Université des Sciences et de la Technologie Houari Boumediene


*Master 1*
*Systèmes Informatiques Intelligents*

# *Méta-Heuristiques et Algorithmes évolutionnaires*

## Rapport du projet N°1

# *Intelligence en essaim*

*Groupe 2*

MEGHNI Mohamed El Amine
MENOUER Abdallah

# Introduction générale

En informatique théorique, le problème de satisfiabilité (en abrégé **SAT**), est un problème de décision, qui détermine, à partir d'une formule booléenne donnée, s'il existe un assignement de valeur (**Vrai | Faux)** aux littéraux de la formule, qui évalue la formule à **Vrai**.

Le problème de satisfiabilité est le premier qui fut démontré **NP-Complet** en 1971 par *Stephan Cook et Leonid Levin*, c'est-à-dire tous les problèmes de classe **NP,** notamment des problèmes de décision et d'optimisation, sont difficiles de résoudre que **SAT,** et ces problèmes peuvent être convertis par une transformation en un problème **SAT**. Aucun algorithme est connu qui résout de manière polynomiale, le problème de satisfiabilité, il est généralement admit, qu'un tel algorithme n'existe pas, malgré que cette affirmation n'a pas été prouvée mathématiquement, la question de savoir que si **SAT** est polynomial, est équivalente à la question de savoir si la classe **P = NP.**

De nos jours, plusieurs algorithmes existent afin résoudre le problème de satisfiabilité en suivant plusieurs méthodes et principes. Dans cette partie du projet, nous allons nous concentrer sur les méthodes de recherche aveugles (recherche en profondeur, largeur), et les méthodes de recherche guidée (heuristiques).

# Chapitre 1

# Les méthodes Aveugles

## 1.1 Introduction

Les méthodes aveugles sont des méthodes exhaustives, visant à découvrir le chemin qui mène à la solution, sans aucune information sur le chemin qui semble le plus court, elle exploite, s'il le faut, tous les chemins possibles, si cette solution existe.
Parmi ces méthodes, on trouve la recherche en **profondeur,** et la recherche en **largeur**.

## 1.2 Recherche en profondeur

Dans ce type de recherche, les nœuds de l'ensemble sont triés en ordre décroissant suivant leur profondeur dans l'arbre de recherche, c'est-à-dire, les plus profonds sont classés en premier. Les nœuds de même profondeur sont classés arbitrairement sans être informé du bon chemin à prendre en premier.
La recherche en profondeur exploite l'arbre de recherche branche par branche jusqu'à arriver à la solution (si elle existe), dans le cas échéant, elle continue à parcourir parcourt tout l'arbre de recherche jusqu'au dernier puis elle renvoie nil.

## 1.3 Recherche en largeur

La recherche en largeur est plutôt similaire à la recherche en profondeur, sauf que, les nœuds de la liste Ouvert sont triés par ordre croissant de leur profondeur dans l'arbre.
La recherche en largeur exploite l'arbre de recherche niveau par niveau jusqu'à arriver au dernier niveau, qui est le niveau des solutions (feuilles).

## 1.4 Algorithme de recherche en profondeur

Algorithme Profondeur

```
Entrée : Clauses à évaluer
Sortie : solution si existe, nil sinon
Var :
Open : liste vide de noeuds;
state : noeud;
Début

        Initialiser state à l'état initial de l'arbre;

        Ajouter state à Open;
        Tant que(nonVide(Open)) Faire

                        ///Take first element of Open
                        state = Premier element de Open;

                        ///Remove it from Open
                        Retirer state de Open;

                        Si(Isleaf(state)) alors
                                Si(IsSolution(state)) alors
                                        Retourner state;
                                fSi;
                        Sinon
                                ///Develop the current state if not a leaf
                                Sons = Fils de state ;
                                Ajouter les éléments de Sons au débud'Open
                        fSi;
        Fait;
        Retourner nil;
Fin
```

## 1.4.1 Complexité de recherche en profondeur

L'algorithme précédent vise à explorer, s'il le faut, l'arbre de recherche en entier afin de trouver une solution au problème. Les solutions possibles se situent dans le dernier niveau de l'arbre, une solution au problème est donc une feuille de l'arbre de recherche.

Une solution au problème peut se présenter comme la première feuille découverte par le programme dans le meilleur cas, ou la dernière dans le mauvais cas (explorer l'arbre en entier).

Dans le meilleur cas, l'algorithme parcours une seule branche de l'arbre, ce qui fait une complexité de O(n) / n : nombre de littéraux constituant une solution

Dans le pire des cas, l'algorithme parcours tous les nœuds de l'arbre, ce qui fait une complexité de O ($2^n$)

# 1.5 Algorithme de recherche en profondeur

Algorithme Largeur

```
Entrée : Clauses à évaluer
Sortie : solution si existe, nil sinon
Var :
Open : liste vide de noeuds;
state : noeud;
Début
        Initialiser state à l'état initial de l'arbre;
        Ajouter state à Open;
        Tant que(nonVide(Open)) Faire
                        ///Take first element of Open
                        state = Premier element de Open;
                        ///Remove it from Open
                        Retirer state de Open;
                        ///Develop the current state if not a leaf
                        Si(Isleaf(state)) alors
                                Si(IsSolution(state)) alors
                                        Retourner state;
                                fSi;
                        Sinon
                                Sons = Fils de state ;
                                Ajouter les éléments de Sons à la fin d'Open
                        fSi;
         Fait;
       Retourner nil; Fin
```

## 1.5.1 Complexité de recherche en largeur

L'algorithme précédent vise à explorer au minimum, tous les nœuds non feuilles de l'arbre de recherche (n niveaux / n : nombre de littéraux constituant une solution). Une solution au problème peut se présenter comme la première feuille découverte par le programme dans le meilleur cas, ou la dernière dans le mauvais cas (explorer l'arbre en entier).
Dans le meilleur cas, l'algorithme parcours tous les nœuds non feuilles et la feuille solution, ce qui fait une complexité de $O(2^n-1)$ / n : nombre de littéraux constituant une solution
Dans le pire des cas, l'algorithme parcours tous les nœuds de l'arbre, ce qui fait une complexité de $O(2^n)$

# Chapitre 2

# Résolution par méthode aveugle

## 2.1 Introduction

Dans ce chapitre, nous allons voir les différentes structures de données utilisées pour implémenter un solveur SAT en exploitant les méthodes aveugles vues dans le chapitre précédent.

## 2.2 Représentation des formules booléennes

Dans le problème de satisfiabilité, la formule booléenne est une conjonction de disjonction de littéraux.

Exemple :

Soit l'ensemble de variables {a,b,c} et la formule
F = (a ∨ b) ∧ ( c ∨ b ).
(a ∨ b) ∧ ( c ∨ b ) sont deux clauses avec deux littéraux par clause. Leur conjonction F est une formule normale conjonctive.

## 2.3 Représentation matricielle

Une représentation possible des formules conjonctive sur machine est une matrice d'entier **M** de dimension **n x m** tel que **n** représente le nombre de clauses et **m** représente le cardinal de l'ensemble des variables.
Tout d'abords, toutes les cases sont initialisées a « -1 ». Chaque clause est représentée par un vecteur, chaque littéral présent est représentée par « 1 » respectivement « 0 » si sa valeur est « Vrai » respectivement « Faux » dans sa case correspondante (l'index de la colonne).

Reprenons l'exemple précédent :
Ensemble de variables = {a (1), b (2), c (3)}
F = (a ∨ b) ∧ ( c ∨ non b ).
Soit la matrice M représentant cette formule en mémoire central :

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

## 2.3 Représentation des états

Une solution à une instance SAT, est un assignement de valeur de vérité aux variables de la formule conjonctive. Chaque variable prend soit la valeur vrai, faux sinon.

**Exemple :**

Ensemble de variables $\gamma$ = {a (1), b (2), c (3)}
Soit la formule conjonctive F = (a ∨ b) ∧ ( non c ∨ non b ).
Soit l'assignement $\Gamma$ = { a = Vrai ; b = Vrai ; c = Faux }
L'assignement $\Gamma$ satisfait F.

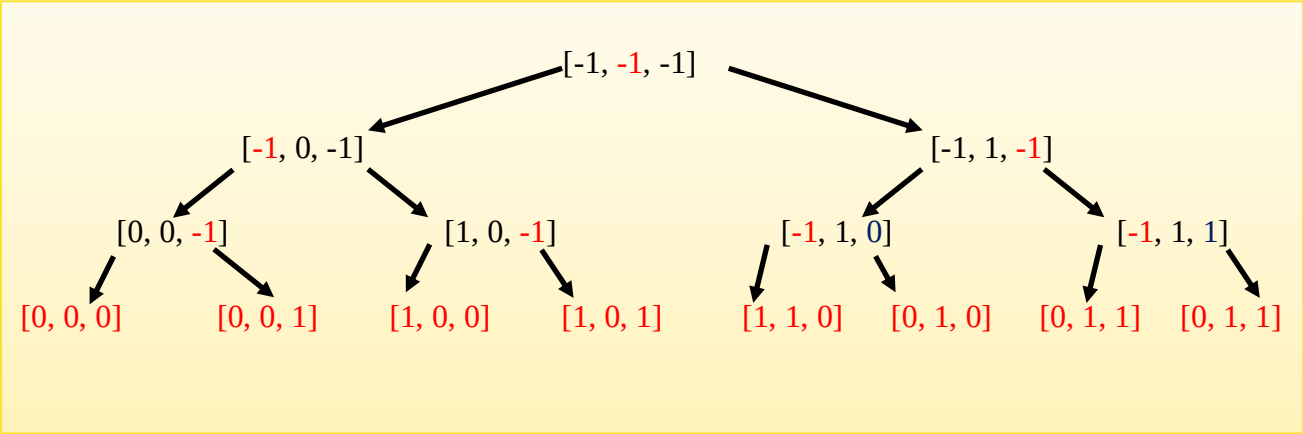
En machine, l'assignement $\Gamma$ sera représenté par un vecteur de 3 colonnes tel que :
V = [1, 1, 0]
V[i] = 1 désigne que la variable à l'index i de l'ensemble $\gamma$ a l'assignement **Vrai.**
V[i] = 0 désigne que la variable à l'index i de l'ensemble $\gamma$ a l'assignement **Faux.**

## 2.3 Règles de transition

A partir de chaque état **i** dans l'arbre, on peut transiter vers deux autre états successeurs, en choisissant aléatoirement un littéral non évalué et lui assigner la valeur **Vrai** dans le premier fils, et la valeur **Faux** dans le second. Si tous les littéraux de l'état **i** sont évalués, alors l'état courant est une feuille.

## 2.4 Arbre de recherche

Soit l'ensemble de variables $\gamma$ = {a (1), b (2), c (3)}
Soit la formule conjonctive F = (a ∨ b) ∧ ( non c ∨ non b ).
Soit $\alpha$ l'état initial : $\alpha$ = [-1, -1, -1]
L'arbre de recherche d'un assignement aux variables de $\gamma$ qui satisfait la formule F en commençant par la racine $\alpha$ est :




Soit $\Omega$ = {[0, 0, 0],[0, 0, 1],[1, 0, 0],[1, 0, 1],[1, 1, 0],[0, 1, 0],[0, 1, 1],[0, 1, 1]}l'espace des solutions possibles à satisfaire F obtenu par le biais d'une méthode aveugle.

Au final, l'assignement [1, 1, 0] satisfait la formule F, tel que a = **Vrai, b = Vrai, c = Faux**

# Chapitre 3

# Les méthodes guidées

## 3.1 Introduction

Les méthodes aveugles, visant à découvrir le chemin qui mène à la solution, ne sont pas utilisées en pratique car elles développent trop de nœuds avant d'arriver au but, il faut trouver des alternatives plus efficaces que la recherche aveugle.

## 3.2 Recherche guidée

La recherche guidée ou **heuristique,** largement utilisée dans des problèmes qui sont caractérisés par une explosion combinatoire d'états, utilise un ensemble d'information qui permet d'évaluer la probabilité qu'un chemin allant du nœud courant au nœud cible soit la meilleure que les autres. Ces heuristiques permettent de réduire considérablement l'effort de recherche mais ne garantissent pas que le chemin choisit soit le chemin de moindre coût.

## 3.3 Algorithme A*

L'algorithme A* (A star), est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un état but, en se basant sur une heuristique, cet algorithme permet de générer une solution (chemin) parmi les meilleurs chemins possibles.
L'heuristique utilisée par A* est la fonction f définie par :

$$\mathcal{F}(n) = \mathcal{O}(n) + \hbar(n)$$

Où n est un nœud représentant l'état du système, $\mathcal{O}(n)$ est le coût de la chaine allant de l'état initial s à n, et $\hbar(n)$ appelée heuristique est une estimation de coût de la chaine reliant n avec l'état final.
L'heuristique est spécifique au domaine d'application et nécessite une bonne maîtrise du problème à résoudre.

## 3.4 Heuristique de satisfiabilité

Pour notre problème de satisfiabilité, on utilisera comme heuristique, le nombre de clause satisfaite par chaque état du système.

Le point de départ de la recherche sera un état généré de manière aléatoire, nous choisirons le nœud qui satisfait le plus grand nombre des clauses, et nous le développerons jusqu'à aboutir à un état qui satisfait toutes les clauses (solution positive).

## 3.5 Algorithme A*

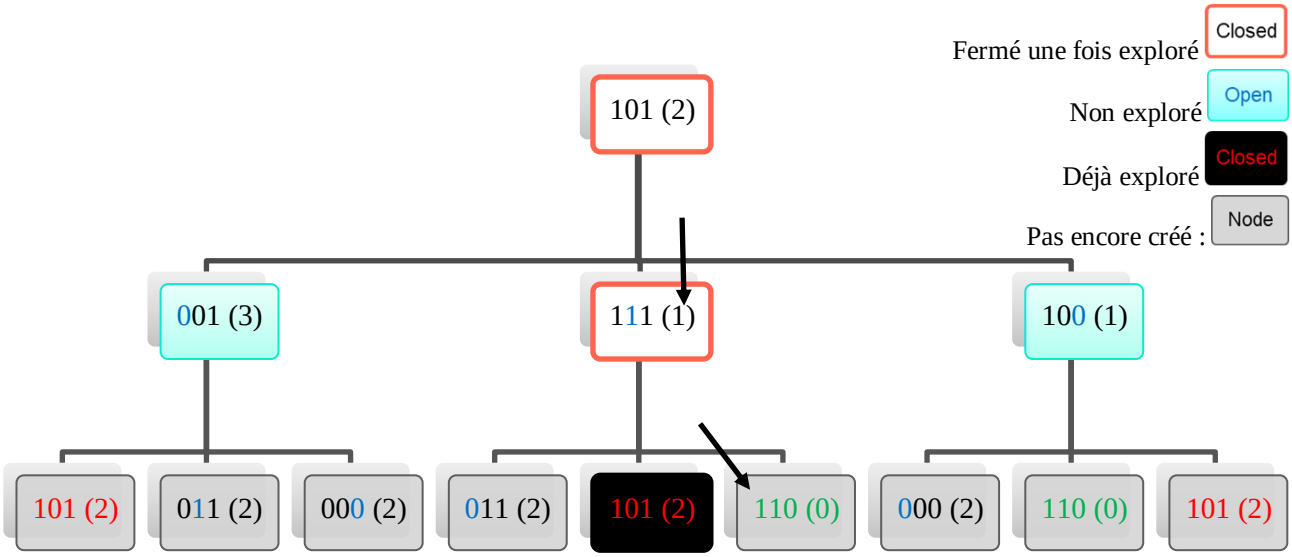
Algorithme A star

```
Entrée : Clauses à évaluer
Sortie : solution si existe, nil sinon
Var :
Open,Close : liste vide de noeuds;
state : noeud;
Début

        State : = Random_Value() ;
        Ajouter state à Open;
        Tant que(nonVide(Open)) Faire

                        ///Take first element of Open
                        state := Premier element de Open;

                        ///Remove it from Open
                        Retirer state de Open;

                        //Add it to clause
                        Ajouter state à Close;
                        Si(IsSolution(state)) alors
                                Retourner state;
                        fSi;
                        Sinon
                          Sons = Fils de state ;
                           Trier la liste Sons en fonction de l'heuristique ;
                           Ajouter les éléments de Sons qui n'appartient pas à Close au débud'Open ;
                        fSi;
         Fait;
       Retourner nil; Fin
```

### 3.5.1 Complexité A*

L'algorithme A*, en différence avec les algorithmes cités précédemment (profondeur et largeur), ne traite que les nœuds dont les littéraux sont tous défini.
Comme le premier nœud (état initial) est généré aléatoirement, le meilleur cas est l'état initial est une solution au problème 0(1), dans le mauvais (pire) cas, le problème n'est pas satisfaisable.
La complexité est calculée : $1 + n + n^2 + \ldots. + n^m = \sum_{i=0}^{m} n^i = (1 - n^{m+1}) / (1 - n)$
/ m : nombre de niveaux et n nombre de littéraux.

## 3.6 Arbre de recherche A*



# Chapitre 4

# Les algorithmes génétiques

## 4.1 Définition

Les algorithmes génétiques, initiés par John Holland, sont des algorithmes d'optimisation, se basant sur le mécanisme de l'évolution de la nature, plus précisément, en génétique (croisement, mutation et sélection)

## 4.2 Fonctionnement des Algorithmes génétiques

Les algorithmes génétiques sont utilisés dans des problèmes d'optimisation, c'est-à-dire, la solution optimal d'un problème n'est pas calculable de façon algorithmique ou analytique.

Selon cette méthode, des milliers de solutions (gènes), sont créées au hasard, puis sont soumise à une évaluation de pertinence de la solution. Les espèces les plus adaptées (solution qui se tendent vers la plus optimal), survivent que celles qui le sont moins adaptées, la population évolue donc par générations successives en croisant les meilleures solutions, ou de muter certaines espèces de la population afin de diversifier les nouvelles générations à venir.

## 4.2 La sélection

Les solutions les plus enclins à produire de meilleures solutions sont sélectionnées pour donner vies à la nouvelle génération, tandis que les solutions les moins efficaces meurent, ce qui améliore globalement l'adaptation. Plusieurs mécanismes de sélection existent, nous citerons :

**Sélection à loterie :** similaire au jeu de loterie, chaque solution a une chance d'être sélectionnée proportionnelle avec sa valeur de fitness, les solutions les plus performantes auront une grande chance d'être sélectionnées.

**Sélection élitiste :** les solutions les plus performantes seront sélectionnées.

**Sélection à tournoi :** dans cette sélection, deux solution seront choisi de manière aléatoire, on sélectionne parmi ces deux solutions celle la plus performante.

## 4.3 Le croisement

Dans cette opération, deux solutions s'échangent des parties de leurs informations afin de générer de nouvelles solutions, deux parents sont alors recombinés de façon à former deux fils possédant des caractéristiques issues des deux parents.

### 4.3.1 Le croisement en 1 point

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

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

### 4.3.2 Le croisement en 2 point

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

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

### 4.3.3 Le croisement uniforme

Le croisement uniforme consiste à utiliser un **masque** pour chaque fils, d'où on précisera par bit que'elle valeur de parent, le fils héritera.

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

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Masque fils haut |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | Masque fils bas |

1 : bit du père en rouge.

0 : bit du père en bleu.