



Rapport de TP

Module : Swarm Intelligence

Master 1 SII

TP

**Résolution du problème de satisfiabilité
avec des méthodes constructives
(Approche par espace des états)**

- Réalisé par :

BENHADDAD Wissam

BOURAHLA Yasser

23-02-2018

Table des matières

1	Introduction :	2
1.1	Problématique :	2
1.2	Définitions	2
1.2.1	Problème SAT	2
1.2.2	Stratégie de recherche dans l'espace des états	3
1.2.3	Stratégie de recherche aveugle	3
1.2.4	Stratégie de recherche guidée	4
2	Implémentation	5
2.1	Structures de données	5
2.1.1	Représentation du problème SAT	5
2.1.2	Représentation des états	6
2.1.3	Développement des états	7
2.2	Conception et pseudo-code	8
2.2.1	Gestion de la liste open	9
3	Expérimentations	11
3.1	Donnés	11
3.1.1	Format DIMACS	11
3.1.2	Exemple	12
3.1.3	Type d'instances	12
3.2	Environement de travail	13
3.2.1	Machines	13
3.2.2	Outils utilisés	14
3.3	Résultats	14
3.3.1	En largeur d'abord :	14
3.4	Statistiques	16
3.5	Comparaison entres les quatres méthodes	16
4	Conclusion	17

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 satisfiabilité (**SAT**), est un problème de décision ou il s'agit d'assigner des valeurs de vérité à des littéraux¹ tel qu'un ensemble de clauses en forme normale conjonctive FNC² préalablement défini soit satisfiable, en d'autres termes, que toutes les clauses soient vraies pour les mêmes valeurs de vérité de leurs littéraux, ce problème est le premier à avoir été démontré comme étant **NP-Complet**, et cela par **Stephen Cook** dans [1], et qui a donc posé les fondements de l'informatique théoriques et de la théorie de la complexité.

1. Une variable logique ou bien sa négation

2. Une conjonction de disjonction de littéraux

1.2.2 Stratégie de recherche dans l'espace des états

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

1.2.3 Stratégie de recherche aveugle

Cette catégories englobe les stratégies où il est question de passer par toutes les solutions et les tester une à une, dans ce TP nous nous intéresserons plus particulièrement aux algorithmes/méthodes suivants :

Par profondeur d'abord (DFS)

L'algorithme de parcours en profondeur d'abord consiste à visiter un noeud de départ (souvent appelé **racine**), puis ensuite visiter 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 ensembles **Open** et **Closed** qui représentent respectivement l'ensemble des noeuds du graphe qui n'ont pas encore été développés et ceux déjà développés, cet ajout permet à l'algorithme d'éviter de boucler indéfiniment sur un ensemble de noeuds.

En Largeur d'abord (BFS)

Cet algorithme diffère de son prédécesseur 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 précédemment visités durant la recherche, ce qui peut causer un débordement de la mémoire lors de l'exécution sur machine (Ce point sera rediscuté dans [3.3](#))

Par coût uniforme

Le principe est simple, au fur et à mesure que l'algorithme avance et développe des noeuds, il garde en mémoire le coût¹, 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 noeud, l'algorithme est alors équivalent à celui de la recherche en largeur d'abord

1. Fonction retournant le coût pour passer du noeud de départ (la racine) au noeud courant

1.2.4 Stratégie de recherche guidée

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

Recherche gloutonne (Greedy algorithm)

Cet algorithme est basé sur la notion d'heuristique¹, au lieu de parcourir de façon "naïve" l'ensemble des nœuds dans l'espace de recherche, il choisit à chaque itération sur l'ensemble **open** le nœud 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 coût d'un nœud n
- h est la fonction qui estime le coût d'un nœud n vers le but

Le principe de l'algorithme est donc de prendre le nœud dans **open** qui possède la valeur minimale de f , assurant ainsi de trouver le chemin optimal **ssi**. l'heuristique h choisie est admissible²

1. Une fonction d'estimation de la distance séparant le nœud courant au but
2. Ne surestime jamais le coût réel pour atteindre le but, elle est **optimiste**

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 $i^{\text{ème}}$ 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 :

$$\begin{aligned}x_1 \vee \neg x_2 \vee x_5 \\ \neg x_2 \vee x_4 \vee x_5 \\ \neg x_1 \vee x_2 \vee \neg x_3\end{aligned}$$

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 $i^{\text{ème}}$ 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.

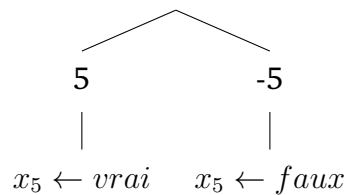
Pour le même exemple vu précédemment on obtient les *Bitsets* suivants :

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

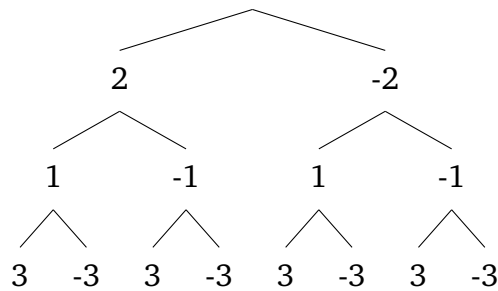
2.1.2 Représentation des états

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.

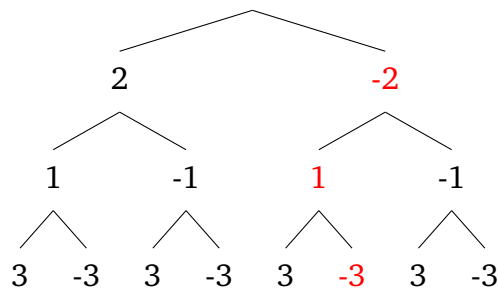


2.1.3 Développement des états

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 : $x_1 = \text{vrai}$, $x_2 = \text{faux}$, $x_3 = \text{faux}$ 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.

<i>Bitset</i> du parent	1	0	1	1
-------------------------	---	---	---	---

OR

<i>Bitset</i> du littéral	1	0	0	1
---------------------------	---	---	---	---

↓

<i>Bitset</i> 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 différentes méthodes est présenté ci-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     insérer 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 ci-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

Profondeur d'abord

La recherche en profondeur d'abord consiste à choisir le noeud avec la profondeur la plus élevée de l'arbre, ceci revient 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 :



Sélection d'un noeud :



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 :



Sélection d'un noeud :



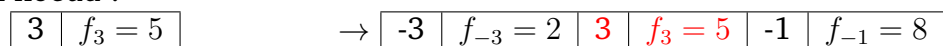
Recherche En se basant sur une fonction d'évaluation

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 :



Sélection d'un noeud :



Complexité de l'insertion : $o(n)$.

Complexité de la sélection : $o(1)$.

Tas Les noeuds sont organisés dans une structure de tas¹. La racine du tas est sélectionnée pour l'évaluation, tandis que l'insertion se fait par entassement du nouveau élément. Les deux opérations se font en $O(\log(n))$.

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

Chapitre 3

Expérimentations

3.1 Données

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 sur [2], et sont 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 littéral au total dans l'instance, à noter que chaque littéral x_i sera représenté par son indice i .
 3. **nbclause** le nombre total de clause présentes dans l'instance.
- chaque ligne représente une conjonction de littéraux $(x_i | \neg x_i)$ indentifiés par un numéro i , séparés par un blanc, et le 0 à la fin dénote la fin d'une ligne.

1. Représentation conventionnelle 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 [2] 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, avec XX = nombre de littéraux et YY = nombre de clauses

3.2 Environnement de travail

3.2.1 Machines

Pour les tests nous avons utilisé deux machines pour chaque groupes d'instances, autrement dit une machine pour effectuer les tests sur un ensemble d'instances satisfiables [UF75-325\[2\]](#) et une autre sur les instances contradictoires (non satisfiables) [UUF75-325\[2\]](#), les caractéristiques de chaque machine sont données dans les figures [3.1](#) et [3.2](#) suivantes :



FIGURE 3.1 – Machine A pour les instances contradictoires

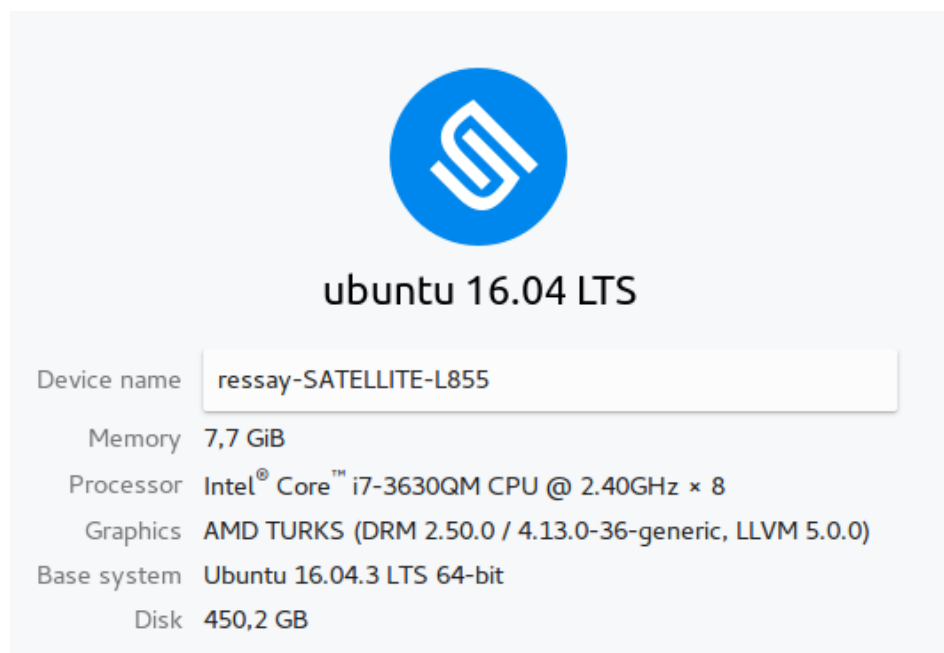


FIGURE 3.2 – Machine B pour les instances satisfiables

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 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** il est caractérisé par sa forte simplicité d'utilisation et les nombreuses plugins et extensions qui lui sont dédiées.

3.3 Résultats

Pour chacun des groupes d'instances(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 :

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 :

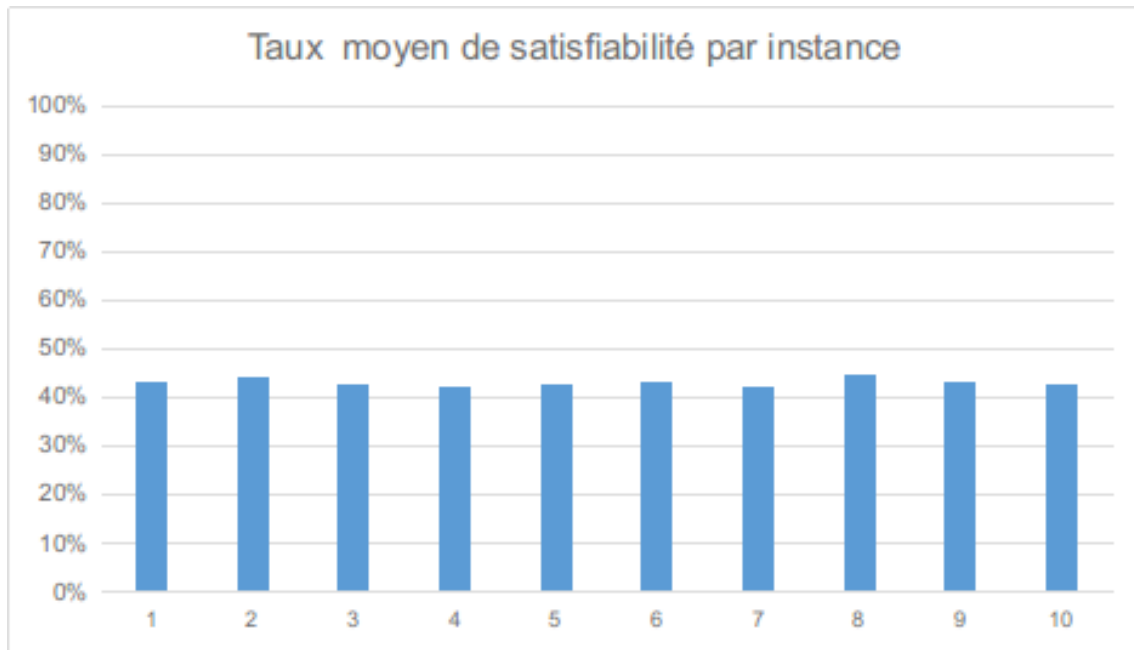


FIGURE 3.3 – Illustration des données de ??

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

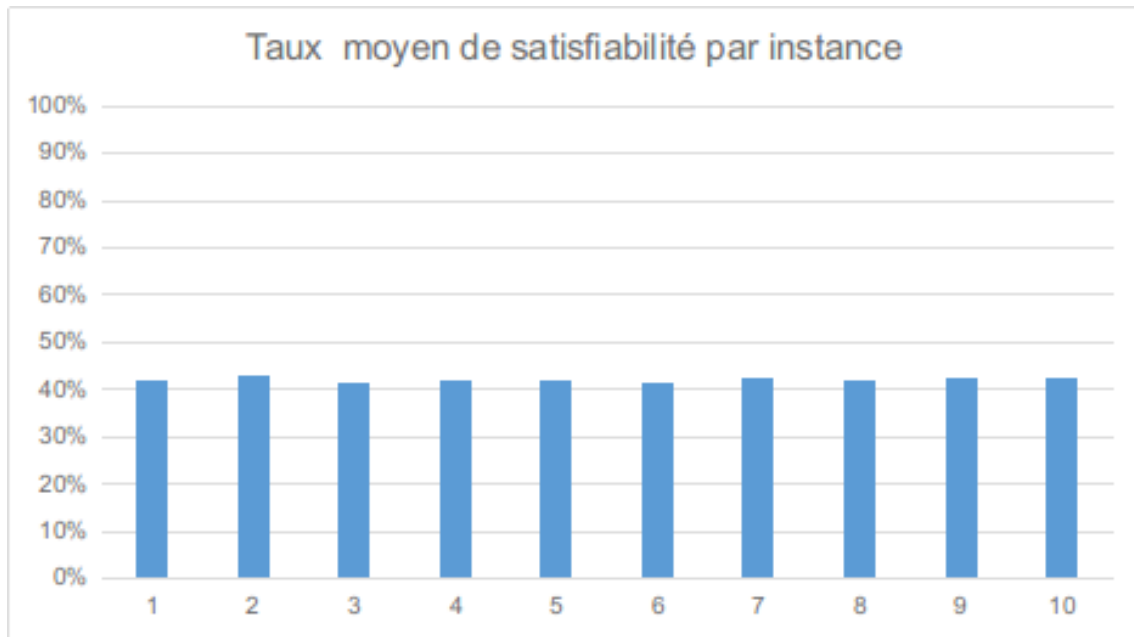


FIGURE 3.4 – Illustration des données de ??

3.4 Statistiques

3.5 Comparaison entres les quatres méthodes

Chapitre 4

Conclusion

Table des figures

3.1	Machine A pour les instances contradictoires	13
3.2	Machine B pour les instances satisfiables	13
3.3	Illustration des données de ??	15
3.4	Illustration des données de ??	16

Bibliographie

- [1] Stephen A. Cook. *The Complexity of Theorem-proving Procedures*. 1971.
- [2] **SATLIB** - benchmark for **SAT** problems. <http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html>.