# Algorithmique - Algorithme évolutionnaire pour la résolution du problème 3-SAT

Luc Libralesso

March 31, 2016

**Abstract:** *A l'heure actuelle, le problème de satisfaction de formules booléennes (SAT) est* NP-Complet. *Cela signifie qu'il n'existe pas d'algorithme exact permettant de le résoudre en un temps polynomial. Cependant, ce problème est présent dans de nombreuses applications ce qui en fait un enjeu de taille. Une autre approche à ce problème est de faire appel à des méthodes approchées qui donnent des résultats satisfaisants en un temps acceptable. Ce document présente une approche basée sur des algorithmes évolutionnaires pour résoudre le problème SAT.*

**Introduction:** Ce document s'intéresse à résoudre le problème *SAT*[1] avec l'aide d'algorithmes évolutionnaires, particulièrement plébicités ces dernières années.

Il aborde dans une première partie le format DIMACS utilisé pour représenter les instances. Dans une deuxième partie le design de l'algorithme évolutionnaire utilisé, les résultats de l'implémentation de cet algorithme sur un benchmark et enfin des pistes d'amélioration de cet algorithme en vue d'obtenir encore de meilleurs résultats.

## 1 Format DIMACS

Ci dessous un exemple de fichier au format DIMACS :

```
c
c start with comments
c
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

- Les premières lignes correspondent à des commentaires, elles ne sont pas utiles pour le solveur.
- `p cnf 5 3` signifie que l'instance sera de la forme CNF[2]. Cette ligne informe également que seront disponibles 5 variables dans ce problèmes et 3 contraintes.
- Les lignes suivantes représentent les différents contraintes (de trois variables comme nous sommes en 3-SAT). Ces lignes se terminent par 0 et les variables précédées par des '-' sont la négation de ces variables.

Dans l'exemple ci-dessus, une résolution possible serait :

- 1 : Vrai
- 2 : Vrai
- 3 : Vrai
- 4 : Faux
- 5 : Vrai

**Benchmarks Existants :** Un des benchmarks les plus connus pour SAT est *SATLIB*[3]. Il regroupe un grand nombre d'instances plus ou moins difficiles dont certaines qui représentent d'autres problèmes *NP-Complets* comme des problèmes de coloration de graphe ou encore des problèmes de planning.

## 2 Algorithme présenté

Tout au long de cette section, seront utilisées :

**n:** nombre de variables pour l'instance

**m:** nombre de contraintes pour l'instance

**I:** ensemble des individus de la population

L'algorithme présenté aura les paramètres suivants :

**Représentation des individus:** Chaque individu codera une solution à la satisfaction de formule logique. Chaque individu aura $n$ variables booléennes codées en lui. Une pour chaque variable dans le problème de départ.

**Fonction de fitness:** La fonction de fitness utilisée sera le nombre de contraintes qui sont violées. Il s'agit donc d'un problème de minimisation de la fonction objectif pour la faire tendre vers 0. En effet, si aucune contrainte n'est violée par une solution, on peut dire que cette solution répond au problème et nous pouvons arrêter l'algorithme.

[1] SAT est un problème NP-Complet. Il est particulièrement utilisé et dispose de son propre format d'instance standardisé : le format DIMACS. C'est pourquoi il est choisi dans ce document.

[2] Conjonctive Normal Form. Soit de la forme $(a \vee b \vee c) \wedge (d \vee e \vee f) \dots$

[3] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

**Nombre d'invididus:** Pour ce problème, nous choisirons 100 individus pour former notre population.

Notre algorithme évolutionnaire suivra la boucle de fonctionnement présente sur la figure 1.

- Une première phase d'initialisation de la population : Chaque individu est généré aléatoirement. Le fait de ne pas introduire de biais ici est important pour ne pas se retrouver piégé dans un minimum local. Les $n$ variables seront générées suivant une *Bernoulli* de paramètre $p = \frac{1}{2}$.
- L'évaluation et la sélection des solutions se fera par une fonction de fitness qui compte le nombre de contraintes violées. Dans un but de limiter la pression de sélection du système, nous utiliserons un algorithme de roulette. Chaque individu $i$ ayant pour fonction de fitness $f(i)$ aura $m - f(i)$ chances d'être sélectionné. Soit $p(i)$ la probabilité de cet individu a être sélectionné :

$$p(i) = \frac{m - f(i)}{\sum_{k \in I}(m - f(k))}$$

  A la fin de cette étape resteront 100 enfants.
- La phase de mutation va modifier chaque individu avec une probabilité de $\frac{1}{10}$. Si un individu est affecté par la mutation, chaque variable aura une probabilité $\frac{1}{2}$ d'être inversée dans son code génétique.
- la phase de reproduction doublera la taille de la population (il y aura 100 individus de générés). 100 couples seront sélectionnés avec remise. Un individu pourra être présent deux fois dans le couple et pourra être le parent de plusieurs enfants. À la fin de cette phase, existeront 200 enfants. L'opération de reproduction prend au hasard une variable parmi les parents. Comme pour la sélection, les parents les plus adaptés auront plus de chances d'être sélectionnés.

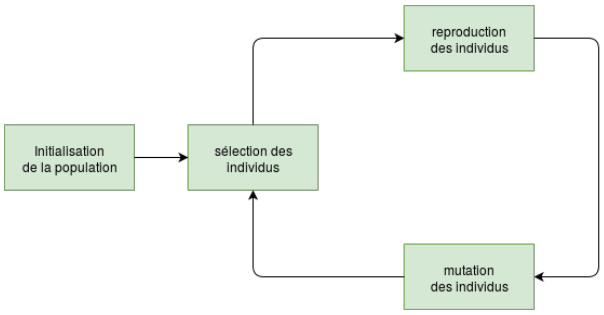


Figure 1: Boucle de fonctionnement de l'algorithme évolutionnaire présenté

## 2.1 Critères de design d'un algorithme évolutionnaire

Pour cette étude, nous prendrons trois instances générées aléatoirement présentes dans le benchmark de SATLIB[4]. A celles-ci, nous testerons nos différentes version de cet algorithme et comparerons les résultats au fur et à mesure des optimisations.

pour guider les optimisations, les principes pour la conception d'algorithmes évolutionnaires dans [**?**] seront utilisés.

Nous verrons aussi en quoi l'algorithme proposé répond à ces critères.

1. *Il doit être facile de générer une solution acceptable.* Dans notre cas, chaque solution générée aléatoirement est acceptable. Elle respectera plus ou moins bien les contraintes proposées.

2. *Pour chaque solution $s$, il est possible de trouver un chemin vers la solution optimale $s*$.* En effet, les opérations de mutation permettent de faire changer les bits d'une solution vers n'importe quelle autre. Il est donc bien possible de trouver un chemin depuis une solution quelconque vers une solution optimale.

3. *Les solutions dans le voisinage d'une solution $s$ doivent être 'proche' de $s$.* Cette ligne de conduite est respectée par l'opérateur de mutation ou de reproduction. En effet, l'opérateur de mutation fait varier aléatoirement des bits dans la solution avec une certaine probabilité. La nouvelle solution après mutation ressemblera très fortement à la solution initiale.

4. *La topologie induite par la fonction de coût ne doit pas être trop plate.* Ce point est respecté. En effet, l'espace de valeurs admissibles pour la fonction de coût varie de 0 au nombre de contraintes. Il pourrait être cependant intéressant d'essayer d'obtenir des fonctions de coût plus élaborées, notamment en introduisant une pénalité pour certaines solutions qui ont beaucoup de solutions lui ressemblant. Cela permettrait de favoriser la diversité des solutions et donc d'avoir plus de chances de toucher la solution optimale.

5. *L'information pertinente devrait être propagée durant la phase de coopération.* Dans le cadre d'algorithmes évolutionnaires, la phase de coopération est la phase de reproduction des individus.

6. *Un enfant issu de deux parents qui se ressemblent doit ressembler à ses parents.* Un point qu'il serait possible d'améliorer ici est que plutôt donner aléatoirement des valeurs des variables parmi les gènes des parents serait de fixer les gènes des parents dans l'enfant et d'appliquer une méthode affectant les valeurs restantes de manière à minimiser les conflits avec des contraintes.

[4] `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/RND3SAT/uf20-91.tar.gz`, les trois premières instances du package

7. *La diversité de la population devraît être préservée.* Ici, rien n'empêche plusieurs solutions d'être identiques et de cohabiter. une solution pour régler ce problème est de donner un malus aux solutions qui sont la réplique de solutions déjà existantes.

De ces critères, nous pouvons déduire deux améliorations possibles :

- Appliquer une méthode de *Hill Climbing* lors de la génération des enfants afin de permettre d'être proches de leurs parents tout en proposant un choix de variable cohérent. Une méthode pour cela serait de choisir une affectation de variable qui crée le moins de conflits dans les contraintes. Une telle approche aurait une complexité en temps au pire des cas de $O(m^2)$, $m$ étant le nombre de contraintes.
- Appliquer une pénalité pour les solutions qui ont des solutions qui lui sont identiques. Dans l'implémentation de cette méthode, nous penserons bien à ne pas abaisser une seule solution pour que le gène reste dominant.

### 2.2 Résultats préliminaires

Après exécution de l'algorithme, nous obtenons un résultat final violant en moyenne pour 100 individus et 25 générations :

| fichier | nb contraintes violées |
|---|---|
| uf20-01.cnf | 0.7 |
| uf20-02.cnf | 0.0 |
| uf20-03.cnf | 0.7 |
| uf20-04.cnf | 1.6 |
| uf20-05.cnf | 0.4 |
| uf20-06.cnf | 1.0 |

## 3 Conclusion & Perspectives

Ce document présente une méthode approchée pour résoudre le problème *SAT* en utilisant un algorithme évolutionnaire. Les résultats fournis par cette approche sont plutôt bons et permettent de résoudre des problèmes ayant près de 100 contraintes très facilement.

De plus, sont possibles un certain nombre d'améliorations telles que le *Hill Climbing* qui permet de choisir des individus encore plus efficacement, et une méthode de pondération qui permettrait d'éliminer les individus qui se ressemblent trop en vue d'améliorer l'exploration de l'espace de recherche.

## 4 Références


- `http://www.satcompetition.org/2009/format-benchmarks2009.html`