



# 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

<b>1</b>	<b>Introduction :</b>	<b>2</b>
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
<b>2</b>	<b>Implémentation</b>	<b>5</b>
2.1	Structures de données	5
2.1.1	Représentation du problème SAT	5
2.2	Conception et pseudo-code	6
2.2.1	Profondeur d'abord	6
2.2.2	Largeur d'abord	6
2.2.3	Recherche gloutonne	6
2.2.4	Algorithme A*	6
<b>3</b>	<b>Expérimentations</b>	<b>7</b>
3.1	Donnés	7
3.1.1	Format DIMACS	7
3.1.2	Exemple	8
3.1.3	Type d'instances	8
3.2	Environement de travail	9
3.2.1	Machines	9
3.2.2	Outils utilisés	10
3.3	Résultats	10
3.3.1	En l'argur d'abord :	10
3.4	Statistiques	12
3.5	Comparaison entres les quatres méthodes	12
<b>4</b>	<b>Conclusion</b>	<b>13</b>

# 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 littéraux<sup>1</sup> tel qu'un ensemble de clauses en forme normale conjonctive FNC<sup>2</sup> 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 coût<sup>1</sup>, 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'heuristique<sup>1</sup>, 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 admissible<sup>2</sup>

---

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 :

$$x_1 \vee \neg x_2 \vee x_5$$

$$\neg x_2 \vee x_4 \vee x_5$$

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

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 pour 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 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

## 2.2 Conception et pseudo-code

### 2.2.1 Profondeur d'abord

### 2.2.2 Largeur d'abord

### 2.2.3 Recherche gloutonne

### 2.2.4 Algorithme A\*

# 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**<sup>1</sup> (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 numero  $i$ , 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

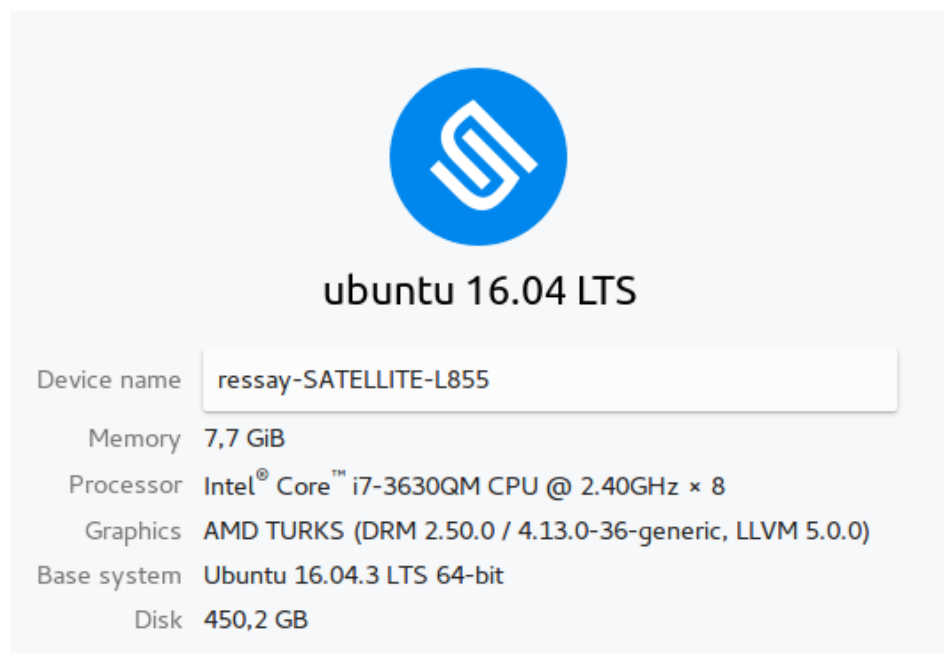


FIGURE 3.2 – Machine B pour les instances satisfiables

### 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 :

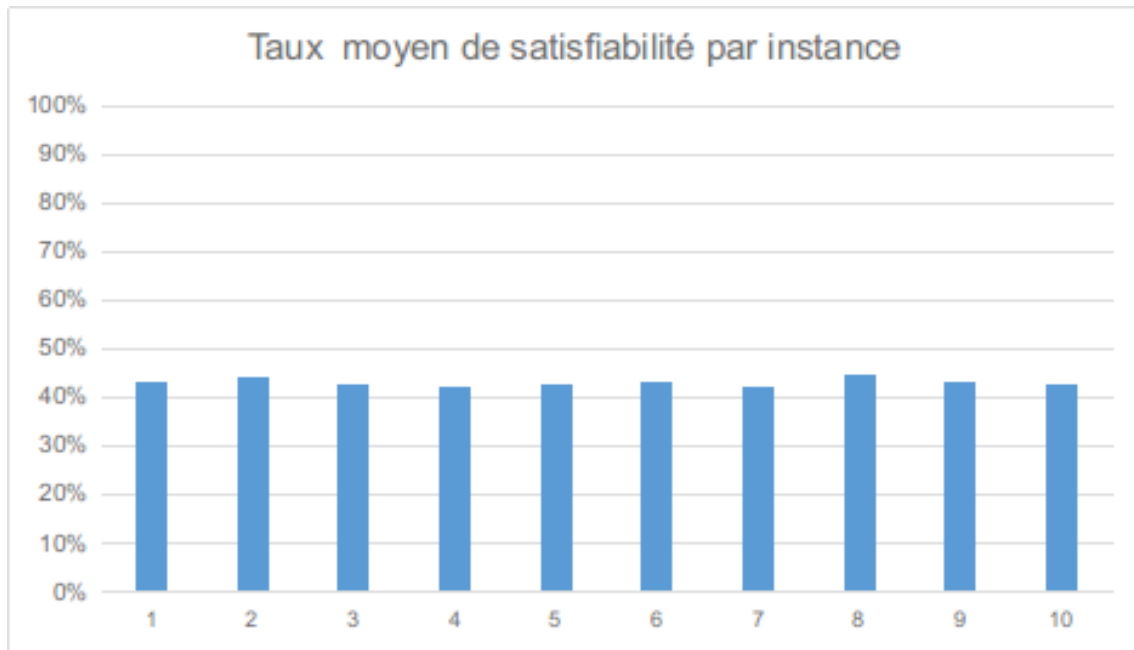


FIGURE 3.3 – Illustration des données de ??

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

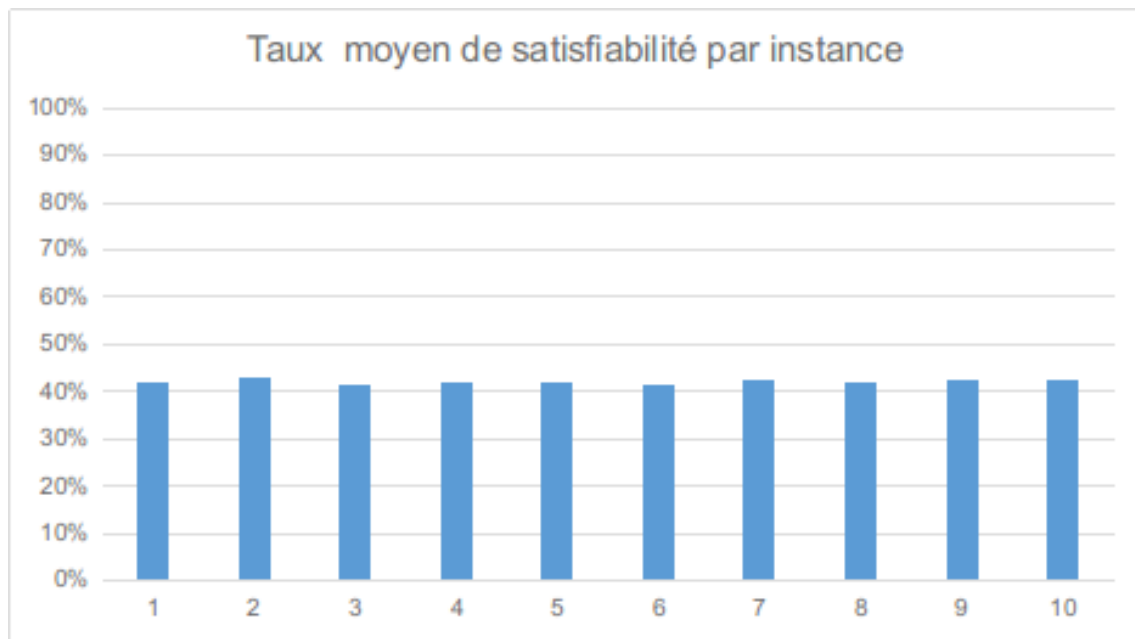


FIGURE 3.4 – Illustration des données de ??

### 3.4 Statistiques

### 3.5 Comparaison entres les quatres méthodes

## **Chapitre 4**

## **Conclusion**

# Table des figures

3.1	Machine <b>A</b> pour les instances contradictoires . . . . .	9
3.2	Machine <b>B</b> pour les instances satisfiables . . . . .	9
3.3	Illustration des données de ?? . . . . .	11
3.4	Illustration des données de ?? . . . . .	12

# 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>.