\}$

Donc c'est un problème de minimisation dont L'Algorithme de résolution est le suivant :

**A\_ETOILE ( )**

**Entrée :**

- Littéral (Nom, Valeur)
- Clause vecteur de littéraux [3]
- Sat vecteur de clauses [325]
- $G(x_i, x_j)$  : coût entre le noud  $x_i$  et  $x_j$ .
- $H(x)$  : heuristique.

**Sortie :** « YES SAT » si satisfaite, « UNSAT » sinon.

**Variables :**

- Ouvert : liste triée selon  $F(x)$  initialisée vide.
- Fermé : liste vide.

## **DEBUT**

Tab [] = tableau contenant le nombre d'occurrence de chaque littéral selon l'indice+1

Ajouter Sat a la liste Ouvert ;

Si (Ouvert = vide) alors échec ;

**Tant que** (Ouvert  $\neq$  vide) **faire**

Père= (min nombre de clause non SAT) élément d'Ouvert ;

Insérer Père dans Fermé ;

**Si** (Père est satisfiable) **alors** Ecrire « YES SAT » et EXIT ;

**Sinon** (si Père.profondeur <75) **alors**

Fils1 =Clone (Père)

Fils2 =Clone (Père)

Fils1.profondeur +1 ;

Fils2.profondeur +1 ;

Fils1.Litteral[Fils1.profondeur]=1 ; //pour la valeur contraire c'est 0

Fils2.Litteral[Fils2.profondeur]=0 ; // pour la valeur contraire c'est 0

**Si** Fils1 existe pas dans Fermé **alors**

Ajouter Fils1 à Ouvert ;

**FSi** ;

**Si** Fils2 existe pas dans Fermé **alors**

Ajouter Fils2 à Ouvert ;

**FSi** ;

**FSi** ;

**Faite** ;

**FIN**

### Comparaison

La complexité spatiale et temporelle au pire cas est :

Soit :

B : Le nombre maximum de successeurs d'un nœud.

D : la profondeur du nœud but le moins profond.

M : la profondeur maximale de l'espace de recherche.

Algorithme	BFS	DFS	A* heuristique 1	A* heuristique 2
Complexité				
En temps	$O(B^D) = (2^{75})$	$O(B^M) = (2^{75})$	$(2^{75})$	$(2^{75})$
En espace	$O(B^D) = (2^{75})$	$O(B^M) = (2^{75})$	$(2^{75})$	$(2^{75})$

Tableau 2-complexité des algorithmes

Pour analyser les résultats obtenus de l'exécution des trois algorithmes implémentés sur les benchmarks donnés, Nous allons calculer le taux de satisfiabilité de chaque instance de benchmarks dans des laps de temps différents t comparer les résultats par la suite.

#### Statistiques de BFS

Il est à noter que l'exécution n'a pas pu atteindre les 15minutes. Au bout de quelques minutes (envers 11min) le programme s'arrête, affichant une insuffisance d'espace mémoire (OutOfMemory Error), comme le montre la figure ci-dessous.

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at java.util.Vector.<init>(Unknown Source)
  at java.util.Vector.<init>(Unknown Source)
  at java.util.Vector.<init>(Unknown Source)
  at alger1.isii.algo.BFS.BFS_Algo(BFS.java:78)
  at alger1.isii.mainSat.Main.main(Main.java:209)
```

Figure 11- Out Of Memory Error

Le tableau ci-dessous décrit le taux de satisfiabilité maximale pour chaque instance des benchmarks donnés pendant 5 et 10 minutes d'exécution :

Benchmarks	Taux de satisfiabilité	En 5min	En 10min	En 15min	En 20min
	Instance				
uf75	01	86 → 26.46%	98 → 30.15%	/	/
	02	91 → 28%	91 → 28%	/	/
	03	92 → 28.30%	94 → 28.92%	/	/
	04	97 → 29.84%	98 → 30.15%	/	/
	05	84 → 25.84%	91 → 28%	/	/
	06	100 → 30.76%	105 → 32.30%	/	/
	07	92 → 28.30%	94 → 28.92%	/	/
	08	101 → 30.07%	101 → 30.07%	/	/
	09	79 → 24.03%	82 → 25.23%	/	/
	10	83 → 23.53%	94 → 28.92%	/	/
uuf75	01	98 → 30.15%	100 → 30.76%	/	/
	02	82 → 25.23%	83 → 23.53%	/	/
	03	90 → 27.69%	90 → 27.69%	/	/
	04	101 → 30.07%	105 → 32.30%	/	/
	05	83 → 23.53%	91 → 28%	/	/
	06	79 → 24.03%	82 → 25.23%	/	/
	07	94 → 28.92%	94 → 28.92%	/	/
	08	92 → 28.30%	98 → 30.15%	/	/
	09	84 → 25.84%	91 → 28%	/	/
	10	98 → 30.15%	100 → 30.76%	/	/

Tableau 3-Statistiques BFS

Comme rien ne vaut un simple schéma, les voici :

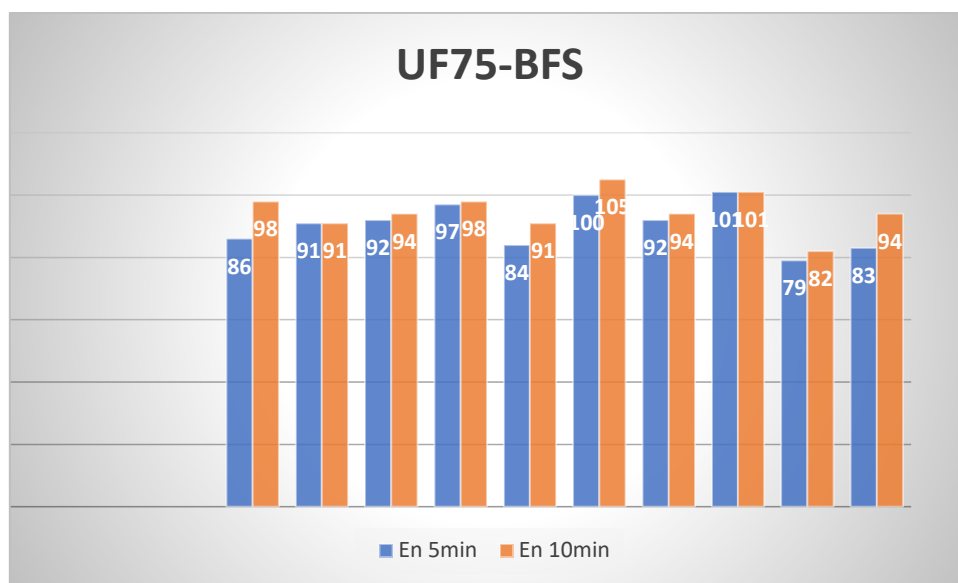


Figure 12- Histogramme montrant les résultats du test BFS pour le problème SAT sur les fichiers de Benchmarks UF-75

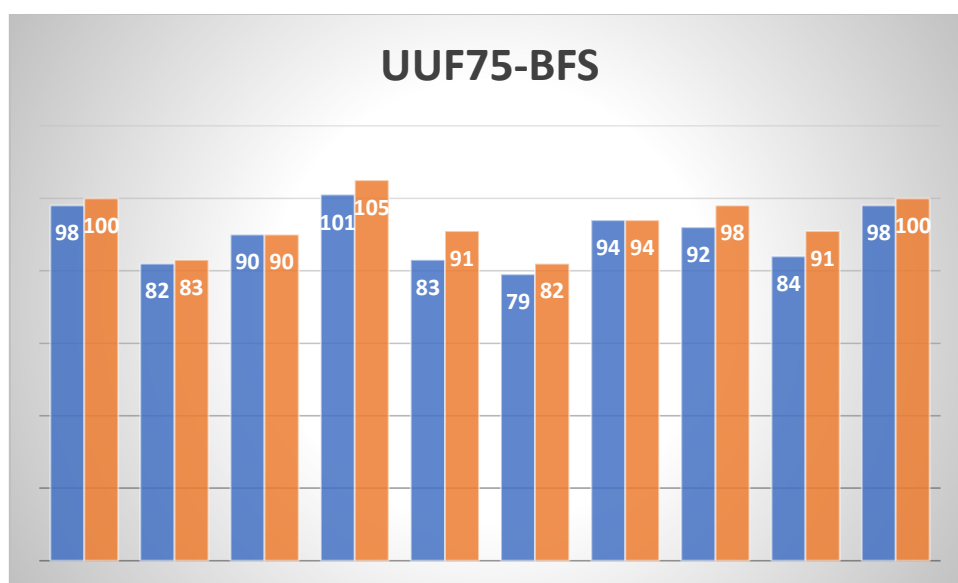


Figure 13- Histogramme montrant les résultats du test BFS pour le problème SAT sur les fichiers de Benchmarks UUF-75

Benchmark	Taux en 5min	Taux en 10min
<b>uf75</b>	27.84%	29.17%
<b>uuf75</b>	27.72%	28.73%

Tableau 4-Résumé de statistiques de BFS

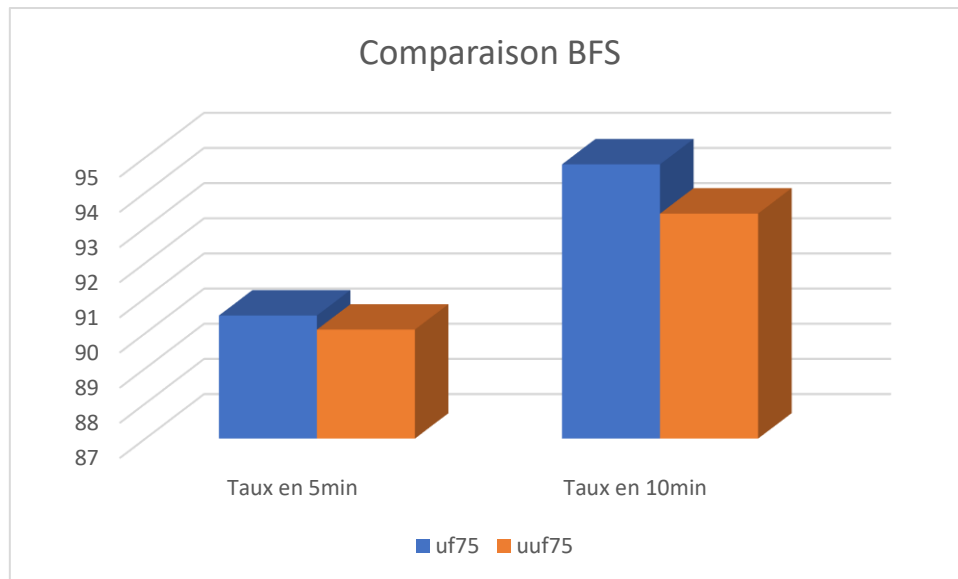


Figure 14-Comparaison avec BFS

#### Statistiques de DFS

Avec cet algorithme de profondeur d'abord, nous allons augmenter le temps d'exécution et utiliser une seule instance (UF75-01) afin d'obtenir le meilleur résultat possible tant qu'il n'y a pas le problème de la saturation de mémoire.

L'image ci-dessous décrit le nombre maximal des clauses satisfaites pendant l'exécution de l'algorithme DFS pendant 30minutes :

```

nombre de clauses satisfaites 204
nombre de clauses satisfaites 210
nombre de clauses satisfaites 218
nombre de clauses satisfaites 234
nombre de clauses satisfaites 249
nombre de clauses satisfaites 251
nombre de clauses satisfaites 254
nombre de clauses satisfaites 262
nombre de clauses satisfaites 267
nombre de clauses satisfaites 275
nombre de clauses satisfaites 277
nombre de clauses satisfaites 279
nombre de clauses satisfaites 282
nombre de clauses satisfaites 286
nombre de clauses satisfaites 291
nombre de clauses satisfaites 293
nombre de clauses satisfaites 294
nombre de clauses satisfaites 297
nombre de clauses satisfaites 298
nombre de clauses satisfaites 299

```

*Figure 15-Nombre de Clauses Satisfiables avec l'algorithme DFS*

Le tableau ci-dessous décrit le taux maximal de satisfiabilité pour l'instance (UF75-01) pendant 30 minutes d'exécution avec une vérification continue de résultats chaque 5 minutes :

	Uf75-01 avec DFS
Taux de satisfiabilité en 5min	275 → 84.61%
Taux de satisfiabilité en 10min	279 → 85.84%
Taux de satisfiabilité en 15min	286 → 88%
Taux de satisfiabilité en 20min	291 → 89.53%
Taux de satisfiabilité en 25min	294 → 90.46%
Taux de satisfiabilité en 30min	299 → 92%

*Tableau 5-Statistiques DFS pour UF75-01*

Présentant ces résultats avec un histogramme :

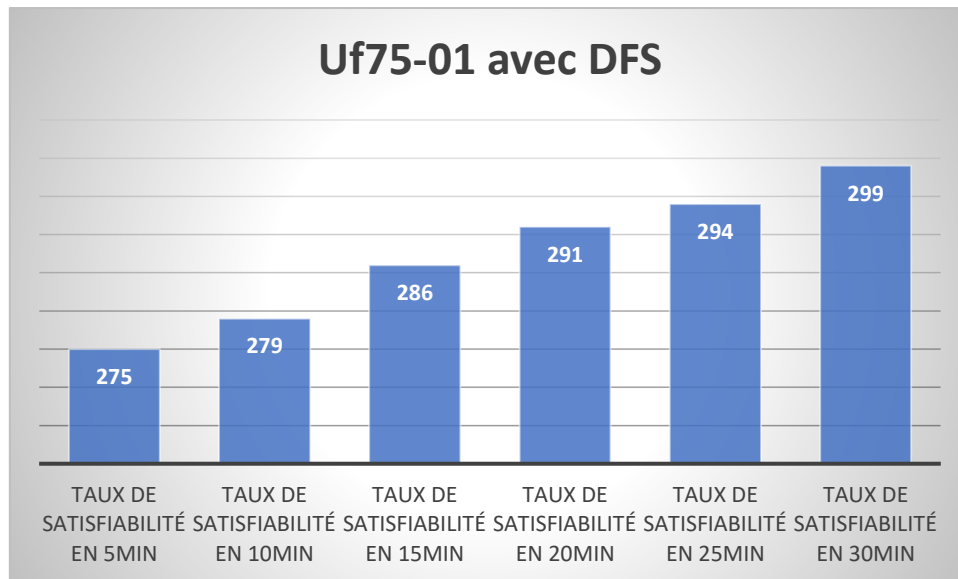


Figure 16-Figure 14-Histogramme montrant les résultats du test DFS pour le problème SAT sur l'instance de Benchmarks UF75-01

Comme a déjà montré dans les figures ci-dessus, L'algorithme de recherche par profondeur d'abord à donner meilleur résultat par rapport a l'algorithme de recherche par largeur d'abord, de plus il n'a pas causé le problème de saturation e la mémoire.

#### Statistiques de A\*

Comme pour l'algorithme basé sur la recherche en profondeur d'abord, l'algorithme A\* n'a pas pu atteindre le but au bout de 30 minutes. Mais le nombre global de clauses satisfaites est meilleur à celui obtenu en utilisant la recherche en profondeur (Nous avons utilisé la même instance avec DFS « UF75-01 » afin de pouvoir comparer les résultats par la suite) et en largeur comme le montre les statistiques avec les deux heuristiques utilisées ci-dessous :



Algo DFS :	nombre de clauses satisfaites	212
Algo DFS :	nombre de clauses satisfaites	213
Algo DFS :	nombre de clauses satisfaites	219
Algo DFS :	nombre de clauses satisfaites	229
Algo DFS :	nombre de clauses satisfaites	232
Algo DFS :	nombre de clauses satisfaites	247
Algo DFS :	nombre de clauses satisfaites	258
Algo DFS :	nombre de clauses satisfaites	264
Algo DFS :	nombre de clauses satisfaites	273
Algo DFS :	nombre de clauses satisfaites	279
Algo DFS :	nombre de clauses satisfaites	280
Algo DFS :	nombre de clauses satisfaites	281
Algo DFS :	nombre de clauses satisfaites	286
Algo DFS :	nombre de clauses satisfaites	301
Algo DFS :	nombre de clauses satisfaites	307
Algo DFS :	nombre de clauses satisfaites	309
Algo DFS :	nombre de clauses satisfaites	311
Algo DFS :	nombre de clauses satisfaites	312
Algo DFS :	nombre de clauses satisfaites	313
Algo DFS :	nombre de clauses satisfaites	314
Algo DFS :	nombre de clauses satisfaites	315

*Figure 17-Nombre de Clauses Satisfiables avec l'algorithmes A\* avec la 1ere heuristique*

nombre de clauses satisfaites 150  
 nombre de clauses satisfaites 154  
 nombre de clauses satisfaites 159  
 nombre de clauses satisfaites 169  
 nombre de clauses satisfaites 170  
 nombre de clauses satisfaites 176  
 nombre de clauses satisfaites 185  
 nombre de clauses satisfaites 186  
 nombre de clauses satisfaites 199  
 nombre de clauses satisfaites 203  
 nombre de clauses satisfaites 213  
 nombre de clauses satisfaites 215  
 nombre de clauses satisfaites 218  
 nombre de clauses satisfaites 221  
 nombre de clauses satisfaites 233  
 nombre de clauses satisfaites 245  
 nombre de clauses satisfaites 246  
 nombre de clauses satisfaites 254  
 nombre de clauses satisfaites 256  
 nombre de clauses satisfaites 257  
 nombre de clauses satisfaites 267  
 nombre de clauses satisfaites 274  
 nombre de clauses satisfaites 277  
 nombre de clauses satisfaites 278  
 nombre de clauses satisfaites 289  
 nombre de clauses satisfaites 293  
 nombre de clauses satisfaites 294  
 nombre de clauses satisfaites 295  
 nombre de clauses satisfaites 296

Figure 18-Nombre de Clauses Satisfiables avec l'algorithme A\* avec la 2eme heuristique

	Uf75-01 avec A* 01	Uf75-01 avec A* 02
Taux de satisfiabilité en 5min	281 →86.46%	159 →48.92%
Taux de satisfiabilité en 10min	286 →88%	199 →61.23%
Taux de satisfiabilité en 15min	295 →90.76%	256 →78.76%
Taux de satisfiabilité en 20min	307→94.46%	277 →85.23%
Taux de satisfiabilité en 25min	312→96%	293 → 90.15%
Taux de satisfiabilité en 30min	315→96.92%	296 →91.07%

Tableau 6-Statistiques A\* pour l'instance UF75-01

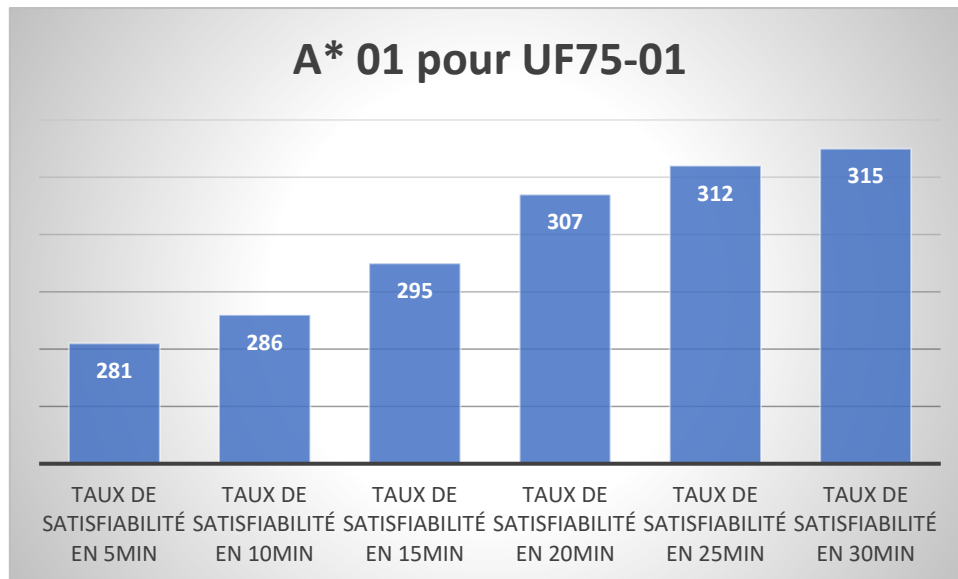


Figure 19-Histogramme montrant les résultats du test A\* avec la 1ere heuristique pour le problème SAT sur l'instance de Benchmarks UF75-01

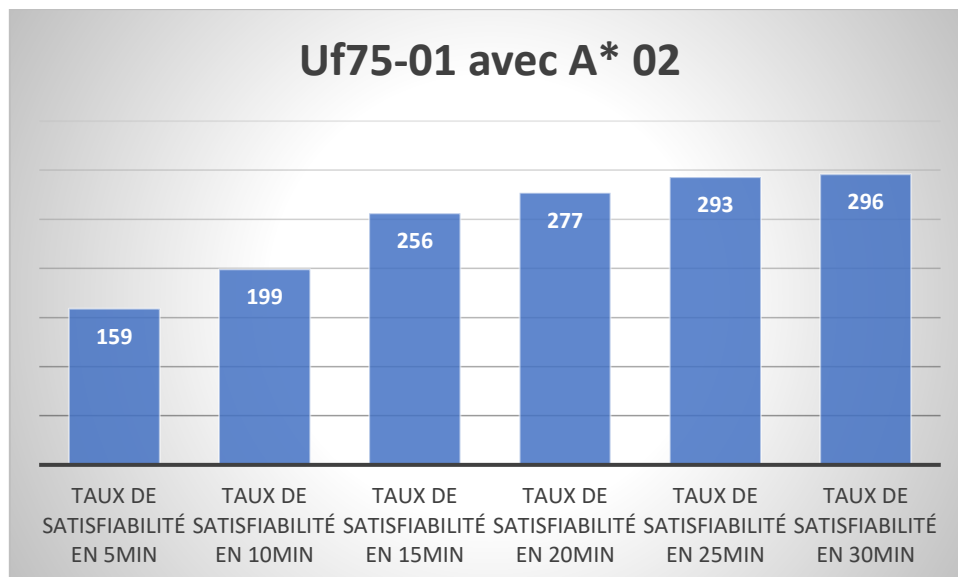


Figure 20-Histogramme montrant les résultats du test A\* avec la 2eme heuristique pour le problème SAT sur l'instance de Benchmarks UF75-01

Ya une différence entre le taux de satisfiabilité entre les deux heuristiques utilisées.

#### Interprétation et comparaison de résultats

Avant de passer à la comparaison il faut noter que le pourcentage d'échec n'est pas uniquement lié à la méthode utilisée, mais aussi à la taille d'espace d'état qui est volumineux et à la performance de la machine, pour notre cas nous avons utilisé l'ordinateur le moins performant.

Après observation et analyse des résultats obtenus des tests des trois algorithmes de résolutions basés sur les méthodes aveugles et informés sur les benchmarks choisis, on ne

constate clairement qu'aucune des stratégies n'a pu atteindre le but dans l'intervalle de temps fixé (30 minutes).

La recherche en largeur d'abord amène tout le temps à l'explosion combinatoire, ceci confirme le fait que cette stratégie est très gourmande coûteuse) en espace mémoire et est donc inefficace pour des données de grande taille (la moins efficace pour le problème de satisfiabilité).

La recherche en profondeur quant à elle, ne mène pas à une explosion de l'espace mémoire, Elle est meilleure que la BFS mais reste complexe en temps de recherche. En effet, pour un espace d'états de grande taille, il est très difficile d'atteindre un but.

L'algorithme A\*, qui est à la base plus efficace, performante et optimale que les deux autres stratégies, s'avère aussi ne pas être très efficace sur les grands espaces de de recherche.

Une petite comparaison complète entre les algorithmes utilisées jusqu'à maintenant permettra de mieux visualiser la différence entre eux :

	En 10 min	En 30 min
BFS	11,36%	/
DFS	32,37%	32 ,85%
A* 01	33,18%	34,62%
A* 02	23,09%	32,53%

Tableau 7-Comparaison entre les algorithmes de partie 01

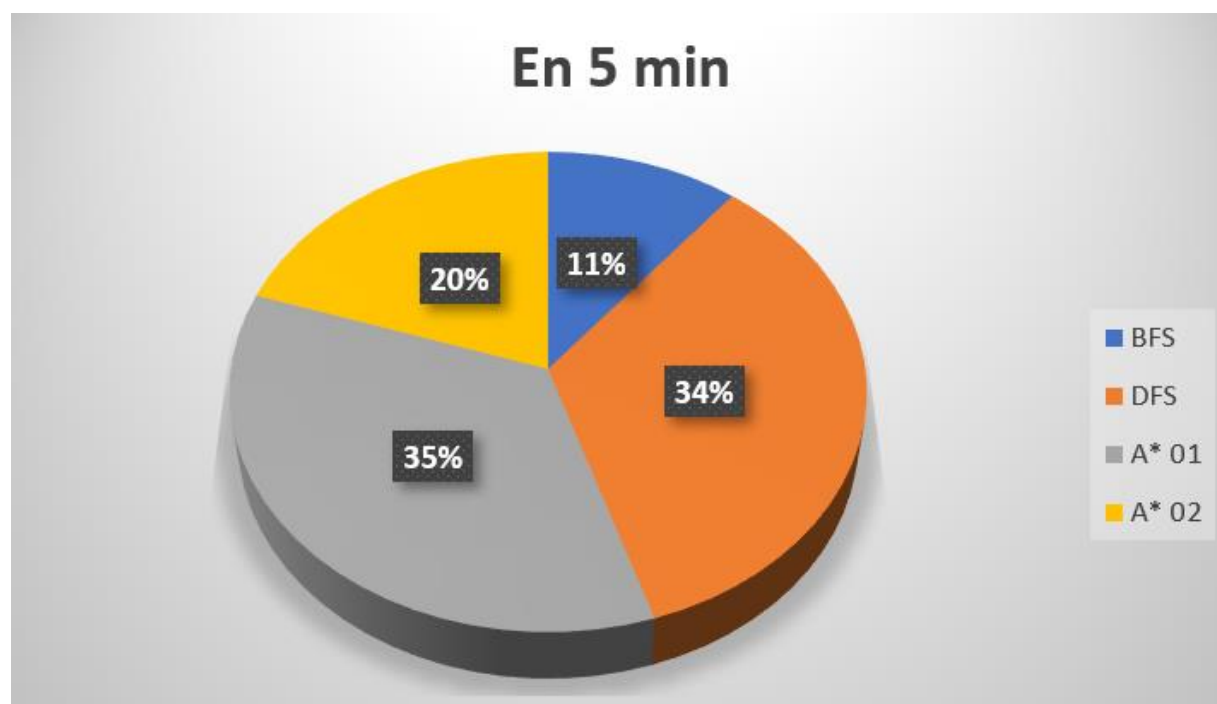
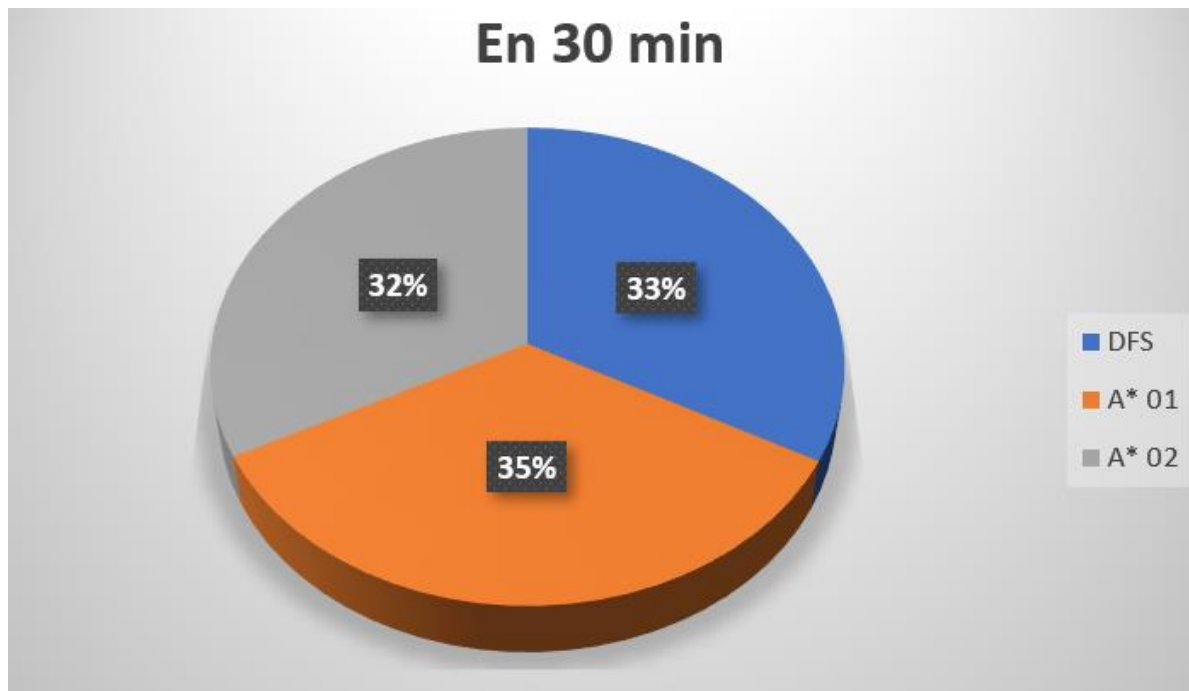


Figure 21-Comparaison avec secteur entre les algorithmes de partie 01 dans les premiers 5minutes



*Figure 22-Comparaison avec secteur entre les algorithmes de partie 01 après 30minutes d'exécutions*

On peut conclure donc que quand le volume de données est trop volumineux, les méthodes aveugles et les heuristiques qui sont performants pour les problèmes complexes deviennent inefficaces.

## Partie 02 – Les algorithmes génétiques

### Introduction

Les algorithmes génétiques sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle : croisements, mutations, sélection, etc.

Un algorithme génétique recherche le ou les extrema d'une fonction définie sur un espace de données. Pour l'utiliser, on doit disposer des cinq éléments suivants :

1. Un principe de codage de l'élément de population : Le choix du codage des données conditionne le succès des algorithmes génétiques.
2. Un mécanisme de génération de la population initiale.
3. Une fonction à optimiser : est appelée fitness ou fonction d'évaluation de l'individu.
4. Des opérateurs permettant de diversifier la population au cours des générations et d'explorer l'espace d'état (croisement).
5. Des paramètres de dimensionnement : taille de la population, nombre total de générations ou critère d'arrêt, probabilités d'application des opérateurs de croisement et de mutation.

Le principe général du fonctionnement d'un algorithme génétique est représenté sur la figure ci-dessous. [1]

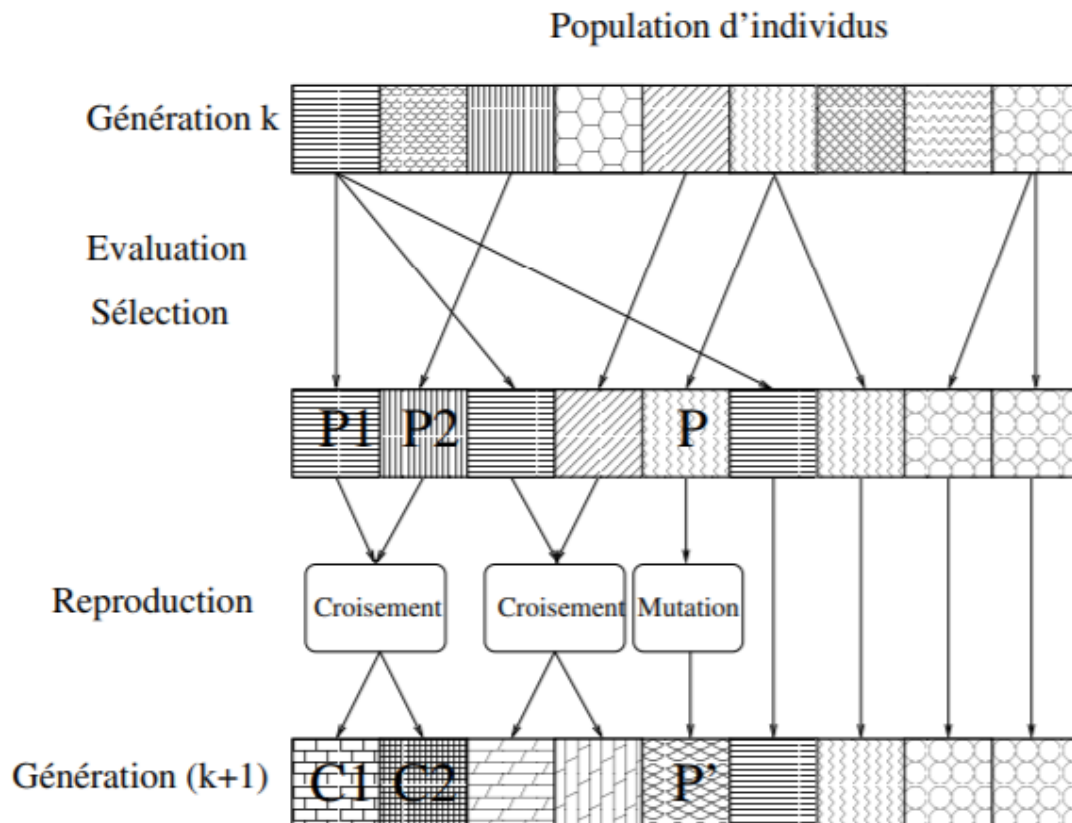


Figure 23-Principe général des algorithmes génétiques

## Description détaillée

### Codage des données

Historiquement, le codage utilisé par les algorithmes génétiques était représenté sous forme de chaînes de bits contenant toute l'information nécessaire à la description, et donc nous allons utiliser un tableau de taille égale au nombre des littéraux des instances et trié de l1 jusqu'à L (avec L représente nombre de littéraux) pour représenter une solution dont chaque case prend une valeur binaire (0 ou 1).

### Génération aléatoire de la population initiale

Le choix de la population initiale d'individus conditionne fortement la rapidité de l'algorithme.

Dans notre solveur SAT une population est présentée sous forme une matrice ( $N \times M$ ) où :

**N** : nombre des chromosomes (individus) définit par l'utilisateur.

**M** : nombre de littéraux = 75 plus une dernière colonne préservée pour définir la valeur de fitness de chaque chromosome.

La figure ci-dessous définit le code source de la fonction qui génère la matrice de population initiale.

```

//definir une population de solutions
public static int [][] Population (int taille)
{
    int [] tabSol=new int [75]; //selon nombre de littéraux
    int [] [] population =new int [taille][76];

    for(int i=0;i<taille;i++)
    {
        for(int j=0;j<75;j++) {
            tabSol= alger1.isii.algo.GENETIQUE.Solution();
            population[i][j]=tabSol[j];
        }
    }
    return population;
}

```

*Figure 24-Fonction d'initialisation d'une population*

#### Gestion des contraintes

Un élément de population qui viole une contrainte se verra attribuer une mauvaise valeur de fitness et aura une probabilité forte d'être éliminé par le processus de sélection.

La fonction fitness utilisée repose sur nombre de clause satisfaites pour les valeurs d'un chromosome, Donc nous allons tester un vecteur de bits binaire (chromosome) de la matrice de la population et récupérer le nombre de clauses satisfaites avec.



```

public static int Evaluation(Satisfaite sat,int[]tempo,int tailleP)
{
    Vector<Literal> Litteraux = new Vector<Literal>();
    //initialiser le vecteur des litteraux avec des -1 et les nommer surtt
    for(int i=-75;i<76;i++)//pour avoir des valeurs des litteraux entre -75 et 75
    {
        if(i!=0) Litteraux.add(new Literal(i)); //apart la valeur 0 qui signifie un saut de ligne
    }
    int nom;
    //cloner sat dans une variable formule
    //creer un vecteur de clauses
    Vector<Clause> clause=new Vector<Clause>();
    for(int i=0;i<sat.getClauses().size();i++)//325
    {
        Vector <Literal> Newlitteraux= new Vector<Literal>();

        for(int j=0;j<sat.getClauses().get(i).getLitteraux().size();j++)//3 litteraux dans chaque clause de 3SAT
        {
            if(sat.getClauses().get(i).getLitteraux().get(j).getNumvar()<0){
                nom=-(sat.getClauses().get(i).getLitteraux().get(j).getNumvar());
                if(tempo[nom-1]==0)
                    Litteraux.get(sat.getClauses().get(i).getLitteraux().get(j).getNumvar()+75).setValeur(1);
                else
                    Litteraux.get(sat.getClauses().get(i).getLitteraux().get(j).getNumvar()+75).setValeur(0);
                Newlitteraux.add(Litteraux.get(sat.getClauses().get(i).getLitteraux().get(j).getNumvar()+75));
            }
            else{
                nom=sat.getClauses().get(i).getLitteraux().get(j).getNumvar();
                //donner une valeur a la variable x
                Litteraux.get(sat.getClauses().get(i).getLitteraux().get(j).getNumvar()+74).setValeur(tempo[nom-1]);
                Newlitteraux.add(Litteraux.get(sat.getClauses().get(i).getLitteraux().get(j).getNumvar()+74));
            }
        }
        clause.add(new Clause(Newlitteraux));
    }
}

```

Figure 25- La fonction fitness1

```

        Newlitteraux.add(Litteraux.get(sat.getClauses().get(i).getLitteraux().
        )
    }
    clause.add(new Clause(Newlitteraux));
}
//remplir les deux sat par les sat de sat
Satisfaite formule = new Satisfaite(clause, Litteraux);

System.out.println("-----");

System.out.println("nombre de clauses satisfaites "+formule.NombreClauseSat());
return formule.NombreClauseSat();
} //end evaluation

```

Figure 26-La fonction fitness2

## Sélection

La sélection permet d'identifier statistiquement les meilleurs individus d'une population et d'éliminer les mauvais. On trouve dans la littérature un nombre important de principes de sélection.

Le type de sélection utilisé est la sélection classée selon les valeurs de la fonction fitness qui sont stocké dans la dernière colonne de la matrice de population, Cette dernière est triée avec un ordre décroissant afin de pouvoir par la suite choisir les (N/2) chromosomes (N est le nombre de chromosome initiale) de reproduction.



```

//trier la matrice de population selon les valeurs de fitness
int tampon = 0;
boolean permut;

do { // hypothèse : le tableau est trié
    permut = false;
    for (int i = 0; i < (tailleP-1); i++)
    {
        // Teste si 2 éléments successifs sont dans le bon ordre ou non
        if (population[i][75] < population[i+1][75])
        {
            // s'ils ne le sont pas, on échange leurs positions
            tampon = population[i][75];
            population[i][75] = population[i + 1][75];
            population[i + 1][75] = tampon;
            permut = true;
        }
    }
} while (permut);

```

Figure 27-trier la matrice de population selon les valeurs de fitness

#### Opérateur de croisement

Le croisement a pour but d'enrichir la diversité de la population en manipulant la structure des chromosomes. Classiquement, les croisements sont envisagés avec deux parents et génèrent deux enfants (voir la figure ci-dessous).

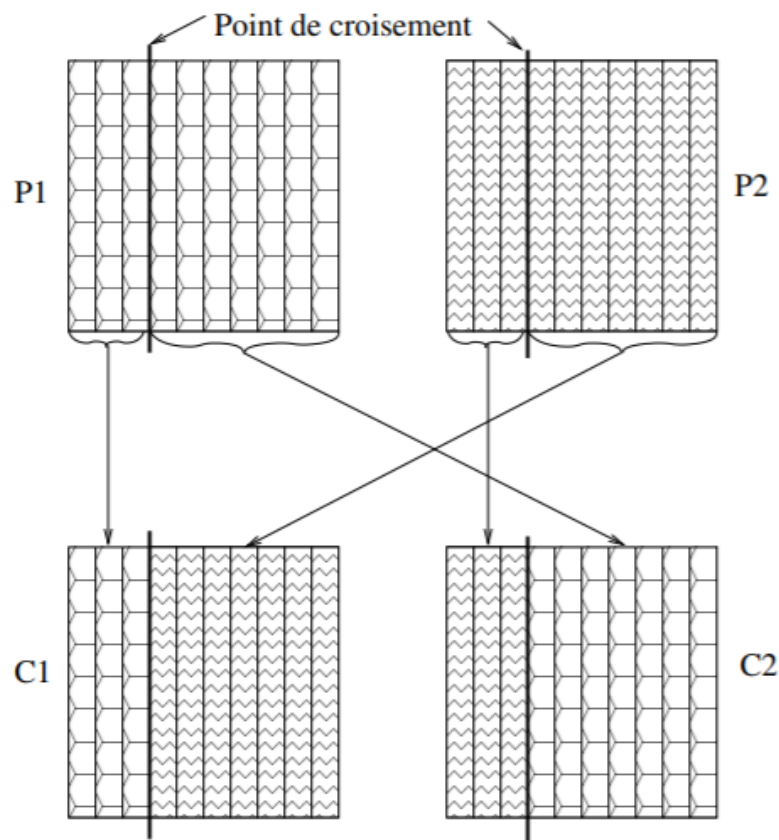
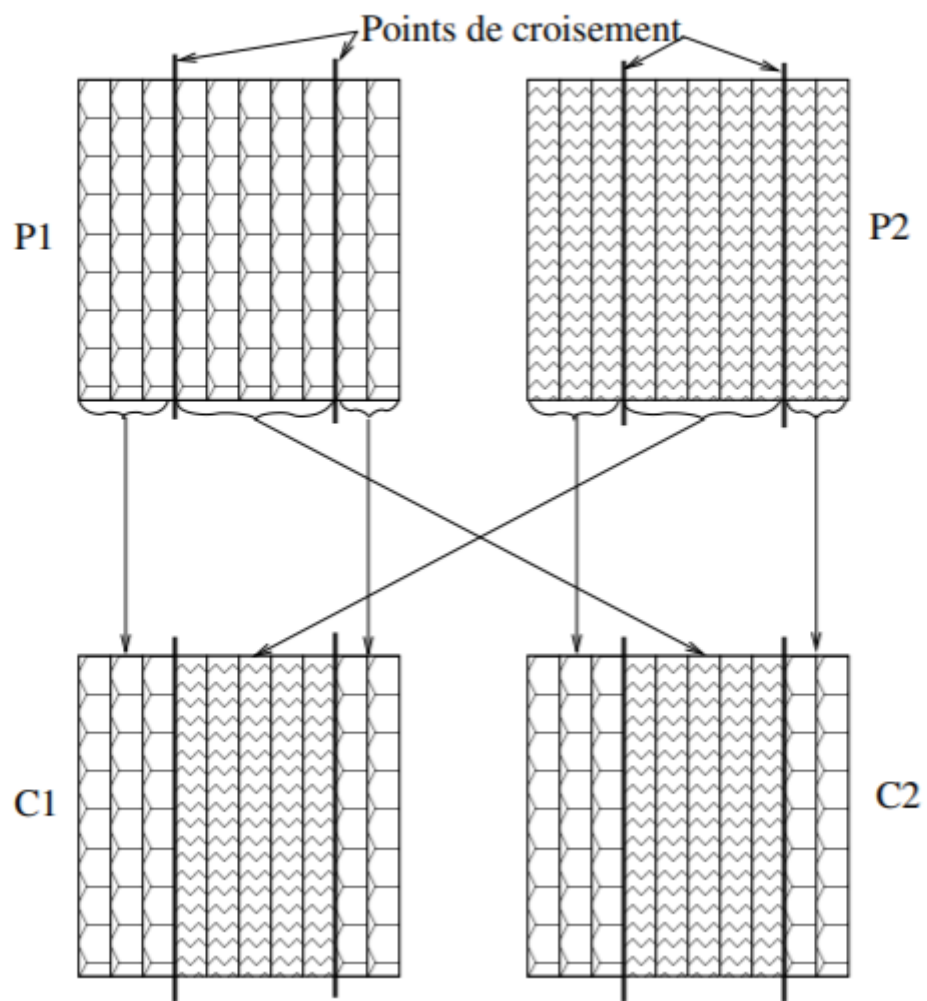


Figure 28- croisement à 1 point



*Figure 29-croisement à 2 points*

Pour notre solveur, le taux de croisement utilisé est 1, sur N positions aléatoire générer avec la fonction Radom.

```

//fonction de croisement
public static int [] croisement1(int[]parent1, int[]parent2)
{
    Random rand=new Random();
    int x=rand.nextInt(75); //generer un nombre aléatoire entre 0 à et 74
    int []child1=new int [75]; //nouveau chromosome
    for(int i=0;i<child1.length;i++)
    {
        if([i<x])
        {
            child1[i]=parent1[i];
        }
        else
            child1[i]=parent2[i];
    }
    return child1;
}

```

Figure 30-fonction de croisement

#### Opérateur de mutation

L'opérateur de mutation apporte aux algorithmes génétiques la propriété d'ergodicité de parcours d'espace. Cette propriété indique que l'algorithme génétique sera susceptible d'atteindre tous les – Principe de l'opérateur de mutation points de l'espace d'état, sans pour autant les parcourir tous dans le processus de résolution.

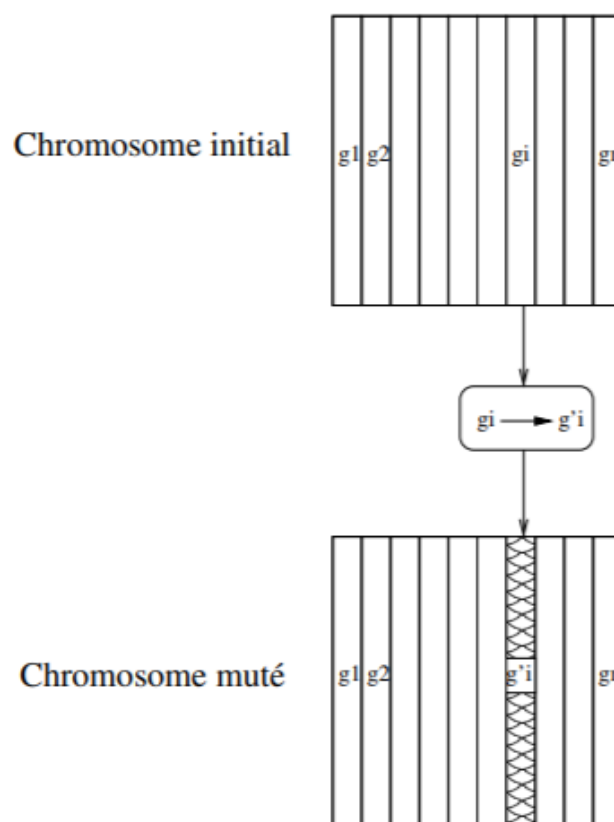


Figure 31 Principe de l'opérateur de mutation

Cependant, Nous allons appliquer la mutation N fois (N est un entier aléatoire) sur K gènes.

```

//mutation
Random rand=new Random();
int x=rand.nextInt(4);
for(int i=0;i<x;i++)
{
    for(int j=0;j<tailleBest;j++)
    {
        for(int e=0;e<mutate.length;e++)
        {
            mutate[e]=fils[j][e];

        }
        mutate=alger1.isii.algo.GENETIQUE.mutation(mutate,tauxMutation);

        for(int a=0;a<75;a++) {
            fils[j][a]=mutate[a];
        }

    }

}
}

```

Figure 32-Appel à la fonction mutation x fois

```

//mutation sur un seul point qui sera appeler X fois
public static int [] mutation(int[]tableau,double tauxMutation)
{
    Random rand=new Random();
    int x=rand.nextInt(75);
    double prob=rand.nextDouble();
    System.out.println("le point de mutation est "+x);
    System.out.println("le point de mutation est "+prob);
    if(tauxMutation>=prob)
    {
        if(tableau[x]==0)
        {
            tableau[x]=1;
        }
        else tableau[x]=0;
    }
    return tableau;
}
}

```

Figure 33-fonction de mutation

Il est à noter que la mutation ne sera appliquée sur un gène si et seulement si le taux de mutation > probabilité de mutation (la probabilité est générée aléatoirement).

### Etude et interprétation des résultats

Nous allons tester plusieurs fois l'algorithme implémenté sur l'instance (UF75-01) avec les deux ordinateurs portables (R est le moins performant et le A est le plus performant), en

changeant à chaque fois les paramètres (taux de mutation, taille de population...) afin de les fixer à des valeurs qui donnent les meilleurs résultats.

Si un paramètre donne un bon résultat pour une valeur donnée, alors il est fixé à cette valeur, et les autres paramètres sont variés. Le processus est appliqué pour chaque paramètre empirique.

Dans un premier temps nous allons jouer sur la taille de la population

Taille population	Nombre d'itérations	Taux de mutation	Nombre de clauses satisfaites	Taux de satisfiabilité
100	1000	0,04	311	95.69%
200	1000	0,04	313	96.30%
300	1000	0,04	316	97.23%
400	1000	0,04	312	96%
500	1000	0,04	316	97.23%
600	1000	0,04	314	96.61%
700	1000	0,04	316	97.23%
800	1000	0,04	315	96.92%
900	1000	0,04	322	99.07%
1000	1000	0,04	317	97.53%

Tableau 8-AG modifier taille de population

On remarque que la taille 900 donne les meilleurs résultats donc on la fixe, et on passe à la modification de nombre d'itération.

Taille population	Nombre d'itérations	Taux de mutation	Nombre de clauses satisfaites	Taux de satisfiabilité
900	100	0,04	315	96.92%
900	200	0,04	316	97.23%
900	300	0,04	316	97.23%
900	400	0,04	316	97.23%
900	500	0,04	317	97.53%
900	600	0,04	317	97.53%
900	700	0,04	318	97.84%
900	800	0,04	317	97.53%
900	900	0,04	317	97.53%
900	1000	0,04	315	96.92%

Tableau 9-AG modifier nombre d'itérations

Le nombre d'itérations 700 donne les meilleurs résultats donc on le fixe et on joue sur le taux de mutation afin de pouvoir augmenter le taux de satisfaisabilité par la suite :

Taille population	Nombre d'itérations	Taux de mutation	Nombre de clauses satisfaites	Taux de satisfaisabilité
900	700	0,01	316	97.23%
900	700	0,02	318	97.84%
900	700	0,03	316	97.23%
900	700	0,04	318	97.84%
900	700	0,05	317	97.53%
900	700	0,06	318	97.84%
900	700	0,07	318	97.84%
900	700	0,08	316	97.23%
900	700	0,09	317	97.53%
900	700	0,1	319	98.15%
900	700	0,2	316	97.23%
900	700	0,3	320	98.46%
900	700	0,4	318	97.84%
900	700	0,5	321	98.76%
900	700	0,6	318	97.84%
900	700	0,7	319	98.15%
900	700	0,8	323	99.38%
900	700	0,9	319	98.15%
900	700	1	320	98.46%

Tableau 10- AG modifier taux de mutation

Après avoir analysé les résultats obtenus de tableaux ci-dessus, nous avons constaté que le meilleur taux de mutation en fonction de taux de satisfaction pour l'instance uf75-01 est 0.8 avec un taux=99.07%.

Il y a une convergence prématurée et stagnation d'amélioration de résultats après un certain nombre d'itération donc on n'a pas besoin d'augmenter le nombre d'itération plus que 1000.

La figure ci-dessus représente le résultat obtenu avec les meilleures paramètres choisies :

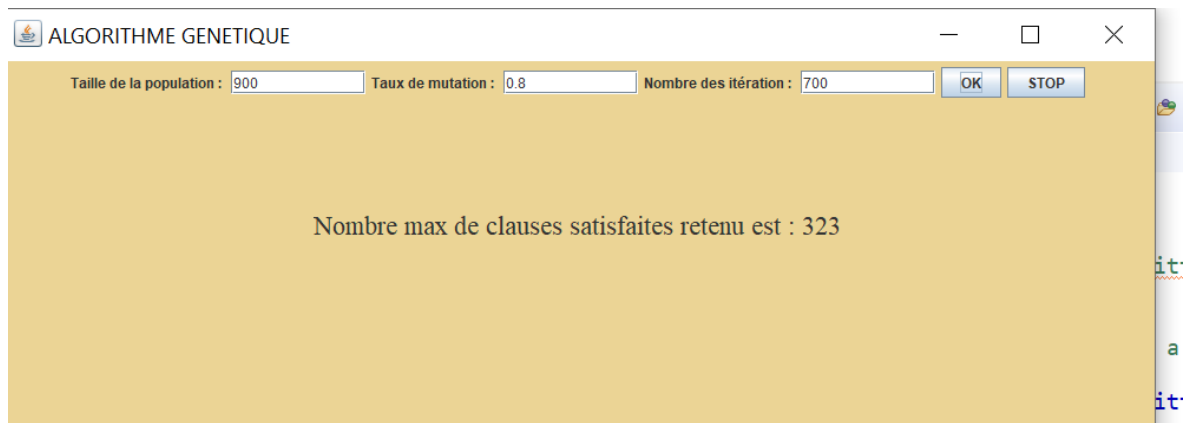


Figure 34-Interface graphique de GA

Essayons maintenant les meilleurs paramètres avec les autres instances, comme le montre le tableau ci-dessous :

Benchmarks	Instance	Nombre de clauses satisfaites	Taux de satisfiabilité
uf75	01	323	99.38%
	02	324	99.69%
	03	322	99.07%
	04	320	98.46%
	05	323	99.38%
	06	323	99.38%
	07	317	97.53%
	08	315	96.92%
	09	318	97.84%
	10	318	97.84%
uuf75	01	320	98.46%
	02	319	98.15%
	03	318	97.84%
	04	319	98.15%
	05	320	98.46%
	06	320	98.46%
	07	322	99.07%
	08	315	96.92%
	09	317	97.53%
	10	309	95.07%

Tableau 11-GA tous les Benchlarks

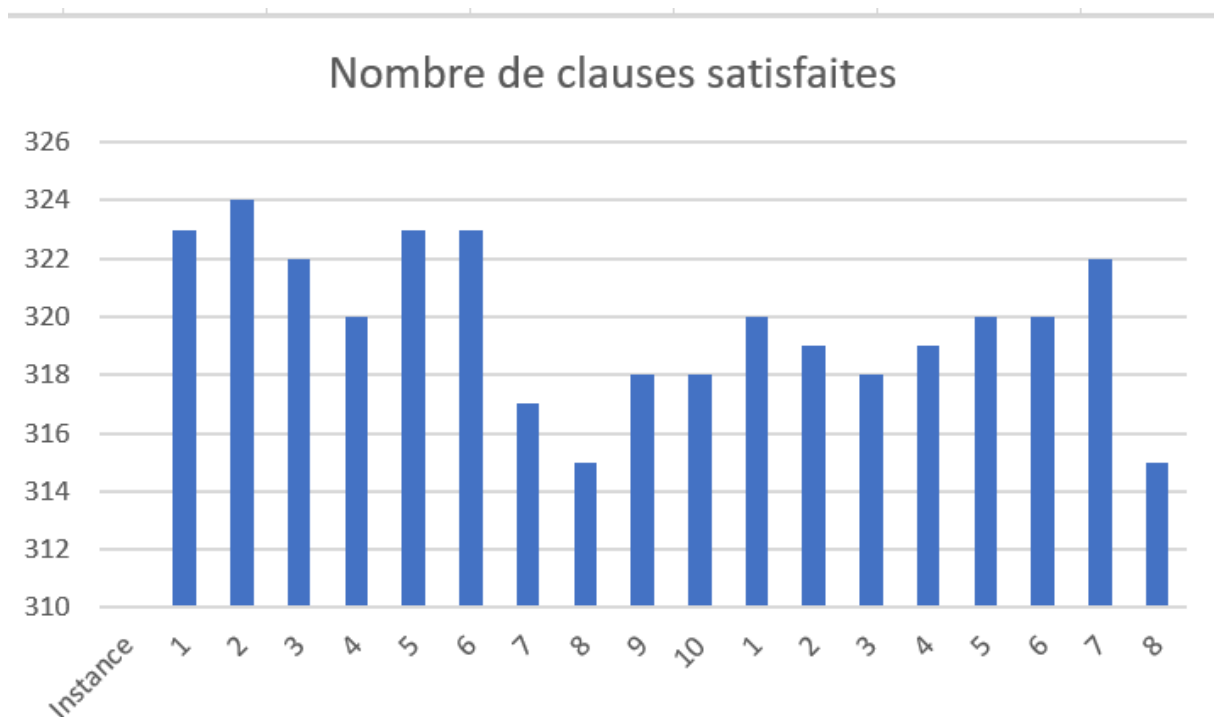


Figure 35-GA tous les benchmarks résultats

### Analyse

L'exécution de l'algorithme de résolution de SAT basé sur la métaheuristique

« Algorithme génétique » a donné des résultats satisfaisants au bout 200 itérations. L'algorithme n'a pas atteint la solution souhaitée mais il a pu satisfaire jusqu'à 99% des clauses.

### Comparaison avec les résultats de la partie 1

Après avoir observé et analyser les résultats obtenus de l'exécution des algorithmes basés sur les méthodes aveugles et l'algorithme A\*, et ceux obtenus en utilisant l'algorithme génétique, sur les instances données de problème SAT, on constate clairement que l'algorithme génétique permet de donner des solutions de meilleure qualité.

Une différence de taille entre les méthodes citées ci-dessus :

En termes de taux de satisfaction l'algorithme génétique a pu atteindre 99%.

En termes de temps, AG est clairement beaucoup plus rapide que les autres méthodes (convergence prématurée) un grand taux de satisfiabilité en moins d'une minute.

En effet, avec les méthodes aveugles, seulement 35% des clauses ont pu être satisfaites en 30minutes. Ce taux, s'est légèrement amélioré après l'utilisation d'une heuristique, mais reste quand-même loin de celui obtenu grâce à GA, à savoir, 99%.

GA a donc permis de trouver des solutions de meilleure qualité en un temps moindre.

### Conclusion

Cette partie du projet nous a permis de confirmer qu'il est souvent impossible de résoudre des problèmes de très grande taille comme le problème SAT, avec des méthodes de résolution exhaustives ou en utilisant des heuristiques, car ces dernières sont coûteuses en termes de temps et/ou espace



mémoire, Pour cela l'utilisation de méthodes plus performantes comme les métaheuristiques qui donnent des résultats de meilleure qualité e temps très réduit devient une nécessité.

## Parie 03-L'algorithme ACS

### Introduction

Les algorithmes de colonies de fourmis (en anglais, *ant colony optimization*, ou *ACO*) sont des algorithmes inspirés du comportement des fourmis, ou d'autres espèces formant un superorganisme, et qui constituent une famille de métaheuristiques d'optimisation.

Les spécialistes réservent ce terme (ACO) à un type particulier d'algorithme. Il existe cependant plusieurs familles de méthodes s'inspirant du comportement des fourmis. En français, ces différentes approches sont regroupées sous les termes : « algorithmes de colonies de fourmis », « optimisation par colonies de fourmis », « fourmis artificielles » ou diverses combinaisons de ces variantes.

D'une façon très générale, les algorithmes de colonies de fourmis sont considérés comme des métaheuristiques à population, où chaque solution est représentée par une fourmi se déplaçant sur l'espace de recherche. Les fourmis marquent les meilleures solutions, et tiennent compte des marquages précédents pour optimiser leur recherche.

On peut les considérer comme des algorithmes multi-agents probabilistes, utilisant une distribution de probabilité implicite pour effectuer la transition entre chaque itération. Dans leurs versions adaptées à des problèmes combinatoires, ils utilisent une construction itérative des solutions [3].

Dans cette partie, nous allons appliquer la métaheuristique « ACS » - Ant Colony System- dans problème de satisfiabilité SAT tout en utilisant les instances données, afin d'étudier la performance de cette métaheuristique, et comparer les résultats obtenus avec ceux des méthodes utilisées dans les parties précédentes.

### L'algorithme ACS

Pour notre solution nous avons suivi les étapes suivantes :

1. Initialisation des paramètres des règles de transition et de mise à jour

```

JButton sendA = new JButton("OK");
sendA.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        //affichage.setText("en calcul...");
        float alfa=Float.parseFloat(alffa.getText());
        float qZero=Float.parseFloat(tf2A.getText());
        float bita=Float.parseFloat(tf1A.getText());
        float Row=Float.parseFloat(troo.getText());
        float pheroo=Float.parseFloat(troo.getText());
        //initialisation des paramètres empiriques
        float alpha = alfa , beta = bita , q0 = qZero , Ro = Row , pheromone0 = pheroo;
        //Vecteur des fourmis.
        Vector<Fourmi> fourmis = new Vector<>(); //vecteur des fourmis
    }
});

```

Figure 36-ACS initialisation depuis l'interface graphique

2. Génération de k fourmis

```

int nb_fourmis = 100 ; //nombre de fourmis
. . .

```

Figure 37-ACS nombre de fourmis

```
//initialiser les the fourmis
for (int i = 0; i < nb_fourmis; i++)//pour chaque fourmi
{
    //ajouter la fourmi dans le vecteur des fourmis
    //initialiser la solution des fourmis null
    //avec une valeur d'evaluation a 0
    fourmis.addElement(new Fourmi(null, 0));
}
```

Figure 38-ACS initialiser les fourmis

- Utilisation d'un nombre fixe d'itération comme condition d'arrêt de l'algorithme

```
int nb_iter_max = 300 ;//nombre max d'itérations
```

```
int max_iter = 300 ;//nombre max d'itérations
```

Figure 39-ACS nombre des itérations

- Tant que le nombre maximum des itérations n'a pas été atteint

```
//Debut du travail
for (int i = 0; i < max_iter; i++)
{
    // ...
}
```

Figure 40-ACS critère d'arrêt

Pour chaque fourmi faire

```
//pour chaque fourmi
for (int j = 0; j < nb_fourmis; j++)
{
    // ...
}
```

Figure 41-ACS pour chaque fourmi

- Construire une solution
- Evaluer la solution construite

```
//pour chaque fourmi
for (int j = 0; j < nb_fourmis; j++)
{
    //initialiser vecteur de solution pour la fourmi k
    Vector<Integer> k = new Vector<>();
    //Construire la solution avec les paramètres
    k = alger1.isii.algo.ACS.Construire( litterauxF,q0,alpha,beta,clauses_benchmark);
    //modifier les parametre de la fourmi
    fourmis.get(j).setSolution(k); //modifier la solution
    fourmis.get(j).setEvaluation(alger1.isii.algo.ACS.evaluation(k,clauses_benchmark)); //evaluer la solution
}
```

Figure 42-ACS Construction et évaluation de solution e chaque fourmi

- Mettre à jours le phéromone « Online delayed update »

```

//la mise à jour online "Online delayed update "
for (int online = 0; online < fourmis.size(); online++)
{
    //initialiser vecteur de solution
    Vector<Integer> solution = new Vector<>();
    //recupere la solution de la fourmi
    solution = fourmis.get(online).getSolution();
    for (int l = 0; l < litterauxF.size(); l++) {
        if(litterauxF.get(l).getLiteral() == solution.get(online)){
            float p = litterauxF.get(l).getPheromone();
            //evaluer la solution
            float evaluat = alger1.isii.algo.ACS.evaluation(solution, clauses_benchmark)/325f;
            float update = (1-Ro)*p + Ro*evaluat;
            //mettre à jour le pheromone
            litterauxF.get(l).setPheromone(update);|
        }
    }
}

```

Figure 43-ACS online delayed update

Faite

Déterminer la meilleure solution obtenue dans l'itération actuelle (pour chaque itération évaluer la solution actuelle si elle est de meilleure qualité que les précédentes on la marque comme meilleure solution)

```

//Choisir la meilleure solution pour cette itération
int best_sol = fourmis.get(0).getEvaluation() , indice_best=0;
for (int a = 1; a < fourmis.size(); a++)
{
    //si la solution obtenu dans cette itération est meilleure que les autres on la definit comme best_sol
    if(fourmis.get(a).getEvaluation() > best_sol){
        //recupere l'evaluation de cette solution
        best_sol = fourmis.get(a).getEvaluation();
        //recupere l'indice de cette solution
        indice_best = a;
    }
}

```

Figure 44-ACS obtenir meilleure solution des fourmis

```

//deinir la solution obtenu comme meilleure et inserer ses info dans une meilleure definit comme meilleure fourmi
if(best_sol > meilleure_fourmi.get(0).getEvaluation())
{
    meilleure_fourmi.get(0).setEvaluation(best_sol); //attribuer l'evaluation a cette fourmi
    meilleure_fourmi.get(0).setSolution(fourmis.get(indice_best).getSolution()); //attribuer la meilleure sol a la fourmi
}

```

Figure 45-ACS meilleure fourmi

Mettre à jours le phéromone « Offline delayed update »

```

//Mettre à jours le phéromone « Offline delayed update »
//initialiser un vecteur de la meilleure solution
Vector<Integer> offline = new Vector<Integer>();
//ajouter la meilleure solution dans offline
offline.addAll(fourmis.get(indice_best).getSolution());
//parcourir le vecteur offline
for (int g = 0; g < offline.size(); g++) {
    //pour chaque littéral
    for (int j = 0; j < litterauxF.size(); j++) {
        if(offline.get(g) == litterauxF.get(j).getLiteral()){
            //recupere la quantité de phéromone
            float p = litterauxF.get(j).getPheromone();
            //evaluer le vecteur offline
            float eval = alger1.isii.algo.ACS.evaluation(offline, clauses_benchmark)/325f;
            //mettre a jour
            float update = (1-Ro)*p + Ro*eval; //meilleure_fourmi-best/325
            //remplacer la valeur de phéromone
            litterauxF.get(j).setPheromone(2*update);
        }
    }
}

```

Figure 46-ACS Offline delayed update

Pour la construction de solution nous avons utilisé l'algorithme suivant

### ***Construction des solutions :***

*Entrée : Liste des littéraux L*

*Sortie : solution S*

*1- La solution S est initialement vide*

*2- Tant que la liste des littéraux L n'est pas vide faire*

- Utiliser la règle de transition pour choisir un littéral*
- Supprimer le littéral et son contraire de L*
- Ajouter le littéral sélectionné à la solution S*

*Fin*

*Fin de la construction (retourner S) ;*

Figure 47-ACS Algorithme de construction de solutions

## Etude et interprétation des résultats

Nous avons testé notre solution basée sur l'algorithme ACS sur la première instance plusieurs fois, en changeant à chaque fois les paramètres afin de les fixer ceux qui donnent les meilleurs résultats.

Comme dans la partie précédente, chaque tableau représente les résultats obtenus en fixant le un paramètre pour pouvoir par la suite sélectionner les meilleures qui donnent un meilleur taux de satisfaisabilité.

Note : pour une meilleure lecture les valeurs changées sont mises en gras à chaque étape. Les meilleures valeurs obtenues sont en rouge.

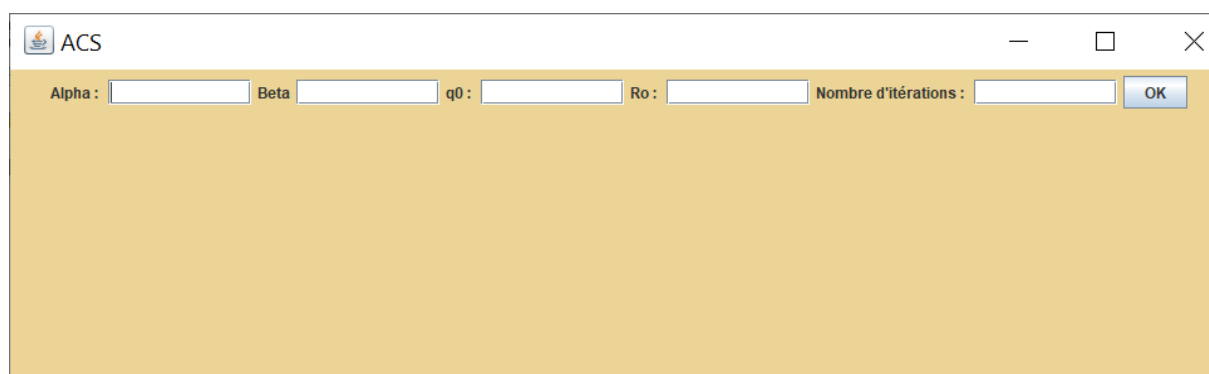


Figure 48-L'interface graphique de ACS

Avec un nombre de fourmis=10.

A	$\beta$	$q_0$	$r_0$	Nbr Itérations	Nombre de clauses satisfaites
0.5	0.5	0.5	0.1	100	297
0.5	0.5	0.5	0.1	<b>500</b>	300
0.5	0.5	0.5	0.1	<b>1000</b>	301
<b>0.7</b>	0.5	0.5	0.1	500	297
<b>0.1</b>	0.5	0.5	0.1	500	299
<b>0.5</b>	<b>0.5</b>	<b>0.8</b>	<b>0.2</b>	500	309
0.5	<b>0.7</b>	<b>0.9</b>	0.2	500	308

Tableau 12-ACS résultats avec 10 fourmis

Nombre de fourmis =30

A	$\beta$	$q_0$	$r_0$	Nbr Itérations	Nombre de clauses satisfaites
0.5	0.5	0.5	0.1	500	303
0.5	0.5	0.5	0.1	<b>300</b>	303
<b>0.01</b>	<b>0.05</b>	0.5	0.1	500	299
<b>0.7</b>	<b>0.7</b>	0.5	0.1	500	300
<b>0.7</b>	<b>0.7</b>	0.5	<b>0.3</b>	500	298
0.7	0.7	<b>0.8</b>	0.3	500	307
<b>0.7</b>	<b>0.7</b>	<b>0.9</b>	<b>0.6</b>	500	310

Tableau 13-ACS résultats avec 30 fourmis

Nombre de fourmis =40

A	$\beta$	$q_0$	$r_0$	Nbr Itérations	Nombre de clauses satisfaites
0.7	0.7	0.9	0.6	500	312
0.7	<b>0.9</b>	0.9	0.6	500	314
0.7	<b>0.3</b>	0.9	<b>0.5</b>	500	309
<b>0.9</b>	<b>0.9</b>	0.9	<b>0.7</b>	500	208
<b>0.6</b>	0.8	0.4	0.5	500	298
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.3</b>	500	315
0.9	0.9	0.9	<b>0.8</b>	500	308

Tableau 14-ACS résultats avec 40 fourmis

Nombre de fourmis =50

A	$\beta$	$q_0$	$r_0$	Nbr Itérations	Nombre de clauses satisfaites
0.5	0.5	0.5	0.1	500	303
0.4	0.4	0.8	0.2	500	310
0.4	0.4	0.8	0.2	<b>300</b>	309
<b>0.7</b>	<b>0.7</b>	0.5	0.1	500	301
<b>0.6</b>	<b>0.6</b>	<b>0.8</b>	<b>0.2</b>	500	304
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.2</b>	500	313
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.4</b>	500	314

Tableau 15-ACS résultats avec 50 fourmis

Nombre de fourmis =60

A	$\beta$	$q_0$	$r_0$	Nbr Itérations	Nombre de clauses satisfaites
0.5	0.5	0.5	0.1	500	303
0.5	0.5	<b>0.8</b>	<b>0.2</b>	500	307
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.3</b>	500	309
0.9	0.9	0.9	<b>0.4</b>	500	315
0.9	0.9	0.9	<b>0.5</b>	500	315
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.4</b>	<b>200</b>	316
0.9	0.9	0.9	<b>0.4</b>	<b>100</b>	315

Tableau 16-ACS résultats avec 60 fourmis

Nombre de fourmis =70

A	$\beta$	$q_0$	$r_0$	Nbr Itérations	Nombre de clauses satisfaites
0.5	0.5	0.5	0.1	500	303
0.5	0.5	<b>0.8</b>	<b>0.2</b>	500	300
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.3</b>	500	301

0.9	0.9	0.9	<b>0.4</b>	500	302
0.9	0.9	0.9	<b>0.5</b>	500	306
<b>0.9</b>	<b>0.9</b>	<b>0.9</b>	<b>0.4</b>	<b>300</b>	<b>316</b>
0.9	0.9	0.9	<b>0.4</b>	<b>100</b>	310

Tableau 17-ACS résultats avec 70 fourmis

Le meilleur taux de satisfiabilité atteint est égal 97% avec les paramètres suivants :

Nombre de fourmis = 60, Nombre d'itérations =300,  $\alpha = 0.9$ ,  $\beta = 0.9$ ,  $q_0 = 0.9$ ,  $r_0 = 0.4$ .

On va fixer nos paramètres empiriques à ces valeurs et les tester sur les autres instances :

Benchmarks	Instance	Nombre de clauses satisfaites	Taux de satisfiabilité
Uf75	01	316	97.2%
	02	315	96.9%
	03	314	96.6%
	04	314	96.6%
	05	315	96.9%
	06	310	95.4%
	07	311	95.6%
	08	311	95.6%
	09	313	96.3%
	10	310	95.4%
UUf75	01	311	95.6%
	02	311	95.6%
	03	315	96.9%
	04	310	95.4%
	05	305	93.8%
	06	310	95.4%
	07	302	92.9%
	08	316	97.2%
	09	311	95.6%
	10	311	95.6%

Tableau 18-ACS tous les Benchmarks

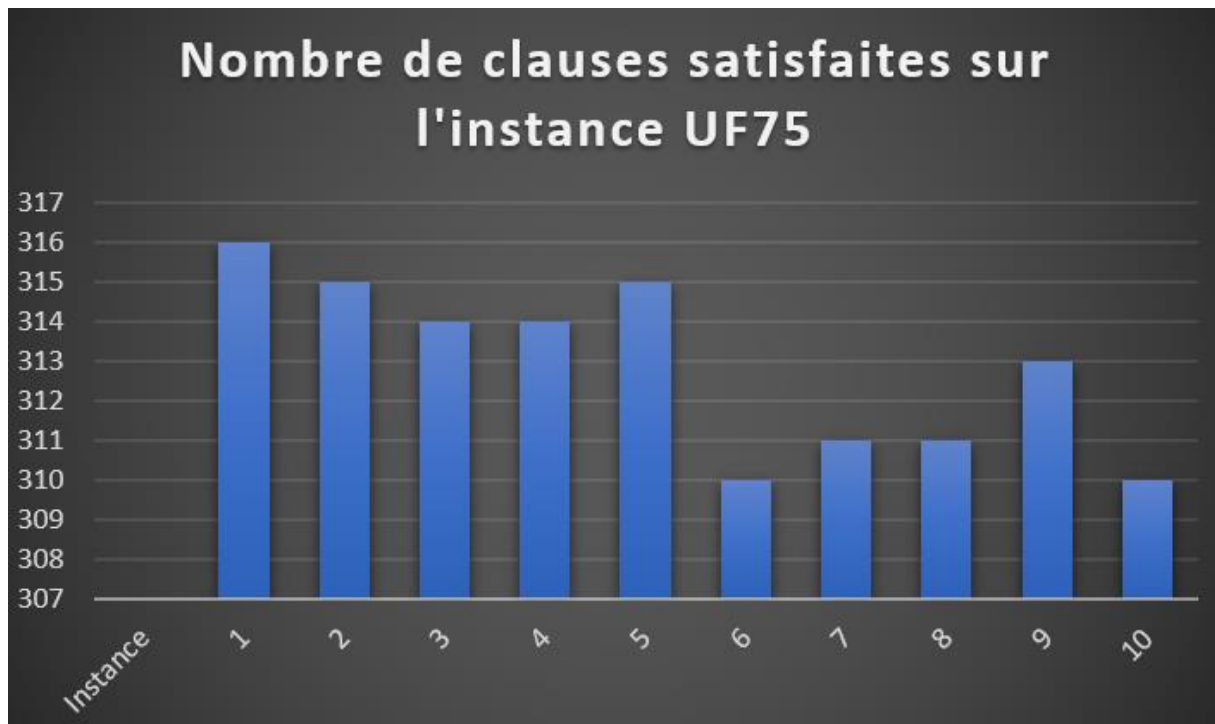


Figure 49-ACS Nombre de clauses satisfaites sur l'instance UF75

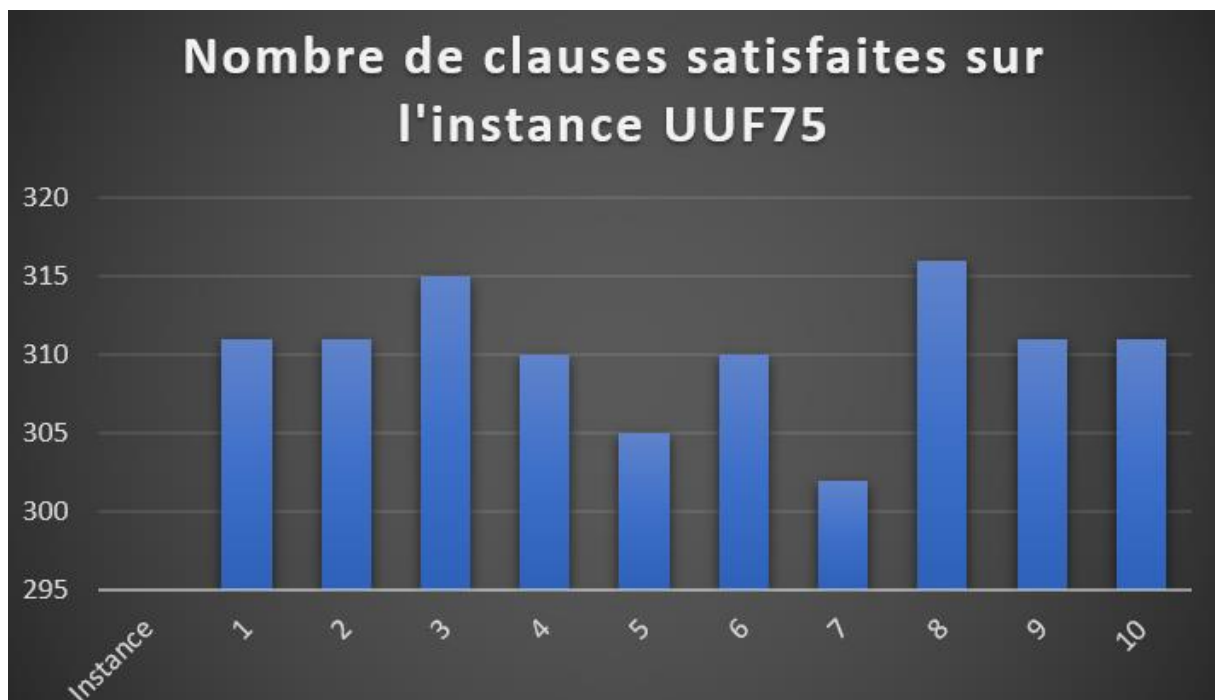


Figure 50-ACS Nombre de clauses satisfaites sur l'instance UUF75

L'exécution de l'algorithme de résolution de SAT basé sur la métaheuristique « ACS » à donner des résultats assez satisfaisants au bout de quelques secondes. L'algorithme a pu satisfaire jusqu'à 97% des clauses pour certaines instances.



### Comparaison des résultats de trois parties

Afin de pouvoir comparer les méthodes implémentées ci-dessus dans les trois parties nous avons choisi la première instance comme le montre la figure ci-dessous

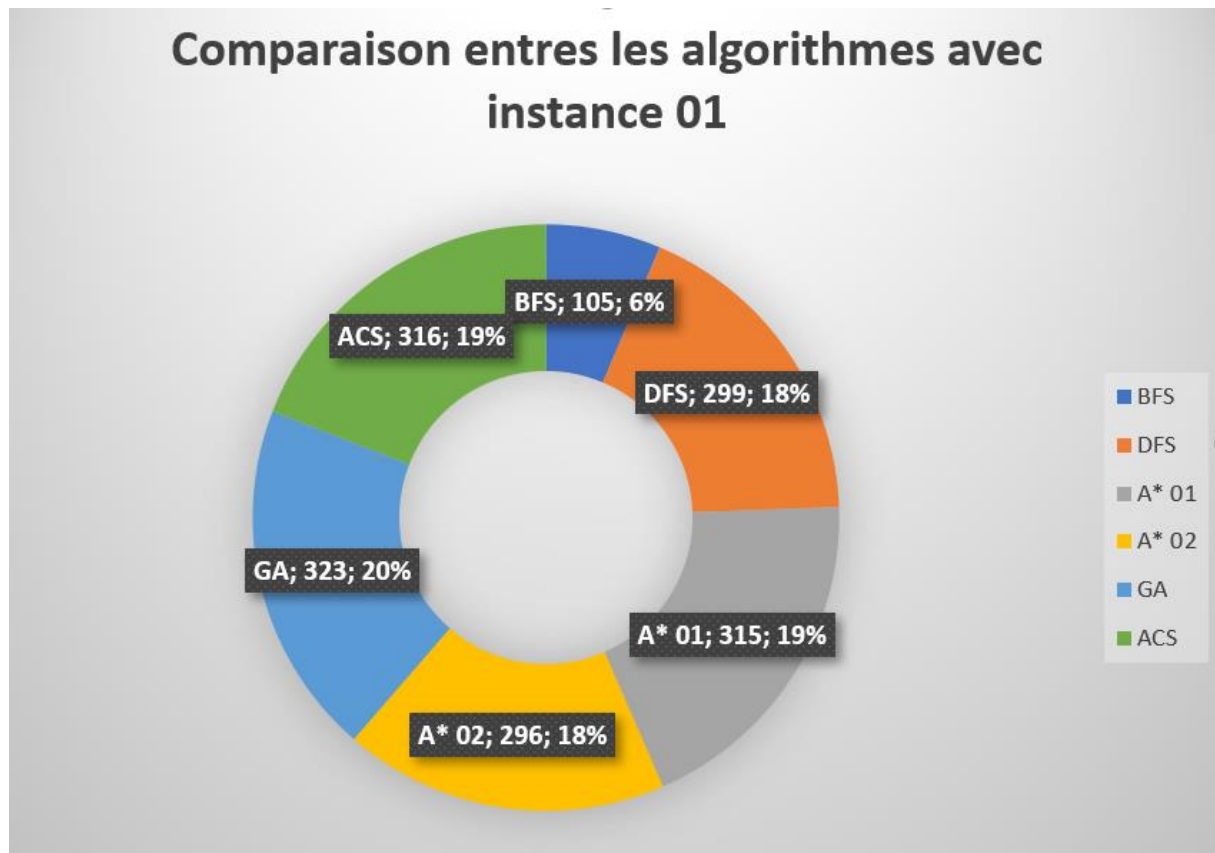


Figure 51-Comparaison

Après avoir bien observé et analysé les résultats obtenus après une série d'exécutions différentes sur les méthodes aveugles, informées et les métaheuristiques, nous avons constaté que les métaheuristiques donnent de meilleurs résultats en termes de taux de satisfaction dans un laps de temps très réduit.

Entre les résultats obtenus dans les deux métaheuristiques implémentés et testés (GA et ACS) on constate qu'il y a une différence minime avec un taux de satisfaisabilité = 99% avec GA et 97% avec ACS ce qui implique que GA est légèrement meilleur que l'ACS en termes de résultats pour le problème SAT sous nos conditions matérielles et paramètres empiriques.

Par contre si on compare les résultats obtenus avec les métaheuristiques et celles obtenues avec les méthodes de la partie une, on remarque une différence de taille soit en termes de taux de satisfaisabilité ou bien en termes de complexité temporelle et/ou spatiale.

### Conclusion

Le travail sur ce projet nous a permis de confirmer qu'il est vraiment impossible de résoudre des problèmes de très grandes tailles tel que le problème SAT, avec des méthodes exhaustives ou purement algorithmiques, qui sont coûteuses en termes de temps et/ou espace mémoire.

Les résultats obtenus avec les métaheuristiques ont prouvé leurs performances, indispensabilité, et efficacité pour progresser de manière essentielle (s'approcher de la solution optimale) dans la résolution des problèmes de grande taille dans un laps du temps très réduit.

## Bibliographie

[1](s.d.). Récupéré sur <https://tel.archives-ouvertes.fr/tel-01293722/document>

[3] *Algorithme de colonies de fourmis*. (s.d.). Récupéré sur wikipedia:  
[https://fr.wikipedia.org/wiki/Algorithme\\_de\\_colonies\\_de\\_fourmis](https://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis)

[2] [https://fr.wikipedia.org/wiki/Java\\_\(langage\)](https://fr.wikipedia.org/wiki/Java_(langage)). (s.d.).