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

Première partie

Approche par espace des états

Chapitre 1

Introduction :

1.1 Problématique :

Dans ce TP, nous allons tenter d'implémenter et de comparer plusieurs méthodes 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 TP.

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 litérraux [2], ce problème est le premier à avoir été démontré comme étant **NP-Complet**, et cela par Stephen Cook dans [**?**], et qui a donc posé les fondements de l'informathique théoriques et de la théorie de la complexité.

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

1.2.2 Stratégie de recherche dans l'espace des états

En cosidérant l'espace de recherche comme étant une arborescence, dont les noeuds sont les differents états du problème, nous pouvons classer les différentes stratégies de recherches en deux grandes catégories :

1.2.3 Stratégie de rercherche aveugle

Cette catégories englobe les stratégies ou il est question de passer par toutes les solutions et les tester une à une, dans ce TP nous nous intéresserons plus particulièrement aux algortihmes/methodes suivant(es) :

Par prodonfeur d'abord (DFS)

L'algorithme de parcours en profondeur d'abord consiste à visité un noeud 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 developper, une variante de cet algorithme utilise deux ensemble **Open** et **Closed** qui représentent réspectivement l'ensemble des noeuds du graphe qui n'ont pas encore étés developpés et ceux déjà développés, cet ajout permet à l'algorithme d'éviter de boucler indéfiniment sur un ensemble de noeuds.

- Complexité temporelle : $O(2^N)$
- Complexité spatiale : $O(2m)$ avec m=nombre de litéraux

En Largeur d'abord (BFS)

Cet algorithme diffère de son prédecesseur par le fait qu'il visite tous les voisins(**successeurs**) d'un noeud 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 3.3.1 page 19 et 3.5 page 32)

- Complexité temporelle : $O(2^N)$
- Complexité spatiale : $O(2^N)$ avec N=nombre de litéraux

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

- Complexité temporelle : $O(2^N)$
- Complexité spatiale : $O(2^N)$ avec N=nombre de litéraux

1.2.4 Stratégie de rercherche 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 algortihmes 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.

- Complexité temporelle : $O(2^N)$
- Complexité spatiale : $O(2^N)$ avec N=nombre de litéraux

Algorithme A*

Contrairement aux précedents algorithmes de recherche qui effectuaient une recherche de façon "naîve", 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]

- Complexité temporelle : $O(2^N)$
- Complexité spatiale : $O(2^N)$ avec N=nombre de litéraux

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

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 $i^{\text{ième}}$ 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 :

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

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*

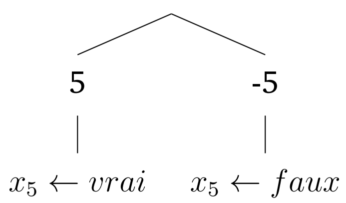
On pourrait aussi aborder la représentation du point de vu littéral, c'est à dire associer à chaque littéral les clauses dans lesquels il est présent. Pour cela un tableau de bits appelé *Bitset* pourrait être utilisé où chaque bit *i* aurait la valeur 1 si la $i^{\text{ième}}$ clause contient le littéral,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

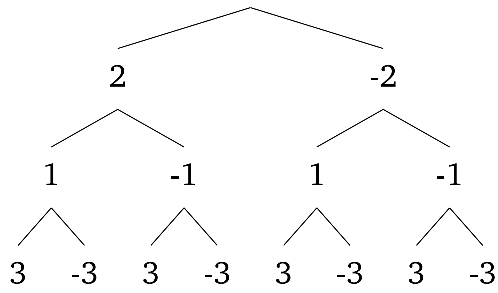
2.1.2 Représentation des états

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 assignés sont consistante avec les clauses de l'instance, négative 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.

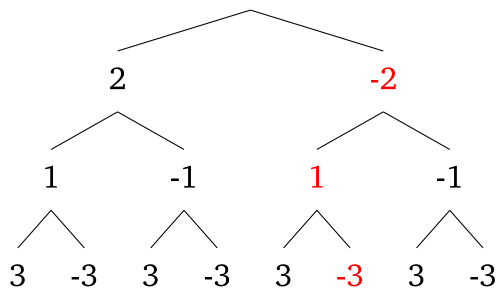


2.1.3 Développement des états

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 présenté 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 ← developper(noeud) ;
10 |  inserer les sueccesseurs 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 cette algorithme nous avons implémenter une procédure de recherche générique prenant en paramètre un type de gestion de liste, un estimateur de coût et d'heuristique et les entrés de l'instance SAT afin d'évaluer les noeuds.

2.2.1 Gestion de la liste open

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 LIFO.
Insertion d'un noeud :

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

Sélection d'un noeud :

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

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 estimé par la fonction d'évaluation. Le premier noeud est toujours sélectionner, l'insertion par contre se fait de tel sorte à garder la liste triée en ordre croissant (resp. décroissant).

Insertion d'un noeud :

$$\begin{array}{|c|c|}\hline 3 & f_3 = 5 \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|c|c|c|}\hline -3 & f_{-3} = 2 & 3 & f_3 = 5 & -1 & f_{-1} = 8 \\ \hline \end{array}$$

Sélection d'un noeud :

$$\begin{array}{|c|c|}\hline -3 & f_{-3} = 2 \\ \hline \end{array} \quad \leftarrow \quad \begin{array}{|c|c|c|c|}\hline 3 & f_3 = 5 & -1 & f_{-1} = 8 \\ \hline \end{array}$$

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

Tas 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

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

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 : $solution$;
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.

2.3 Interface graphique

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

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

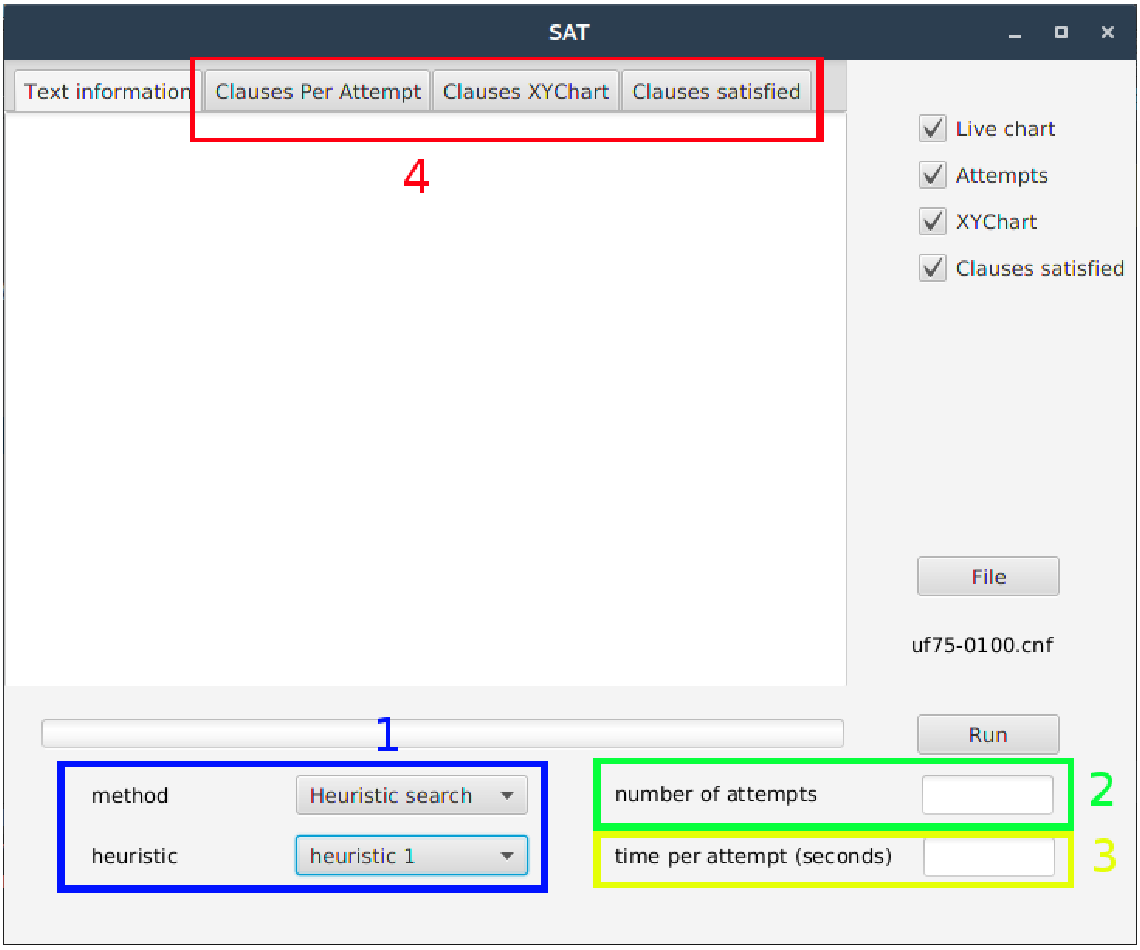


FIGURE 2.1 – Fenêtre principale

Détails

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

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

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

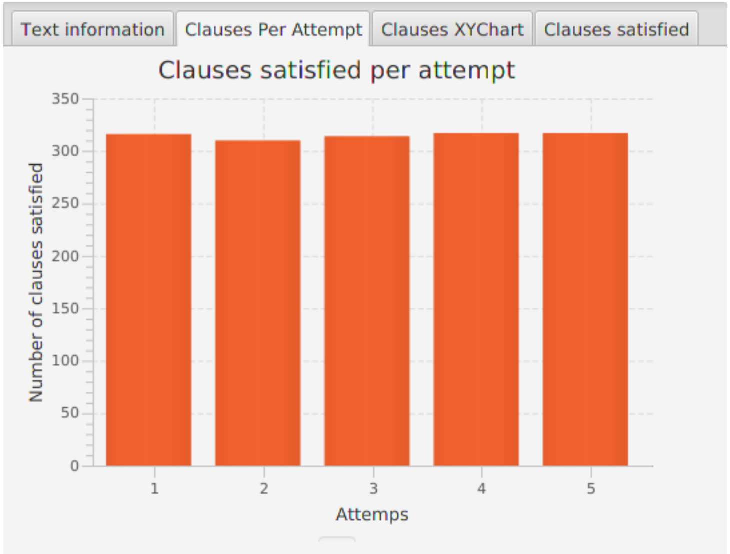


FIGURE 2.2 – Attempts

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

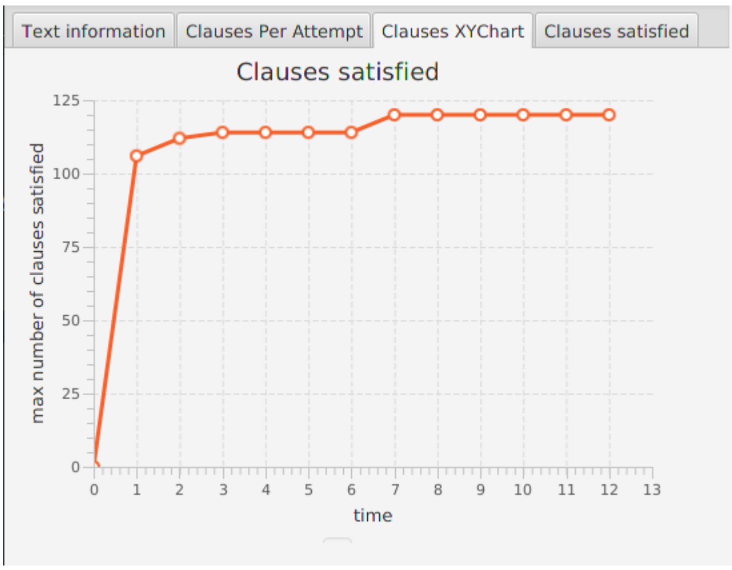


FIGURE 2.3 – XYChart

- **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 :

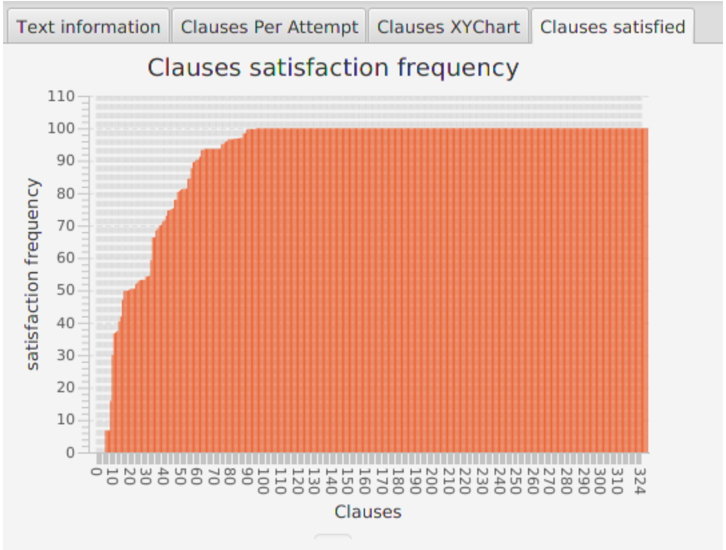


FIGURE 2.4 – Clauses satisfied

Chapitre 3

Expérimentations

3.1 Donnés

Afin de tester notre programme 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 [**?**], 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 litéraux 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 [?] 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 litéraux
    2. YY = nombre de clauses

3.2 Environement de travail

3.2.1 Machines

Pour les tests nous avons utilisé deux machines pour chaques groupes d'instances, autrement dit une machine pour éffecutuer les tests sur un ensembles d'instances satisfiables **UF75-325**[?] et une autre sur les instances contradictoires(non satisfiables) **UUF75-325**[?], les caractéristiques de chaques machines sont données dans les figures 3.1 et 3.2 suivantes :

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

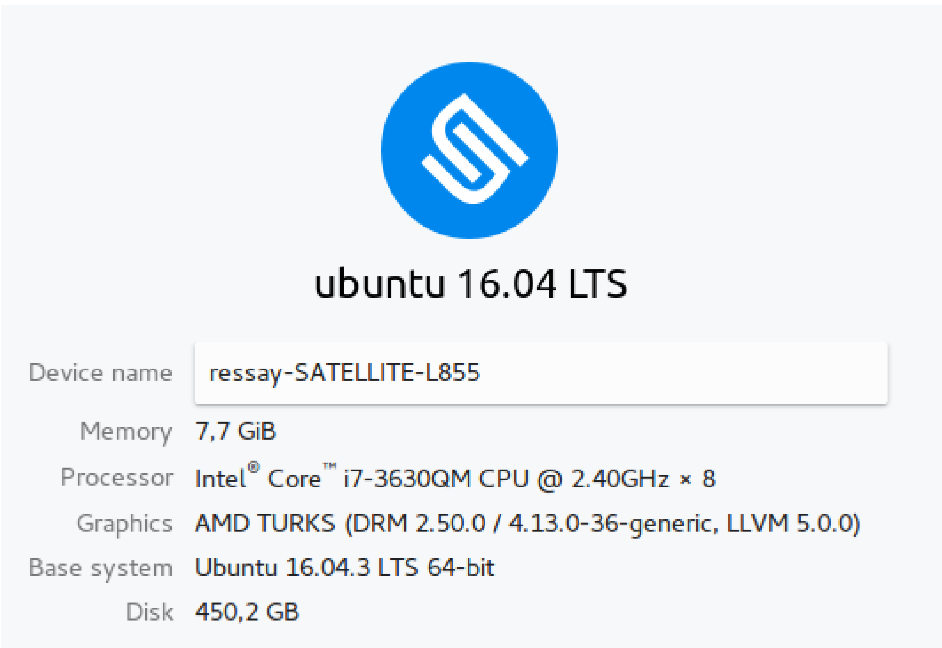


FIGURE 3.2 – Machine **B** pour les instances satisfiables

3.2.2 Outils utilisés

Langage de programmation :

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

IDE :

IntelliJ Idea L'environement de dévelopement 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 extentions qui lui sont dédiées.

3.3 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 :

3.3.1 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	Maximum clauses	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 :

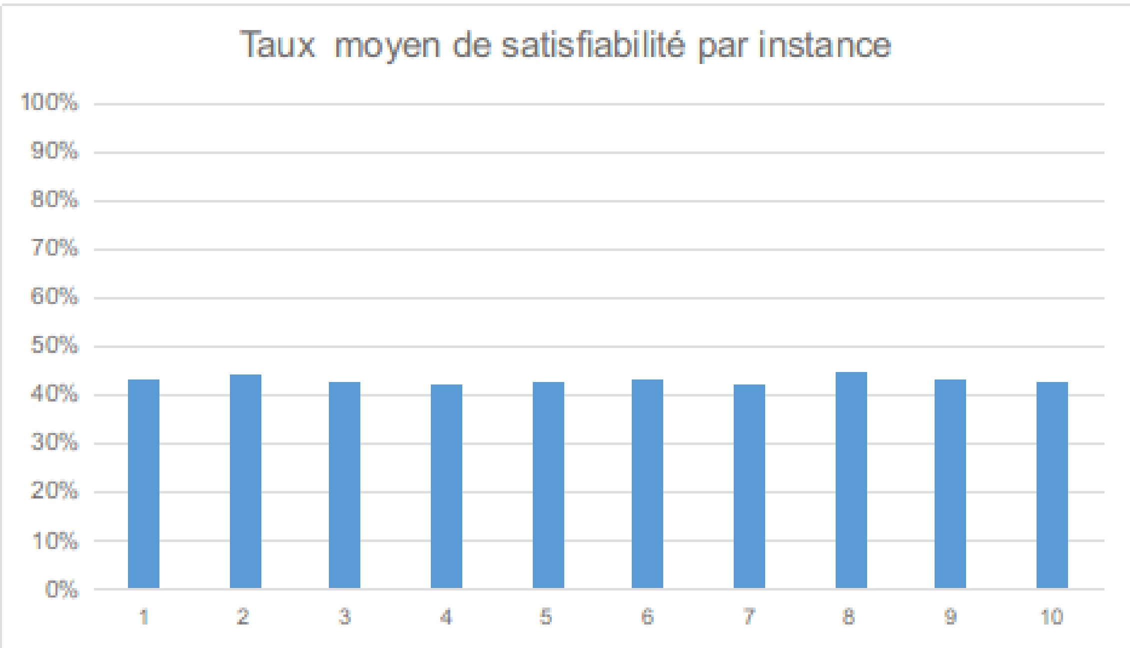


FIGURE 3.3 – Illustration des données de **??**

Pour les instances contradictoires (non sastisfiables) :

Fichiers test	Instance	Maximum clauses	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

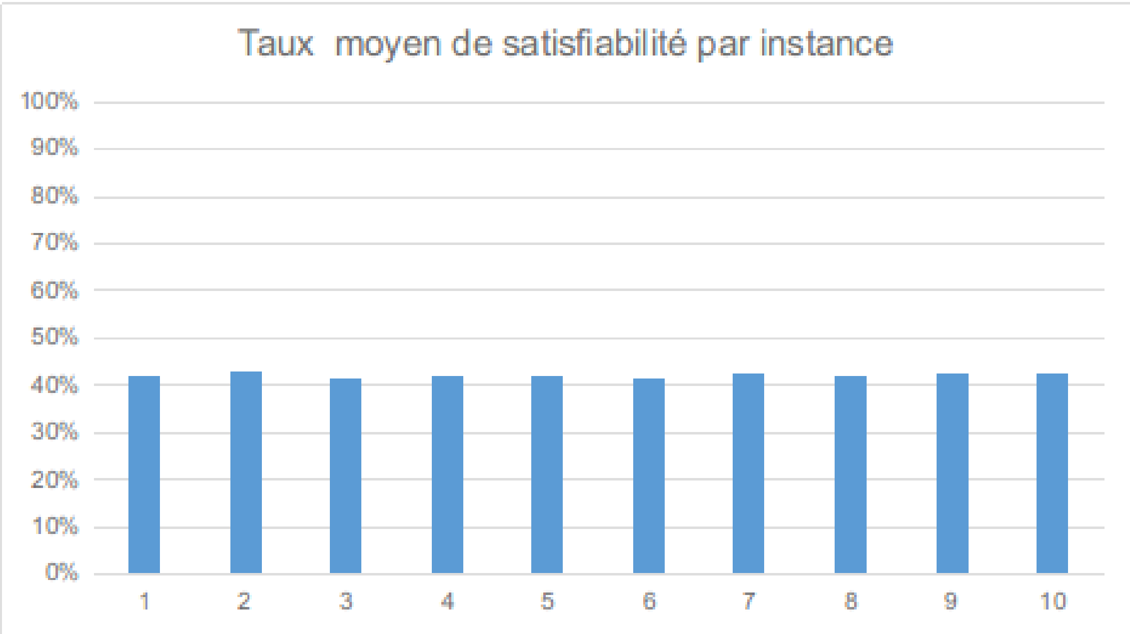


FIGURE 3.4 – Illustration des données de **??**

3.3.2 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	Maximum clauses	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

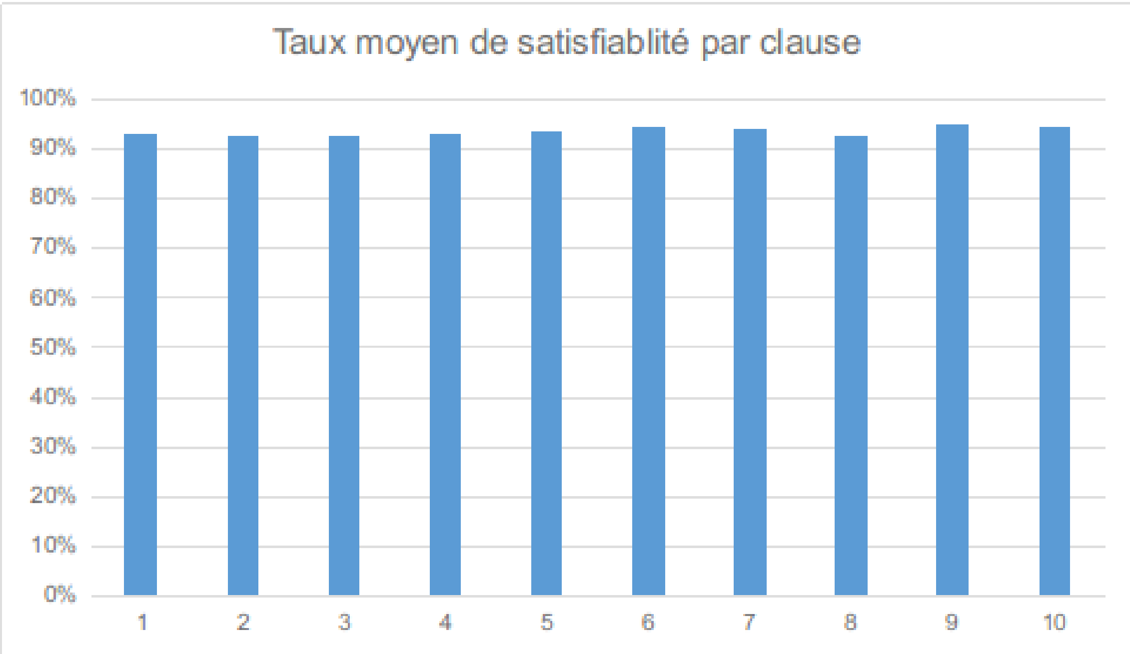


FIGURE 3.5 – Illustration des données de la table 3.3

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

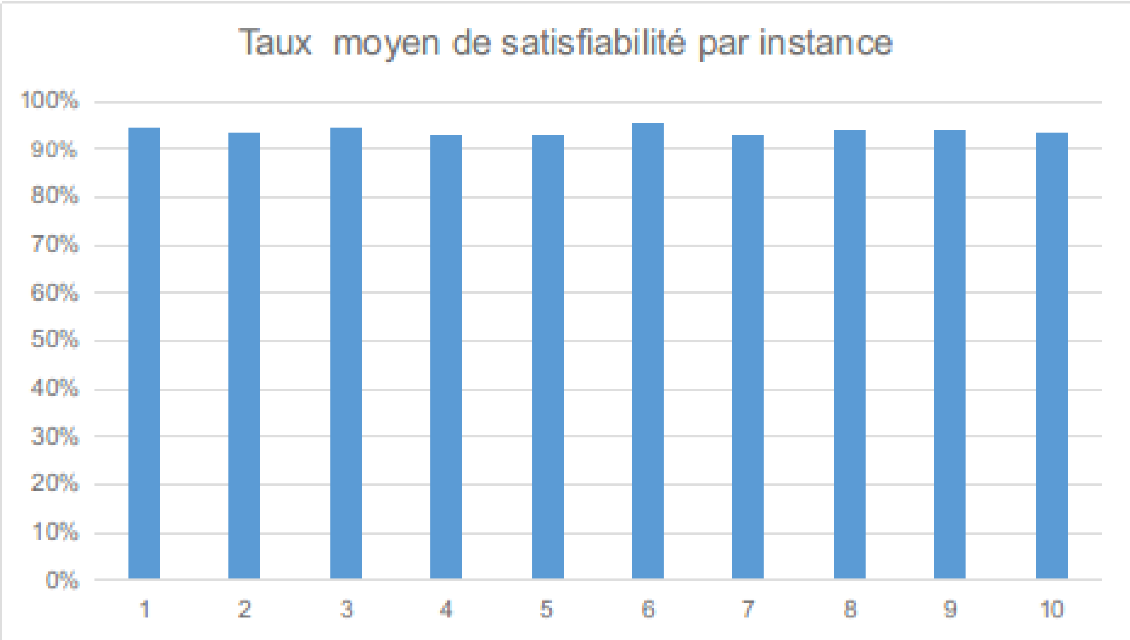


FIGURE 3.6 – Illustration des données de la table 3.4

3.3.3 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

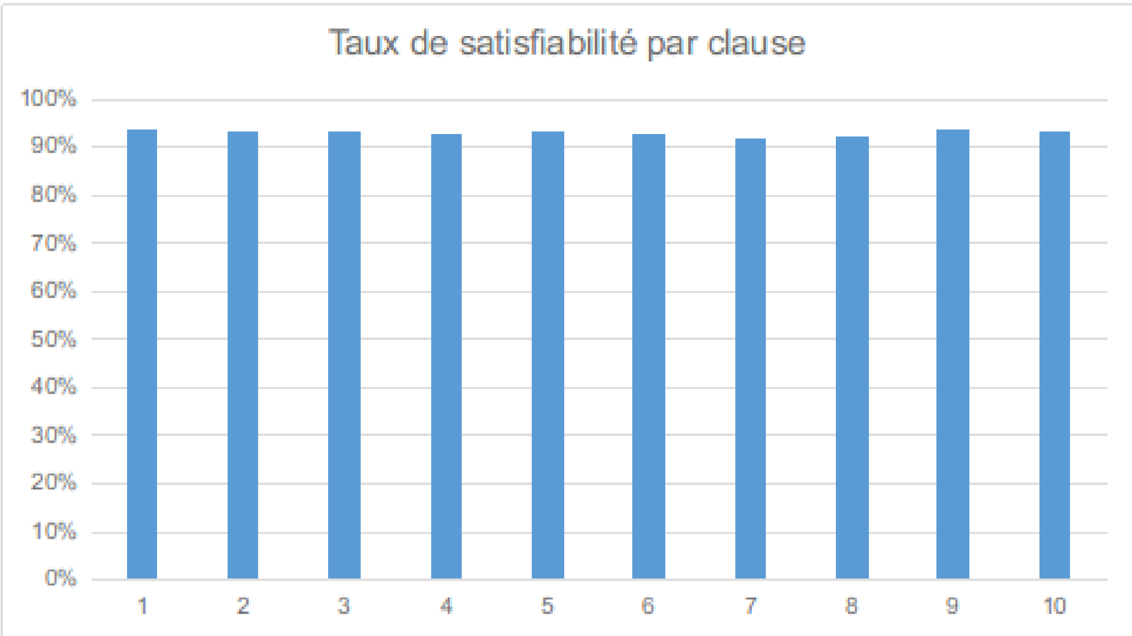


FIGURE 3.7 – Illustration des données de la table 3.5

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

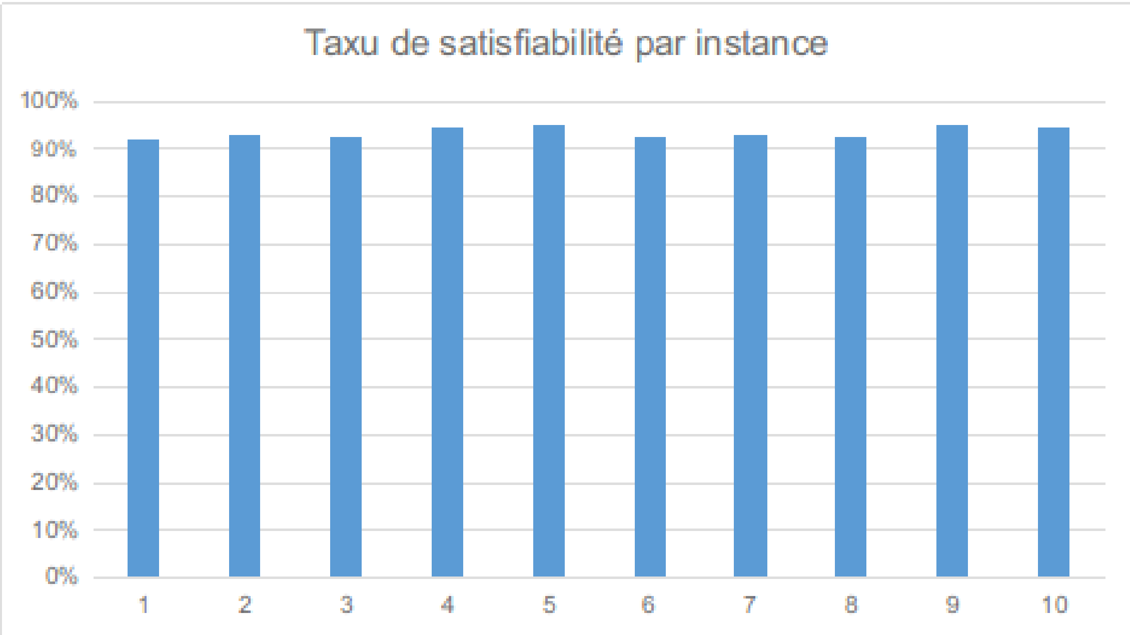


FIGURE 3.8 – Illustration des données de la table 3.6

3.3.4 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

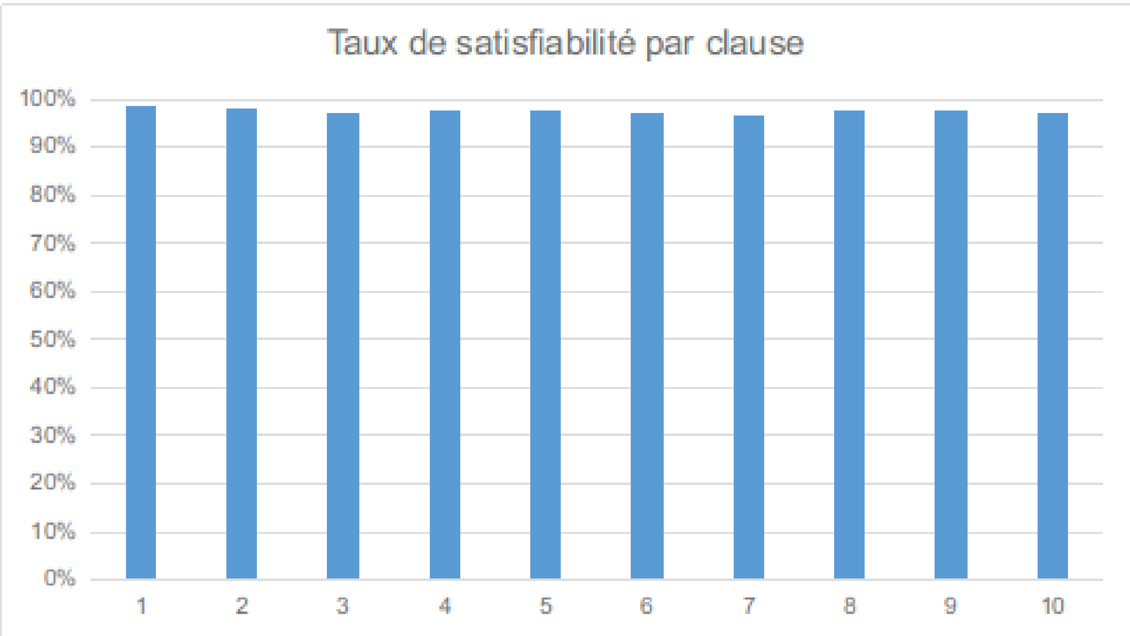


FIGURE 3.9 – Illustration des données de la table 3.7

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

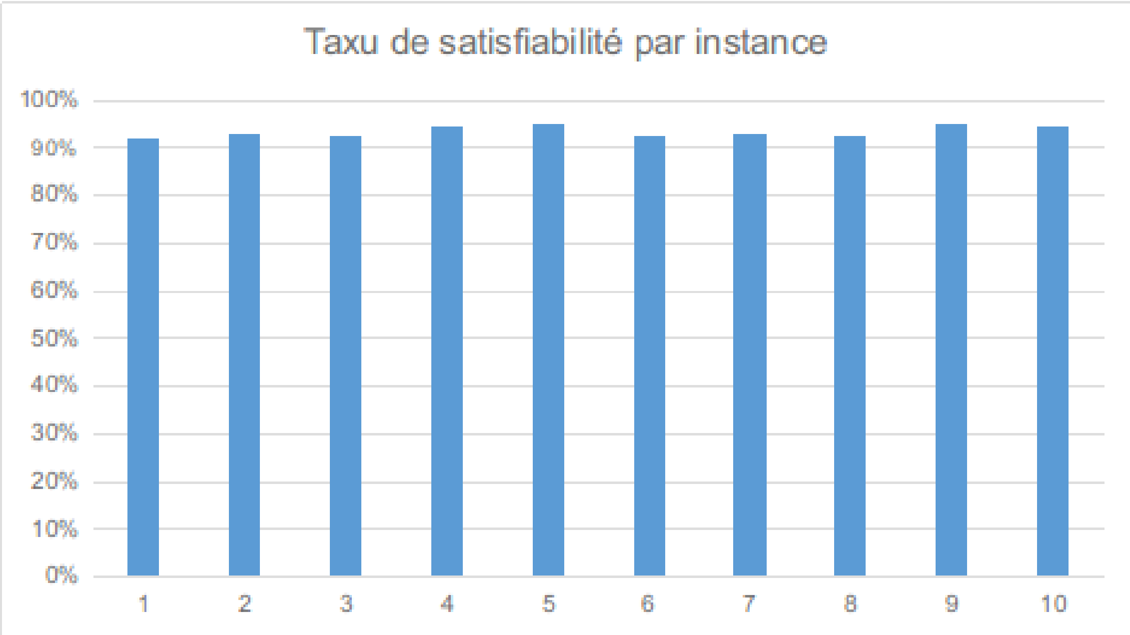


FIGURE 3.10 – Illustration des données de la table 3.8

3.3.5 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

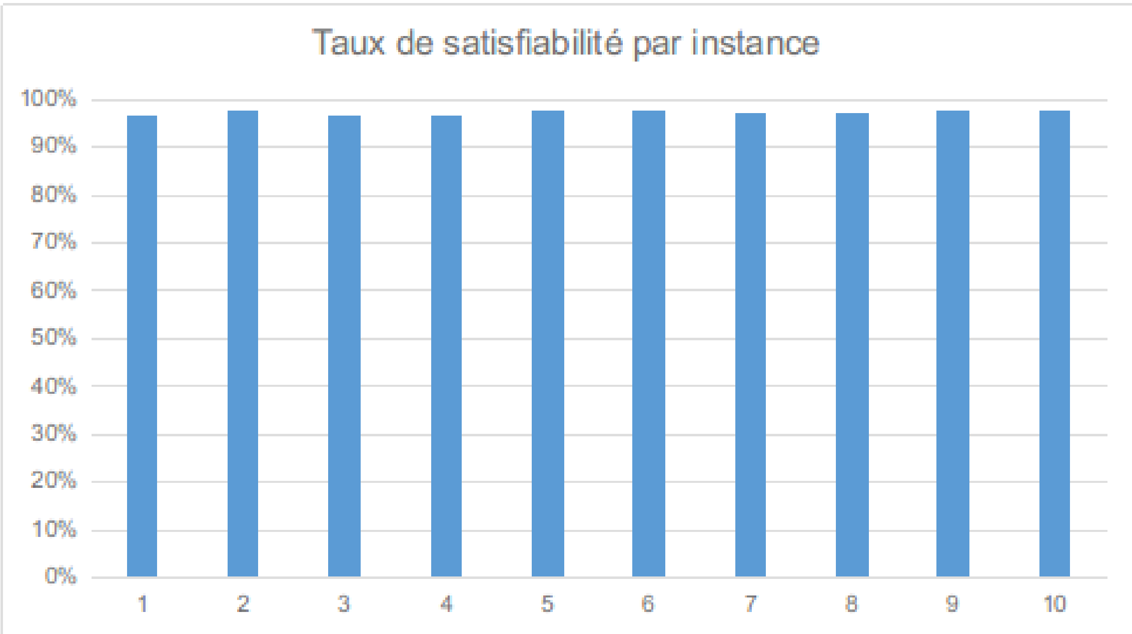


FIGURE 3.11 – Illustration des données de la table 3.9

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

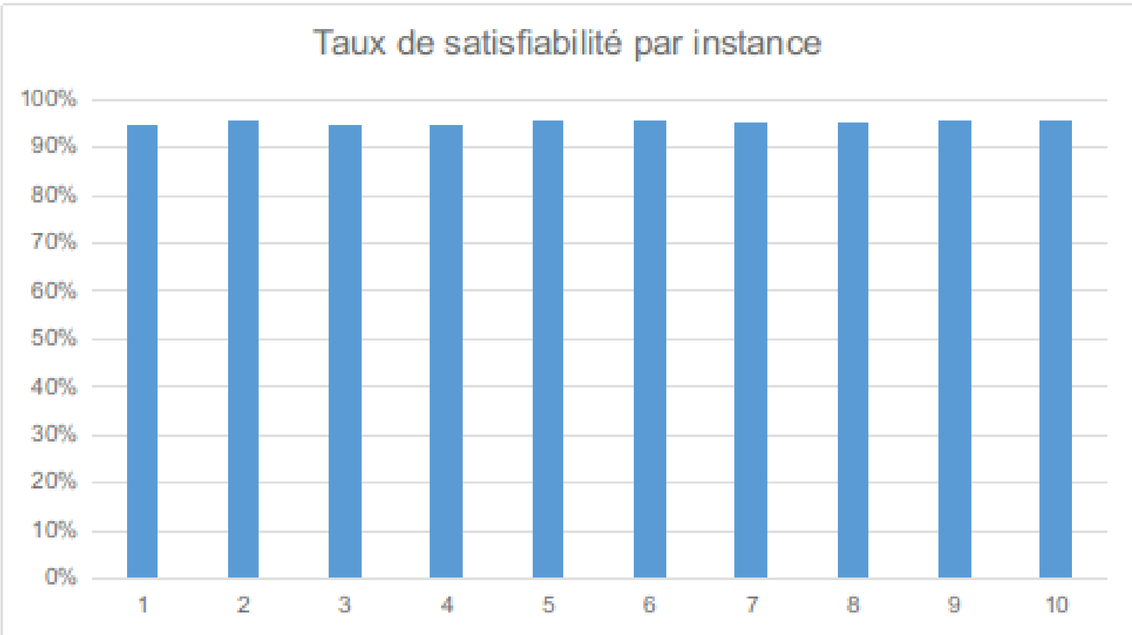


FIGURE 3.12 – Illustration des données de la table 3.10

3.4 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

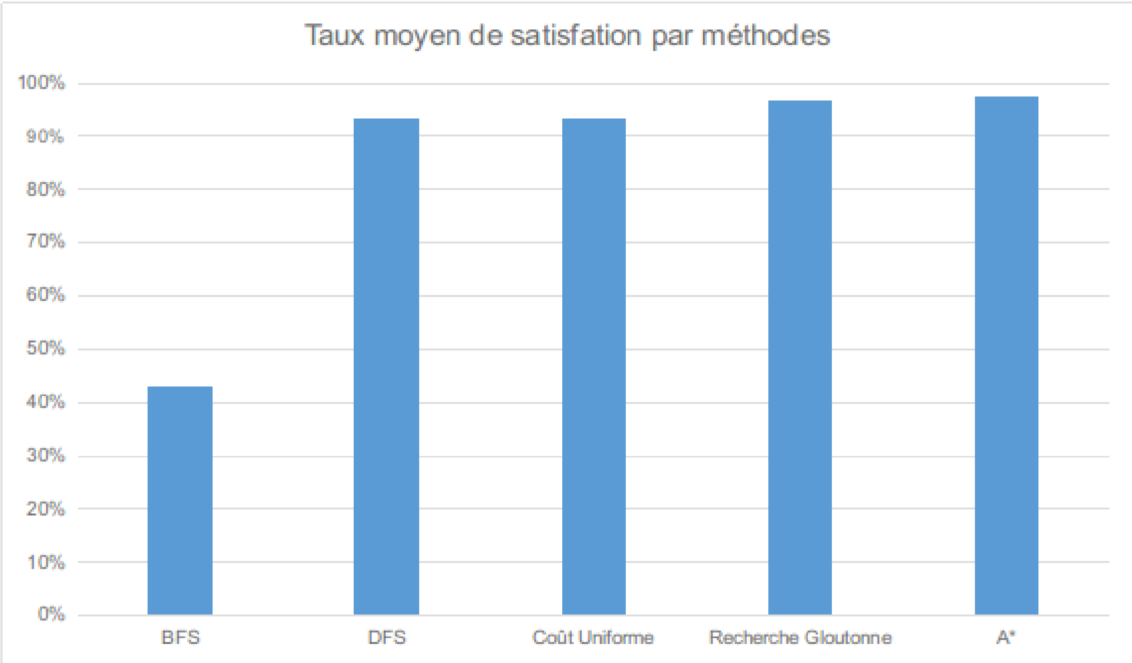


FIGURE 3.13 – Illustration des données de la table 3.11

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

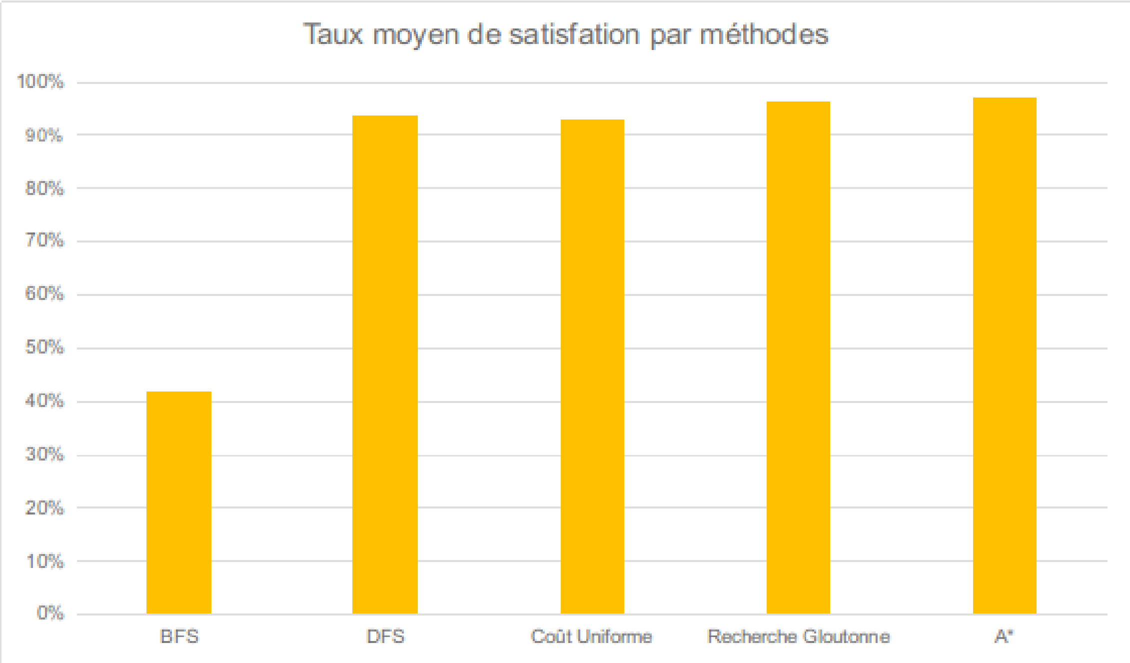


FIGURE 3.14 – Illustration des données de la table 3.12

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 3.4) démontre que la structure du BitSet proposée évalue 15 à 190 fois (selon la géstion de open) plus de clauses en une seconde que la struture de matrice, combiner cette représentation avec une gesion en tas de open, nous a fait gagné un temps assez important lors de l'évaluation et le réarrangement de open.

évaluation par / gestion de open	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

3.5 Comparaison entres les cinq méthodes

Pour conclure ce chapitre, nous allons mainetant 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 énormemment de mémoire, ce qui peut conduire à un débordement de la mémoire très rapidement.

Pour ce qu'il en est de l'agrotithme de recherche par coût uniforme, il se voit être un compromis entre l'agorithme 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 commpte 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 àléatoire, 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 satisfiabilité sur les benchmarks fournis) , cela est dû principalement au fait que les fréquences d'apparitions des litéraux soient très proches les unes des autres, aisni choisir un litéral (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 suffisaint, on ne doit pas oublié que ce taux évolue selon la taille du problème, en effet sur les instances de tailels 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 dérrière cet océan de solutions.

Deuxième partie

Approche par espace des solutions

Chapitre 4

Introduction :

4.1 Problématique :

Limite des méthodes de recherche CLASSIQUES

4.2 Définitions

4.2.1 Espace des solutions

4.2.2 Metaheuristique

4.2.3 Intelligence en essaim (Swarm intelligence)

4.2.4 Bee swarm optimization (BSO)

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 :

$$x_1 \lor \neg x_2 \lor x_4$$
$$\neg x_2 \lor x_3 \lor x_4$$
$$\neg x_1 \lor x_2 \lor \neg x_3$$

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

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 [1] 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.

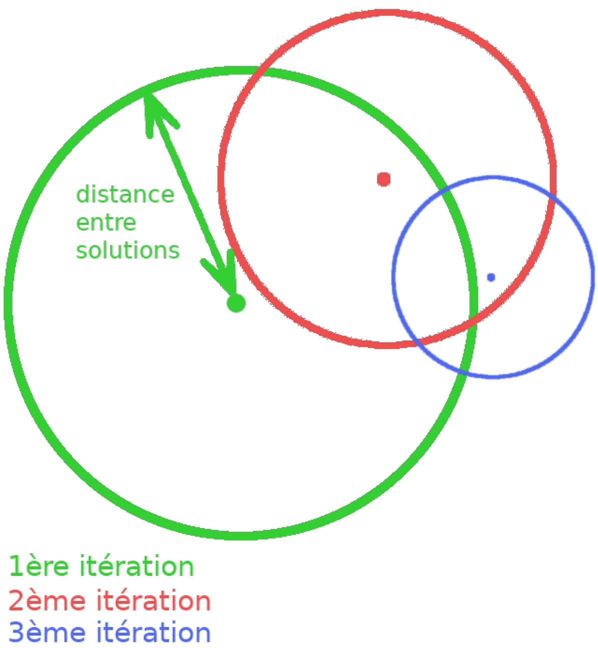


FIGURE 5.1 – Illustration des régions de recherche des différentes itérations

1. Kirkpatrick, S. ; Gelatt Jr, C. D. ; Vecchi, M. P. (1983). "Optimization by Simulated Annealing". Science. 220 (4598) : 671–680. Bibcode :1983Sci...220..671K. doi :10.1126/science.220.4598.671. JSTOR 1690046. PMID 17813860)

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 :

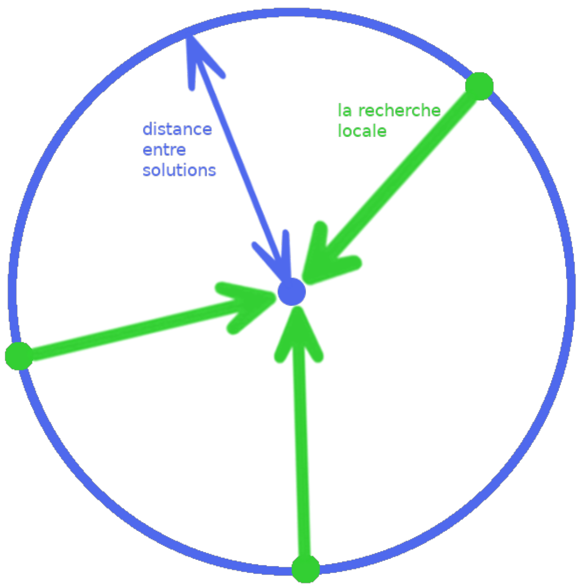


FIGURE 5.2 – Illustration de BSO avec un nombre de recherche locale égale à la distance entre les solutions

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.

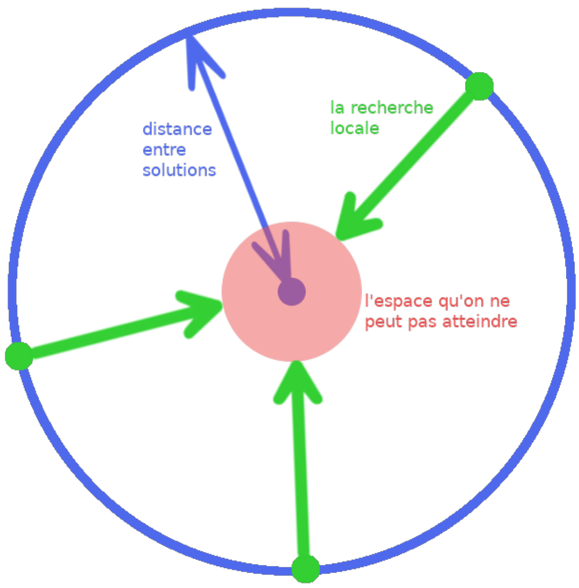


FIGURE 5.3 – Illustration de BSO avec un nombre de recherche locale inférieur à la distance entre les solutions

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

6.2 Résultats

6.2.1 Pour les instances satisfiables

6.2.2 Pour les instances non satisfiables

6.3 Comparaison avec les méthodes de I

Table des figures

Liste des tableaux

Annexe A

Code source

```java
public Node search(Node start, int timeLimit) {
    LinkedList<Node> successors;
    Node current;
    open.add(start);
    while (!open.isEmpty())
    {
        current = open.getNext();
        if(evaluator.isGoal(current))
        {
            return current;
        }
        if(current.getDepth() == maxDepth) continue;
        successors = current.getSuccessors();
        for (Node successor : successors)
//              if(!closed.contains(successor))
            {
                successor.setParent(current);
                successor.setDepth(current.getDepth()+1);
                if(evaluator instanceof HeuristicEvaluator)
                    ((HeuristicEvaluator) evaluator).evaluateF(successor);
                open.add(successor);
            }
//          closed.add(current);
    }
    return null;
}
```

FIGURE A.1 – Algorithme de recherche principal

```java
public class BreadthStorage extends Storage
{
    LinkedList<Node> list = new LinkedList<>();
    @Override
    public void add(Node node) { list.addFirst(node); }

    @Override
    public boolean isEmpty() { return list.isEmpty(); }

    @Override
    public Node getNext() {
        return list.removeLast();
    }
}
```

FIGURE A.2 – Gestion de open en largeur d'abord

```java
public class DepthStorage extends Storage
{
    ArrayList<Node> list = new ArrayList<>();
    @Override
    public void add(Node node) { list.add(node); }

    @Override
    public boolean isEmpty() { return list.isEmpty(); }

    @Override
    public Node getNext() { return list.remove( i: list.size()-1); }
}
```

FIGURE A.3 – Gestion de open par profondeur d'abord

```java
@Override
public void add(Node node)
{
    if(currentSize == heap.length) {
        doubleSize();
    }
    heap[currentSize++] = node;

    for (int i = currentSize-1; i != 0 && compare(heap[i],heap[parent(i)]) < 0; i=parent(i))
        swap(i,parent(i));

}

@Override
public boolean isEmpty() { return currentSize == 0; }

@Override
public Node getNext() {
    if(currentSize == 0)
    return null;
    if (currentSize == 1)
        return heap[--currentSize];
    Node node = heap[0];
    heap[0] = heap[--currentSize];
    heapify( i: 0);
    return node;
}
```

FIGURE A.4 – Gestion de open en tant que tas

```java
public static SATEvaluator loadClausesFromDimacs(String pathToCnfFile) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(pathToCnfFile));
    String line = reader.readLine();
    String first[] = line.split( s: "\\s+");
    SATEvaluator se = new SATEvaluator(Integer.parseInt(first[2]),Integer.parseInt(first[3]));
    se.clauses = new int[se.getNumberOfClauses()][se.getNumberOfVariables()];
    se.variablesBitSet = new BitSet[2][se.getNumberOfVariables()];
    for (int i = 0; i < se.getNumberOfVariables(); i++) {
        se.variablesBitSet[0][i] = new BitSet(se.getNumberOfClauses());
        se.variablesBitSet[1][i] = new BitSet(se.getNumberOfClauses());
    }
    int i = 0;
    while ((line = reader.readLine()) != null && i < se.getNumberOfClauses()) {
        if(line.length()>0 && line.charAt(0) == ' ')
            line = line.substring(1);
        String sLine[] = line.split( s: "\\s+");
        for (int j = 0; j < sLine.length - 1; j++) {
            int i1 = Integer.parseInt(sLine[j]);
            se.clauses[i][Math.abs(i1)-1]
                    = i1 / Math.abs(i1);
            int index = (i1 > 0)?1:0;
            se.variablesBitSet[index][Math.abs(i1)-1].set(i);
        }
        i++;
    }
    return se;
}
```

FIGURE A.5 – Chargement de l’instance depuis le fichier benchmark

```java
public class SATNode extends Node
{
    private int value;
    private int numberOfClausesSatisfied;
    private BitSet bitSet;
    private int index;


    public SATNode(Node parent) {
        super(parent);
        bitSet = new BitSet();

    }

    @Override
    public LinkedList<Node> getSuccessors() {
        LinkedList<Node> successors = new LinkedList<>();
        SATNode n1 = new SATNode( parent: this);
        SATNode n2 = new SATNode( parent: this);
        n1.setValue(1);
        n2.setValue(-1);
        successors.add(n1);
        successors.add(n2);
        return successors;
    }
```

FIGURE A.6 – Structure d'un noeud dans l'arborescence