Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'électronique et d'informatique
Département d'informatique

# Rapport de TP

Module : Swarm Intelligence

Master 1 SII

**TP**

## Résolution du problème de satisfiablitié (Approche par espace des états et par espace des solutions)

- Réalisé par :

**BENHADDAD Wissam**
**BOURAHLA Yasser**

**23-02-2018**

# Table des matières


**I Approche par espace des états** 2

**1 Introduction :** 3

1.1 Problématique : . . . . . 3
1.2 Définitions . . . . . 3
1.2.1 Problème SAT . . . . . 3
1.2.2 Stratégie de recherche dans l'espace des états . . . . . 4
1.2.3 Stratégie de recherche aveugle . . . . . 4
1.2.4 Stratégie de recherche guidée . . . . . 5

**2 Implémentation des méthodes de constructives pour SAT** 6

2.1 Structures de données . . . . . 6
2.1.1 Représentation du problème SAT . . . . . 6
2.1.2 Représentation des états . . . . . 7
2.1.3 Développement des états . . . . . 8
2.2 Conception et pseudo-code . . . . . 9
2.2.1 Gestion de la liste open . . . . . 10
2.2.2 Fonction d'évaluation d'un noeud . . . . . 11
2.2.3 Évaluateur SAT . . . . . 12

**3 Expérimentations** 14

3.1 Donnés . . . . . 14
3.1.1 Format DIMACS . . . . . 14
3.1.2 Example . . . . . 15
3.1.3 Type d'instances . . . . . 15
3.2 Environement de travail . . . . . 16
3.2.1 Machines utilisées pour les expérimentations . . . . . 16
3.2.2 Outils utilisés . . . . . 17
3.3 Interface graphique . . . . . 17
3.4 Résultats . . . . . 20
3.4.1 En largeur d'abord : . . . . . 20
3.4.2 Par profondeur d'abord : . . . . . 23
3.4.3 Cout uniforme . . . . . 25
3.4.4 Recherche gloutonne . . . . . 27
3.4.5 Algorithme A* . . . . . 29
3.5 Statistiques . . . . . 31
3.6 Comparaison entres les cinq méthodes . . . . . 33

**II Approche par espace des solutions BSO** 35

**4 Introduction :** 36
4.1 Problématique : 36
4.2 Définitions 36
4.2.1 Espace des solutions 36
4.2.2 Méta-heuristique 37
4.2.3 Intelligence en essaim (Swarm intelligence) 37
4.2.4 Bee swarm optimization (BSO) 37

**5 Implémentation de l'algorithme BSO pour le problème SAT** 38
5.1 Structures de données : 38
5.1.1 Représentation d'instance et de solutions SAT : 38
5.1.2 La table Dance : 39
5.2 Conception et pseudo-code : 39
5.2.1 Algorithme de recherche : 39
5.2.2 Le paramétrage empirique : 40
5.2.3 Le paramétrage dynamique : 42

**6 Expérimentations** 44
6.1 Données 44
6.2 Résultats 44
6.2.1 BSO avec paramétrage statique 44
6.2.2 BSO avec paramétrage dynamique 45
6.3 Comparaison avec les méthodes de I 48

**III Approche par espace des solutions ACO** 49

**7 Introduction :** 50
7.1 Problématique : 50
7.2 Définitions 50
7.2.1 Ant Colony Optimization(ACO) 50
7.2.2 Phéromones 51

**8 Implémentation des algorithmes AS/ACS pour le problème SAT** 52
8.1 Structures de données : 52
8.1.1 Représentation d'instance et de solutions SAT : 52
8.1.2 La table des Phéromones 53
8.2 Conception et pseudo-code : 54
8.2.1 AS(Ant System) 55
8.2.2 ACS(Ant Colony System) 57
8.3 Actions post-construction(Deamons) 60

**9 Expérimentations** 61
9.1 Données 61
9.2 Machines 61
9.3 Résultats 61
9.3.1 ACS 61
9.3.2 AS 62

9.4 Comparaison entre AS et ACS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

**IV Comparaisons et conclusions générale** **63**

**10 Comparaisons des trois approches** **64**
10.1 Résumé . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**A Code source BSO** **71**

**B Code source ACO** **75**

# Première partie

# Approche par espace des états

# Chapitre 1

# Introduction :

## 1.1 Problématique :

Pour ce projet, nous allons tenter d'implémenter et de comparer plusieurs méthodes de résolutions aveugles, dites aussi **à base d'espace d'états**, Pour la résolution du problème de satisfiabilité, plus communément appelé **Problème SAT**, Ce travail est aussi une application directe des différentes méthodes vues durant le premier semestre en ce qui concerne la **Résolution de problèmes**, mais aussi la **Complexité des algorithmes et les structures de données**.

## 1.2 Définitions

Avant de rentrer dans les détails de la résolution du problème, nous devons d'abord définir ce qu'est le problème SAT, ainsi que les différentes méthodes utilisées pour sa résolution dans ce projet.

### 1.2.1 Problème SAT

Dans le domaine de l'informatique et de la logique, le problème de satisfiablité (**SAT**), est un problème de décision où il s'agit d'assigner des valeurs de vérité à des variables tel qu'un ensemble de clauses en forme normale conjonctives FNC [1] préalablement défini soit satisfiable. En d'autres termes, que toutes les clauses soient vraies pour les mêmes valeurs de vérité de leurs littéraux [2]. Ce problème est le premier à avoir été démontré comme étant **NP-Complet**, et cela par Stephen Cook dans [2], et qui a donc posé les fondements de l'informatique théoriques et de la théorie de la complexité.

1. Une conjonction de disjonction de littéraux
2. Une variables logique ou bien sa négation

### 1.2.2 Stratégie de recherche dans l'espace des états

En considérant l'espace de recherche comme étant une arborescence, dont les nœuds sont les différents états du problème, nous pouvons classer les différentes stratégies de recherche en deux grandes catégories :

### 1.2.3 Stratégie de recherche aveugle

Cette catégorie englobe les stratégies ou il est question de passer par toutes les solutions et les tester une à une. Dans ce projet nous nous intéresserons plus particulièrement aux algortihmes/methodes suivant(es) :

#### Recherche par profondeur d'abord (DFS)

L'algorithme de parcours en profondeur d'abord consiste à visité un nœud de départ (souvent appelé **racine**), puis visite le premier sommet voisin(ou **successeur**) jusqu'à ce qu'une profondeur limite soit atteinte ou bien qu'il n'y ait plus de voisin à développer, une variante de cet algorithme utilise deux ensemble **Open** et **Closed** qui représentent respectivement l'ensemble des nœuds du graphe qui n'ont pas encore étés développés et ceux déjà développés. Cet ajout permet à l'algorithme d'éviter de boucler indéfiniment sur un ensemble de nœuds.

#### Recherche en Largeur d'abord (BFS)

Cet algorithme diffère de son prédécesseur par le fait qu'il visite tous les voisins(**successeurs**) d'un nœud avant de passer au noeud suivant, ce qui revient à gérer l'ajout et la suppression de l'ensemble Open comme une file, donc en mode **FIFO** (En supposant bien sûr qu'on dispose des deux ensembles open et closed), cet approche permet de sauvegarder tous les noeuds précèdemment visité durant la recherche, ce qui peut causer un débordement de la mémoire lors de l'exécution sur machine(Ce point sera rediscuté dans 2.1.3 page 8 et 3.4.1 page 20 et 3.6 page 33)

#### Par coût uniforme

Le principe est simple, au fur et à mesur que l'algorithme avance et développe des noeuds, il garde en mémoire le coût [1], le noeud qui sera ensuite choisi sera celui dont le coût accumulé est le plus bas, assurant ainsi de toujours choisir le chemin le plus optimal, si le coût pour passer d'un noeud à n'importe quel autre de ses voisins est le même quelque soit le neoud, l'algorithme est alors équivalent à celui de la recherche en largeur d'abord

### 1.2.4 Stratégie de recherche guidée

Cette catégorie englobe quant à elle les stratégies ou il est question de parcourir une plus petite partie de l'espace de recherche dans l'espoir de trouver la solution optimal en un temps plus réduit, les algorithmes sont les suivants :

#### Recherche gloutonne (Greedy algorithm)

Cet algorithme est basé sur la notion d'heuristique [1], au lieu de parcourir de façon "naïve" l'ensemble des noeuds dans l'espace de recherche, il choisit a chaque itérration sur l'ensemble **open** le noeud le plus **prometteur** en terme de distance par rapport au but recherché.

#### Algorithme A*

Contrairement aux précédents algorithmes de recherche qui effectuaient une recherche de façon "naïve", l'algorithme **A*** propose une vision un peu nouvelle, il utilise la notion de coût et celle d'heuristique, la fonction d'évaluation $f$ est donc définie comme étant la somme de deux fonctions $g$ et $h$ ou :

- $g$ est la fonction qui retourne le cout d'un noeud $n$
- $h$ est la fonction qui estime le cout d'un noeud $n$ vers le but

Le principe de l'algorithme est donc de prendre le noeud dans **open** qui possède la valeur minimal de $f$, assurant ainsi de trouver le chemin optimal **ssi.** l'heuristique $h$ choisie est consistante [2]

1. Fonction retournant le cout pour passer du noeud de départ(la racine) au noeud courant
1. Une fonction d'éstimation de la distance séparant le noeud courant au but
2. Ne surestime jamais le coût réel pour passer d'un nœud à un de ses successeurs

# Chapitre 2

# Implémentation des méthodes de constructives pour SAT

## 2.1 Structures de données

La stratégie de recherche avec graphe requiert une représentation des entrées du problème, des états construisant une solution potentielle à ce dernier ainsi que le développement de ces états.

### 2.1.1 Représentation du problème SAT

Une instance du problème SAT peut être considérée comme un ensemble de clauses, chacune de ces clauses est une disjonction de littéraux. Dans ce rapport Nous proposons deux structures différentes pour les représenter que nous comparerons par la suite.

#### Représentation matricielle

Une première représentation serait d'associer à chaque clause de l'instance un tableau de taille égale au nombre de variables logiques utilisés dont la $i^{\text{ème}}$ case aura la valeur 1 si la variable *i* est présente dans la clause, -1 si sa négation est présente, 0 sinon. Ainsi en représentant toutes les clauses on obtient une matrice dont chaque ligne est associée à une clause.
L'exemple suivant montre une instance du problème SAT et sa représentation matricielle :

$$X = x_1, x_2, x_3, x_4, x_5$$

$$C = c_1, c_2, c_3$$

$$x_1 \vee \neg x_2 \vee x_5$$
$$\neg x_2 \vee x_4 \vee x_5$$
$$\neg x_1 \vee x_2 \vee \neg x_3$$

Ces clauses vont être représentée comme suit :

| | | | | |
|---|---|---|---|---|
| 1 | -1 | 0 | 0 | 1 |
| 0 | -1 | 0 | 1 | 1 |
| -1 | 1 | -1 | 0 | 0 |

**Représentation par *Bitset***

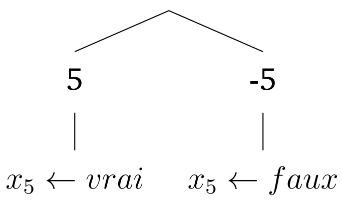
On pourrait aussi aborder la représentation du point de vu littéral, c'est à dire associer à chaque littéral les clauses dans lesquels il est présent. Pour cela un tableau de bits appelé *Bitset* pourrait être utilisé où chaque bit *i* aurait la valeur 1 si la $\text{i}^{\text{ième}}$ clause contient le littéral, la valeur 0 sinon. On obtient donc un tableau de taille 2 fois le nombre de variables utilisés dont les entrées représentent les *Bitsets* des littéraux.
Pour le même exemple vu précédemment on obtient les *Bitsets* suivants :

| | | | |
|---|---|---|---|
| $x_1$ | 1 | 0 | 0 |
| $x_2$ | 0 | 0 | 0 |
| $x_3$ | 0 | 0 | 0 |
| $x_4$ | 0 | 1 | 0 |
| $x_5$ | 1 | 1 | 0 |

| | | | |
|---|---|---|---|
| $\neg x_1$ | 0 | 0 | 1 |
| $\neg x_2$ | 1 | 1 | 0 |
| $\neg x_3$ | 0 | 0 | 1 |
| $\neg x_4$ | 0 | 0 | 0 |
| $\neg x_5$ | 0 | 0 | 0 |

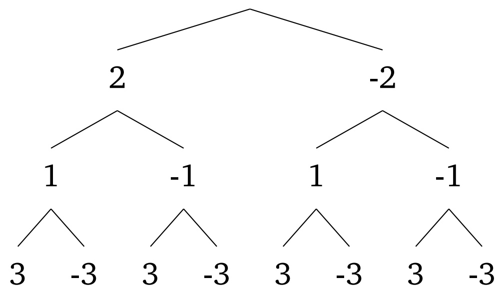
### 2.1.2 Représentation des états

Une solution à une instance du problème SAT se réduit à l'assignation des valeurs de vérités aux variables logiques de cette instance. On peut considérer un état dans l'espace de recherche comme étant le choix de la valeur de vérité d'une des variables logiques, on obtient après une succession de choix une solution au problème qui peut être positive si les valeurs assignées sont consistantes avec les clauses de l'instance, négatives sinon. Nous allons représenter un état avec un noeud qui contient le numéro de la variable choisie, multiplié par -1 pour désigner l'assignation de la valeur $faux$ à la variable, il reste inchangé sinon.

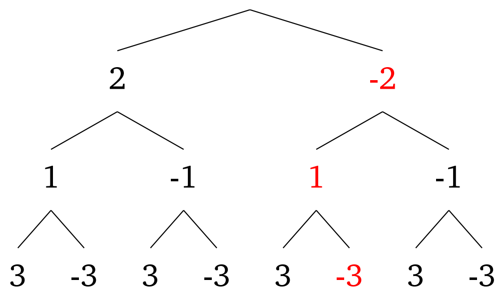


### 2.1.3 Développement des états

A partir de chaque état on peut faire le choix de la valeur de vérité d'une variable logique choisie aléatoirement. Le développement d'un noeud donne deux successeurs, un pour chaque valeur de vérité assignée à la prochaine variable. On obtient après l'exploration de l'espace de recherche un arbre d'états, l'exemple suivant est un arbre associé à une instance SAT contenant trois variables.




Une solution est représentée par une branche de l'arbre, par exemple la solution : $x_1 = vrai$, $x_2 = faux$, $x_3 = faux$ est représentée dans l'arbre précédent comme suit :




Pour pouvoir construire une solution à partir de n'importe quel noeud, on doit y sauvegarder l'adresse de son parent, ainsi on peut récupérer les valeurs assignées aux noeuds précédents jusqu'à la racine. L'enregistrement suivant représente un noeud de l'arbre :

```
struct {
int valeur;
struct noeud* parent;
} noeud;
```

**Remarque1 :** Un inconvénient que nous avons déjà cité de la recherche en largeur d'abord était la saturation rapide de la mémoire, cela est dû au fait de garder tous les noeuds dans la mémoire. Ce problème est évité dans la recherche en profondeur d'abord car dès l'évaluation d'un noeud se trouvant dans la profondeur maximale, ce dernier est supprimé de la mémoire. notons que la structure du noeud déjà présentée ne contient pas les adresses de ses successeurs, ceci nous permet d'éviter de garder tous l'arbre d'états dans la mémoire mais juste les branche susceptible d'être évaluée par la suite.

**Remarque2 :** Dans la deuxième représentation du problème SAT, une optimisation serait d'ajouter un $Bitset$ dans la structure du noeud et y garder les clauses qu'il satisfait ainsi que celles de ses parents, celui là peut être obtenu en appliquant l'opération OU logique sur le $Bitset$ du noeud parent et celui du littéral choisi.

| | | | | |
|---|---|---|---|---|
| $Bitset$ du parent | 1 | 0 | 1 | 1 |

OR

| | | | | |
|---|---|---|---|---|
| $Bitset$ du littéral | 1 | 0 | 0 | 1 |

$\downarrow$

| | | | | |
|---|---|---|---|---|
| $Bitset$ du noeud | 1 | 0 | 1 | 1 |

## 2.2 Conception et pseudo-code

Dans cette partie nous allons présenter l'implémentation des algorithmes de recherche avec graphe, un algorithme générique qui englobe les différente méthodes est présenté si dessous :

```
Algorithme 1 : Algorithme de recherche avec graphe
Résultat : retourne la solution ou échec
1  open ← état initial;
2  initialiser l'ensemble closed à vide;
3  tant que ¬vide open faire
4      noeud ← choisir_noeud(open);
5      si noeud_but(noeud) alors
6          retourner solution(noeud);
7      fin
8      ajouter(noeud,closed);
9      successeurs ← développer(noeud) ;
10     inserer les successeurs qui n'appartiennent pas à closed dans open
11 fin
12 retourner echec;
```

La différence entre les algorithmes de recherche réside dans la manière dont on sélectionne le noeud à évaluer, ligne 4 dans l'algorithme si dessus, ainsi que l'estimation du coût et de l'heuristique, s'ils existent, avant l'insertion, ligne 10.

En se basant sur cet algorithme nous avons implémenté une procédure de recherche générique prenant en paramètre un type de gestion de liste, un estimateur de coût et d'heuristique et les entrées de l'instance SAT afin d'évaluer les noeuds.

### 2.2.1 Gestion de la liste open

#### Recherche par profondeur d'abord

La recherche en profondeur d'abord consiste à choisir le noeud avec la profondeur la plus élevé de l'arbre, ceci reviens à sélectionner l'élément le plus récemment inséré dans la liste open, c'est à dire, la gérer avec une politique FIFO.
Insertion d'un noeud :

$$\boxed{3} \quad \rightarrow \quad \boxed{3}\boxed{-3}\boxed{-2}\boxed{-1}$$

Sélection d'un noeud :

$$\boxed{3} \quad \leftarrow \quad \boxed{-3}\boxed{-2}\boxed{-1}$$

#### Recherche en largeur d'abord

Contrairement à la recherche en profondeur d'abord, les noeuds sont visités de tel sorte à parcourir l'arbre niveau par niveau, cela peut être réalisé par la sélection du noeud le moins récemment insérer dans open, d'où une gestion LIFO de la liste.
Insertion d'un noeud :

$$\boxed{3} \quad \rightarrow \quad \boxed{3}\boxed{-3}\boxed{-2}\boxed{-1}$$

Sélection d'un noeud :

$$\boxed{-1} \quad \leftarrow \quad \boxed{3}\boxed{-3}\boxed{-2}$$

#### Recherche En se basant sur une fonction d'évaluation

Dans ce type de recherche, la sélection d'un noeud se fait sur la base d'une fonction d'évaluation. Le noeud sélectionné est celui avec la valeur minimale (resp. maximale) de la fonction d'évaluation. Nous utilisons ce type de gestion afin d'implémenter les algorithmes : recherche à coût uniforme, recherche gloutonne et l'algorithme A*.
Nous avons implémenter ce type de gestion avec deux structures différentes que nous comparerons dans la suite de ce rapport.

**Liste triée** Les noeuds sont triés dans une liste selon leur valeur estimée par la fonction d'évaluation. Le premier noeud est toujours sélectionné, l'insertion par contre se fait de telle sorte à garder la liste triée en ordre croissant (resp. décroissant).
Insertion d'un noeud :

$$\boxed{3}\boxed{f_3 = 5} \quad \rightarrow \quad \boxed{-3}\boxed{f_{-3} = 2}\boxed{3}\boxed{f_3 = 5}\boxed{-1}\boxed{f_{-1} = 8}$$

Sélection d'un noeud :

$$\boxed{-3}\boxed{f_{-3} = 2} \quad \leftarrow \quad \boxed{3}\boxed{f_3 = 5}\boxed{-1}\boxed{f_{-1} = 8}$$

Complexité de l'insertion : $o(n)$.
Complexité de la sélection : $o(1)$.

**Remarque** Les noeuds sont organisé dans une structure de tas [1]. La racine du tas est sélectionner pour l'évaluation, tandis que l'insertion se fait par entassement du nouveau élément. Les deux opérations se font en $o(log(n))$.

### 2.2.2 Fonction d'évaluation d'un noeud

La fonction d'évaluation f d'un noeud n est généralement définit à l'aide de deux autres fonctions g et h. La première désigne le coût nécessaire pour atteindre le noeud n à partir de la racine, tandis que la deuxième est une heuristique qui estime le coût restant avant d'arriver au but.

#### Recherche gloutonne

La fonction d'évaluation dans ce cas f est égale à h, on se contente de la valeur estimé par l'heuristique pour décider le prochain noeud à développer. Une heuristique pour le problème SAT qui peut mesurer la distance des noeuds par rapport au noeud but serait de calculer le nombre de clauses pas encore satisfaites, le noeud avec la valeur minimale de cette heuristique est le noeud qui satisfait le plus de clauses et donc le plus proche de satisfaire toutes les clauses.

#### Recherche à coût uniforme

Contrairement à la recherche gloutonne, la recherche à coût uniforme n'utilise que la fonction g, permettant ainsi de développer le noeud le plus proche de la racine en terme de coût. Cependant trouver une fonction d'estimation du coût pour le problème SAT s'avère délicat comme on ne peut pas vraiment déterminer une distance entre un noeud et la racine. Ceci dit, une fonction de coût qui calcule le nombre de clauses devant être satisfaite par un noeud mais qui sont déjà satisfaite par ses parents peut être utilisé. Cela représente la perte d'une branche contenant des littéraux qui satisfont les même clauses de l'instance SAT, plus le coût est élevé, moins les chances que cette branche nous mène au but.

1. un tas est un arbre équilibré dont chaque noeud a une clé supérieur (resp. Inférieur) à celle de ses fils

| | | | | | $\rightarrow$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *Bitset* du parent | 1 | 0 | 0 | 1 | $\rightarrow$ | *Bitset* du littéral | 1 | 1 | 0 | 0 |

coût = 1

#### Algorithme A*

L'algorithme A* combine les deux fonctions g et h citées précédemment afin d'évaluer les noeuds en prenant en considération le nombre de clauses déjà satisfaites ainsi que le coût de la branche dans laquelle il se trouve.

### 2.2.3 Évaluateur SAT

Dans cette partie nous présentons deux méthodes d'évaluation du noeud but basé sur les deux structures représentatives des instances SAT citées précédemment.

#### Évaluation par matrice

La première méthode consiste à parcourir la matrice des clauses et chercher pour chaque clause si un de ses littéraux a été évalué vrai par les noeuds de la solution. Si dessous l'algorithme correspondant.

---

**Algorithme 2 :** Algorithme d'évaluation par matrice

---

**Résultat :** retourne un booléen : vrai si la solution est positive, faux sinon

**entré** : $solutionpartielle$;
**pour** $clause \in matrice$ **faire**
    $satC \leftarrow faux$ ;
    **pour** $noeud \in solution$ *et* $\neg satC$ **faire**
        **si** $clause[\textbf{\textit{abs}}(noeud.valeur)] \times noeud.valeur > \mathit{0}$ **alors**
            $satC \leftarrow vrai$;
        **fin**
    **fin**
    **si** $satC$ **alors**
        $cpt \leftarrow cpt + 1$;
    **fin**
**fin**
**si** $cpt = \textbf{\textit{taille}}(matrice)$ **alors**
    **retourner** $vrai$;
**fin**
**retourner** $faux$;

---

### Évaluation par Bitset

Comme vu précédemment, en utilisant la structure Bitset pour représenter l'instance SAT chaque noeud contient un Bitset des clauses satisfaites par sa branche, il suffit donc de calculer le nombre de bits à 1 pour décider si c'est un noeud but ou pas. Nous utilisons pour cela l'algorithme "Hamming Weight" permettant de calculer le nombre de bits à 1 dans un entier en une complexité constante.

# Chapitre 3

# Expérimentations

## 3.1 Donnés

Afin de tester notre solveur nous avons opté pour l'utilisation de fichiers benchmark qui vont représenter des instances du problème, dorénavant, et pour être plus conforme avec la terminologie du problème, nous utiliserons le terme **INSTANCE** pour désigner ces dits fichiers.

Les instances nous sont présentées sous forme de fichiers au format **DIMACS** [1](plus de détails dans 3.1.1) et sont disponibles en téléchargement gratuitement et librement dans [1], et sont également le fruit du travail de nombreux chercheurs dévoués.

### 3.1.1 Format DIMACS

Un fichier en format **DIMACS** est un fichier dont l'extension est **.cnf**, et est structuré de la manière suivante :

- Le fichier peut commencer avec des commentaires, un commentaire sur une ligne commence par le caractère **'c'**
- La première ligne du fichier(après les commentaires) doit être structurée de la manière suivante : **p cnf** **nbvar** **nbclause**
  1. **p cnf** pour indiquer que l'instance est en forme normale conjonctive **FNC**.
  2. **nbvar** indique le nombre de variable au total dans l'instance, à noté que chaque literal $x_i$ sera représenté par son indice $i$.
  3. **nbclause** le nombre total de clauses présentes dans l'instance.
- chaque ligne représente une conjonction de litéraux $(x_i|\neg x_i)$ indentifiés par un numero $i$, séparés par un blanc, avec un 0 à la fin pour marquer la fin de la ligne.

1. Représentation convetionnelle d'une instance du problème SAT

### 3.1.2 Example

```
c
c Un commentaire
c
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

### 3.1.3 Type d'instances

Dans [1] nous avons à notre disposition deux types d'instances pour chaque taille du problème :

- Un ensemble d'instances satisfiable dans un fichier dénommé UFXX-YY
- Un ensemble d'instances satisfiable dans un fichier dénommé UUFXX-YY
- avec :
  1. XX = nombre de variable
  2. YY = nombre de clauses

## 3.2 Environement de travail

### 3.2.1 Machines utilisées pour les expérimentations

Pour les tests nous avons utilisé deux machines pour chaque groupe d'instances, autrement dit une machine pour effectuer les tests sur un ensemble d'instances satisfiables **UF75-325**[1] et une autre sur les instances contradictoires(non satisfiables) **UUF75-325**[1], les caractéristiques de chaque machines sont données dans les figures 3.1 et 3.2 :


FIGURE 3.1 – Machine **A** pour les instances contradictoires

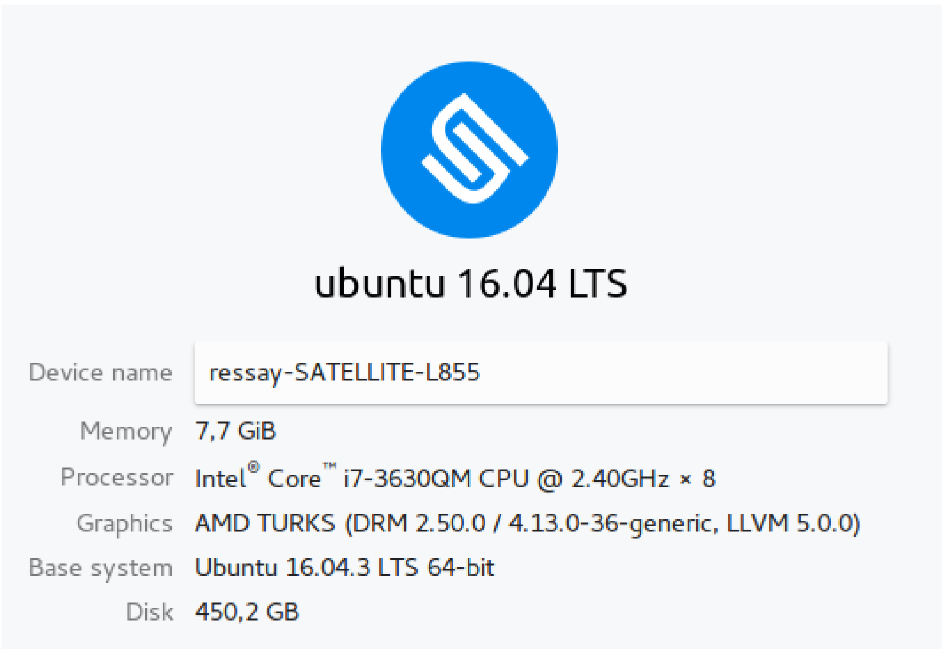



FIGURE 3.2 – Machine **B** pour les instances satisfiables

### 3.2.2 Outils utilisés

**Langage de programmation :**

Nous avons opté pour le langage Java, car il offre une grande flexibilité et facilite l'implémentation qui est due au fait qu'il soit totalement orienté-objet.

**IDE :**

**IntelliJ Idea** L'environnement de développement choisit est IntelliJ IDEA, spécialement dédié au développement en utilisant le langage Java. Il est proposé par l'entreprise JetBrains et est caractérisé par sa forte simplicité d'utilisation et les nombreux plugins et extensions qui lui sont dédiées.

## 3.3 Interface graphique

Afin de faciliter l'utilisation des méthodes, et la visualisation en temps réel du comportement de ces dernière, nous avons mis au point une interface graphique simple d'utilisation.

La fenêtre principale se décompose en quatre sections ( voir figure ci dessous **??** )

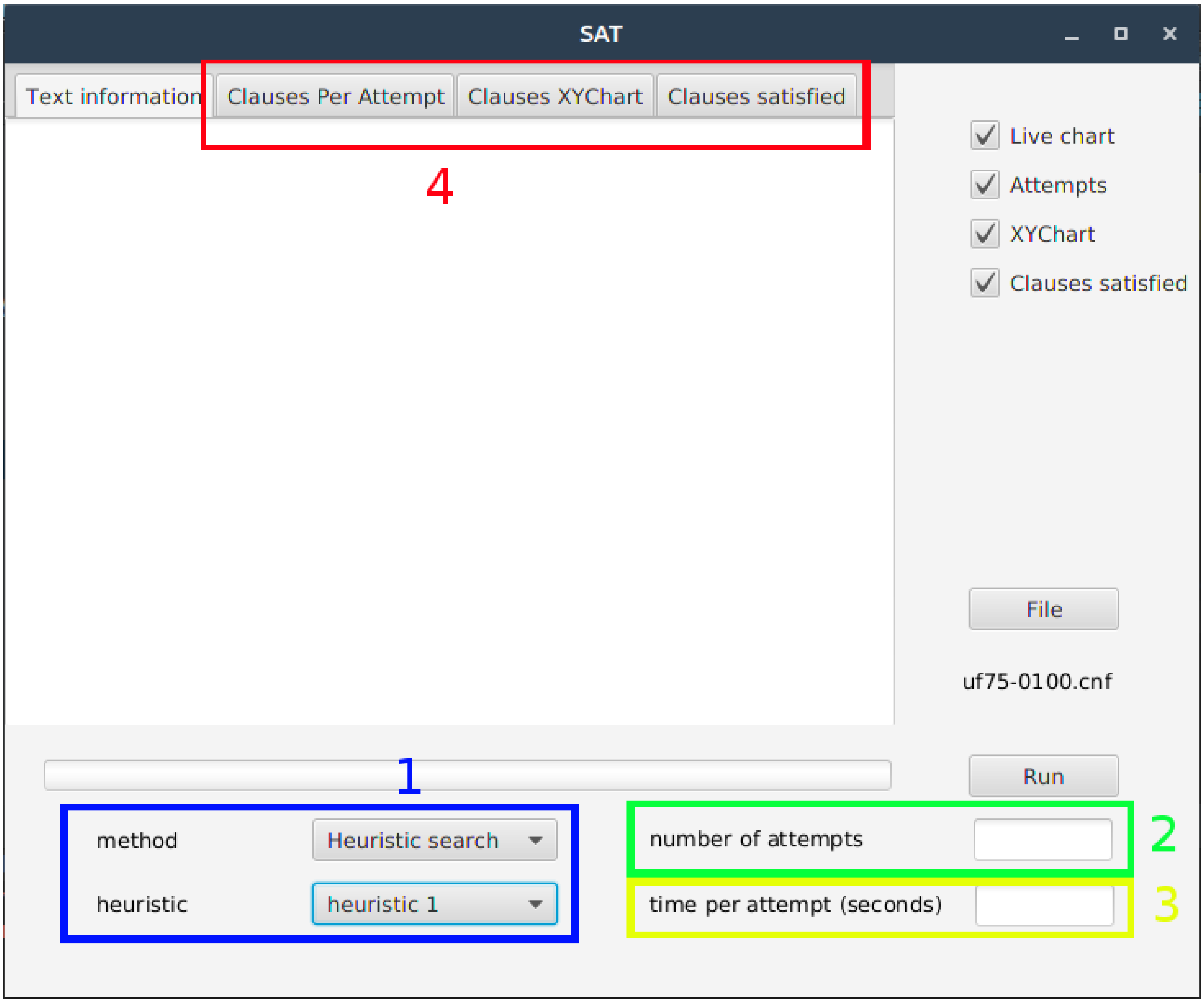



**FIGURE 3.3 –** Fenêtre principale

**Détails**

1. Une liste déroulante pour choisir la méthode de recherche désirée.
2. Le nombre d'exécutions sur une même instance.
3. La durée (en secondes) d'une exécution sur une instance.
4. Un groupe d'onglets dédiés à l'affichage de trois types de graphiques illustratifs.

Pour ce qui en est des groupes d'onglets, nous avons trois types de graphiques :

- **Attempts :** un histogramme montrant le taux de satisfiabilité pour chaque exécution sur une instances :

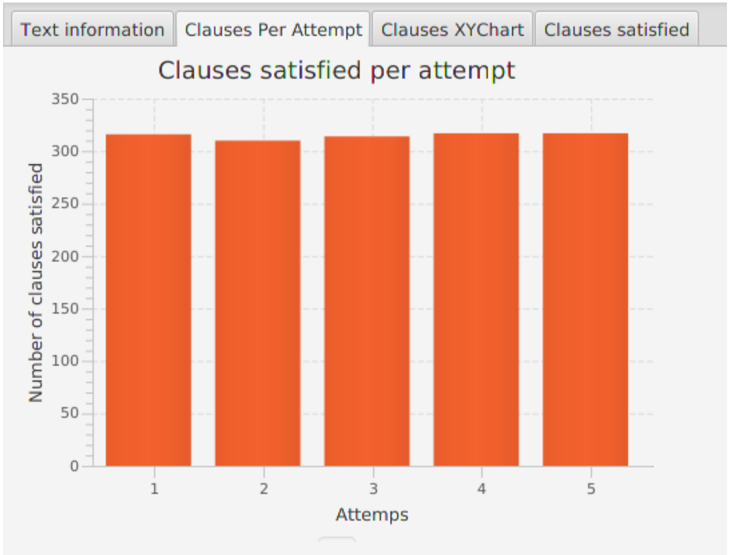



**FIGURE 3.4 –** Attempts

- **XYChart :** une courbe pour suivre l'évolution du taux de satisfiabilité pour chaque tentative :

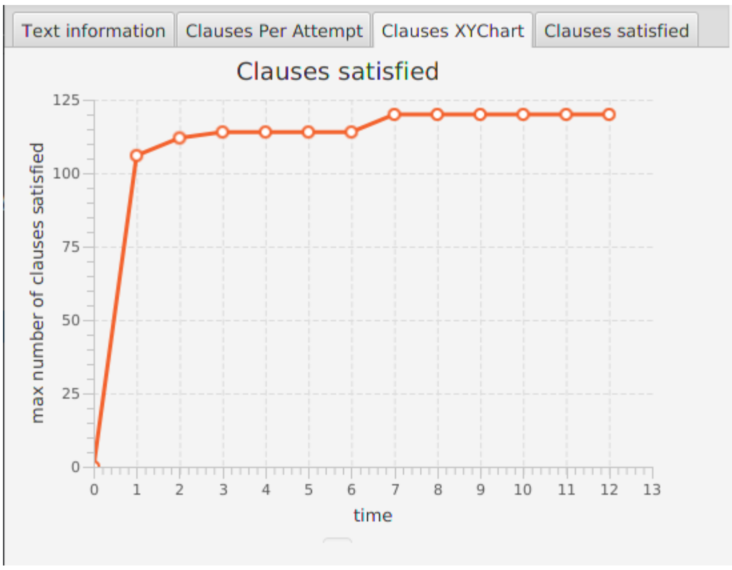



**FIGURE 3.5 –** XYChart

- **Clauses satisfied :** un histogramme qui montre la fréquence de satisfiabilité d'une clause $c_i$ durant une tentative sur l'instance courante, l'histogramme est trié pour mieux observer les données :

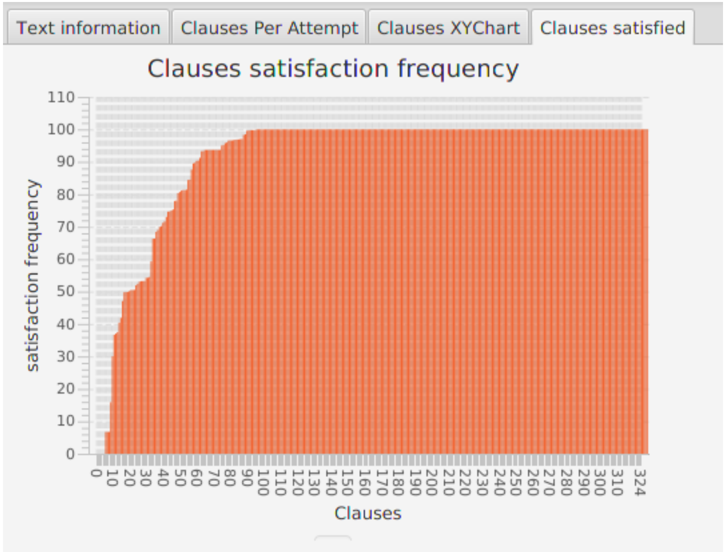



**FIGURE 3.6 –** Clauses satisfied during evaluation of execution of UF75-325-01

## 3.4 Résultats

Pour chacun des groupes d'instancs(i.e UF75-325 et UUF75-325) nous avons lancé les machines dédiées sur les 10 premières instances, avec 10 exécutions de durées égales à 10 mins pour chaque instance et pour chaque méthodes, les résultats sont les suivants :

### 3.4.1 En largeur d'abord :

Les résultats sont présentés d'abord sous forme de tables puis illustrés dans des histogrammes :

**Remarque :** En ce qui concerne cet algorithme, nous avons eu une saturation de la mémoire après 1 min d'exécution avec la structure d'évaluation en **Bitset** (voir 16 page 13) cela est principalement dû au fait que cette structure permet d'évaluer un plus grand nombre de clauses en un lapse de temps très court là où la structure d'évaluation matrcielle ( voir 2.2.3 page 12) prend plus de temps pour faire le traitement, en conséquence le débordement de la mémoire survient mais après un temps plus conséquant, les résultats obtenus sont donc ceux observé avant le débordement.

**Pour les instances satisfiables :**

| Fichiers test | Instance | Max clauses satisfaites | Taux moyen de satisfiabilité |
|---|---|---|---|
| UF75-325 | 1 | 153 | 42,83% |
| | 2 | 152 | 43,94% |
| | 3 | 147 | 42,31% |
| | 4 | 140 | 42,25% |
| | 5 | 146 | 42,46% |
| | 6 | 146 | 43,05% |
| | 7 | 144 | 41,91% |
| | 8 | 160 | 44,37% |
| | 9 | 152 | 43,04% |
| | 10 | 144 | 42,58% |

**TABLE 3.1 –** Tableau récapitulatif des résultats pour les instances satisfiables

Pour mieux visualiser les données du tableau, le graphe suivant est proposé :

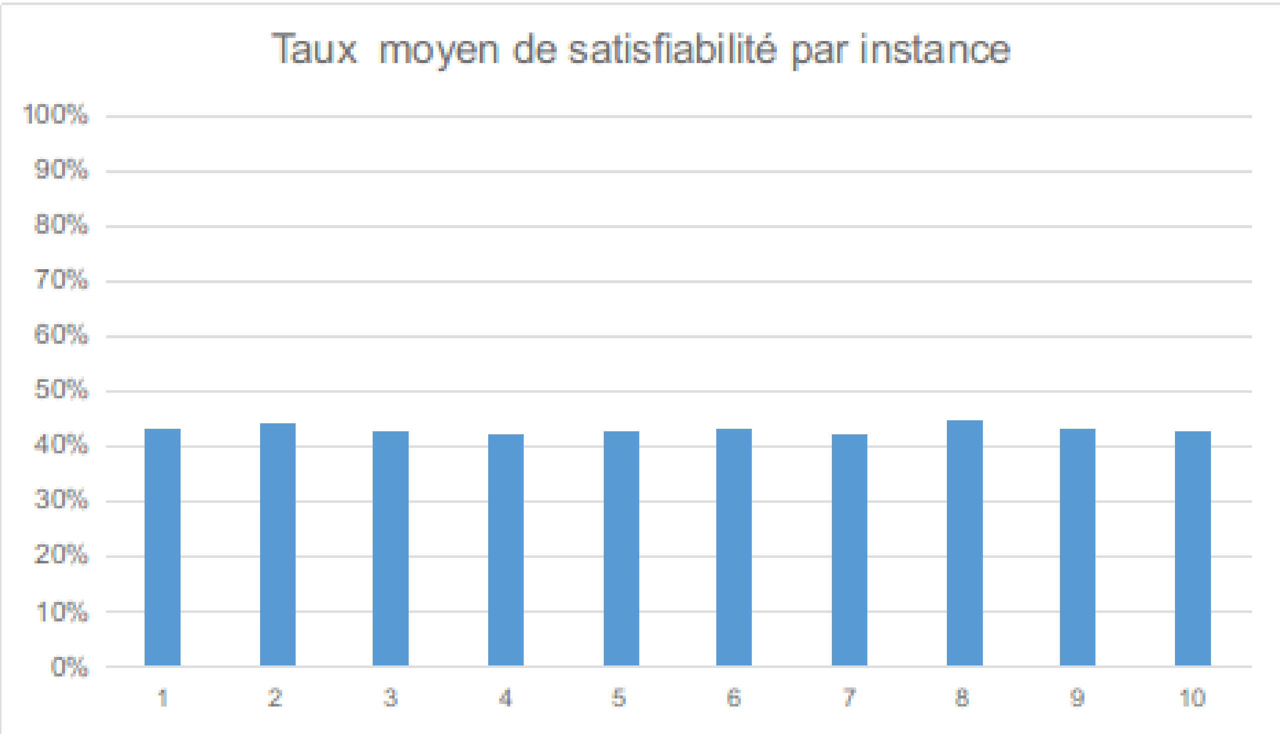



**FIGURE 3.7 –** Illustration des données de ??

**Pour les instances contradictoires ( non sastisfiables ) :**

| Fichiers test | Instance | Max clauses satisfaites | Taux moyen de satisfiabilité |
|---|---|---|---|
| UUF75-325 | 1 | 143 | 41,60% |
| | 2 | 147 | 42,95% |
| | 3 | 151 | 41,23% |
| | 4 | 136 | 41,48% |
| | 5 | 148 | 41,72% |
| | 6 | 144 | 41,05% |
| | 7 | 145 | 42,15% |
| | 8 | 145 | 41,82% |
| | 9 | 154 | 42,37% |
| | 10 | 142 | 42,15% |

**TABLE 3.2 –** Tableau récapitulatif des résultats pour les instances non-satisfiables

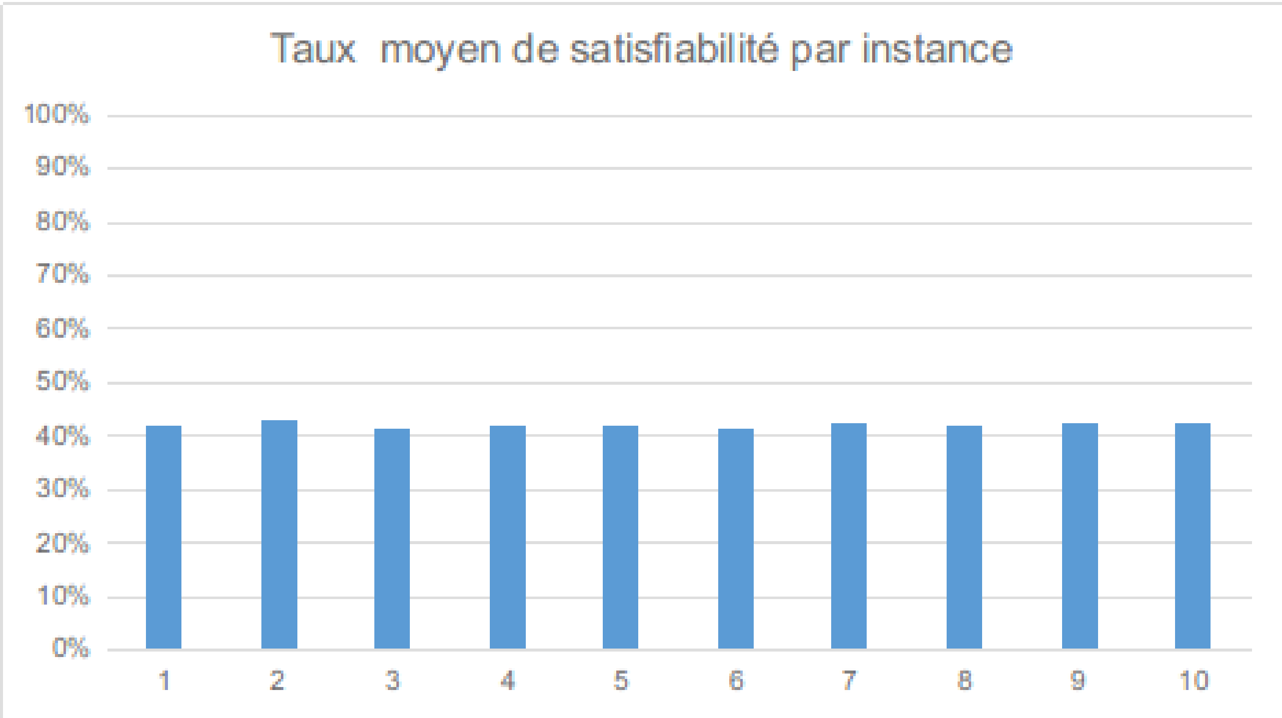



**FIGURE 3.8 –** Illustration des données de **??**

### 3.4.2 Par profondeur d'abord :

Les résultats sont présentés d'abord sous forme de tables puis illustrés dans des histogrammes :

**Pour les instances satisfiables :**

| Fichiers test | Instance | Max clauses satisfaites | Taux moyen de satisfiabilité |
|---|---|---|---|
| UF75-325 | 1 | 312 | 92,95% |
| | 2 | 306 | 92,37% |
| | 3 | 309 | 92,46% |
| | 4 | 306 | 92,65% |
| | 5 | 308 | 93,14% |
| | 6 | 310 | 94,18% |
| | 7 | 305 | 93,75% |
| | 8 | 308 | 92,49% |
| | 9 | 310 | 94,46% |
| | 10 | 306 | 94,22% |

**TABLE 3.3 –** Tableau récapitulatif des résultats pour les instances satisfiables

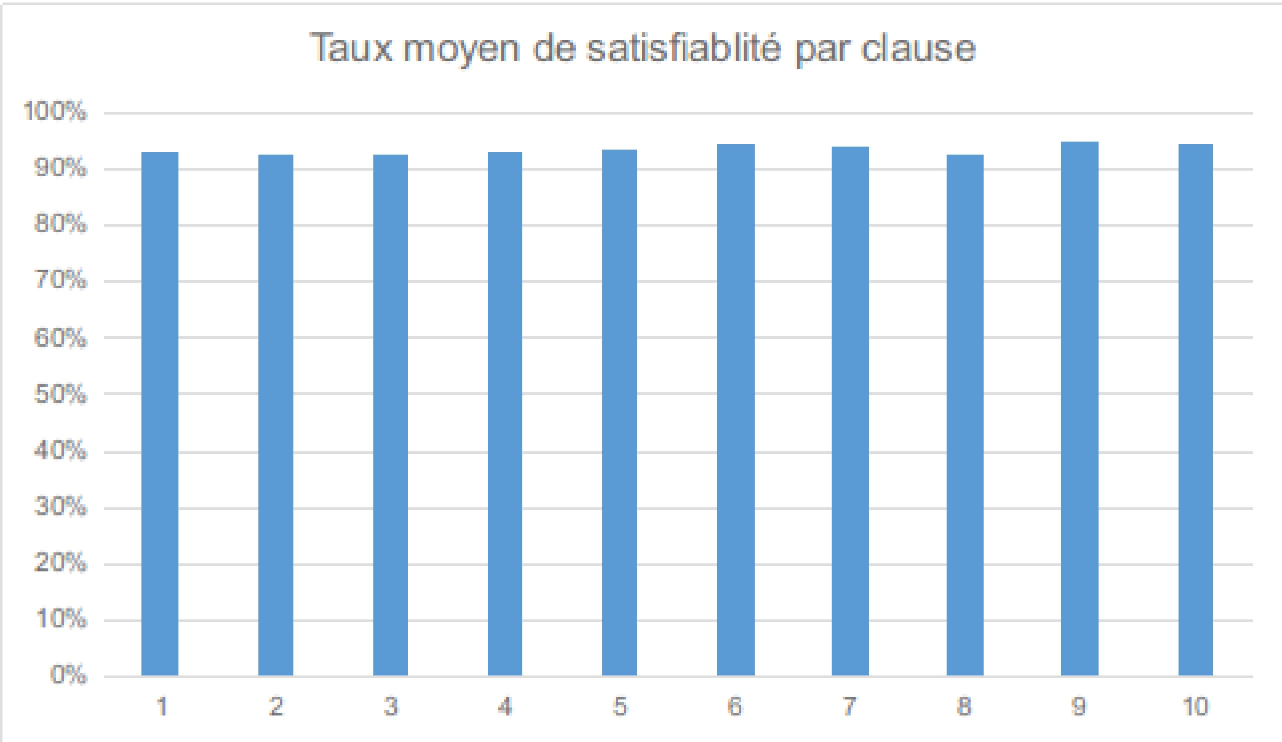



**FIGURE 3.9 –** Illustration des données de la table 3.10

**Pour les instances contradictoires ( non sastisfiables ) :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UF75-325 | 1 | 312 | 94,37% |
| | 2 | 306 | 93,29% |
| | 3 | 309 | 94,34% |
| | 4 | 306 | 92,83% |
| | 5 | 308 | 92,61% |
| | 6 | 310 | 95,38% |
| | 7 | 305 | 92,83% |
| | 8 | 308 | 93,66% |
| | 9 | 310 | 93,60% |
| | 10 | 306 | 93,33% |

**TABLE 3.4 –** Tableau récapitulatif des résultats pour les instances non-satisfiables

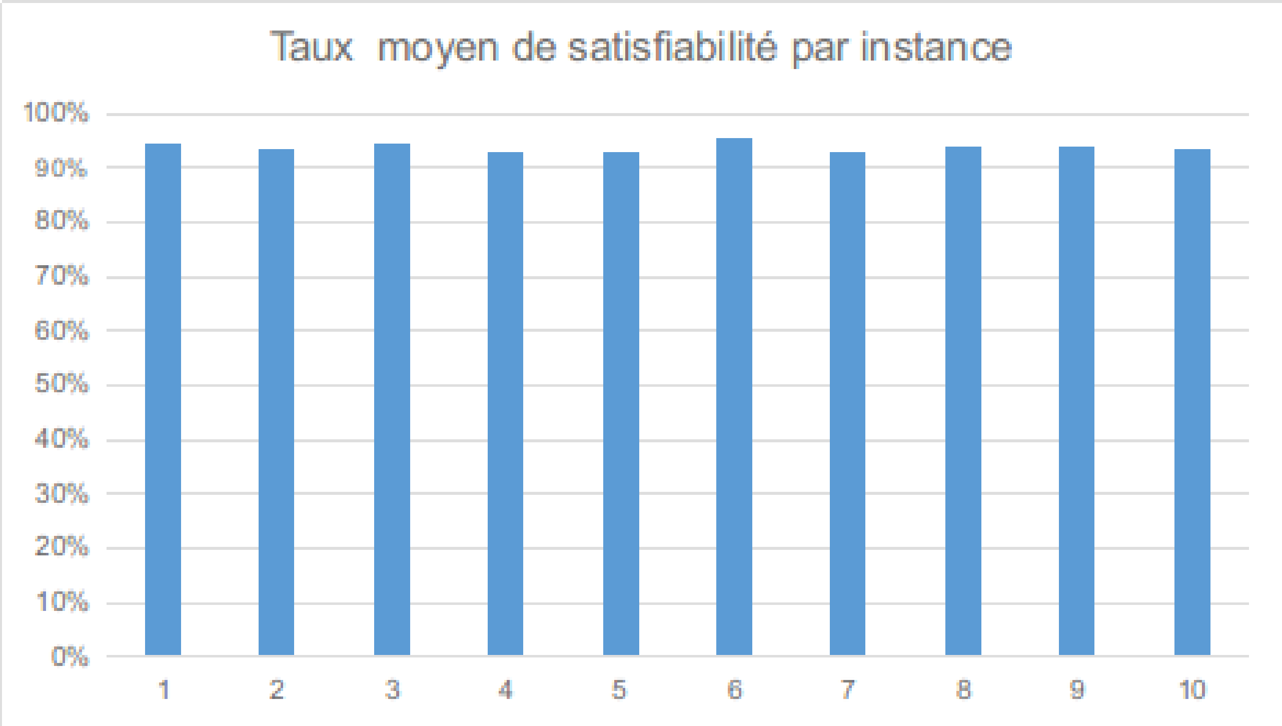



**FIGURE 3.10 –** Illustration des données de la table 3.4

### 3.4.3 Cout uniforme

: Les résultats sont présentés d'abord sous forme de tables puis illustrés dans des histogrammes :

**Pour les instances satisfiables :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UF75-325 | 1 | 307 | 93,38% |
| | 2 | 304 | 93,23% |
| | 3 | 307 | 93,23% |
| | 4 | 304 | 92,77% |
| | 5 | 303 | 93,08% |
| | 6 | 302 | 92,77% |
| | 7 | 299 | 91,69% |
| | 8 | 301 | 92,00% |
| | 9 | 307 | 93,54% |
| | 10 | 305 | 93,08% |

**TABLE 3.5 –** Tableau récapitulatif des résultats pour les instances satisfiables

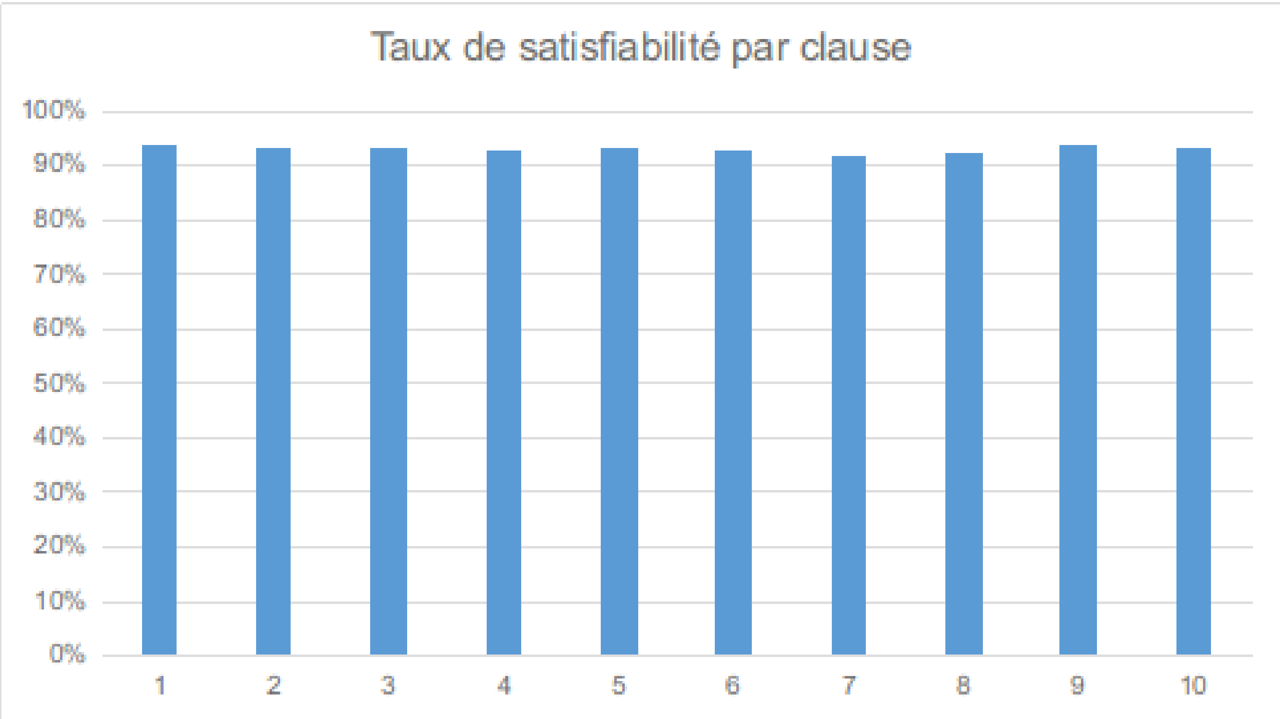



**FIGURE 3.11 –** Illustration des données de la table 3.5

**Pour les instances contradictoires ( non sastisfiables ) :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UUF75-325 | 1 | 299 | 91,69% |
| | 2 | 305 | 92,77% |
| | 3 | 301 | 92,00% |
| | 4 | 307 | 94,15% |
| | 5 | 309 | 94,92% |
| | 6 | 300 | 92,00% |
| | 7 | 303 | 92,92% |
| | 8 | 301 | 92,46% |
| | 9 | 310 | 94,62% |
| | 10 | 308 | 94,42% |

**TABLE 3.6 –** Tableau récapitulatif des résultats pour les instances non-satisfiables

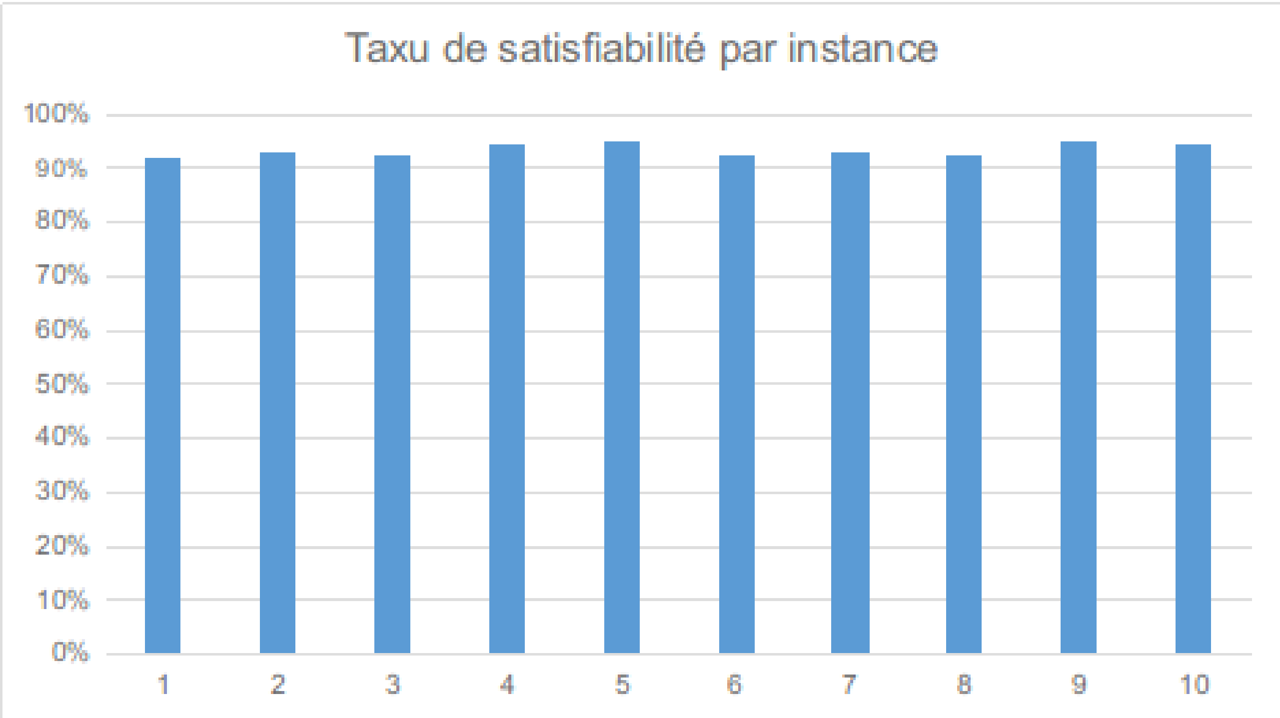



**FIGURE 3.12 –** Illustration des données de la table 3.6

### 3.4.4 Recherche gloutonne

: Les résultats sont présentés d'abord sous forme de tables puis illustrés dans des histogrammes :

**Pour les instances satisfiables :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UF75-325 | 1 | 320 | 98,15% |
| | 2 | 319 | 97,85% |
| | 3 | 316 | 96,77% |
| | 4 | 317 | 97,23% |
| | 5 | 317 | 97,23% |
| | 6 | 317 | 97,08% |
| | 7 | 315 | 96,46% |
| | 8 | 318 | 97,23% |
| | 9 | 318 | 97,54% |
| | 10 | 316 | 97,08% |

**TABLE 3.7 –** Tableau récapitulatif des résultats pour les instances satisfiables

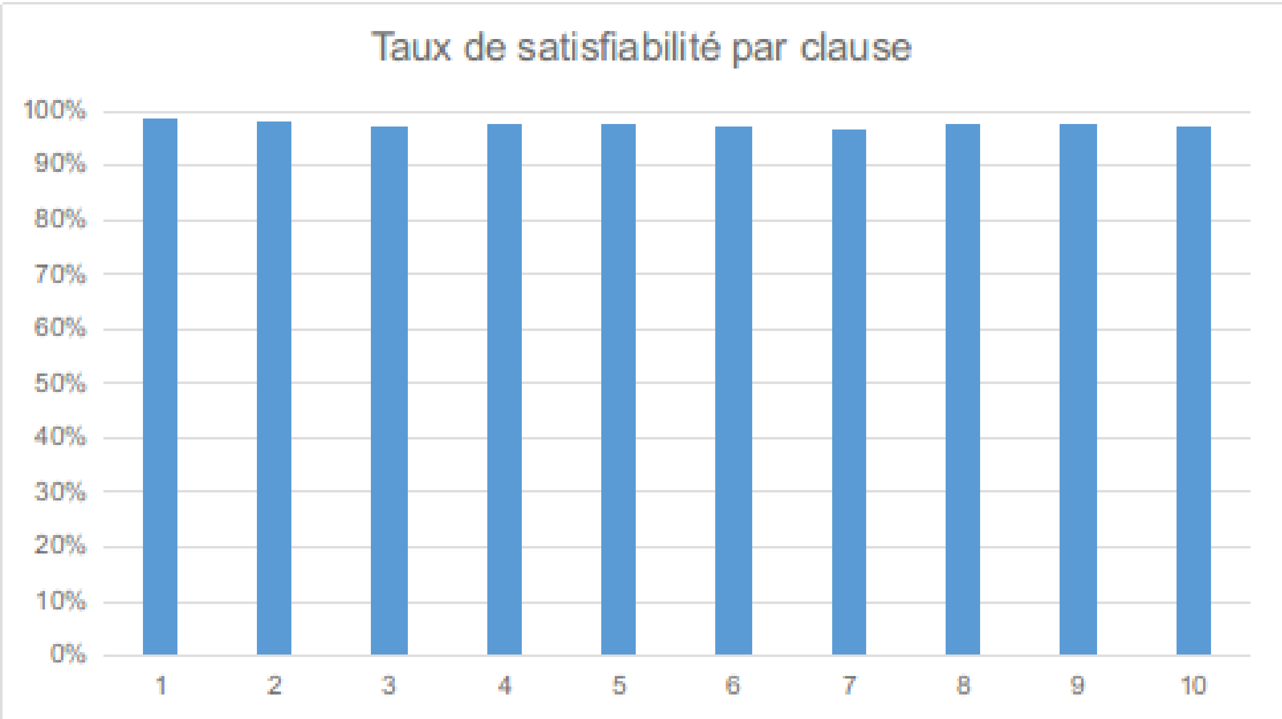



**FIGURE 3.13 –** Illustration des données de la table 3.7

**Pour les instances contradictoires ( non sastisfiables ) :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UUF75-325 | 1 | 316 | 96,31% |
| | 2 | 315 | 96,77% |
| | 3 | 316 | 96,77% |
| | 4 | 313 | 96,31% |
| | 5 | 315 | 96,77% |
| | 6 | 311 | 95,08% |
| | 7 | 313 | 95,85% |
| | 8 | 314 | 96,00% |
| | 9 | 320 | 98,00% |
| | 10 | 310 | 94,92% |

**TABLE 3.8 –** Tableau récapitulatif des résultats pour les instances non-satisfiables

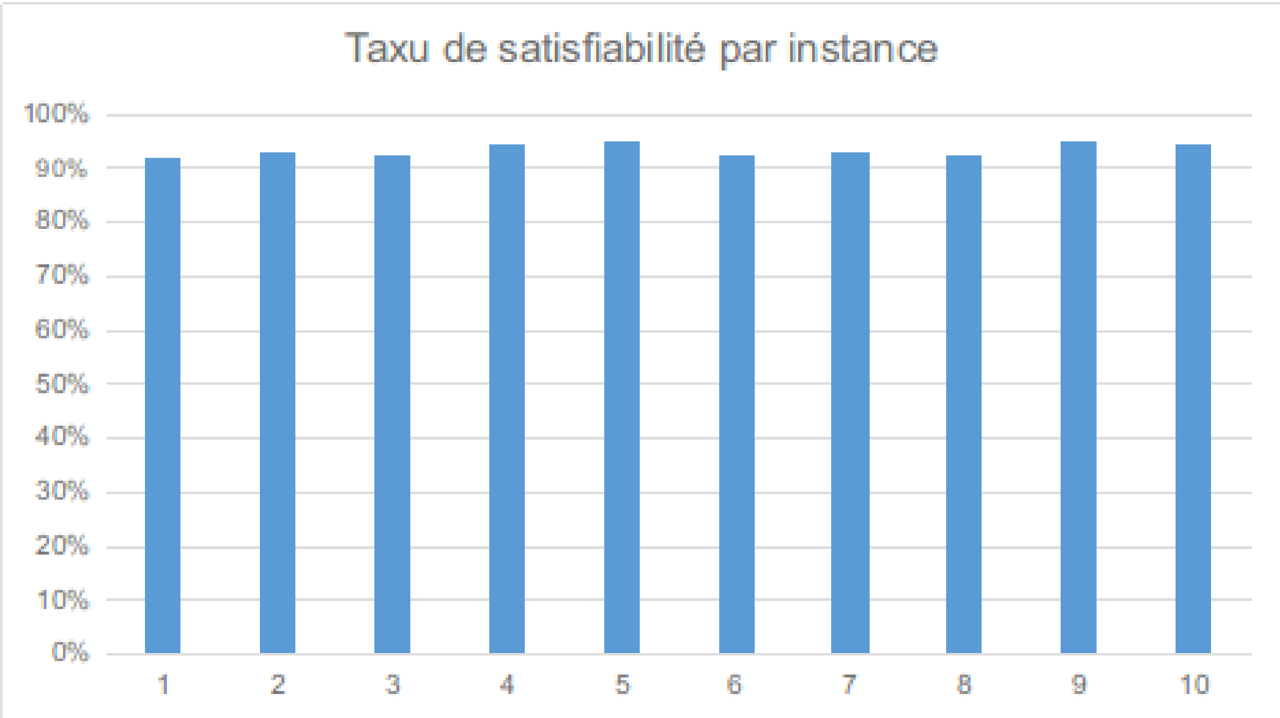



**FIGURE 3.14 –** Illustration des données de la table 3.8

### 3.4.5 Algorithme A*

: Les résultats sont présentés d'abord sous forme de tables puis illustrés dans des histogrammes :

**Pour les instances satisfiables :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UF75-325 | 1 | 318 | 96,58% |
| | 2 | 318 | 97,26% |
| | 3 | 316 | 96,25% |
| | 4 | 316 | 96,31% |
| | 5 | 320 | 97,42% |
| | 6 | 320 | 97,20% |
| | 7 | 318 | 96,80% |
| | 8 | 319 | 96,83% |
| | 9 | 319 | 97,29% |
| | 10 | 319 | 97,54% |

**TABLE 3.9 –** Tableau récapitulatif des résultats pour les instances satisfiables

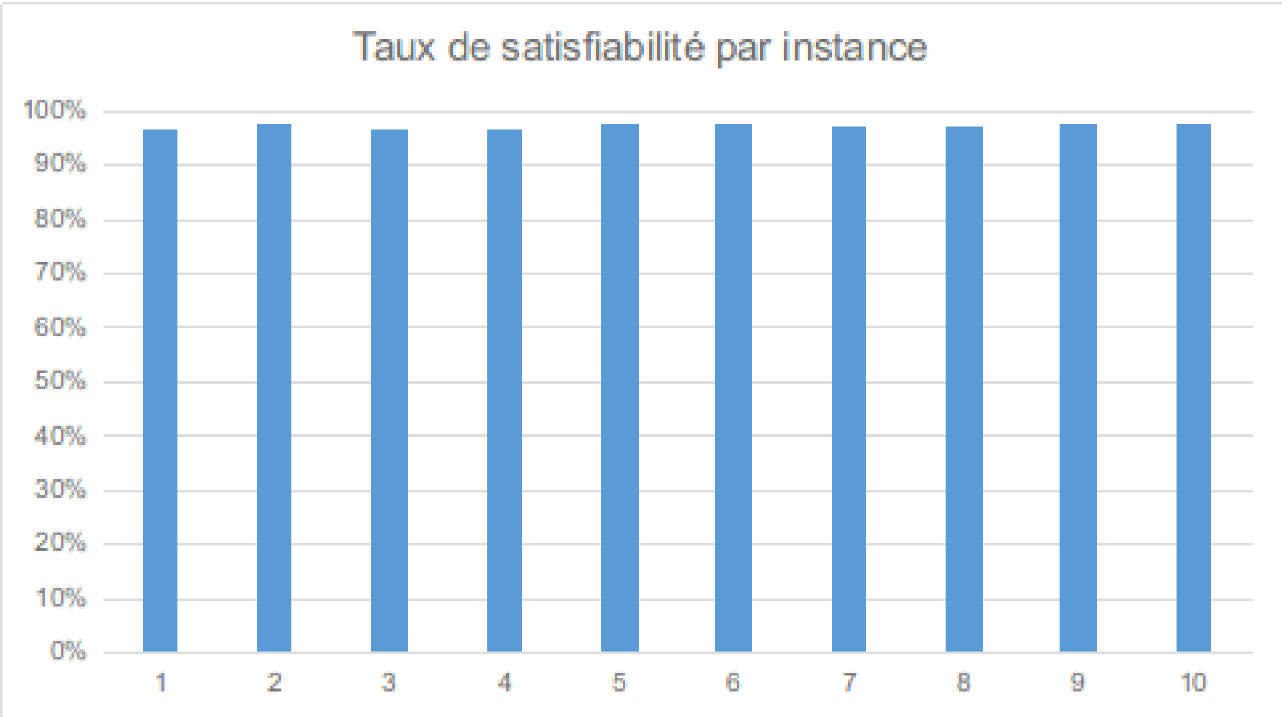



**FIGURE 3.15 –** Illustration des données de la table 3.9

**Pour les instances contradictoires ( non sastisfiables ) :**

| Fichiers test | Instance | Maximum clauses | Taux moyen de satisfiabilité |
|---|---|---|---|
| UUF75-325 | 1 | 316 | 94,65% |
| | 2 | 317 | 95,31% |
| | 3 | 315 | 94,33% |
| | 4 | 315 | 94,38% |
| | 5 | 320 | 95,47% |
| | 6 | 320 | 95,26% |
| | 7 | 317 | 94,86% |
| | 8 | 319 | 94,89% |
| | 9 | 318 | 95,34% |
| | 10 | 318 | 95,59% |

**TABLE 3.10 –** Tableau récapitulatif des résultats pour les instances non-satisfiables

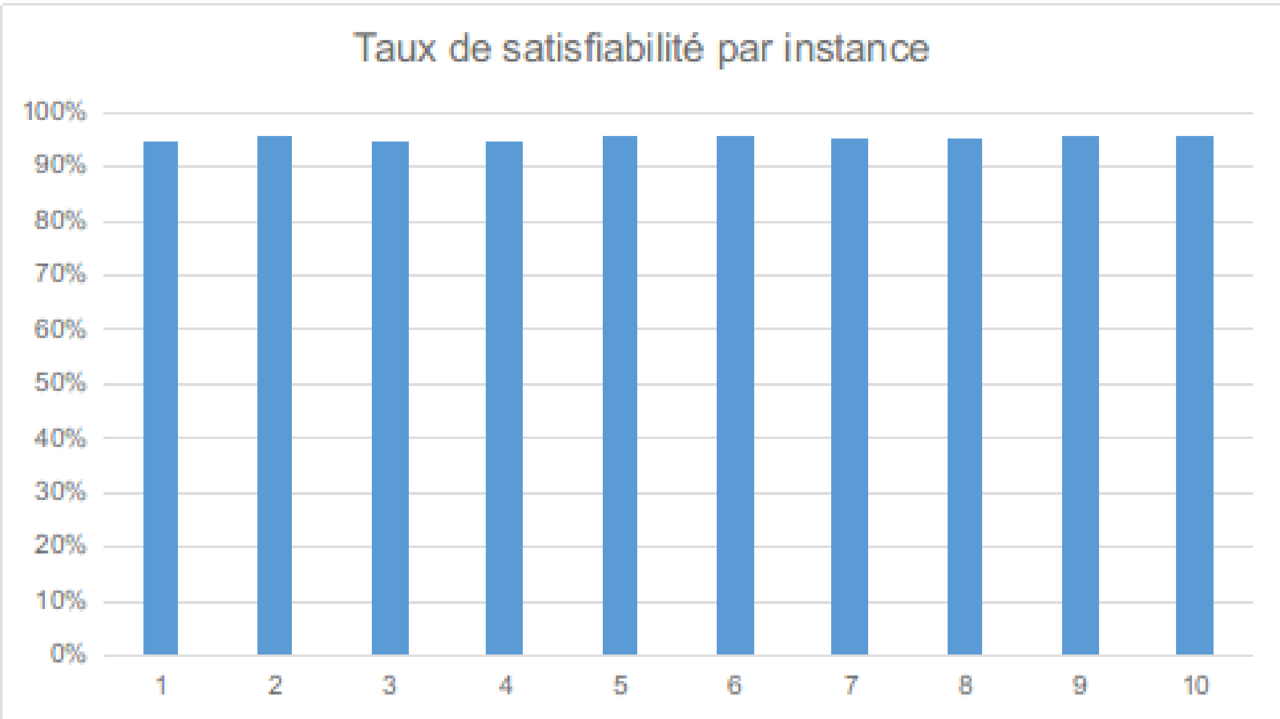



**FIGURE 3.16 –** Illustration des données de la table 3.10

## 3.5 Statistiques

Étant donné le très grand nombre de données et de résultats obtenus, nous avons décidé de récapitulé ces dérniers dans un tableau statistiques, puis dans un graphique de type **Boites-à-moustaches**

| UF75-325 | | | | | |
|---|---|---|---|---|---|
| Mesure | BFS | DFS | Coût Uniforme | Recherche Gloutonne | A* |
| Nombre moyen de clauses satisfaites | 139,4 | 303,1 | 303,0 | 314,0 | 316,1 |
| Taux Moyen de satisfiablié | 42,9021% | 93,2677% | 93,2154% | 96,6000% | 97,2615% |

**TABLE 3.11 –** Tableau de mesures statistiques pour les instances satifsiables

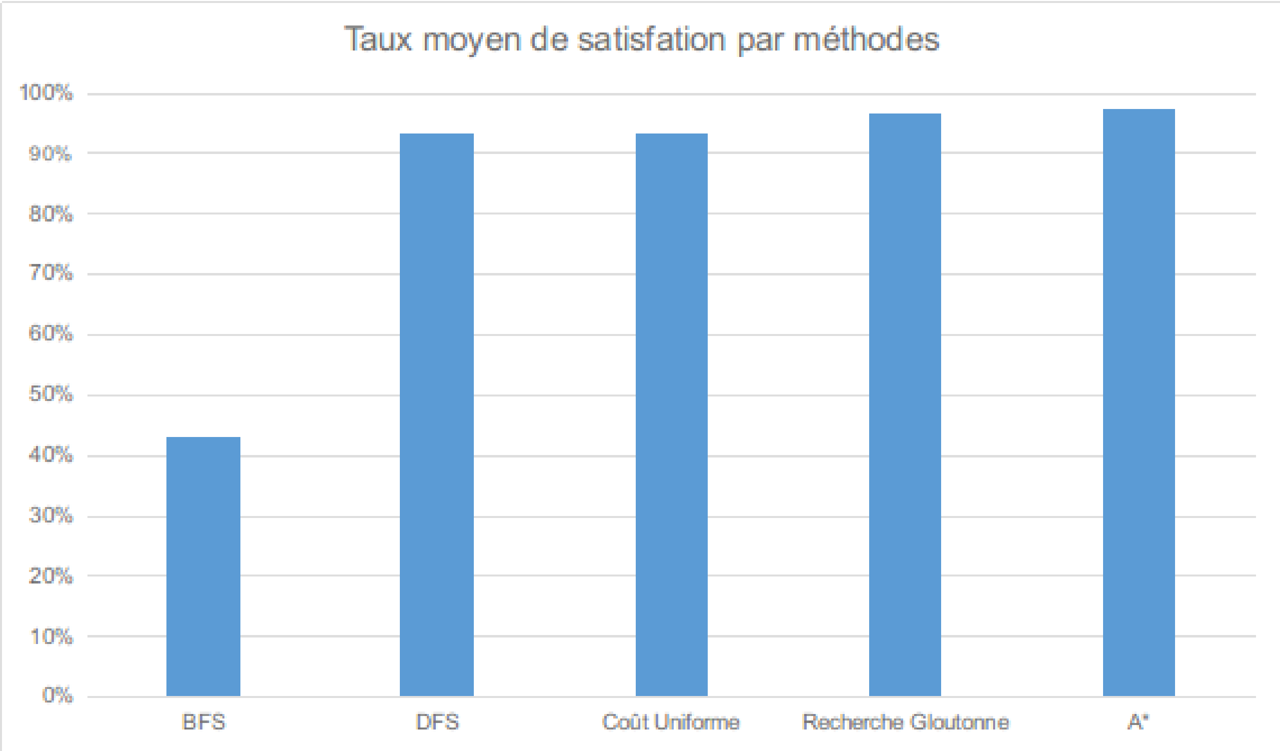



**FIGURE 3.17 –** Illustration des données de la table 3.11

| UUF75-325 | | | | | |
|---|---|---|---|---|---|
| Mesure | BFS | DFS | Coût uniforme | Recherche gloutonne | A* |
| Nombre moyen de clauses satisfaites | 136,0 | 304,3 | 301,9 | 312,9 | 315,1 |
| Taux Moyen de satisfiablié | 41,8523% | 93,6246% | 92,8769% | 96,2769% | 96,9477% |

**TABLE 3.12 –** Tableau de mesures statistiques pour les instances non-satifsiables

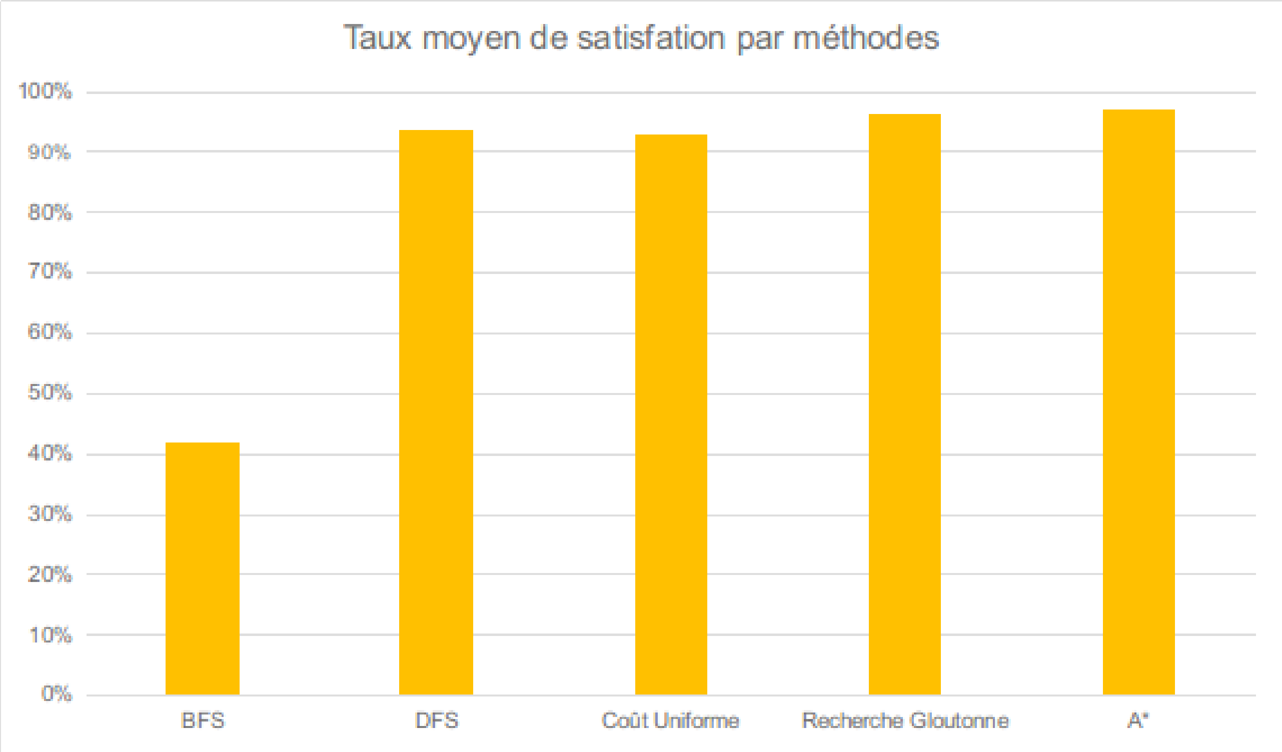



**FIGURE 3.18 –** Illustration des données de la table 3.12

### Améliorations avec BitSet

Nous avons tenté de comparé les résultats expérimentaux en essayant différentes structures de données pour l'évaluation d'une solution et la gestion de la liste open, le tableau suivant ( voir table 3.5 ) démontre que la structure du BitSet proposée évalue 15 à 190 fois ( selon la gestion de open ) plus de clauses en une seconde que la structure de matrice, combiner cette représentation avec une gestion en tas de open, nous a fait gagné un temps assez important lors de l'évaluation et le réarrangement de open.

| gestion de open \ évaluation par | Matrice | Bitset |
|---|---|---|
| liste triée | 205124 éval/s | 11952330 éval/s |
| tas | 237532 éval/s | 37252319 éval/s |
| FIFO | 238403 éval/s | 3149722 éval/s |
| LIFO | 213397 éval/s | 40879427 éval/s |

**TABLE 3.13 –** Nombre d'évaluations par seconde

## 3.6 Comparaison entres les cinq méthodes

Pour conclure ce chapitre, nous allons maintenant comparer les différentes méthodes selon la rapidité d'exécution, l'espace mémoire utilisé et le taux de satisfiabilité enregistré.

Nous avons remarqué à travers les nombreux tests que les deux catégories de stratégies de recherche avaient des forces et des lacunes, pour citer des exemples, la recherche par profondeur d'abord et en largeur d'abord de part sa simplicité, sont de bonnes stratégies de recherche, mais dont les limites sont vites atteintes, la première est certe peu gourmande en espace mémoire, mais ne trouve pas la solution en un temps assez rapide, la deuxième quant à elle nous garantie (si le coût pour passer d'un noeud à un autre est le même quelques soient les noeuds choisis) de trouver la solution avec le plus petit nombre de litéraux possible, mais en contre partie consomme énormément de mémoire, ce qui peut conduire à un débordement de la mémoire très rapidement.

Pour ce qu'il en est de l'algorithme de recherche par coût uniforme, il se voit être un compromis entre l'algorithme DFS[2] (voir 1.2.3) et l'algorithme BFS([3] (voir 1.2.3, il assure de trouver la solution avec un potentiel débordement de mémoire, mais peut aussi prendre un temps exponentiel pour trouver la solution (1.2.3), les expérimentations réalisés en sont la preuve.

Quand on bascule vers la deuxième catégorie, on se rend vite compte que l'ajout d'une heuristique peut réduire le temps de recherche d'une façon significative, ce que fait l'algorithme DFS en 10-15 mins peut être fait en quelques secondes avec l'algorithme de recherches gloutonne ou bien A*, cependant le gain en rapidité ne masque pas le fait que l'espace mémoire reste aussi soumis à un débordement ( moins fréquemment mais ça reste un risque potentiel), de plus la difficulté de trouver de bonnes heuristiques ( admissibles par exemple ) demeure un challenge du point de vue théorique et pratique, à noté aussi que très souvent, l'algorithme A* se limite à une recherche dans un maximum local, ce qui peut ralentir le processus de recherche de solutions optimales.

Une remarque à faire concernant l'ensemble des méthodes utilisées est que les résultats, malgré le fait que le choix des noeuds soit aléatoire, ne diffèrent pas d'une exécution à une autre sur une même instance ( pour A* par exemple on est dans les 96%-97% de taux de falsifiabilité sur les benchmarks fournis) , cela est dû principalement au fait que les fréquences d'apparitions des littéraux soient très proches les unes des autres, ainsi choisir un littéral (ou sa négation) plutôt qu'un autre n'influe pas vraiment sur le résultat final.

2. Depth first search
3. Breadth first search

# Conclusion

En conclusion de ce travail, nous pouvons dire malgré la simplicité apparente d'un problème, il est très souvent impossible de le résoudre à l'aide de méthodes dites **classiques**, il est vrai qu'un taux de réussite de 97% par exemple peut paraître suffisait, on ne doit pas oublié que ce taux évolue selon la taille du problème, en effet sur les instances de tailles moyenne vue dans cette partie du tp, il aurait été préférable de trouver des méthodes qui avoisinent les 99% de taux de réussite, mais il est évident que ces méthodes représentent les limites des méthodes classiques, c'est ainsi de façon naturelle et sensée, que nous allons passé des méthodes heuristiques aux méta-heuristiques, une évolution nécessaire pour ne serait ce qu'approximer de façon plausibles et suffisante la solution optimale cachée derrière cet océan de solutions.

# Deuxième partie

# Approche par espace des solutions BSO

# Chapitre 4

# Introduction :

## 4.1 Problématique :

**Limites des méthodes de recherche classiques** Bien que les méthodes présentées précédemment (voir I) présentent des résultats assez satisfaisants, ils ne permettent pas de résoudre le problème en un temps assez petit, cela nous a conduit à explorer une toute autre famille d'algorithmes appelés les méta-heuristiques.

## 4.2 Définitions

### 4.2.1 Espace des solutions

L'espace des solutions $S$ d'un problème donné est l'ensemble de toutes les solutions possibles pour ce dernier (qu'elles soient positives ou négatives), une solution étant le résultat produit par les méthodes vues dans la partie I

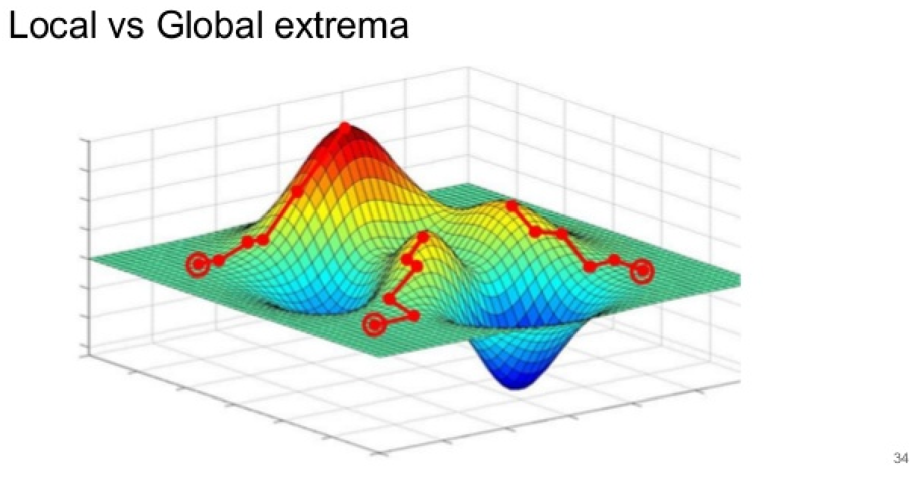



**FIGURE 4.1 –** Exemple d'un espace de solution multidimensionnel

### 4.2.2 Méta-heuristique

Les méta-heuristiques sont des algorithmes d'optimisation de solutions visant à résoudre des problèmes hautement combinatoires et dont la complexité ne permet pas leur résolution en un temps raisonnable par des méthodes dite ***classiques***, elles ont la particularité de faire une recherche dans l'espace des solutions $S$(4.2.1), contrairement aux méthodes de I qui construisent les solutions pas à pas en explorant l'espace des états d'une solution en particulier

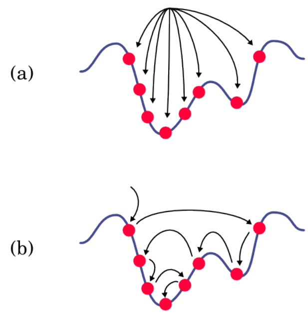



**FIGURE 4.2 –** Exemples de recherche ans l'espace des solutions selon une fonction d'évaluation

### 4.2.3 Intelligence en essaim (Swarm intelligence)

L'intelligence en essaim consiste à étudier et à construire des sociétés d'individus artificiels simples (généralement appelés Agents [1]) qui sont capables collectivement de fournir une réponse complexe et parviennent a travers des interactions(aussi appelés **synergie**) simples à prendre des décisions intelligentes.

### 4.2.4 Bee swarm optimization (BSO)

C'est un algorithme de recherche de solutions à base de populations d'agents artificielles qui imitent le comportement des abeilles dans leur façon de rechercher la nourriture d'une façon organisé et optimale, la nourriture étant l'analogie d'une solution dans le domaine de l'intelligence artificielle(Résolution de problèmes).[4]

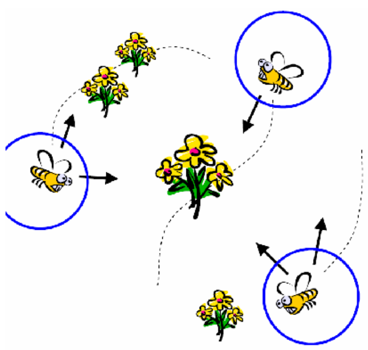


**FIGURE 4.3 –** Abeilles communiquant pour la recherche de nourriture

1. une entité autonome capable de percevoir son environnement grâce à des capteurs et aussi d'agir sur celui-ci via des effecteurs afin de réaliser des buts

# Chapitre 5

# Implémentation de l'algorithme BSO pour le problème SAT

## 5.1 Structures de données :

Comme vu dans l'approche par espace des états la représentation du problème et les structures de données ont un impact considérable sur les performances de l'implémentation d'un algorithme.
Dans cette partie nous allons voir les structures de données adéquates à notre implémentation de BSO.

### 5.1.1 Représentation d'instance et de solutions SAT :

Nous allons utilisé la représentation par Bitset vu précédemment dans laquelle on représente une instance SAT en gardant pour chaque littéral les clauses qu'il satisfait dans un Bitset, et une solution SAT par un Bitset de taille égale au nombre de variables de l'instance SAT et pour chaque variable on lui associe un bit qui est met à 1 si la variable est vrai, à 0 sinon. Pour résumé tout cela, soit l'instance SAT suivante :

$$\begin{aligned} x_1 \vee \neg x_2 \vee x_4 \\ \neg x_2 \vee x_3 \vee x_4 \\ \neg x_1 \vee x_2 \vee \neg x_3 \end{aligned}$$

Et la solution suivante :

$$x_1 \leftarrow true,\ x_2 \leftarrow false,\ x_3 \leftarrow true,\ x_4 \leftarrow false$$

La représentation :

| $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 |

Solution

| | | | |
|---|---|---|---|
| $x_1$ | 1 | 0 | 0 |
| $x_2$ | 0 | 0 | 1 |
| $x_3$ | 0 | 1 | 0 |
| $x_4$ | 1 | 1 | 0 |

| | | | |
|---|---|---|---|
| $\neg x_1$ | 0 | 0 | 1 |
| $\neg x_2$ | 1 | 1 | 0 |
| $\neg x_3$ | 0 | 0 | 1 |
| $\neg x_4$ | 0 | 0 | 0 |

Instance

On peut calculer les clauses satisfaites par la solution en utilisant le or logique entre les Bitset de ses littéraux :

| | | | |
|---|---|---|---|
| $x_1$ | 1 | 0 | 0 |

OR

| | | | |
|---|---|---|---|
| $x_3$ | 0 | 1 | 0 |

OR

| | | | |
|---|---|---|---|
| $\neg x_2$ | 1 | 1 | 0 |

OR

| | | | |
|---|---|---|---|
| $\neg x_4$ | 0 | 0 | 0 |

$\downarrow$

| | | | |
|---|---|---|---|
| *Bitset* résultat | 1 | 1 | 0 |

### 5.1.2 La table Dance :

Comme la plupart des méta-heuristique, BSO travaille sur une solution qu'il essaye d'améliorer à chaque itération. Une table contenant les meilleures solutions, appelée Dance, est utilisée. Nous avons opté à organiser cette table sous forme de tas, ainsi à chaque itération la racine du tas est choisie pour le traitement, suite à cela, les meilleures solutions trouvées par les abeilles à la fin de l'itération sont insérées dans la table.

## 5.2 Conception et pseudo-code :

Nous présentons dans la suite les parties essentielles constituant la méthode BSO.

### 5.2.1 Algorithme de recherche :

Comme expliqué précédemment, à chaque itération on essaye d'améliorer une solution initiale. L'itération commence par générer des solution équidistante de la solution initial, et pour chaque solution générée on fait une recherche locale. Ensuite, chacune des solutions trouvées localement est insérées dans la table Dance cité précédemment. L'itération suivante fera le même traitement en commençant par la meilleure solution de Dance. Cela est répété jusqu'à ce qu'on arrive à la solution optimale ou à une condition d'arrêt, nombre maximum d'itération atteint par exemple.

**Algorithme 3 :** Algorithme de recherche BSO

```
Résultat : retourne la meilleure solution trouvée
1  sRef ←solution aléatoire;
2  meilleureSolution ← sRef;
3  tant que ¬fin() faire
4      ajouter(listeTabou,sRef);
5      abeilles ← determinerRégionDeRecherche(sRef);
6      pour chaque abeille ∈ abeilles faire
7          solutionLocale ← rechercheLocale(abeille);
8          ajouter(Dance,solutionLocale);
9      fin
10     sRef ←meilleureDeDance(Dance);
11     si sRef > meilleure alors
12         meilleureSolution ← sRef;
13     fin
14 fin
15 retourner meilleureSolution;
```

Nous allons à présent détailler les différentes lignes de cet algorithme :

1. Ligne 1 : Une solution aléatoire est générée.
2. Ligne 3 : la condition d'arrêt peut être : solution optimale trouvée, nombre maximale d'itération atteint, temps limite dépassé etc.
3. Ligne 4 : La solution sur laquelle on fait une itération est ajoutée dans une liste tabou pour éviter la stagnation dans un minimum local.
4. Ligne 5 : On détermine les régions de recherche, représentées par des abeilles, à partir de la solution initial en utilisant un paramètre de distance = 1/flip. Cette fonction génère flip+1 solutions équidistantes ce qui va permettre par la suite de faire des recherches dans plusieurs régions différentes et ainsi augmenter les chances d'arriver à une solution optimale.
5. Lignes 6-9 : Dans cette partie on boucle sur les abeilles en appliquant une recherche tabou sur chacune des régions. Les solutions résultats sont insérées dans la table Dance.
6. Ligne 10 : On sélectionne la meilleure solution de la table Dance. Si l'algorithme est dans un état de stagnation, c'est à dire la meilleure solution en terme de qualité ne s'améliore pas, on choisit la meilleure solution en terme de diversité.

### 5.2.2 Le paramétrage empirique :

Le pseudo-code si dessus utilise des paramètre tel que flip, nombre maximale d'itération globale/locale ainsi que des paramètres permettant de détecter l'état de stagnation.

Nous détaillons maintenant le rôle de chaque paramètre que nous montrerons ultérieurement comment ajuster la valeur expérimentalement.

**Flip :**

Ce paramètre permet à la fois de spécifier le nombre d'abeilles ainsi que la distance entre les régions de recherche de ses abeilles.

Flip itérations sont exécutée pour créer flip nouvelles solutions à partir de la solution initiale. Chaque itération i commence par inverser le ième bit de la solution initiale ensuite tous les bits d'indice i + n*flip < nombre de variables. Ainsi on obtient flip solution chacune a une distance de hamming de 1/flip de toutes les autres.
Exemple pour flip = 3.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |

Première itération :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |

Deuxième itération :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

Troisième itération :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Nombre maximale d'itérations globales :**

C'est le nombre d'itérations de la boucle de recherche de BSO. Plus ce nombre est grand plus le temps d'exécution est important, et la probabilité d'améliorer la meilleure solution augmente. Nous devant donc trouver un compromis entre le temps d'exécution et la qualité de la solution en ajustant ce nombre.

**Nombre maximale d'itérations locale :**

C'est le nombre d'itération de la recherche tabou. Il représente à quel point on recherche localement dans une des régions générées précédemment. La recherche tabou améliore rapidement une solution, mais elle est largement affecté par la solution de départ, c'est à dire si on commence à partir d'une solution lointaine du but on risque de stagner pendant longtemps vu qu'on recherche toujours dans le voisinage, d'où une première intuition serait de garder ce nombre relativement petit par rapport au nombre d'itération globale.

**Paramètre de stagnation :**

Pour éviter de stagner pendant longtemps, si les solutions de Dance ne s'améliore pas durant un certain nombre d'itérations, on choisit la solution la plus distante du reste des solutions, et ainsi permettre à BSO d'explorer de nouvelles solutions. Ce nombre d'itérations limite est lui aussi un paramètre empirique que nous utiliserons dans cette implémentation de BSO.

### 5.2.3 Le paramétrage dynamique :

Une autre solution serait de régler les paramètres dynamiquement pendant l'exécution. Nous nous sommes basés sur l'algorithme du recuit simulé [3] pour varier les valeurs des paramètres de BSO.

Le principe est simple, On commence BSO avec une distance entre les solutions relativement grande, la distance ensuite diminue plus le nombre d'itération augmente. Cela permet de remplir initialement la table Dance avec des solutions lointaines les unes des autres, ensuite avec le passage du temps la recherche se fait de plus en plus au voisinage des meilleures solutions entre elles. Si l'algorithme arrive à un état de stagnation, la distance entre les solutions est réinitialiser à une grande distance pour permettre à l'algorithme d'explorer d'autres régions de solutions.

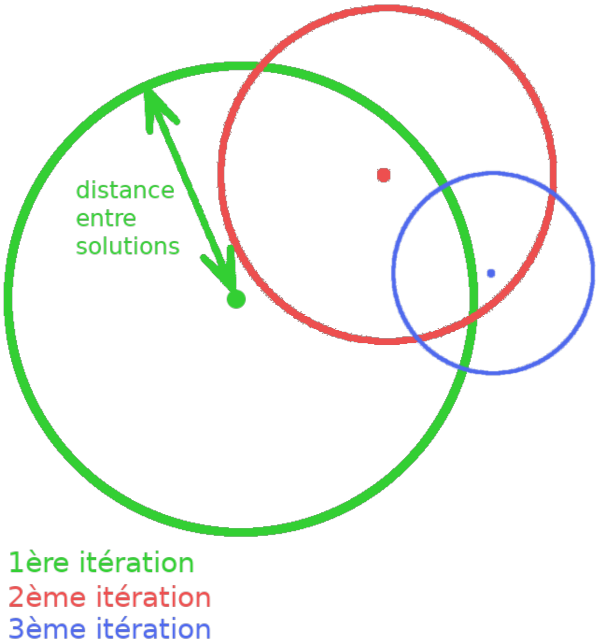



**FIGURE 5.1 –** Illustration des régions de recherche des différentes itérations

#### Nombre maximale d'itérations locale :

Dans cette implémentation le nombre d'itération locale lui aussi varie en fonction de la distance. L'intuition était de choisir un nombre égale à la distance entre les solutions afin de mieux couvrir l'espace entre les elles tout en restant optimale par rapport au temps d'exécution. l'idée derrière c'est de donner la possibilité à la recherche locale d'arriver à toutes les solutions entre la solution sur laquelle on applique la recherche locale, et celle à partir de la quelle on a commencer l'itération de BSO.
On peut illustrer un nombre d'itération locale égale à la distance comme suit :

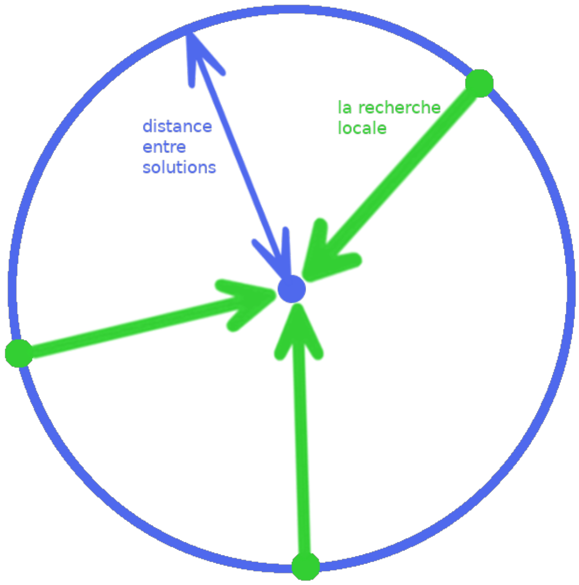



**FIGURE 5.2 –** Illustration de BSO avec un nombre de recherche locale égale à la distance entre les solutions

Par contre dans le cas où le nombre d'itération locale est inférieur à la distance on ne pourra jamais arriver à la solution initiale puisque la recherche locale change un bit chaque itération, et pour arriver à la solution initiale on doit changer un nombre de bits égale à la distance entre les deux solutions.

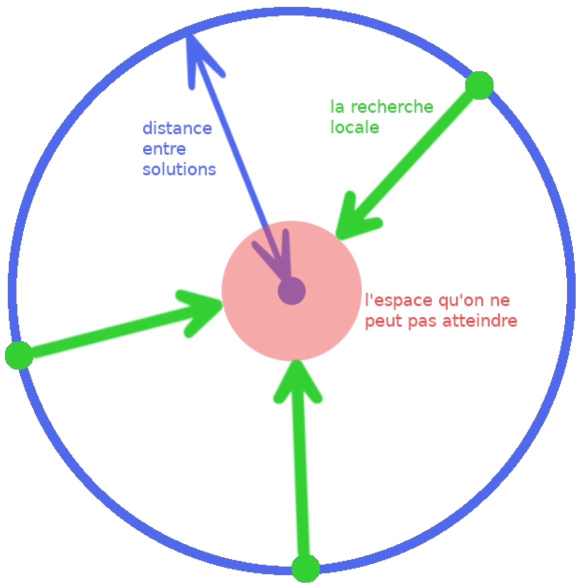



**FIGURE 5.3 –** Illustration de BSO avec un nombre de recherche locale inférieur à la distance entre les solutions

Dans la suite de ce rapport nous allons comparer les deux implémentations entre elles expérimentalement ainsi qu'avec les solutions heuristique vu dans le premier chapitre.

# Chapitre 6

# Expérimentations

## 6.1 Données

Les données de tests sont les même vues dans I3.1 aux quelles nous avons rajouté des instances avec plus de variables et plus de clauses pour tester les limites des améliorations apportées par BSO.

## 6.2 Résultats

### 6.2.1 BSO avec paramétrage statique

Nous avons testé notre solveur sur deux types d'instances, **(u)uf75-325** et **(u)uf100-430** et **jnh**(ici), avec un total de dix instances par paramètres et une limite de dix(ou cinq) essaies par instance, les tableaux et figures suivantes montres quels jeux de paramètre ont abouti aux meilleurs résultats durant les test.

| MaxIterations | flip | Nb. d'abeilles | maxChances | Recherches locales | Clauses satis. en moyennes | Taux de satisf. moyen | Temps moyen (s) |
|---|---|---|---|---|---|---|---|
| 1000 | 5 | 30 | 7 | 15 | 324.72 | 0.9991384615 | 2.65978 |
| 1000 | 5 | 30 | 9 | 20 | 324.72 | 0.9991384615 | 3.77651 |
| 1000 | 5 | 30 | 9 | 25 | 324.72 | 0.9991384615 | 4.405 |
| 1000 | 5 | 30 | 11 | 20 | 324.7 | 0.9990769231 | 3.58024 |
| 1000 | 5 | 30 | 11 | 30 | 324.7 | 0.9990769231 | 5.73142 |
| 1000 | 5 | 30 | 7 | 30 | 324.69 | 0.9990461538 | 5.29142 |

**TABLE 6.1 –** Meilleures combinaisons des paramètres empiriques pour les instances uf75-325

**Importance de chaque paramètre** Pour mieux se rendre de compte de l'impacte de chaque paramètre, nous avons décidé de fixer chaque valeur d'un paramètre et de faire varier les valeurs des autres paramètres restants et d'en tirer la moyenne, les tableaux suivants résument le travail réalisé :

| MaxItter | Moyenne clauses saisf. |
|---|---|
| 400 | 324.4436111111 |
| 700 | 324.4419444444 |
| 1000 | 324.4641666667 |

TABLE 6.2 – Impact du paramètres **MaxItteraitions**

| flip | Moyenne clauses saisf. |
|---|---|
| 5 | 324.4247222222 |
| 7 | 324.4419444444 |
| 9 | 324.4641666667 |

TABLE 6.3 – Impact du paramètres **Flip**

| nbr abeilles | Moyenne clauses saisf. |
|---|---|
| 7 | 324.4641666667 |
| 9 | 324.4419444444 |
| 11 | 324.4247222222 |

TABLE 6.4 – Impact du paramètres **Nombre d'abeilles**

| recherches locales | Moyenne clauses saisf. |
|---|---|
| 15 | 324.4251851852 |
| 20 | 324.4248148148 |
| 25 | 324.4644444444 |
| 30 | 324.46 |

TABLE 6.5 – Impact du paramètres **Nombre de recherches locales**

### 6.2.2 BSO avec paramétrage dynamique

#### Traitement non-parallèle

Après avoir améliorer le choix des paramètres, en le rendant dynamique et dépendant de l'exécution courante, nous avons obtenus de bien meilleures résultats, non seulement sur les même instances mais aussi sur d'autres de nature beaucoup plus complexe, les conditions sont toujours les mêmes voir 9.3.1

| Instance | clauses sais. | Taux | Temps(s) |
|---|---|---|---|
| uf75-01 | 325 | 1 | 1.5592 |
| uf75-02 | 325 | 1 | 2.0478 |
| uf75-03 | 325 | 1 | 8.4525 |
| uf75-04 | 324.9 | 0.9996923077 | 11.878 |
| uf75-05 | 325 | 1 | 7.8659 |
| uf75-06 | 325 | 1 | 6.1555 |
| uf75-07 | 325 | 1 | 6.3884 |
| uf75-08 | 325 | 1 | 6.0486 |
| uf75-09 | 325 | 1 | 2.4297 |
| uf75-010 | 325 | 1 | 1.7065 |
| Moyenne | 324.99 | 0.9999692308 | 5.45321 |

**TABLE 6.6 –** Résume pour les instances **uf75-325**

| instance | clauses satis. | Taux | Temps(s) |
|---|---|---|---|
| uf100-01 | 430 | 1 | 25.8205 |
| uf100-02 | 429.6 | 0.9990697674 | 49.4153 |
| uf100-03 | 430 | 1 | 19.8098 |
| uf100-04 | 430 | 1 | 11.4756 |
| uf100-05 | 429.9 | 0.9997674419 | 36.687 |
| uf100-06 | 430 | 1 | 2.0742 |
| uf100-07 | 430 | 1 | 9.2083 |
| uf100-08 | 429.4 | 0.9986046512 | 56.0873 |
| uf100-09 | 430 | 1 | 14.9025 |
| uf100-010 | 430 | 1 | 18.9783 |
| Moyenne | 429.89 | 0.999744186 | 24.44588 |

**TABLE 6.7 –** Résume pour les instances **uf100-430**

| instance | clauses satis. | Taux | Temps(s) |
|---|---|---|---|
| jnh201 | 800 | 1 | 11.704 |
| jnh202 | 798.8 | 0.9985 | 82.441 |
| jnh203 | 798.8 | 0.9985 | 81.8918 |
| jnh204 | 800 | 1 | 40.7106 |
| jnh205 | 800 | 1 | 34.9264 |
| Moyenne | 799.52 | 0.9994 | 50.33476 |

**TABLE 6.8 –** Résume pour les instances **jnh200-800**

Pour illustrer l'importance de cette modification, la figure suivante montre le taux de satisfiabilité selon la taille du problème(nombre de variables/de clauses) :

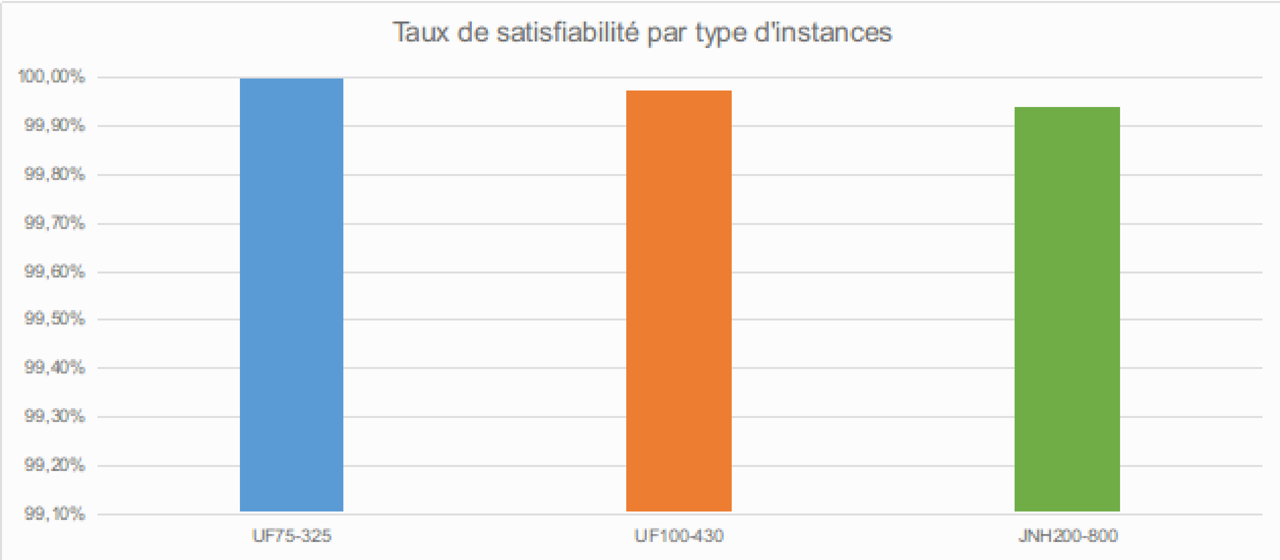



**FIGURE 6.1 –** Taux de satisfiabilité selon la nature de l'instance

### Traitement parallèle

Nous avons décidé de tenter une approche parallèle à la résolution du problème, en considérant chaque abeille comme un thread pouvant réaliser son travail indépendamment des autres abeilles, les résultats en terme de taux de satisfiabilité n'ont pas beaucoup changé, mais une nette amélioration du temps d'exécution peut être soulignée, le graphe et le tableau suivants comparent les deux technique(parallèle vs non-parallèle) en terme de ce dernier :

| UF75-325 | UF75-325 PAR | UF100-430 | UF100-430 PAR | JNH200-800 | JNH200-800 PAR |
|---|---|---|---|---|---|
| 5,45321 | 2,27845 | 24,44588 | 13,01019 | 312,4600 | 50,3348 |

**TABLE 6.9 –** Temps moyen passer à l'évaluation (DBSO vs DBSO-Parallèle)

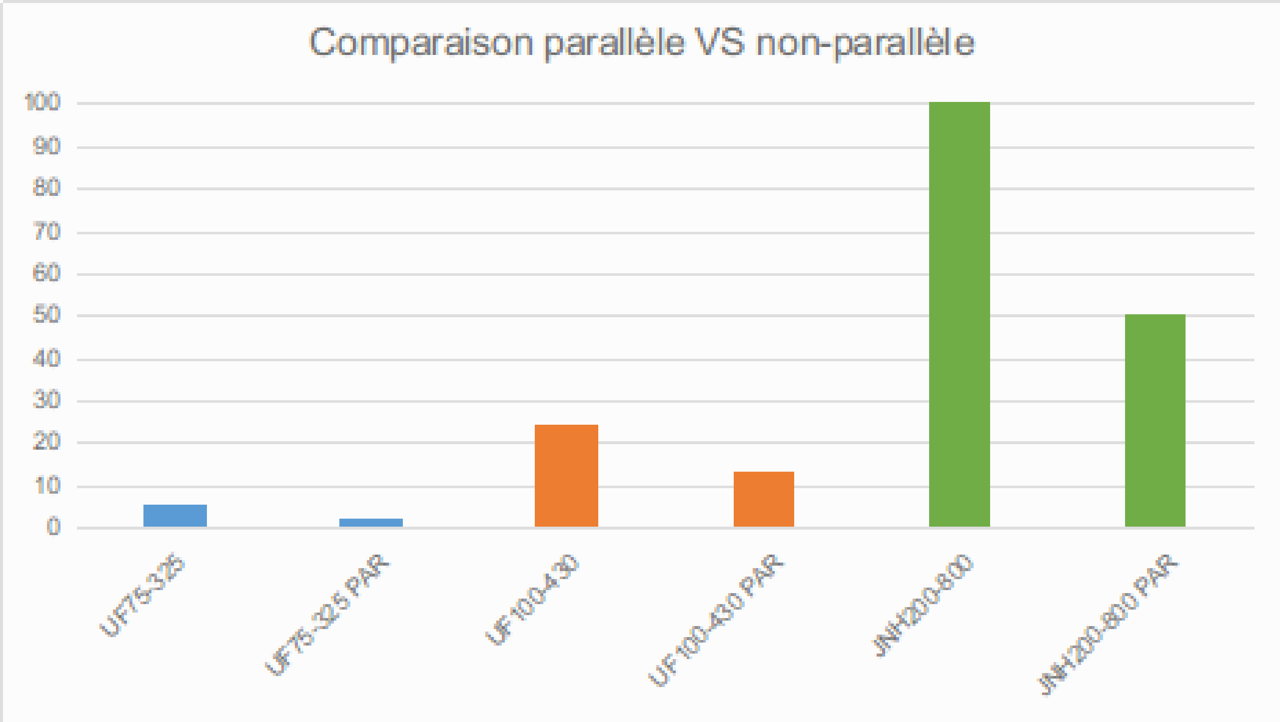



**FIGURE 6.2 –** Apport du parallélisme

## 6.3 Comparaison avec les méthodes de I

Il est évident que l'approche vu dans cette partie surpasse de loin celle vu en I, là oû la meilleure des méthodes constructives (A*) ne donnait pas d'assez bon résultats, l'algorithme BSO lui y est clairement supérieur, que ce soit en terme d'exactitude ou d'exploitation de ressources(i.e mémoire et temps d'exécution). N'est en moins, cette algorithme admet des limites, et sa nature stochastique n'en fait pas une méthode fiables a 100%( risque de stagnation prématurée, optimums locaux, réglage des paramètres couteux en temps .. ), cela nous a amené a explorer un autre algorithme de la même famille(méta-heuristiques) mais qui exploite l'approche de construction de solution en même temps, à savoir ACO (Ant Colony Optimization).

# Troisième partie

# Approche par espace des solutions ACO

# Chapitre 7

# Introduction :

## 7.1 Problématique :

**Limite de BSO** Bien l'algorithme BSO (voir **??**)présente des résultats très satisfaisants, il a cependant quelques points faibles qui sont liés à la recherche des bonnes valeurs des paramètres empiriques , l'ajustement dynamique a permit de palier a ce problème, mais il reste aussi l'aspect stochastique aléatoire très imprévisible des méta-heuristiques, c'est là qu'entre en scène une nouvelle familles de M.H [1] appelées ACO (**Ant Colony Optimization**) pour essayer de marier méthodes constructives et méthode évolutionnaires.

## 7.2 Définitions

### 7.2.1 Ant Colony Optimization(ACO)

Les algorithmes de colonies de fourmis sont des algorithmes inspirés du comportement des fourmis, et qui constituent une famille de méta-heuristiques d'optimisation principalement conçues pour des problèmes de **path-finding** [2]. Dans cette partie du projet nous allons nous intéresser à deux implémentation d'une M.H ACO, à savoir **AS**(Ant System) et **ACS**(Ant Colony System).

1. Méta-heuristique
2. Problème visant a trouver le chemin le plus court d'un point de départ A à un point d'arrivée B

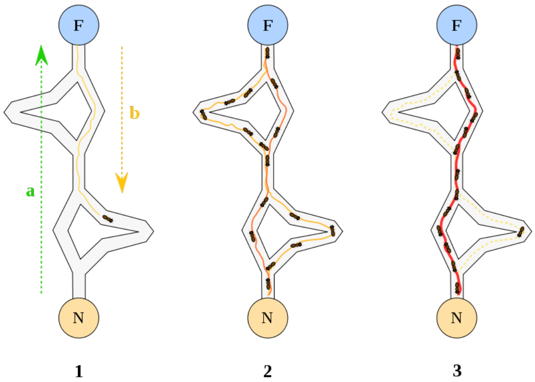



**FIGURE 7.1 –** Abeilles communiquant pour la recherche de nourriture

#### Ant System(AS)

Cette variante d'ACO fut l'une des première a être développée, elle se base sur une approche probabiliste du choix du chemin a parcourir par la fourmi, en effet une fourmi en temps normal réagit à un stimulus naturel qui est la **Phéromone**, elle suivra instinctivement la trace de phéromones la plus forte, et cela la plus part du temps trace précédente.

#### Ant Colony System(ACS)

Pensé comme une amélioration d'**AS**, ACS permet une modélisation plus fidèle à la vie réelle en introduisant les principes suivants :

- La mise a jour de la trace de phéromones enligne(effectuée par chaque fourmi lors de son passage sur un état) et hors-ligne(effectué à la fin de la construction de toutes les solutions des fourmi)
- La marche aléatoire(Random Walk[3]) qui servira de règle de transition pour aller d'un état à un autre.
- Exploitation, c'est le fait de suivre la trace de phéromones la plus forte( le stimulus le plus fort).
- Exploration, c'est le fait de prendre l'initiative d'explorer de nouveaux états sans pour autant tenir compte de la trace de phéromones la plus forte

### 7.2.2 Phéromones

Pour finir il nous faut introduire le concept de la phéromone. Dans la nature, la phéromone es une molécule chimique produite par un organisme, qui induit un comportement spécifique chez un autre membre de la même espèce, dans notre cadre de la recherche de solutions optimales pour une problème donnée, elle peut être perçu comme une valeur numérique attaché un état de la solution(cela reste une interprétation générale qui peut varier selon le problème).

3. Modèle mathématique d'un système possédant une évolution composée d'une succession de pas aléatoires, ou effectués « au hasard ».

# Chapitre 8

# Implémentation des algorithmes AS/ACS pour le problème SAT

## 8.1 Structures de données :

Vu les très bonnes performance réalisée par les structures de donnée vues dans I, nous avons logiquement opté pour les même représentations pour l'implémentation d'AS/ACS, à savoir :

### 8.1.1 Représentation d'instance et de solutions SAT :

$$x_1 \vee \neg x_2 \vee x_4$$
$$\neg x_2 \vee x_3 \vee x_4$$
$$\neg x_1 \vee x_2 \vee \neg x_3$$

Et la solution suivante :

$$x_1 \leftarrow true,\ x_2 \leftarrow false,\ x_3 \leftarrow true,\ x_4 \leftarrow false$$

La représentation :

| $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 |

Solution

| | | | |
|---|---|---|---|
| $x_1$ | 1 | 0 | 0 |
| $x_2$ | 0 | 0 | 1 |
| $x_3$ | 0 | 1 | 0 |
| $x_4$ | 1 | 1 | 0 |

| | | | |
|---|---|---|---|
| $\neg x_1$ | 0 | 0 | 1 |
| $\neg x_2$ | 1 | 1 | 0 |
| $\neg x_3$ | 0 | 0 | 1 |
| $\neg x_4$ | 0 | 0 | 0 |

Instance

On peut calculer les clauses satisfaites par la solution en utilisant le or logique entre les Bitset de ses littéraux :

| | | | |
|---|---|---|---|
| $x_1$ | 1 | 0 | 0 |

OR

| | | | |
|---|---|---|---|
| $x_3$ | 0 | 1 | 0 |

OR

| | | | |
|---|---|---|---|
| $\neg x_2$ | 1 | 1 | 0 |

OR

| | | | |
|---|---|---|---|
| $\neg x_4$ | 0 | 0 | 0 |

$\downarrow$

| | | | |
|---|---|---|---|
| *Bitset* résultat | 1 | 1 | 0 |

### 8.1.2 La table des Phéromones

La différence entre BSO et ACO réside dans le fait qu'une solution n'est pas obtenue selon le voisinage d'une autre solution, mais est construite pas à pas selon une règle de transition d'un état à un autre. Pour garder trace des phéromones déposées par les fourmis lors de la construction de leur solutions respectives, nous pouvons utilisée un tableau a deux dimensions(Matrice) dont les lignes sont les variables de la solution, et les colonnes les deux litéraux de la variable associée. le schéma suivant traduit cela :

| $i$ | $x_i$ | $\neg x_i$ |
|---|---|---|
| 1 | 0.1 | 0.1 |
| 2 | 0.1 | 0.1 |
| 3 | 0.1 | 0.1 |
| 4 | 0.1 | 0.1 |

**TABLE 8.1 –** Table des phéromones initiale

$\Downarrow$

| $i$ | $x_i$ | $\neg x_i$ |
|---|---|---|
| 1 | 0.1 | 0.1 |
| 2 | 0.1 | 0.1 |
| 3 | 0.1 | 0.1 |
| 4 | 0.325 | 0.102 |

**TABLE 8.2 –** Table des phéromones après le passage d'une fourmi

La figure illustre le processus de dépôt de phéromones, celui de l'évaporation sera vu plus tard car il dépend du choix de l'algorithme dérivant d'ACO(i.e soi AS ou bien ACS).

## 8.2 Conception et pseudo-code :

L'algorithme ACO a une structure de base très simple, elle se résume à l'algorithme suivant :

**Algorithme 4 :** Algorithme de recherche ACO

```
Résultat : retourne la meilleure solution trouvée
init(pheromons);
init(allParameters);
meilleureSolution ← randomSolution;
tant que ¬fin() faire
    pour chaque fourmi ∈ fourmis faire
        contruireSolution(fourmi);
        si isValide(fourmi.solution) alors
            retourner fourmi.solution;
        fin
    fin
    postConstructionActions(fourmis);
    // sera vu plus en détails dans AS/ACS
    ;
    si bestSolution(fourmis)>meilleureSolution alors
        meilleureSolution ← bestSolution(fourmis);
    fin
    miseAjourt(pheromons);
fin
retourner meilleureSolution;
```

Maintenant que nous avons vu le principale fonctionnement d'ACO, il est temps de spécifier le comportement d'AS et ACS :

### 8.2.1 AS(Ant System)

Il repose comme cité dans 7.2.1 sur une règle de transition probabiliste que l'on va détailler après avoir montré le pseudo code suivant :

**Algorithme 5 :** Algorithme de recherche AS

**Résultat :** retourne la meilleure solution trouvée

1 **init**($pheromons$);<br>
2 **init**($allParameters$) $meilleureSolution \leftarrow randomSolution$;<br>
3 **pour** $t = 1$ **à** *maxItter* **faire** **pour chaque** $fourmi \in fourmis$ **faire**<br>
4     **répéter**<br>
5         $P_{i,j} \leftarrow$ **calculerProba()**;<br>
6         $nextState \leftarrow$ **selectNextBy(**$P_{i,j}$**)**;<br>
7     **jusqu'à** $fourmil$ *construit sa solution*;<br>
8     $evaluation$ = **evaluer(**$fourmlle.solution$**)**;<br>
9     **si** $evaluation = bestScore$ **alors**<br>
10         **retourner** $fourmi.solution$;<br>
11     **fin**<br>
12     **onlinePheromonsUpdate(**$pheromons$**)**;<br>
13 **fin**<br>
14 **si** ***bestSolution***($fourmis$)>$meilleureSolution$ **alors**<br>
15     $meilleureSolution \leftarrow$ **bestSolution(**$fourmis$**)**;<br>
16 **fin**<br>
17 ;<br>
18 **retourner** $meilleureSolution$;

Nous allons maintenant détailler l'algorithme :

1. Ligne 1-2 : initialiser les paramètres empiriques et la table des phéromones a une valeur très petite arbitraire(0.1 par exemple).
2. Ligne 5 : calculer la probabilité du litéral $j$ de la variable $i$ selon la formule suivante :

$$P_{i,j} = \frac{[T_{i,j}]^{\alpha} * [\mu_{i,j}]^{\beta}}{\sum\limits_{lit \in Literals} [T_{i,lit}]^{\alpha} * [\mu_{i,lit}]^{\beta}} \tag{8.1}$$

   Autrement dit : le literal $x_i$ (réspec. $\neg x_i$) aura une probabilité $P_{x_i}$ (respec. $P_{\neg x_i}$ ) d'être choisi comme prochain état de la solution en cours de construction.
3. Ligne 6 : pour simuler un processus aléatoire, il suffit de tirer au hasard un nombre aléatoire $q$, si $q < P_{x_i}$ alors le prochain literal à être choisis sera $x_i$, sinon ce sera $\neg x_i$ ( on prend la densité de probabilités la plus proche du nombre aléatoire $q$ )
4. Ligne 12 : la mise a jour en-ligne de la table de phéromones se fait selon la formule suivante :

$$P_{i,j} = (1 - \rho)T_{i,j} + \rho \sum_{a \in Ants} \Delta_a T_{i,j} \tag{8.2}$$

où :

- $\rho$ est le taux d'évaporation de la phéromone
- $\Delta_a T_{i,j}$ est le taux de phéromones ajouté par la fourmi $a$ sur le litéral $j$ de la variable $x_i$ dans notre cas nous avons prit :

$$\Delta_a T_{i,j} = nbrClauseSatisfaites(x_{i,j})/nbrClauseTotal \tag{8.3}$$

le but étant de déposer un taux de phéromones plus élevé si le literal nous rapproche(en théorie) de la solution positive.

le processus de construction à un instant $t$ est illustré comme suit :

| |
|---|
| ⋮ |
| 1 |
| 0 |
| 0 |
| ? |

| | |
|---|---|
| ⋮ | ⋮ |
| 0.254 | 0.154 |
| 0.001 | 0.155 |
| 0.1 | 0.512 |
| 0.244 | 0.2 |

$\downarrow$ $\downarrow$

| |
|---|
| ⋮ |
| 1 |
| 0 |
| 0 |
| 1 |

| | |
|---|---|
| ⋮ | ⋮ |
| 0.254 | 0.154 |
| 0.001 | 0.155 |
| 0.1 | 0.512 |
| 0.244 + 0.2 | 0.2+0.05 |

Nous pouvons noter que ce comportement n'est pas toujours représentatif de la réalité, en effet selon cet algorithme la fourmi ne fait qu'explorer bêtement les traces de phéromones, il serait intéressant de modéliser le phénomènes d'exploitation de la plus forte trace de phéromone, cela a été introduit grâce à l'algorithme ACS.

### 8.2.2 ACS(Ant Colony System)

Comme cité dans 7.2.1, ACS est une amélioration de AS dans le sens ou une fourmi peut adopter une comportement aléatoire lors de l'application de la règle de transition, elle pourra soit explorer de nouvelles opportunités ou bien exploiter la meilleure trace à sa disposition, la fourmi n'étant pas doté d'une intelligence assez développé pour faire ce choix systématique, elle choisira selon l'instant donné totalement au hasard, bien sur ce choix est conditionné par le fait que la plus part du temps une fourmi choisira la trace de phéromones la plus forte instinctivement. ACS se démarque aussi par l'ajout d'une mise a jour hors-ligne de la table des phéromones lorsque toute les fourmis auront finit la construction de leurs solutions, le principe est que la fourmi ayant fournit la meilleure solution ajoute la quantité de phéromone proportionnelle à sa solution, en gardant aussi la mise à jour en-ligne(step by step ) des fourmis lors de la construction de leurs solutions respectives.

---

**Algorithme 6 :** Algorithme de recherche ACS

---

**Résultat :** retourne la meilleure solution trouvée<br>
1 **init**($pheromons$);<br>
2 **init**($allParameters$);<br>
3 $meilleureSolution \leftarrow randomSolution$;<br>
4 **pour** $t = 1$ **à** *maxItter* **faire** **pour chaque** $fourmi \in fourmis$ **faire**<br>
5     **répéter**<br>
6         $P_{i,j} \leftarrow$ **calculerProba()**;<br>
7         $nextState \leftarrow$ **selectNextBy(**$P_{i,j}$**)**;<br>
8     **jusqu'à** $fourmi$ *construit sa solution*;<br>
9     $evaluation$ = **evaluer(**$fourmlle.solution$**)**;<br>
10     **si** $evaluation = bestScore$ **alors**<br>
11         **retourner** $fourmi.solution$;<br>
12     **fin**<br>
13     **onlinePheromonsUpdate(**$pheromons$**)**;<br>
14 **fin**<br>
15 **si** ***bestSolution***($fourmis$)>$meilleureSolution$ **alors**<br>
16     $meilleureSolution \leftarrow$ **bestSolution(**$fourmis$**)**;<br>
17 **fin**<br>
18 **offLinePheromonsUpdate(**$pheromons, meilleurFourmille$**)**;<br>
19 ;<br>
20 **retourner** $meilleureSolution$;

---

1. Ligne 1-2 : initialiser les paramètres empiriques et la table des phéromones a une valeur très petite arbitraire(0.1 par exemple).
2. Ligne 5 : calculer la probabilité du litéral $j$ de la variable $i$ selon la formule suivante :

$$P_{i,j} = \frac{[T_{i,j}]^\alpha * [\mu_{i,j}]^\beta}{\sum\limits_{lit \in Literals} [T_{i,lit}]^\alpha * [\mu_{i,lit}]^\beta} \tag{8.4}$$

   Autrement dit : le literal $x_i$ (réspec. $\neg x_i$) aura une probabilité $P_{x_i}$ (respec. $P_{\neg x_i}$ ) d'être choisi comme prochain état de la solution en cours de construction.
3. Ligne 6 : pour simuler un processus de la marche aléatoire, on tire au hasard un nombre $q$, si $q < q_0$ alors on choisit le literal avec la combinaison **pheromone|heuristique**

maximale, sinon si $q < P_{x_i}$ alors le prochain literal à être choisis sera $x_i$, sinon ce sera $\neg x_i$ ( on prend la densité de probabilités la plus proche du nombre aléatoire $q$ ).
plus formellement on a :

si $q \leq q_0$

$$P_{i,j} = \begin{cases} 1 & \text{si } i = argmax\{[T_{i,j}]^\alpha * [\mu_{i,j}]^\beta\} \\ \\ 0 & \text{sinon} \end{cases}$$

si $q > q_0$

$$P_{i,j} = \frac{[T_{i,j}]^\alpha * [\mu_{i,j}]^\beta}{\sum\limits_{lit \in Literals} [T_{i,lit}]^\alpha * [\mu_{i,lit}]^\beta}$$

4. Ligne 12 : la mise a jour en-ligne de la table de phéromones se fait selon la formule suivante :

$$P_{i,j} = (1 - \rho)T_{i,j} + \rho\tau_0 \tag{8.5}$$

où : $\tau_0$ est le taux de phéromones initial

5. Ligne 17 : la mise à jour offline de la table des phéromones est réalisée de la manière suivante :

$$P_{i,j} = (1 - \rho)T_{i,j} + \rho\Delta T_{i,j}^{bestAnt}$$

- $\rho$ est le taux d'évaporation de la phéromone
- $\Delta_a T_{i,j}$ est le taux de phéromones ajouté par la fourmil $a$ sur le litéral $j$ de la variable $x_i$ dans notre cas nous avons prit :

$$\Delta Ti, j^{bestAnt} = nbrClauseSatisfaites(x_{i,j}^{bestAnt})/nbrClauseTotal \tag{8.6}$$

l'idée est que la fourmi avec le plus haut taux de réussite dépose plus de phéromones après la construction des solutions.

le schéma suivant aide à mieux comprendre :

| |
|---|
| ⋮ |
| 1 |
| 0 |
| 0 |
| ? |

| | |
|---|---|
| ⋮ | ⋮ |
| 0.254 | 0.154 |
| 0.001 | 0.155 |
| 0.1 | 0.512 |
| 0.244 | 0.2 |

↓ ↓

| |
|---|
| ⋮ |
| 1 |
| 0 |
| 0 |
| 1 |

| | |
|---|---|
| ⋮ | ⋮ |
| 0.254 | 0.154 |
| 0.001 | 0.155 |
| 0.1 | 0.512 |
| 0.244 + 0.1 | 0.2+0.1 |

↓

| | |
|---|---|
| ⋮ | ⋮ |
| 0.254+0.1 | 0.154+0.05 |
| 0.001+0.05 | 0.155+0.1 |
| 0.1+0.05 | 0.512+0.1 |
| 0.344+0.1 | 0.3+0.05 |

## 8.3 Actions post-construction(Deamons)

Pour améliorer encore plus les deux implémentation d'ACO, une séquence d'actions appelées **Deamons** peut être exécutée, dans notre cas nous avons choisis les deux traitements suivants :

- **exploreNeighbours** : elle consiste en une recherche locale d'une solution voisine par une fourmi quelconque, a travers une recherche "Tabou" classique eut coûteuse, afin d'éviter de tomber dans des optimums locaux, la recherche est freiné après un petit nombre d'itérations appelé **step**(dans les 15 20 itérations).

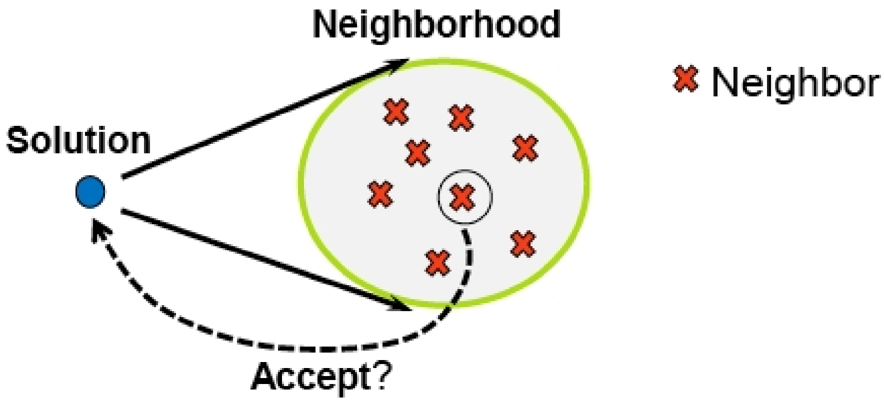



**FIGURE 8.1 –** Exemple de recherche locale

- **improveSolution** : le but est de satisfaire un peu plus de clauses a chaque itérations en inversant la valeur d'une variable si elle apparait dans une clause non satisfaites de l'instance, si la nouvelle solution est meilleure que l'orignal, alors elle deviendra la solution construite par la fourmi.

# Chapitre 9

# Expérimentations

## 9.1 Données

Les données de tests sont les même vues dans I3.1, ici nous traiterons uniquement les instances satisfiables (uf75-325).

## 9.2 Machines

Les machines sont les mêmes que celles utilisées dans 3.2.1 pour rester consistant et faire une comparaison cohérente entres les résultats.

## 9.3 Résultats

### 9.3.1 ACS

Les conditions de test restent inchangées, autrement dit :
Dix essaies par instance pour dix instances en faisant varier les paramètres empiriques à l'exception de $\rho$ le taux d'évaporation fixé à 0.7 pour chaque test :

**Remarque** : la version d'ACS est celle avec les actions **deamons** vus dans 8.3.

| **maxItter** | **maxStep** | **Nbr. fourmis** | **alpha** | **beta** | **$q_0$** | **Moyenne** | **Taux moy.** | **Temps moy(s)** |
|---|---|---|---|---|---|---|---|---|
| 500 | 30 | 30 | 0.8 | 0.3 | 0.6 | 324.7 | 0.9990769231 | 2.51844 |
| 500 | 30 | 30 | 1 | 0 | 0.6 | 324.66 | 0.9989538462 | 2.55154 |
| 500 | 25 | 30 | 0.8 | 0 | 0.6 | 324.65 | 0.9989230769 | 2.30747 |
| 500 | 25 | 30 | 0.8 | 0.3 | 0.6 | 324.64 | 0.9988923077 | 2.22491 |
| 500 | 30 | 30 | 1 | 0.3 | 0.6 | 324.6 | 0.9987692308 | 2.47566 |
| 500 | 25 | 30 | 1 | 0 | 0.6 | 324.57 | 0.9986769231 | 2.28502 |

**TABLE 9.1 –** Meilleurs jeux de paramètres pour l'ensemble des instances choisies (ACS)

### 9.3.2 AS

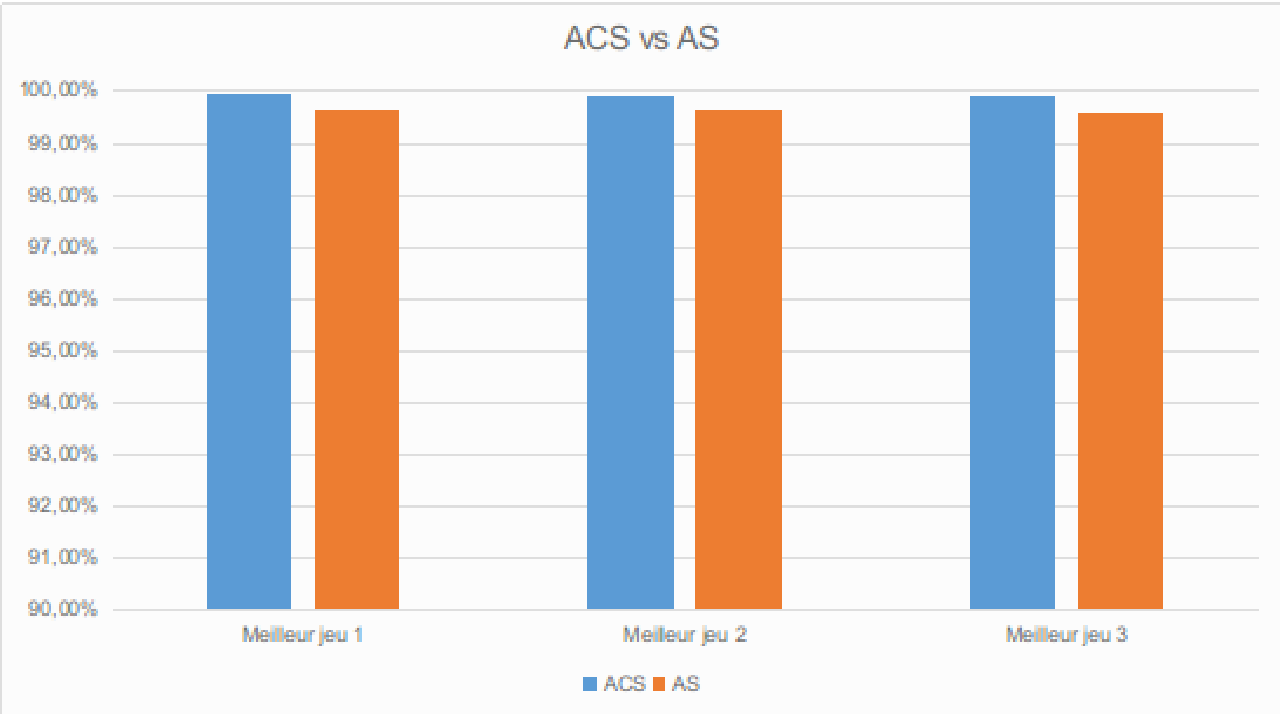
Les conditions restent inchangées.

| maxItter | maxStep | Nbr. fourmis | alpha | beta | Moy. Sat | Taux moy. | Temps moy(s) |
|---|---|---|---|---|---|---|---|
| 100 | 30 | 30 | 1 | 0.3 | 323.75 | 0.9961538462 | 0.52013 |
| 100 | 25 | 30 | 1 | 0.3 | 323.67 | 0.9959076923 | 0.45238 |
| 500 | 30 | 30 | 1 | 0 | 323.65 | 0.9958461538 | 2.50055 |
| 100 | 30 | 30 | 0.8 | 0 | 323.65 | 0.9958461538 | 0.56023 |
| 500 | 25 | 30 | 0.8 | 0 | 323.64 | 0.9958153846 | 2.0897 |
| 100 | 25 | 30 | 0.8 | 0.3 | 323.61 | 0.9957230769 | 0.43755 |

**TABLE 9.2 –** Meilleurs jeux de paramètres pour l'ensemble des instances choisies (AS)

## 9.4 Comparaison entre AS et ACS

Le graphe suivant compare les taux de satisfiabilité moyens des trois meilleurs jeux de paramètres pour AS et ACS :




Comme le prévoyait l'aspect théorique, ACS est nettement supérieure en terme de taux de satisfiabilité à son homologue AS, cela est dû principalement au fait que ce dernier n'est pas une représentation fidèle du comportement réel des fourmis, contrairement à ACS qui, par le bias de la marche aléatoire, se veut plus réaliste et représentatif de la vie réelle. bien que la différence est négligeable, cela reste une comparaison a petite échelle, dans les cas réels les instances sont beaucoup plus complexes et difficiles à résoudre.

# Quatrième partie

# Comparaisons et conclusions générale

# Chapitre 10

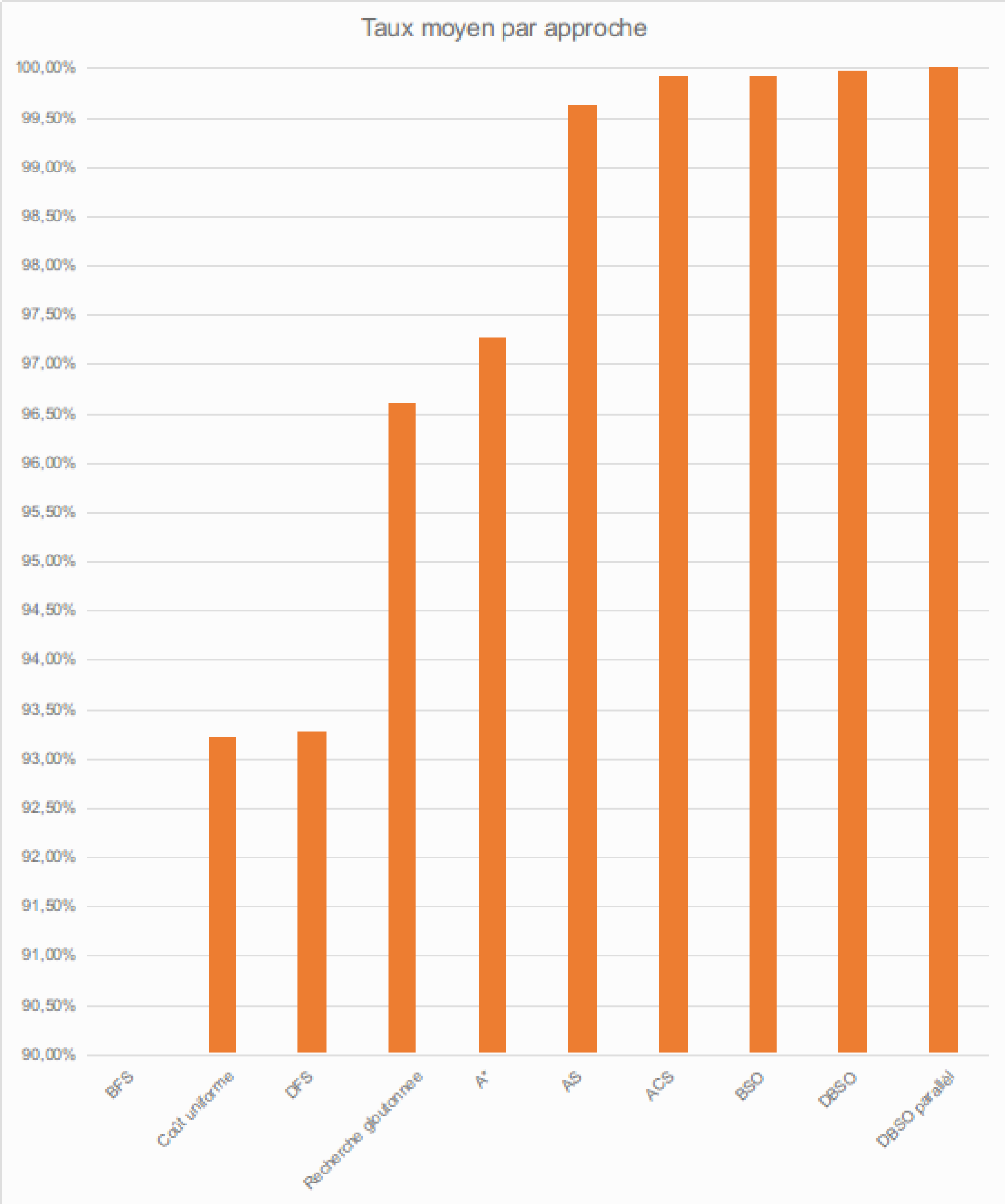
# Comparaisons des trois approches

## 10.1 Résumé

Afin de résumer et de visualiser les comparaisons entre les trois approches, le tableau récapitulatif suivant est proposé(accompagné de son graphe) :

| **Approche** | **Taux moy.** |
|---|---|
| A* | 97,262% |
| ACS | 99,908% |
| AS | 99,615% |
| BFS | 42,902% |
| BSO | 99,914% |
| Coût uniforme | 93,215% |
| DBSO | 99,969% |
| DBSO parallèle | 100,000% |
| DFS | 93,268% |
| Recherche gloutonne | 96,600% |

**TABLE 10.1 –** Récapitulatif de toutes les approches




**Commentaires :** Une vite observations peut être faite, c'est que les méthode de l'approche de la partie I sont très en dessous de la norme désirée, à titre d'exemple **BFS** n'atteint même pas la barre des 50%, la meilleure méthode étant la modification dynamique de BSO que nous avons appelé **D-BSO**(Dynamic Bee Swarm Optimization) avec traitement parallèle dont le taux **MOYEN** de satisfaction est égale à 100%.

# Conclusion générale

Arrivé à la fin de ce projet, et après beaucoup de temps passé à apprendre, modifier et tester les différents algorithmes vu en cours, nous pensons avoir achevé un travail que nous jugeons assez complet, nous avons exploré différents aspects de la résolution de problème, en partant des méthodes basique aux méthode plus avancées, nous pouvons résumé notre travail aux points suivants :

- Les méthodes classiques du début de l'ère de l'intelligence artificielle ont prouvé leur efficacité jusqu'au jour d'aujourd'hui, cependant leurs limite s'est vu apparaître en même temps que l'apparition de problèmes beaucoup plus complexes et surtout plus volumineux.
- De nouvelles méthodes ont fait leur apparition, sacrifiant le désir de trouver une solution exacte(out optimale) qui peut prendre un temps inconcevable pour être détermine, au profit de solutions, certe moins optimales mais qui demeurent une alternative raisonnable.
- Malgré le coté aléatoire et probabiliste des nouvelles approches méta-heuristique, leur façon de fonctionner en fait une représentation fidèle de la vie réelle en générale.
- Les meilleures méthodes classique ne sont pas encore a jeté à la poubelle, car elles peuvent encore êtres utilisées pour améliorer les méthodes modernes, à l'instar de ACO qui a su marier méthodes évolutionnaire et constructives

Talk about the good sides of BSO and its bad sides, talk about the potential that resides in the metaheuristics and swarm intelligence.

# Table des figures

3.1 Machine **A** pour les instances contradictoires . . . . . 16
3.2 Machine **B** pour les instances satisfiables . . . . . 16
3.3 Fenêtre principale . . . . . 18
3.4 Attempts . . . . . 19
3.5 XYChart . . . . . 19
3.6 Clauses satisfied during evaluation of execution of UF75-325-01 . . . . . 20
3.7 Illustration des données de **??** . . . . . 21
3.8 Illustration des données de **??** . . . . . 22
3.9 Illustration des données de la table 3.10 . . . . . 23
3.10 Illustration des données de la table 3.4 . . . . . 24
3.11 Illustration des données de la table 3.5 . . . . . 25
3.12 Illustration des données de la table 3.6 . . . . . 26
3.13 Illustration des données de la table 3.7 . . . . . 27
3.14 Illustration des données de la table 3.8 . . . . . 28
3.15 Illustration des données de la table 3.9 . . . . . 29
3.16 Illustration des données de la table 3.10 . . . . . 30
3.17 Illustration des données de la table 3.11 . . . . . 31
3.18 Illustration des données de la table 3.12 . . . . . 32

4.1 Exemple d'un espace de solution multidimensionnel . . . . . 36
4.2 Exemples de recherche ans l'espace des solutions selon une fonction d'évaluation 37
4.3 Abeilles communiquant pour la recherche de nourriture . . . . . 37

5.1 Illustration des régions de recherche des différentes itérations . . . . . 42
5.2 Illustration de BSO avec un nombre de recherche locale égale à la distance entre les solutions . . . . . 43
5.3 Illustration de BSO avec un nombre de recherche locale inférieur à la distance entre les solutions . . . . . 43

6.1 Taux de satisfiabilité selon la nature de l'instance . . . . . 47
6.2 Apport du parallélisme . . . . . 48

7.1 Abeilles communiquant pour la recherche de nourriture . . . . . 51

8.1 Exemple de recherche locale . . . . . 60

A.1 Algorithme de recherche BSO générique . . . . . 71
A.2 Algorithme de recherche tabou générique . . . . . 72
A.3 Structure de Dance en tant que Tas . . . . . 73
A.4 Schéma d'une abeille générique . . . . . 74

B.1 Algorithme de recherche ACS générique . . . . . 75

B.2 Algorithme de recherche AS générique . . . . . . . . . . . . . . . . . . . . . . 76
B.3 Schéma d'une fourmi générique . . . . . . . . . . . . . . . . . . . . . . . . . . 76
B.4 Algorithme de construction par une fourmi ACS . . . . . . . . . . . . . . . . . 77
B.5 Algorithme de construction par une fourmi AS . . . . . . . . . . . . . . . . . . 78
B.6 Mise à jour en-ligne ACS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
B.7 Mise à jour en-ligne AS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
B.8 Schéma générique de la phéromone . . . . . . . . . . . . . . . . . . . . . . . 79
B.9 Calcul de $P_{i,j}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
B.10 Calcul du taux de phéromone . . . . . . . . . . . . . . . . . . . . . . . . . . 79
B.11 Calcul du taux de phéromone a être déposé sur un litéral . . . . . . . . . . . . 80

# Liste des tableaux

3.1 Tableau récapitulatif des résultats pour les instances satisfiables . . . . . . . . 21
3.2 Tableau récapitulatif des résultats pour les instances non-satisfiables . . . . . 22
3.3 Tableau récapitulatif des résultats pour les instances satisfiables . . . . . . . . 23
3.4 Tableau récapitulatif des résultats pour les instances non-satisfiables . . . . . 24
3.5 Tableau récapitulatif des résultats pour les instances satisfiables . . . . . . . . 25
3.6 Tableau récapitulatif des résultats pour les instances non-satisfiables . . . . . 26
3.7 Tableau récapitulatif des résultats pour les instances satisfiables . . . . . . . . 27
3.8 Tableau récapitulatif des résultats pour les instances non-satisfiables . . . . . 28
3.9 Tableau récapitulatif des résultats pour les instances satisfiables . . . . . . . . 29
3.10 Tableau récapitulatif des résultats pour les instances non-satisfiables . . . . . 30
3.11 Tableau de mesures statistiques pour les instances satifsiables . . . . . . . . . 31
3.12 Tableau de mesures statistiques pour les instances non-satifsiables . . . . . . . 32
3.13 Nombre d'évaluations par seconde . . . . . . . . . . . . . . . . . . . . . . . . 32

6.1 Meilleures combinaisons des paramètres empiriques pour les instances uf75-325 44
6.2 Impact du paramètres **MaxItteraitions** . . . . . . . . . . . . . . . . . . . . . 45
6.3 Impact du paramètres **Flip** . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.4 Impact du paramètres **Nombre d'abeilles** . . . . . . . . . . . . . . . . . . . 45
6.5 Impact du paramètres **Nombre de recherches locales** . . . . . . . . . . . . . 45
6.6 Résume pour les instances **uf75-325** . . . . . . . . . . . . . . . . . . . . . . 46
6.7 Résume pour les instances **uf100-430** . . . . . . . . . . . . . . . . . . . . . 46
6.8 Résume pour les instances **jnh200-800** . . . . . . . . . . . . . . . . . . . . . 46
6.9 Temps moyen passer à l'évaluation (DBSO vs DBSO-Parallèle) . . . . . . . . . 47

8.1 Table des phéromones initiale . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
8.2 Table des phéromones après le passage d'une fourmi . . . . . . . . . . . . . . 53

9.1 Meilleurs jeux de paramètres pour l'ensemble des instances choisies (ACS) . . 61
9.2 Meilleurs jeux de paramètres pour l'ensemble des instances choisies (AS) . . . 62

10.1 Récapitulatif de toutes les approches . . . . . . . . . . . . . . . . . . . . . . 64

# Bibliographie

[1] **SATLIB** - benchmark for **SAT** problems. `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

[2] S. A. Cook. *The Complexity of Theorem-proving Procedures*. 1971.

[3] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. *Optimization by Simulated Annealing*. 1983.

[4] S. S., D. H., A. E. H. O., and K. A. *ABSO : Advanced Bee Swarm Optimization Metaheuristic and Application to Weighted MAX-SAT Problem.* 2011.

# Annexe A

# Code source BSO

```
public abstract class BSOAbstract<T>
{
    protected TabuList<T> tabuList;
    protected Dances<T> dances;

    public BSOAbstract(TabuList<T> tabuList, Dances<T> dances) {
        this.tabuList = tabuList;
        this.dances = dances;
    }
    abstract protected boolean end(T solution);
    abstract protected List<Bee<T>> determineSearchPoints(T solution);
    public T search(Bee<T> beeInit)
    {
        T sRef = beeInit.init();
        T bestSolution = sRef;
        double evaluationBest = dances.evaluate(sRef);
        while (!end(sRef))
        {
            tabuList.add(sRef);
            List<Bee<T>> bees = determineSearchPoints(sRef);

            for(Bee<T> bee : bees)
                dances.add( bee.search() );

            do {
                sRef = dances.getBest();
            }while (sRef != null && tabuList.contains(sRef));

            if(sRef == null)
                sRef = beeInit.init();

            double evaluation = dances.evaluate(sRef);
            if(evaluation < evaluationBest) {
                bestSolution = sRef;
                evaluationBest = evaluation;
            }
        }
        return bestSolution;
    }
```

**FIGURE A.1 –** Algorithme de recherche BSO générique

```java
public abstract class TabuSearchAbstract<T>
{
    TabuList<T> tabuList;

    public TabuSearchAbstract(TabuList<T> tabuList) { this.tabuList = tabuList; }

    // get neighbors of parameter solution
    abstract protected List<T> getNeighbors(T solution);
    // fitness function to be minimized
    abstract protected double evaluate(T solution);

    // if stopping condition is met e.g: optimal solution found, time limit or maximum number of iterations reached
    abstract protected boolean end(T solution);

    public T search(T start)
    {
        T sBest = start;
        T currentBest = start;
        tabuList.add(start);
        while(!end(currentBest))
        {
            List<T> neighbors = getNeighbors(currentBest);

            for (T neighbor : neighbors)
                if(!tabuList.contains(neighbor) &&
                        (currentBest == null || evaluate(neighbor) < evaluate(currentBest)))
                    currentBest = neighbor;

            if(evaluate(currentBest) < evaluate(sBest))
                sBest = currentBest;

            tabuList.add(currentBest);
        }
        return sBest;
    }
}
```

**FIGURE A.2 –** Algorithme de recherche tabou générique

```java
public abstract class DancesHeap<T> extends Dances<T>
{
    protected Heap<T> heap = new Heap<T>() {
        @Override
        public int compare(T n1, T n2) {
            return compareSolutions(n1,n2);
        }
    };


    @Override
    public void add(T solution)
    {
        heap.add(solution);
    }

    @Override
    public T getBest() {
        return heap.getRoot();
    }

    protected int compareSolutions(T sol1,T sol2)
    {
        double e1 = evaluate(sol1),e2 = evaluate(sol2);
        if(e1 > e2)
            return 1;
        if(e1 < e2)
            return -1;
        return 0;
    }
}
```

**FIGURE A.3 –** Structure de Dance en tant que Tas

```java
/**
 *  T is the solution type
 *  S is the searchPoint
 * Created by ressay on 29/03/18.
 */
public abstract class Bee<T>
{
    protected T searchZone;
    protected abstract T init();
    protected abstract T search();
}
```

**FIGURE A.4 –** Schéma d'une abeille générique

# Annexe B

# Code source ACO

```java
/**
 * Abstract behaviour of the ACS algorithm according to a solution of type T
 *
 * @param <T> the nature of the solution
 */
public abstract class ACS<T> extends ACO<T>
{

    @Override
    public T startResearch()
    {
        this.numberOfItterations = 0;
        for (; ; )
        {
            ants = new TreeSet<>();
            initAnts();
            for (Ant<T> ant : ants)
            {
                ant.constructSolution();
                deamons(ant);
                if (isValidSolution(ant.solution))
                {
                    return ant.solution;
                }

            }
            Ant<T> bestAnt = getBestAnt();
            if (bestAnt.compareTo(this.bestAnt) < 0)
                this.bestAnt = bestAnt;
            System.out.println("THE BEST SO FAR IS " + evaluateSolution(this.bestAnt.solution));
            offlinePheromonUpdate(this.bestAnt);
            if (end(this.bestAnt.solution))
            {
                return this.bestAnt.solution;
            }
        }
    }
```

**FIGURE B.1 –** Algorithme de recherche ACS générique

```java
private void onlineStepByStepPheromonUpdate(int variable, int literal)
{
    double Ti = instance.getPheromons().getPheromonValues()[variable][literal];
    double cost = variable;
    instance.getPheromons().getPheromonValues()[variable][literal] =
            (1 - instance.getPheromons().getRo()) * Ti
                    +
            instance.getPheromons().getRo() * instance.getPheromons().getTo();
}
```

**FIGURE B.2 –** Algorithme de recherche AS générique

```java
/**
 * CREATED BY wiss ON 22:24
 **/

/**
 * Abstract structure of any Ant
 *
 * @param <T> nature of the solution the ant is building
 */
public abstract class Ant<T> implements Comparable<Ant<T>>
{

    public T solution;

    public abstract void constructSolution();

    public abstract void improveSolution();

    public abstract void exploreNeighbors(int maxStep);
}
```

**FIGURE B.3 –** Schéma d'une fourmi générique

```java
@Override
public void constructSolution()
{
    Arrays.fill(done, Boolean.FALSE);
    for (int i = 0; i < instance.getNumberOfVariables(); i++)
    {
        if (done[i])
        {
            continue;
        }
        double proba = getProba(i, literal: 1);
        double probaNot = getProba(i, literal: 0);
        double q = ThreadLocalRandom.current().nextDouble();

        if (q <= qProba)
        {
            int[] argmax = getArgmax();
            if (argmax[0] == 1)
            {
                solution.set(argmax[1]);
            } else
            {
                solution.clear(argmax[1]);
            }
        } else
        {
            done[i] = true;
            q = ThreadLocalRandom.current().nextDouble( bound: 1);
            if (q < proba)
            {
                solution.set(i);
                onlineStepByStepPheromonUpdate(i, literal: 1);
            } else
            {
                solution.clear(i);
                onlineStepByStepPheromonUpdate(i, literal: 0);
            }
        }
    }
```

**FIGURE B.4 –** Algorithme de construction par une fourmi ACS

```
public class AntSATAS extends AntSAT
{
    public AntSATAS(SATInstance instance) { super(instance); }

    @Override
    public void constructSolution()
    {
        boolean[] done = new boolean[solution.length()];
        for (int i = 0; i < instance.getLiteralsBitSet()[0].length; i++)
        {
            if (done[i])
                continue;
            double proba = getProba(i,  literal: 1);
            double probaNot = getProba(i,  literal: 0);
            double q = ThreadLocalRandom.current().nextDouble();
            done[i] = true;
            if (q < proba)
            {
                solution.set(i);
                onlineStepByStepPheromonUpdate(i,  literal: 1);
            } else
            {
                solution.clear(i);
                onlineStepByStepPheromonUpdate(i,  literal: 0);
            }
        }
    }
```

**FIGURE B.5 –** Algorithme de construction par une fourmi AS

```
private void onlineStepByStepPheromonUpdate(int variable, int literal)
{
    double Ti = instance.getPheromons().getPheromonValues()[variable][literal];
    double cost = variable;
    instance.getPheromons().getPheromonValues()[variable][literal] =
            (1 - instance.getPheromons().getRo()) * Ti
                    +
            instance.getPheromons().getRo() * instance.getPheromons().getTo();
}
```

**FIGURE B.6 –** Mise à jour en-ligne ACS

```
private void onlineStepByStepPheromonUpdate(int variable, int literal)
{
    double Ti = instance.getPheromons().getPheromonValues()[variable][literal];
    double cost = variable;
    instance.getPheromons().getPheromonValues()[variable][literal] =
            (1 - instance.getPheromons().getRo()) * Ti
                    +
            instance.getPheromons().getRo() * this.getDelta(variable, literal);
}
```

**FIGURE B.7 –** Mise à jour en-ligne AS

```java
/**
 * Abstract structure of the Pheromon information
 *
 * @param <T> the initial value of all the pheromons
 * @param <S> the type of structure where the pheromons are stocked
 */
public abstract class Pheromons<T, S>
{
    public T initValue;
    public S pheromonValues;

    public Pheromons(T initValue) { this.initValue = initValue; }

    public abstract void init(T initValue);
}
```

**FIGURE B.8 –** Schéma générique de la phéromone

```java
public double getProba(int variable, int literal)
{
    return (double) getPherHeur(variable, literal) / getTotalPherHeur(variable);
}

protected double getTotalPherHeur(int variable)
{
    double sum = 0;
    for (int j = 0; j < 2; j++)
    {
        sum += getPherHeur(variable, j);
    }
    return sum;
}
```

**FIGURE B.9 –** Calcul de $P_{i,j}$

```java
protected double getPherHeur(int variable, int literal)
{
    double mu = (double) (instance.getNumberOfClauses() -
            instance.getLiteralsBitSet()[literal][variable].cardinality());

    double muBeta = Math.pow(mu, instance.getPheromons().getBeta());

    double ti = (double) instance.getPheromons().getPheromonValues()[variable][literal];

    double tiAlpha = Math.pow(ti, instance.getPheromons().getAlpha());

    return tiAlpha * muBeta;
}
```

**FIGURE B.10 –** Calcul du taux de phéromone

```java
public double getDelta(int variable, int literal)
{
    return (double) (instance.getLiteralsBitSet()[literal][variable].cardinality());

}
```

**FIGURE B.11 –** Calcul du taux de phéromone a être déposé sur un litéral