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 Informatique

Master 1 — SII — Groupe 2

Module : Méta-heuristiques et algorithmes évolutionnaires (Intelligence en essaim)

Rapport du TP :

Implémentation des solutions pour le problème de satisfiabilité (méthodes aveugles, heuristiques et méta-heuristiques)

Réalisé par :

BENAMARA SOUFIAN PRZEMYSLAW (201500008989)

DERARDJA MOHAMED ELAMINE (201500008274)

15 / 07 / 2019

Sommaire

Chapitre I – Problématique et généralités

1 Introduction

Le problème de satisfiabilité (SAT) a une importance capitale dans le domaine de la complexité mais également en l'intelligence artificielle. Il fut le premier problème à avoir été démontré comme un problème NP-Complet (par "Stephen Cook" en 1971). C'est donc le pionnier et l'ancêtre de tous les problèmes NP-Complets car à partir de SAT, la classe des problèmes NP-Complets a pu être concrétisée.

En effet, le problème SAT est représenté par une instance caractérisée par un nombre de variables qui appartiennent à un ensemble de clauses, chacune composée de littéraux, sous forme normale conjonctive (FNC). Le but (la solution) est de trouver une distribution (assignation de valeurs booléennes) de variables rendant toutes les clauses vraies dans la version originale du problème ou à maximiser les clauses satisfaites dans le cas du Max-SAT.

La nature du SAT peut conduire à croire qu'il est purement théorique. Cependant, ce problème a de nombreuses applications pratiques et cela grâce au théorème de "Karp" qui affirme que tout problème NP-Complet peut être réduit à tout autre problème NP-Complet (réductibilité entre les problèmes combinatoires). Ainsi, par exemple, une instance du Sudoku peut être transformée en une formule de logique propositionnelle à satisfaire.

Au fil du temps, un grand ensemble des approches ont été développées pour la résolution du problème SAT, en commençant par des méthodes exhaustives (lourdes et peu efficaces) passant par des heuristiques (difficiles à trouver, non générales qui diffèrent d'un problème à autre) et finissant pas les méta-heuristiques qui cherchent à trouver un compromis entre le temps d'exécution, ressources utilisées et le taux de satisfiabilité. D'ailleurs, des recherches scientifiques sont toujours publiées cherchant à améliorer la résolution du problème SAT.

2 Définitions

Le SAT est un problème de décision, fondé sur la logique propositionnelle, donc utilise sa terminologie (vocabulaire) par conséquence. Pour cela, des définitions fondamentales seront présentées pour que la suite soit plus claire.

● ***Variable propositionnelle :*** Une variable booléenne qui peut être soit vraie ou fausse, elle est l'élément fondamental des formules propositionnelles.

● ***Formule propositionnelle :*** Une expression construite à partir de connecteurs et de variables propositionnelles.

● ***Littéral :*** Une variable propositionnelle (littéral positif) ou sa négation (littéral négatif).

● ***Clause :*** Une disjonction de littéraux (forme normale disjonctive).

Ainsi, le problème SAT est une conjonction d'un ensemble fini de clauses.

3 Le solveur SAT

3.1 Objectifs

Ce projet implémente des différentes approches existantes pour la résolution du problème SAT, le but principal est de mesurer l'efficacité (en termes de temps et de taux de satisfiabilité) de chaque méthode en jouant sur ses paramètres et de comparer les résultats obtenus afin de conclure la spécificité (caractéristiques, complexité et la meilleure combinaison des paramètres) de chaque méthode et les cas adaptés pour son utilisation.

3.2 Environnement de travail

Les différents benchmarks (tests) sont réalisés avec une machine (PC) dotée des caractéristiques suivantes :

- Processeur : Intel Core i5-7200U @ 2.50 Ghz (2 cœurs physiques et 2 logiques).
- Ram : 8.00 Go (DDR3, 1600 Mhz).
- Système d'exploitation : Windows 10 Professionnel (Version 1809) x64 bits.

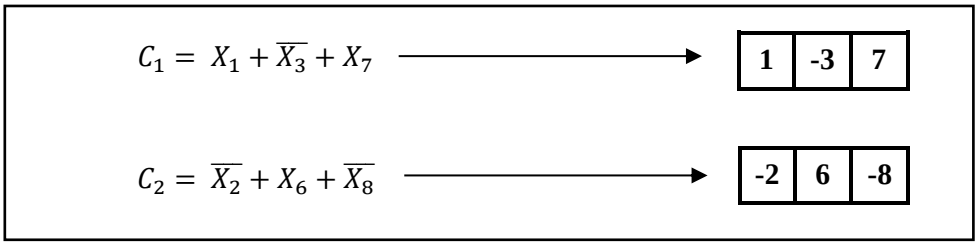
3.3 Structures de données générales

La bonne représentation des différents concepts liés au problème de satisfiabilité est primordiale pour aller le plus loin possible dans la résolution du SAT. Ainsi, le choix des structures de données efficaces garantit de tirer les meilleures performances de chaque approche. Dans ce qui suit, la description détaillée des structures générales adoptées pour l'application.

3.3.1 Représentation d'une clause

Un problème SAT contient un certain nombre de littéraux dans chaque clause. Ainsi, si "K" est le nombre de littéraux, le problème est noté "K-SAT" (K littéraux exactement dans chaque clause).

La modélisation d'une clause a été implémentée sous forme d'un vecteur dynamique contenant "K" cases (K connu au moment de l'exécution). Chaque case est un entier représentant l'indice du littéral, soit positif (pour une valeur de vérité : *vrai*) ou négatif (pour une valeur de vérité : *faux*).



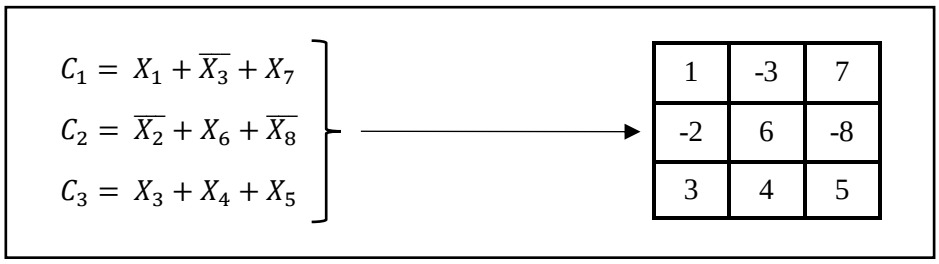
Exemple de la représentation des clauses pour un problème "3-SAT"

Cette structure est la plus adéquate pour la représentation d'une clause, car elle permet d'économiser l'espace mémoire (en représentant que les littéraux qui constituent la clause actuelle) et le temps (en cherchant un littéral d'une clause, il suffit de parcourir "K" éléments au maximum).

Une autre modélisation d'une clause est possible. Celle-ci consiste à modéliser un ensemble (du point de vue mathématique) de telle sorte qu'un vecteur représente tous les littéraux du problème SAT (tableau de valeurs) ; si ce dernier contient "N" littéraux différents, le vecteur sera de taille "2*N" (valeurs *vrai* et *faux*). Ainsi, une clause sera un tableau de "2*N" cases (vecteur représentatif), chaque case est un booléen (ou un bit, "0" ou "1") signifiant l'appartenance de ce littéral à cette clause. Il est clair que cette représentation est plus couteuse que celle proposée ci-dessus en termes d'espace (vecteur de "2*N" cases, au lieu de "K" cases) et temps (parcours de "2*N", au lieu de "K" cases au pire des cas).

3.3.2 Représentation d'un ensemble de clauses

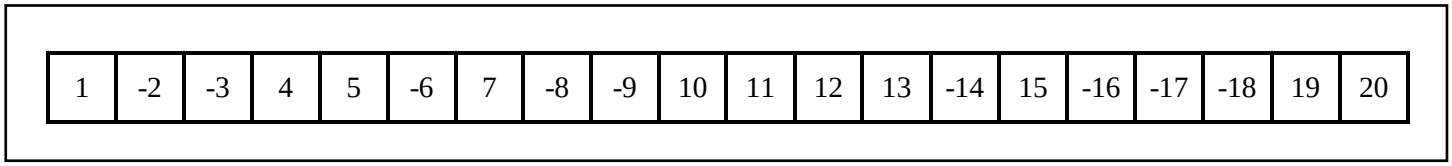
Un ensemble de clauses est le problème SAT lui-même (appelé : une instance). Ce dernier est implémenté dans une matrice dynamique de clauses (même raisonnement qu'une représentation d'une clause).



Exemple de la représentation d'un ensemble de clauses pour un problème 3-SAT

3.3.3 Représentation d'une solution

Une solution pour un problème SAT est un ensemble de valeurs associées aux variables qui permettent de satisfaire toutes les clauses du problème. Une solution potentielle est représentée par un vecteur contenant autant de cases que le nombre de différentes variables du problème SAT donné.



Exemple d'une solution d'un problème SAT contenant 20 différentes variables

Cette modélisation a été choisie à cause de sa lisibilité et facilité d'utilisation dans les différentes méthodes de résolution du SAT. En effet, la correspondance entre l'ensemble des clauses et la solution est clair, pour accéder à un littéral de la solution à partir d'un littéral d'une clause, il suffit de prendre la valeur absolue de ce dernier (et de retrancher un "1", si les cases commencent par l'indice "0").

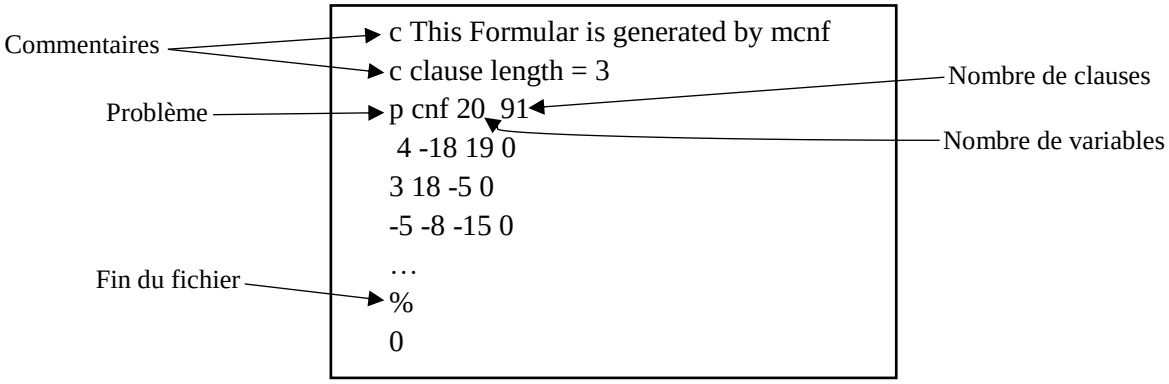
Une autre représentation est possible. Celle-ci consiste à utiliser un vecteur des booléens (ou des bits) dans le but d'économiser l'espace mémoire. Cependant, une solution est généralement non-volumineuse (l'ensemble des différentes variables du SAT est relativement petit), ainsi, la première solution est gardée à cause de sa lisibilité et facilité d'utilisation (accès aux littéraux de la solution).

3.4 Traitements générales

L'exploitation des structures de données précédentes est la prochaine étape qui permet d'utiliser ces structures dans le traitement pratique des entrées/sorties de l'application. Les opérations essentielles seront présentées.

3.4.1 Lecture d'un fichier de test

Une instance du problème SAT est contenue dans un fichier de type DIMACS CNF. Ce dernier est utilisé pour définir une expression booléenne écrite en forme normale conjonctive, en format ASCII, se présente comme suit :



Exemple d'une instance SAT définie par un fichier CNF

Les propriétés de ce type de fichier se résument en :

- Les fichier commence par des commentaires indiquant les caractéristiques de l'instance donnée. Une ligne pareille est identifiée par la lettre "c" minuscule.
- Ensuite, vient la ligne du problème (un "p" minuscule) qui indique le nombre de variables suivi par le nombre de clauses.
- Après, les clauses sont définies en listant les indexes de ses littéraux (positif pour *vrai*, négatif pour *faux*). La définition est terminée par la valeur "0".
- Enfin, le caractère "%" marque la fin du fichier.

Pour l'application, les clauses sont récupérées depuis un fichier CNF. Celui-ci est parcouru ligne par ligne afin d'extraire les littéraux de chaque clause. Les clauses (vecteurs dynamiques d'entiers) sont mises dans une matrice dynamique. Une fois les données de test en RAM, nous pourrons commencer la recherche des solutions pour le problème SAT.

3.4.2 Vérification de la satisfiabilité par une solution

L'algorithme de vérification du nombre des clauses satisfaites d'une solution consiste à parcourir l'ensemble des clauses (matrice, ligne par ligne) du problème SAT donné. Pour chaque clause (ligne de matrice) nous vérifions s'il existe un littéral qui a son même correspondant au niveau de la solution, si la correspondance existe avant d'arriver à la fin de la clause (parcourir tous les littéraux) actuelle, ceci veut dire que cette clause est satisfaite par la solution proposée, dans le cas contraire, elle n'est pas satisfaite. Le processus est clarifié par l'algorithme suivant :

```
Algorithme vérificationSatisfiabilité;
Entrée : I : Instance du problème SAT, S : Solution correspondante aux paramètres du SAT;
Sortie : Le nombre de clauses de I satisfaites par la solution S;
Var : numCL, numLt, numSat : entier;

Début;
    numCl := nombreClauses(I);      numLt := nombreLittérauxParClause(I);          numSat := 0;
    boucleExterne : Pour i := 0 à numCl faire   /* Parcourir toutes les clauses de l'instance "I" */
                        Pour j := 0 à numLt faire   /* Parcourir tous les littéraux de la clause actuelle */
                            Si ( S[valeurAbsolue(I[j])] = I[j] ) alors   /* Correspondance trouvée*/
                                numSat++;   /* Cette clause est satisfaite*/
                                aller à "boucleExterne";
                            Fsi;
                        Fait;
                    Fait;
    retourner(numSat);
Fin;
```

La complexité de cet algorithme est égale au produit des extrema des deux boucles "pour" (interne et externe). Ainsi, si "n" est le nombre des clauses de notre problème et "k" le nombre de littéraux de chacune d'elles (k-SAT), la complexité empirique temporelle dans le pire des cas (aucune clause n'est satisfaite par la solution en entrée) est : $\boldsymbol{n * k}$

Chapitre II – Méthodes exactes pour le SAT

1 Méthodes aveugles

Les techniques de recherche aveugle sont basées sur la vérification de toutes les solutions possibles d'un problème. Ainsi, toutes les combinaisons sont examinées aveuglement, sans aucune information aidante à la prise de décision. Ces approches sont les plus basiques pour résoudre les problèmes dans l'intelligence artificielle.

1.1 Recherche en largeur d'abord

1.1.1 Présentation

L'algorithme de recherche en largeur d'abord (*en Anglais :* Breadth-First Search, BFS) procède par la construction d'un graphe (plus précisément, un arbre) hiérarchique composé d'un ensemble de nœuds et des liens. Un chemin de parcours est établi suivant le principe FIFO (utilisation d'une file) où chaque niveau est examiné entièrement avant d'aller plus profondément dans la recherche. Par conséquence, les itérations de ce processus ne s'arrêtent que si un nœud apporte la solution (le cas de succès) ou que toutes les possibilités ont été examinées (le cas échéant).

1.1.2 Structures et implémentation

Comme mentionné dans la description de cette méthode, BFS utilise la structure de "file" pour stocker les nœuds crées tout au long de la recherche. Cette dernière, est une liste chaînée où les nouveaux éléments arrivés sont placés à la fin (queue) pendant que l'enlèvement concerne uniquement le premier élément (la tête). Cette file est nommée "ouvert" (*en Anglais :* open), elle signifie que ses nœuds sont des candidats, pas encore traités.

Pour le problème SAT, la modélisation est choisie pour qu'un nœud représente une solution tout entière, initialement vide, en ajoutant des littéraux, tirés au hasard, à chaque passage à un niveau plus profond. Le tirage des littéraux est fait de manière unique où chaque numéro de variable est choisi une seule fois dans la branche courante.

Dans le cas général, la recherche en largeur nécessite une autre liste nommée "fermée" (*en Anglais :* closed) qui gardera en mémoire la suite des nœuds développés. Ainsi, elle est mise à jour lors de l'avancement dans la branche courante (ajout d'un nœud) ou le changement de branche (suppression d'un nœud).

Le problème de satisfiabilité ne nécessite pas cette liste, vu qu'au passage à un niveau plus profond, un seul numéro de variable est tiré. Du coup, chaque nœud de l'arbre a exactement deux fils (arbre binaire) qui représentent le littéral choisi (fils gauche) et sa négation (fils droit).

Le schéma ci-dessus explique l'utilisation des structures choisies pour la recherche en largeur d'abord.

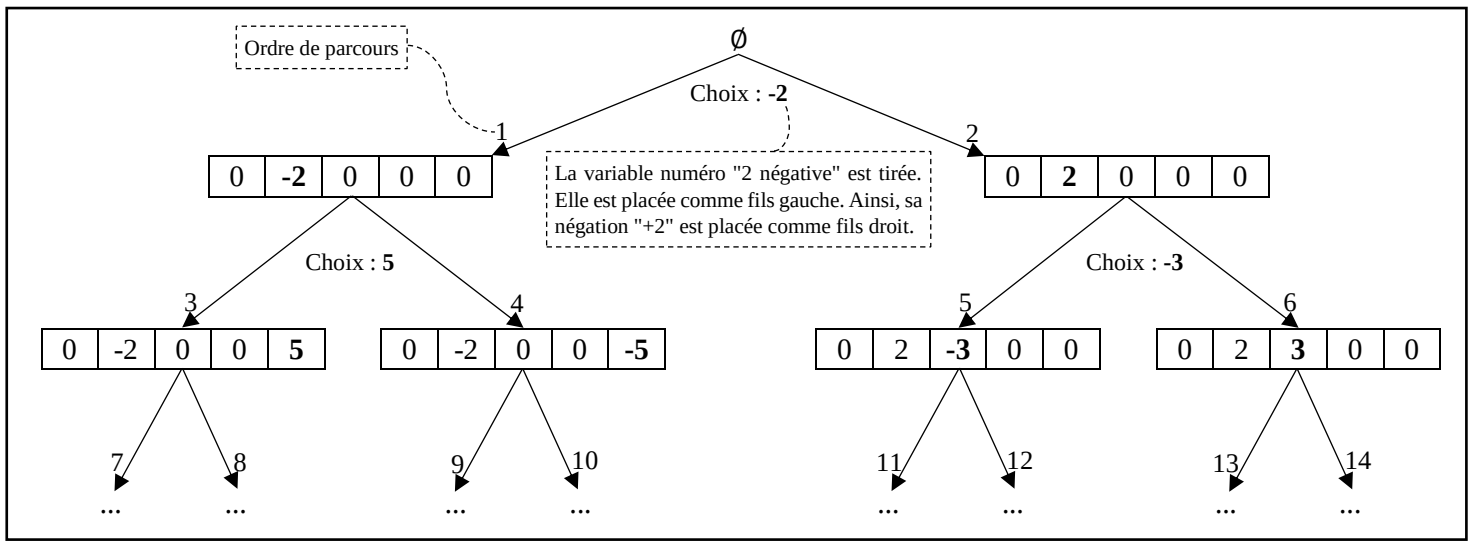


Figure qui montre l'arbre créé en utilisant la recherche en largeur d'abord pour un problème SAT avec 5 variables

L'étude de la complexité du BFS nécessite la définition des situations exactes de son comportement qui dépend d'un côté de l'instance SAT en entrée, et de l'autre du tirage au hasard des variables. Ainsi, deux cas se présentent :

● Meilleur cas : L'instance du problème SAT admet une solution avec un seul littéral qui sera choisi lors de la première itération.

● Pire des cas : L'instance du problème SAT n'est pas satisfiable (n'admet pas de solution).

La ***complexité empirique temporelle*** est calculée suivant le cas d'étude :

● Complexité temporelle au meilleur cas : Une seule itération suffit pour trouver la solution. Du coup, la complexité en notation asymptotique de Landau est constante, égale à : $\boldsymbol{O(1)}$.

● Complexité temporelle au pire des cas : Aucune combinaison n'est satisfiable, toutes les possibilités sont parcourues par la boucle. À chaque itération, deux nouvelles solutions sont ajoutées à la liste "open" (si elles ne représentent pas le dernier niveau). De ce fait, la complexité en notation asymptotique de Landau est exponentielle, égale à : $\boldsymbol{O(2^n)}$ ("n" étant le nombre de variables du problème SAT).

La ***complexité empirique spatiale*** est évaluée en fonction de la taille (nombre d'éléments) de la liste "open", comme suit :

● Complexité spatiale au meilleur cas : La solution est atteinte dès la première itération, aucun élément (nœud) n'est mis dans "open". Du coup, la complexité est constante, égale à : $\boldsymbol{O(1)}$.

● Complexité spatiale au pire des cas : Toutes les combinaisons sont essayées. Chaque itération rajoute deux nouvelles solutions à "open" avec la suppression de l'ancienne (le père). Ainsi, la taille maximale de "open" est atteinte au dernier niveau (numéro "n"). De ce fait, la complexité est exponentielle, égale à : $\boldsymbol{O(2^n)}$.

1.1.3 Expérimentations et performances

Pour les tests, l'ensemble des fichiers "uf75-325" ont servis comme le problème SAT (données d'entrée) à résoudre. Ces fichiers représentent un ensemble d'instances (exactement, 100), toutes satisfiables (admettent au moins une solution) dont les paramètres sont :

● Nombre de variables : 75

● Nombre de clauses : 325

● Taille d'une clause (nombre de variables par clause) : 3

Les expérimentations ont été faites sur uniquement les 5 premières instances, vu que le but est d'estimer le taux de satisfiabilité moyen, et donc ce nombre d'instances est largement suffisant pour le calculer.

Pour les méthodes de recherche exactes, les paramètres qui entrent en jeux sont seulement ceux d'arrêt (pas de hyperparamètre propre à la méthode elle-même) :

● Nombre d'essais : Nombre de tentatives pour une instance donnée (pour un calcul fiable de la moyenne).

● Temps par essai (en secondes) : Délais d'exécution d'une tentative (donc, une exécution s'arrête si une solution a été trouvée ou son délai est atteint).

Le tableau ci-dessous résume la globalité des tests.

Numéro de l'instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
1	5	5	41.4769249 %	5 sec
2	5	5	43.6307716 %	5 sec
3	5	5	40.7384644 %	5 sec
4	5	5	42.0307693 %	5 sec
5	5	5	41.3538437 %	5 sec
La moyenne	**5**	**5**	**41.8461548 %**	**5 sec**

1.2 Recherche en profondeur d'abord

1.2.1 Présentation

L'algorithme de recherche en profondeur d'abord (*en Anglais :* Depth-First Search, DFS) est une autre méthode pour traverser un arbre systématiquement dans le but de trouver la solution. Contrairement au BFS, le parcours suit le principe LIFO (utilisation d'une pile) où le dernier nœud créé est choisi pour être développé. Du coup, une branche est traversée jusqu'au bas (dernier niveau) avant de retourner à l'avant-dernier nœud. Notons, que cette stratégie est plus optimale en espace mémoire car elle s'occupe d'une seule branche avant d'aller aux autres.

1.2.2 Structures et implémentation

La méthode DFS utilise la structure de "pile" pour la sauvegarde des nœuds. Cette dernière est une liste spéciale où que la tête est concernée par les ajouts/retraits. De cette manière, le dernier élément arrivé est susceptible d'être supprimé. De plus, similairement au BFS, la "profondeur d'abord" utilise une liste "open" et "closed".

Pour le problème SAT, l'implémentation de "open" est un petit peu différente qu'au "largeur d'abord". Cette fois ci, un nœud représente l'ensemble de 3 informations (toutes de type "entier") :

- Valeur : Le littéral défini (numéro de variable, positif ou négatif).
- Antécédent : La valeur du père.
- Numéro du fils : Spécifie si c'est le premier (gauche, numéro 1) fils de l'antécédent ou bien le deuxième (droit, numéro 2).

Ce choix économise l'espace mémoire utilisé dans la recherche. En effet, trois entiers sont retenus par nœud au lieu d'une solution complète.

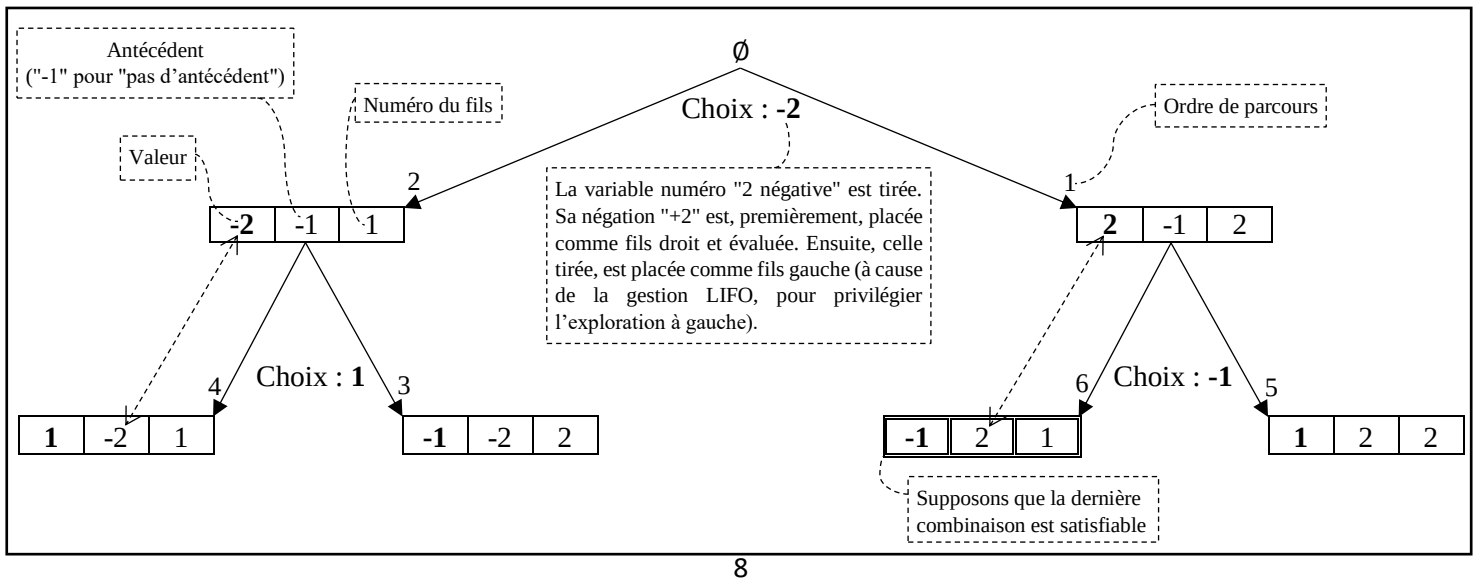


Figure qui montre l'arbre créé en utilisant la recherche en profondeur d'abord pour un problème SAT avec 2 variables

En ce qui concerne l'antécédent et le numéro du fils, ils sont utilisés dans la procédure de suppression récursive. En fait, suite à l'exploitation finale du premier fils (la branche qui passe par ce dernier est arrivée au dernier niveau) et que la solution n'a pas été trouvée, il faut passer automatiquement au deuxième fils, ce qui nécessite la suppression du premier, et ainsi de suite récursivement. Ce problème n'a pas été posé dans BFS car un nœud se constituait d'une solution complète.

La complexité du DFS est définie selon les mêmes deux cas que pour BFS (meilleur cas = solution trouvée à la première itération, pire des cas = solution inexistante).

La ***complexité empirique temporelle*** est calculée suivant le cas d'étude :

● Complexité temporelle au meilleur cas : Une seule itération pour trouver la solution. La complexité en notation asymptotique de Landau est constante, égale à : $\boldsymbol{O(1)}$.

● Complexité temporelle au pire des cas : Toutes les possibilités sont parcourues par la boucle. À chaque itération, deux nouvelles solutions sont ajoutées à la liste "open" (si elles ne représentent pas le dernier niveau). De ce fait, la complexité en notation asymptotique de Landau est exponentielle, égale à : $\boldsymbol{O(2^n)}$ ("n" étant le nombre de variables du problème SAT).

La ***complexité empirique spatiale*** est évaluée en fonction de la taille (nombre d'éléments) de la liste "open", comme suit :

● Complexité spatiale au meilleur cas : La but est atteint dès la première itération, aucun élément (nœud) n'est mis dans "open". Du coup, la complexité est constante, égale à : $\boldsymbol{O(1)}$.

● Complexité spatiale au pire des cas : Toutes les combinaisons sont essayées. Chaque itération rajoute deux nouveaux nœuds (un de la branche courante, l'autre à traiter ultérieurement) à "open". Ainsi, la taille maximale de "open" est atteinte au dernier niveau (numéro "n") où la branche actuelle est explorée complètement. Alors, la complexité est linéaire, égale à : $\boldsymbol{2n \rightarrow O(n)}$.

1.2.3 Expérimentations et performances

La comparaison ultérieure des différentes approches pour la résolution du problème SAT oblige d'utiliser le même fichier benchmark dans tous les tests. Ainsi, "uf75-325" est maintenu avec le même nombre d'instances (cinq) et mêmes paramètres (*nombre d'essais = 5, temps par essai = 5 secondes*). Le tableau suivant résume la globalité des tests.

Numéro de l'instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
1	5	5	91.5076904 %	5 sec
2	5	5	90.6461563 %	5 sec
3	5	5	90.2769318 %	5 sec
4	5	5	91.4461517 %	5 sec
5	5	5	90.1538467 %	5 sec
La moyenne	**5**	**5**	**90.8061554 %**	**5 sec**

2 Méthodes heuristiques

Les techniques de recherche heuristique représentent une amélioration des approches précédentes. Ces dernières, bien qu'ils soient simples à implémenter, elles souffrent d'une explosion combinatoire constatée dans leur lenteur ou les ressources requises pour leur mise en marche. De ce fait, les heuristiques présentent un avantage par l'utilisation des fonctions qui apportent une logique derrière le choix du chemin à emprunter pour trouver la solution plus rapidement. Cependant, le choix de l'heuristique adéquate est un problème lui-même car il nécessite une bonne compréhension du problème traité.

2.1 Algorithme A*

2.1.1 Présentation

Les algorithmes A-étoile, retrouvés souvent dans l'intelligence artificielle, combinent l'efficacité (recherche gloutonne) et l'optimalité (recherche à coût uniforme). Ils sont basés sur une fonction d'évaluation générale notée "f(n)" où la variable "n" est le nœud courant. Cette fonction est calculée en sommant deux mesures heuristiques :

- g(n) : Le coût de la chaine allant de l'état initial jusqu'à le nœud "n".
- h(n) : Une estimation du coût de la chaine reliant le nœud "n" à un état final.

2.1.2 Structures et implémentation

Comme les méthodes aveugles, le A* est une recherche arborescente qui utilise "open" pour le stockage des nouveaux nœuds créés, la différence est que cette liste est triée en ordre croissant selon la fonction "*f*". Ainsi, le prochain pas (exploration) dépend de l'évaluation heuristique qui vise à estimer le meilleur chemin pour atteindre le but.

Pour le problème SAT, une modélisation similaire au BFS est choisie, un nœud représente une solution complète. En ce qui concerne les mesures heuristiques, elles sont définies de manière suivante :

- g(n) : Le nombre de clauses satisfaites par le nœud courant et déjà satisfaites par son père (afin de défavoriser les branches qui satisfont les mêmes clauses).
- h(n) : Le nombre de clauses non-satisfaites par le nœud courant.

Le même pseudo-algorithme est maintenu qu'à la recherche en largeur d'abord, avec la différence que le choix de la tête de "open" est précédé par son tri.

Les cas d'étude de la complexité sont les mêmes qu'auparavant. La ***complexité empirique temporelle*** est calculée suivant le cas d'étude :

- Complexité temporelle au meilleur cas : Une seule itération pour trouver la solution. La complexité en notation asymptotique de Landau est constante, égale à : $\boldsymbol{O(1)}$.
- Complexité temporelle au pire des cas : Malgré la fonction heuristique, la spécification du nombre d'itérations reste impossible (la branche de recherche peut changer plusieurs fois selon l'évaluation). Ainsi, nous considérons que toutes les possibilités seront parcourues par la boucle. La complexité en notation asymptotique de Landau est exponentielle, égale à : $\boldsymbol{O(2^n)}$ ("n" étant le nombre de variables du problème SAT).

La ***complexité empirique spatiale*** est évaluée en fonction de la taille (nombre d'éléments) de la liste "open", comme suit :

- Complexité spatiale au meilleur cas : La but est atteint dès la première itération, aucun élément (nœud) n'est mis dans "open". Du coup, la complexité est constante, égale à : $\boldsymbol{O(1)}$.
- Complexité spatiale au pire des cas : Comme pour la complexité temporelle, toutes les combinaisons risquent d'être essayées. De ce fait, la complexité est exponentielle, égale à : $\boldsymbol{O(2^n)}$.

Notons que les deux cas présentés sont rares. Dans la pratique, le processus de recherche est meilleur (plus rapide et utilise peu d'espace mémoire) comparé aux méthodes aveugles. Malheureusement, l'étude du cas moyen est difficile.

2.1.3 Expérimentations et performances

Le fichier "uf75-325" est maintenu avec le même nombre d'instances (cinq) et mêmes paramètres (*nombre d'essais = 5, temps par essai = 5 secondes*). Le tableau résume la globalité des tests.

Numéro de l'instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
1	5	5	92.0615387 %	5 sec
2	5	5	92.2461472 %	5 sec
3	5	5	90.8307724 %	5 sec
4	5	5	92.6769257 %	5 sec
5	5	5	92.8000031 %	5 sec
La moyenne	**5**	**5**	**92.1230774 %**	**5 sec**

Chapitre III – Algorithme génétique (AG)

1 Méthodes méta-heuristiques

Les méta-heuristiques représentent une alternative aux méthodes exactes. Elles ont été conçues dans le but de pallier le problème de la non-efficacité des solutions classiques pour les problèmes complexes en trouvant un compromis entre le temps d'exécution, ressources utilisées et la qualité du résultat. Ces approches sont génériques (adaptables à tout problème), développées en s'inspirant de l'environnement naturel.

2 Algorithme génétique

2.1 Présentation

L'algorithme génétique (*en Anglais :* Genetic Algorithm, GA) adopte le concept de l'évolution, un phénomène biologique qui apparaît pendant la reproduction. Il démarre par un ensemble de gènes (solutions potentielles). Par la suite, une succession d'actions est répétée dans le but d'améliorer la qualité du groupe initial en choisissant deux candidats élus avec application du croisement (division et combinaison des parties de solutions), mutation (transformation aléatoire de quelques endroits de la solution) et évaluation de nouveaux membres obtenus (retenus s'ils sont meilleurs que d'autres ou bien perdus dans le cas contraire). Ainsi, ce processus est réitéré plusieurs fois tout en gardant la meilleure solution trouvée.

2.2 Implémentation

La projection du GA sur le SAT se traduise par la génération d'une population initiale avec solutions aléatoires différentes. Ensuite, une boucle au nombre défini des itérations est lancée et à chaque fois, deux individus tirés aléatoirement sont désignés pour la reproduction. Cette dernière est composée du croisement qui est effectué uniquement si un taux arbitraire est inférieur à une valeur fixe (définie comme paramètre du GA). Ce traitement consiste à diviser les solutions des individus précédents au même endroit hasardé en liant la 1^{ére} moitié de l'individu 1 avec la 2^{éme} moitié de l'individu 2 et vice-versa. Puis, une mutation des deux résultants est faite, de même que pour le croisement, un taux doit être petit qu'une valeur fixée préalablement. Cette opération est définie par l'inversion d'autant de littéraux que le pourcentage de la mutation (en entrée) par rapport au nombre total des variables. Enfin, si ces solutions modifiées sont meilleures que d'autres dans l'ensemble de début, elles les remplacent.

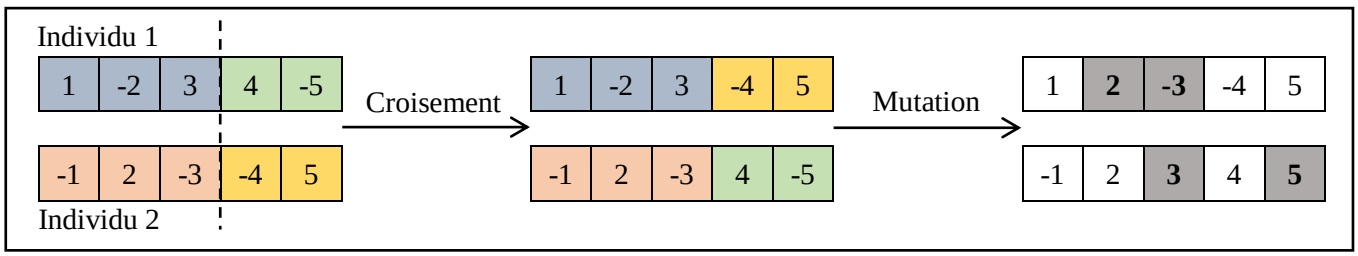


Figure qui montre le processus de reproduction (croisement et mutation) de deux individus

2.3 Expérimentations et performances

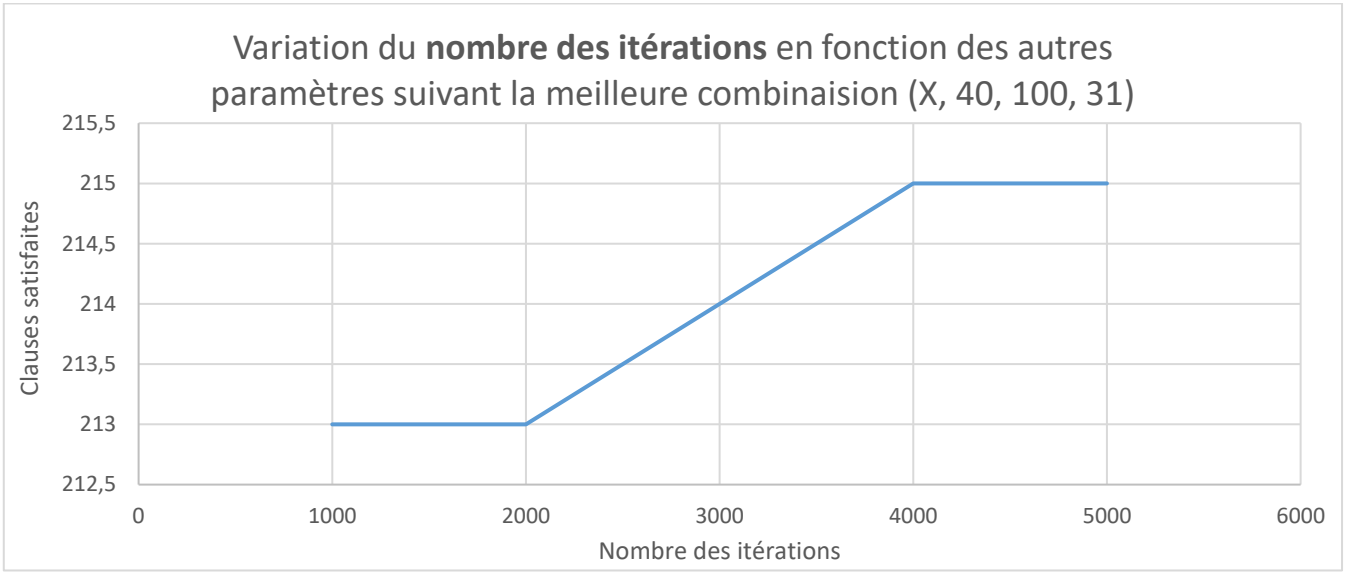
2.3.1 Réglage des paramètres

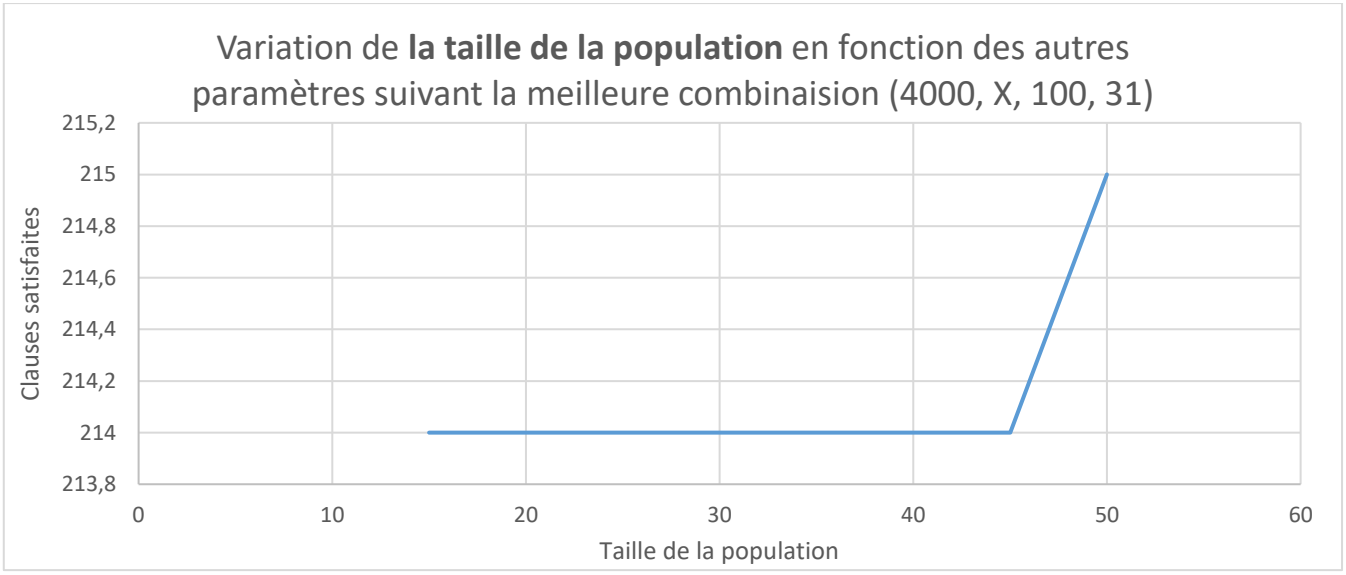
L'efficacité de toute méthode d'optimisation ne dépend pas seulement de son algorithme (idée d'implémentation), mais aussi de ses hyperparamètres. Ces derniers, choisis judicieusement, permettent d'exploiter les possibilités de la technique concernée au maximum.

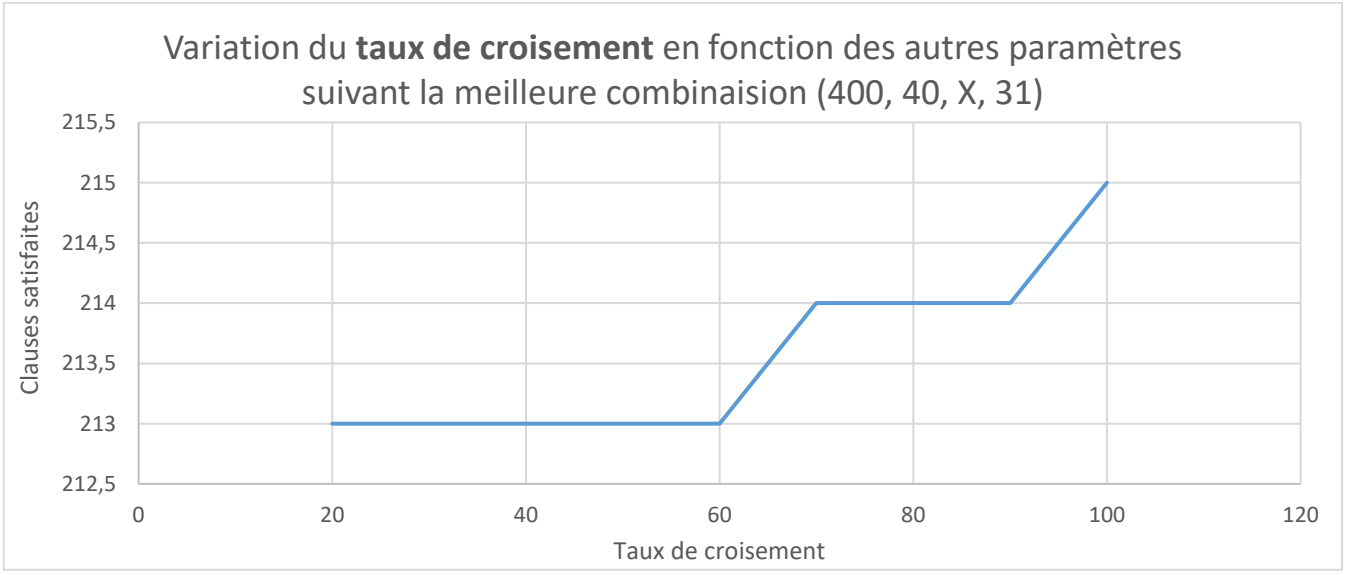
Pour cela, un ajustement des paramètres du GA est fait dans le but de trouver la meilleure combinaison des valeurs d'entrée. La procédure consiste à varier un seul paramètre (avec un pas stable) et fixer les autres à chaque fois en utilisant

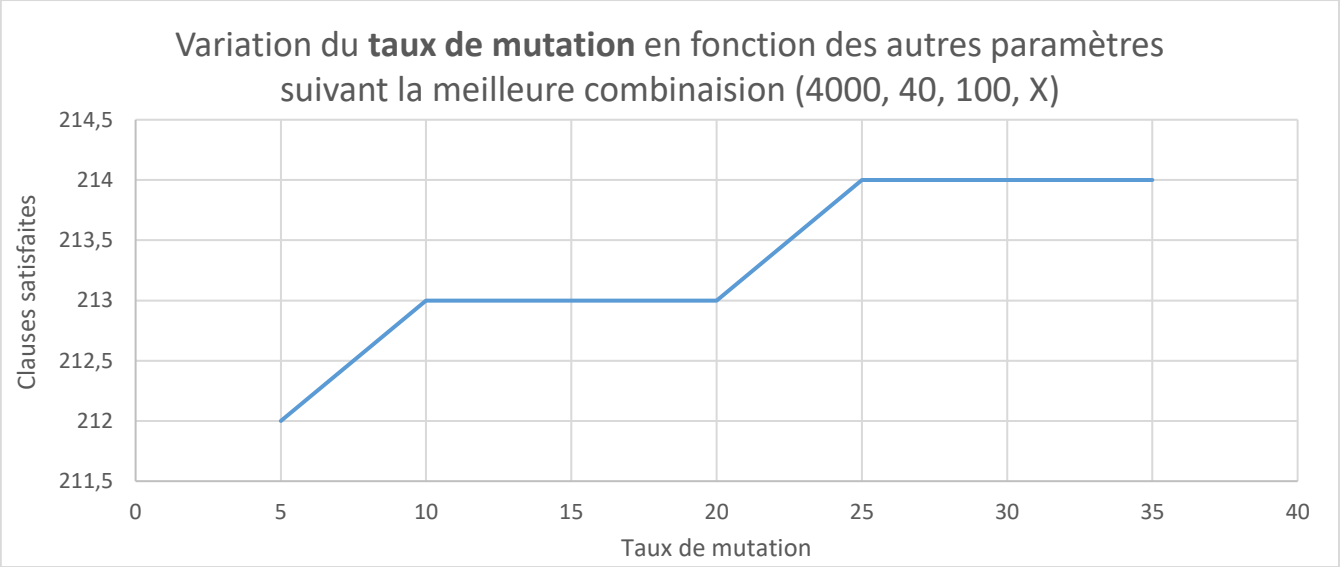
des boucles imbriquées. Ainsi, tous les résultats sont sauvegardés et analysés pour retrouver le meilleur arrangement des paramètres qui satisfait le plus des clauses dans un temps minimal. Notons que ce réglage a pris énormément de temps, compté en heures.

Les graphes ci-dessous résument les résultats les plus pertinents obtenus pour chaque hyperparamètre.









2.3.2 Tests et résultats

Comme pour les méthodes exactes, "uf75-325" est utilisé ; cinq instances avec *"nombre d'essais = 5"* et *"temps par essai = 5 secondes"*. Pour les hyperparamètres, ils sont ajustés sur les meilleurs valeurs trouvés lors de leur réglage :

- Taille de la population : 50
- Taux de croisement : 100%
- Taux de mutation : 31%
- Nombre des itérations : 3900

Ci-dessous, un tableau qui résume la globalité des tests.

Numéro de l'instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
1	5	5	97.969223 %	0.143 sec
2	5	5	98.4615402 %	0.096 sec
3	5	5	98.4000015 %	0.105 sec
4	5	5	98.5230713 %	0.114 sec
5	5	5	98.1538544 %	0.127 sec
La moyenne	**5**	**5**	**98.3015381 %**	**0.117 sec**

Chapitre IV – Particle Swarm Optimization (PSO)

1 Présentation

L'algorithme de l'optimisation par essaims particulaires (*en Anglais :* Particle Swarm Optimization, PSO) s'inspire du comportement des individus sociaux dans un groupe en général, et le déplacement (évolution de position) d'une clique des oiseaux en particulier. Il débute par la création d'un ensemble de particules aux positions aléatoires en sauvegardant cette dernière comme le meilleur emplacement visité. Par la suite, ces positions du groupe initial sont modifiées en jouant sur la vitesse (qui suit une équation mathématique) de chaque particule en actualisant leur emplacement ainsi que la meilleure position globale, le but étant d'explorer l'espace des solutions potentielles. Ces actions sont réitérées un nombre fini de fois.

2 Implémentation

L'adaptation du PSO au SAT est établie en créant une liste des individus avec solutions aléatoires en étant les meilleures localement (*en Anglais :* Particle's Best Position, pBest) pour chacun, et en actualisant le meilleur global (*en Anglais :* Global Best Position, gBest). Après, une boucle est exécutée autant de fois que la valeur de "maximum itérations" (paramètre en entrée) en changeant/améliorant à chaque fois pBest/gBest, et cela en ajustant la vitesse de chaque particule qui sera utilisée pour modifier la solution courante en inversant autant de bits que la valeur de la nouvelle vitesse.

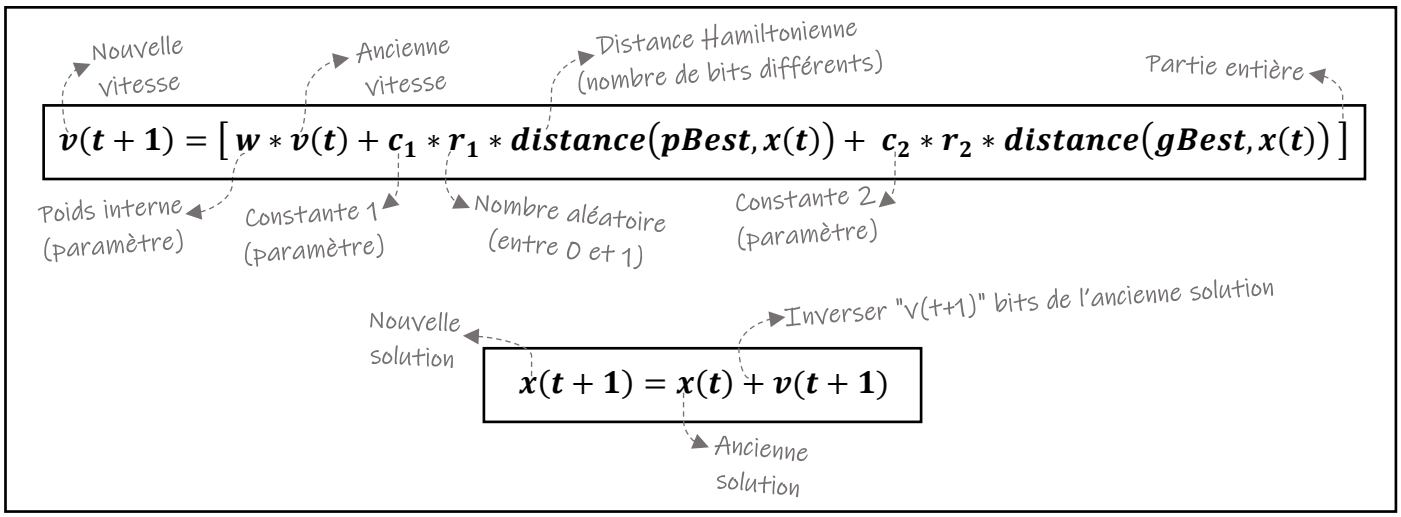


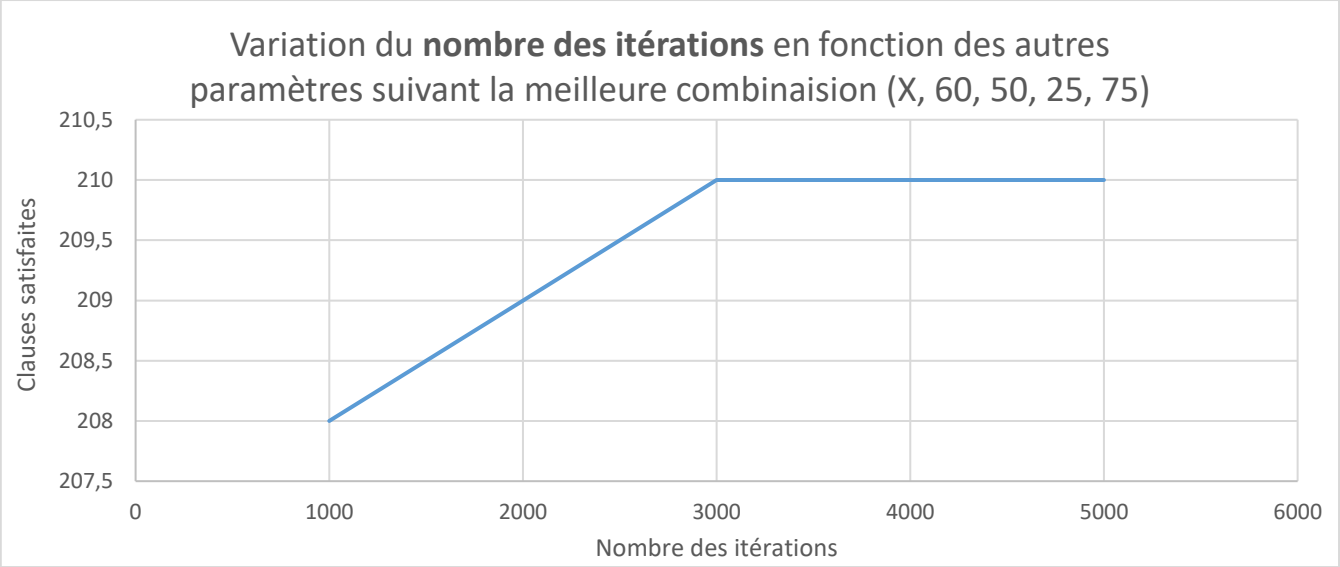
Figure qui montre les équations utilisées et leur signification dans PSO

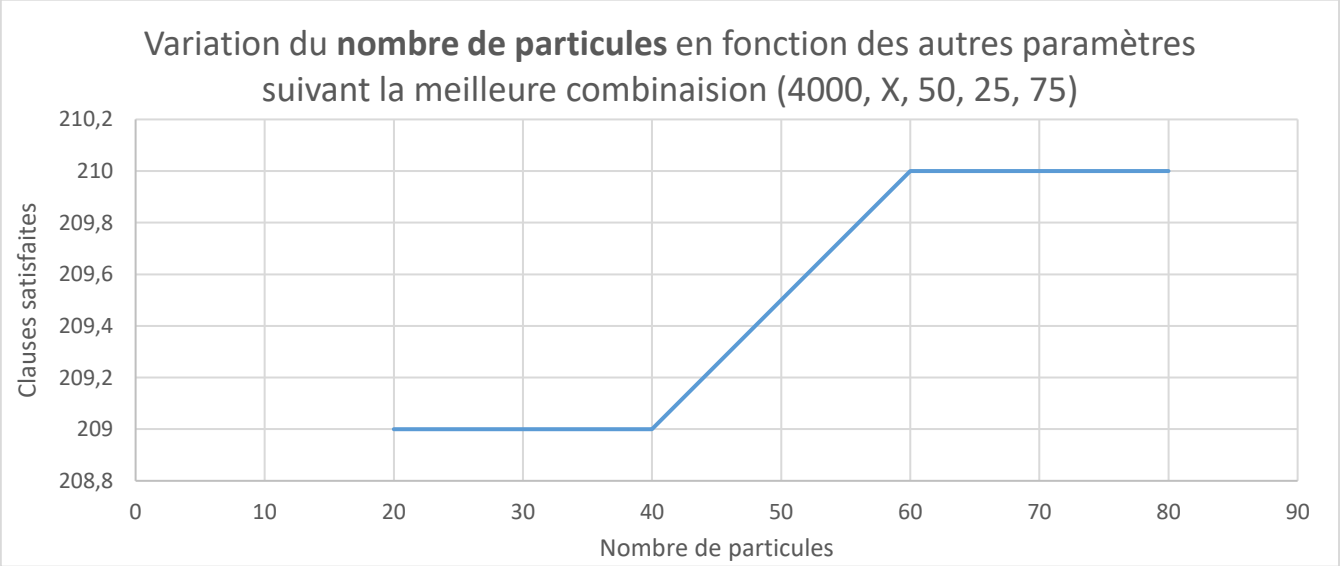
3 Expérimentations et performances

3.1 Réglage des paramètres

Comme pour le GA, un ajustement des paramètres est fait dans le but de trouver la meilleure combinaison des hyperparamètres en entrée du PSO.

Les graphes en bas résument les résultats les plus pertinents obtenus.





3.2 Tests et résultats

Similairement, les expérimentations sont faites sur "uf75-325" (*5 instances*, *nombre d'essais = 5* et *temps par essai = 5 secondes*). Pour les hyperparamètres, ils sont ajustés sur la meilleure combinaison de valeurs trouvés dans la phase du réglage :

- Nombre de particules : 60
- Première constante (c1) : 75
- Deuxième constante (c2) : 20
- Poids interne (w) : 50
- Nombre des itérations : 3000

Le tableau suivant résume les tests et leurs résultats.

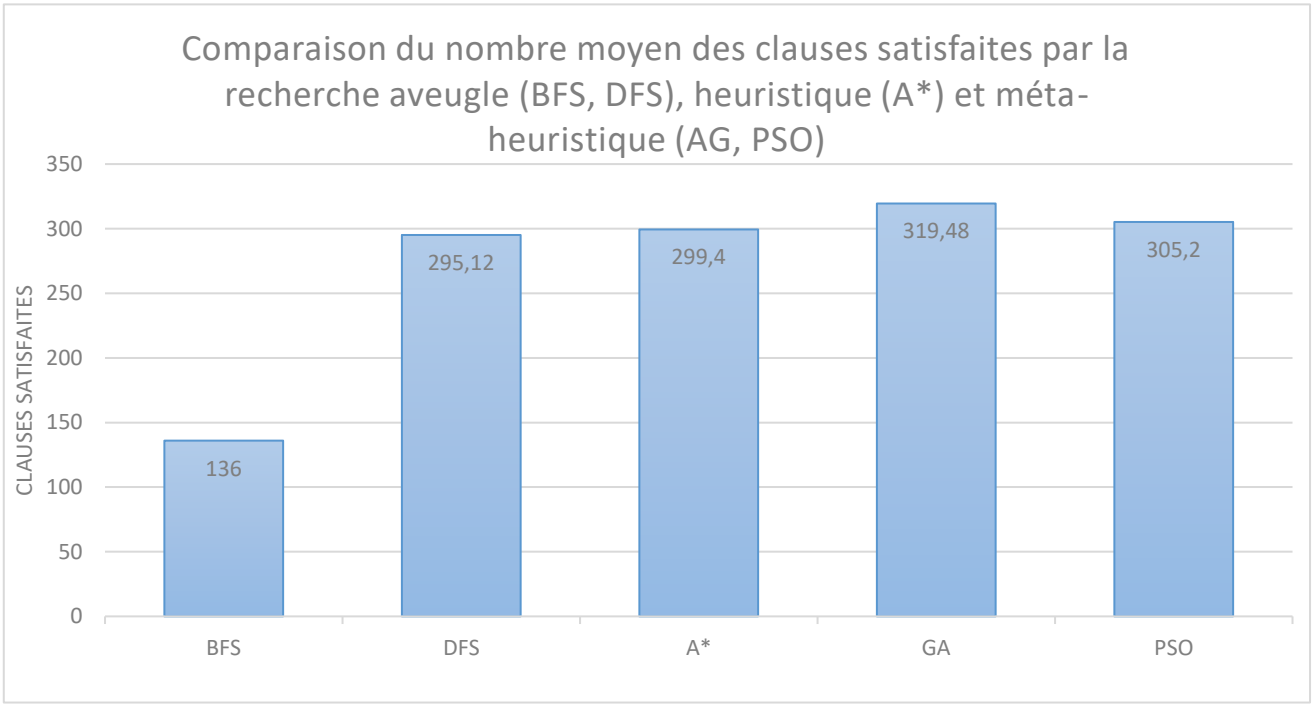
Numéro de l'instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
1	5	5	94.2769241 %	4.621 sec
2	5	5	94.030777 %	4.595 sec
3	5	5	93.6615372 %	4.603 sec
4	5	5	93.5384598 %	4.868 sec
5	5	5	94.030777 %	5 sec
La moyenne	**5**	**5**	**93.907695 %**	**4.7374 sec**

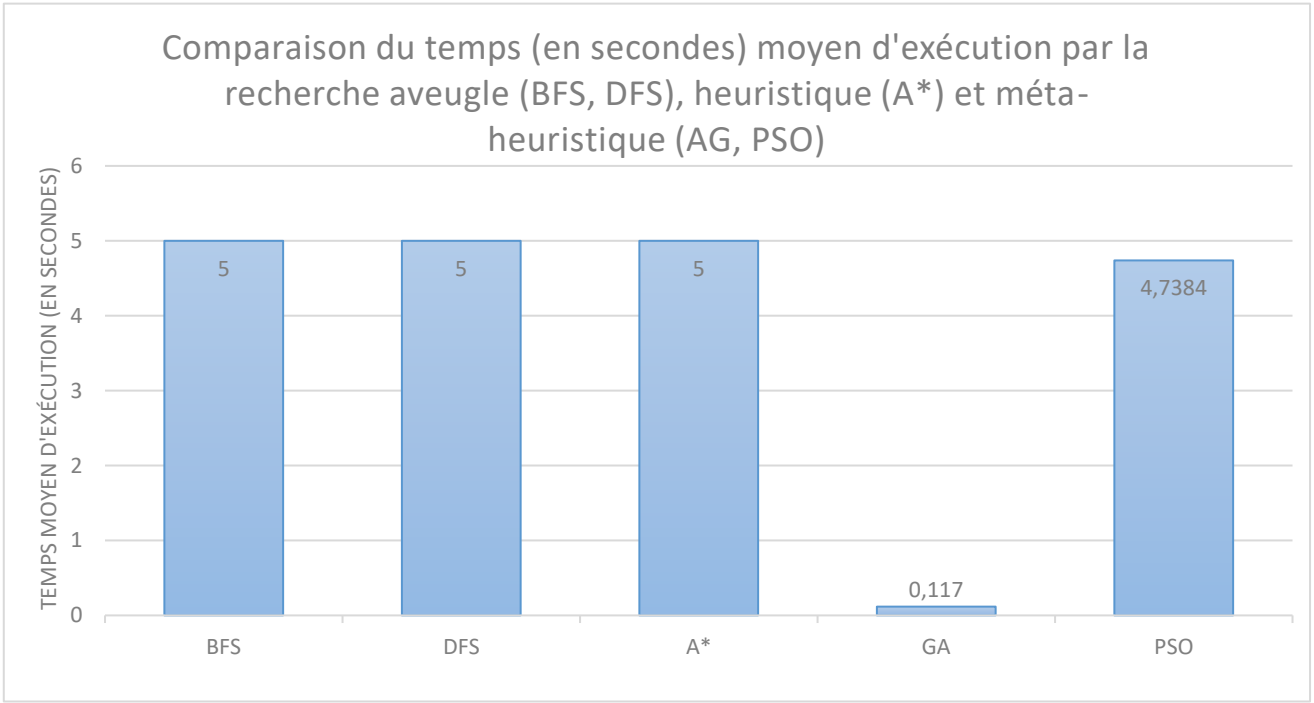
Chapitre V – Evaluation et implémentation

1 Comparatif des tests

Comme mentionné dans le premier chapitre, le but principal de l'implémentation des différentes approches pour résoudre le problème SAT est de mesurer leur efficacité. De ce fait, la comparaison des résultats obtenus par les tests des méthodes aveugles, heuristiques et méta-heuristiques est importante, car elle permet de tirer le meilleur algorithme de résolution SAT.

La comparaison touchera les deux critères de la performance : "nombre moyen de clauses satisfaites" et "temps moyen d'exécution". Les graphes statistiques associés ci-dessous.





L'analyse des deux histogrammes précédents montre l'essor des clauses satisfaites pour DFS par rapport à BFS (une hausse de 117%) avec le même temps d'exécution. Notons aussi une haute exploitation de la mémoire vive (RAM) par la méthode "en largeur d'abord" qui augmente très rapidement au cours du processus de la recherche. Ensuite, une légère évolution de satisfaction des clauses est notée en faveur de A* contre la meilleure recherche aveugle (une différence de 4.28 clauses). Ce résultat logique a été attendu vu l'utilisation d'une simple fonction heuristique.

Pour les approches méta-heuristiques, le GA est largement plus performant que son rival, le PSO. En effet, une augmentation des clauses satisfaites est remarquée pour GA par rapport à PSO, plus précisément, 14.28 clauses. En ce qui concerne l'autre critère, le temps, une chute de 4.6214 secondes est aperçue en faveur de GA, bien-évidemment.

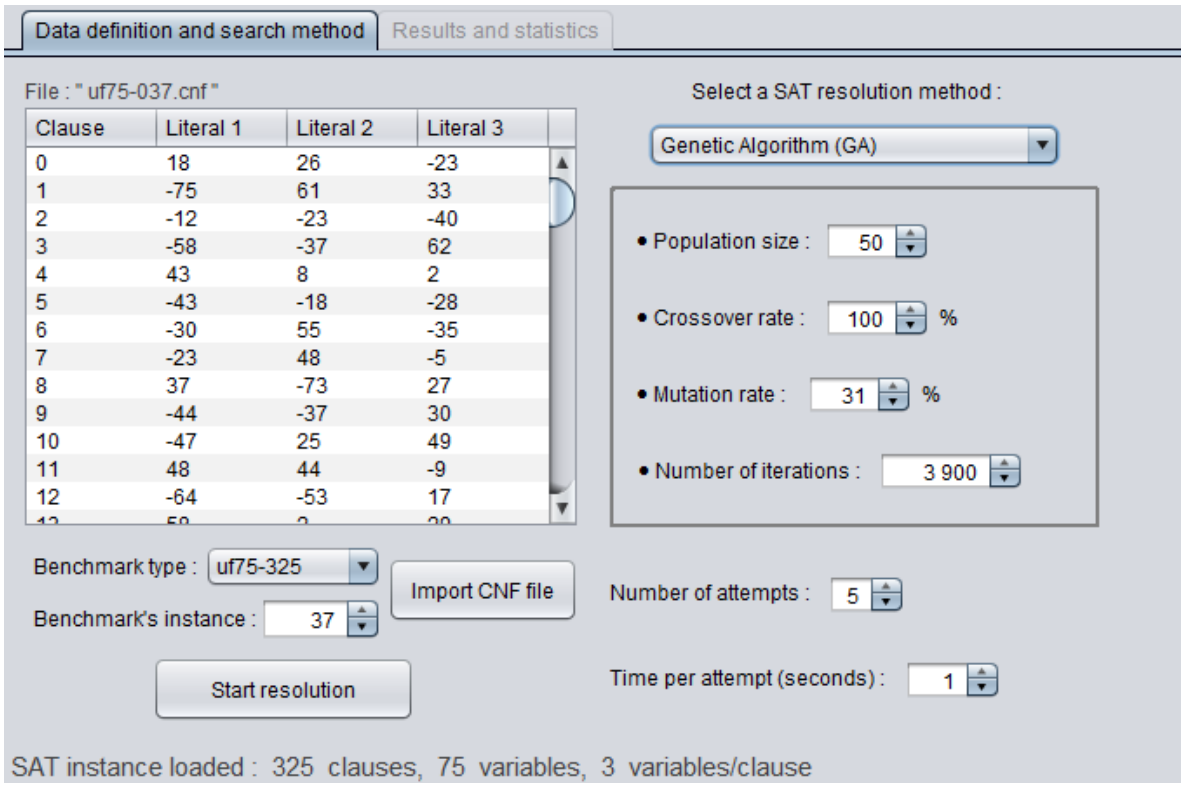
Enfin, la comparaison des recherches exactes avec celles méta-heuristiques est claire, ces dernières sont tout simplement meilleures que les autres, que ce soit en "nombre moyen de clauses satisfaites" (différence allant jusqu'à 183.48 clauses) ou bien "temps moyen d'exécution" (écart de 4.883 secondes).

2 Interface de résolution

L'outil de résolution conçu est programmé entièrement avec le langage orienté objet JAVA (JDK, version 8). Pour l'interface graphique (GUI), elle est construite en utilisant la bibliothèque SWING, bien qu'elle ne soit plus améliorée, elle reste plus légère en la comparant avec JavaFX.

Le développement d'une simple interface graphique est fait dans le but de faciliter à l'utilisateur final l'interaction avec notre application. Elle est composée de deux onglets principaux qui se présentent comme suit :

2.1 Définition de données et méthode de recherche



Les différentes fonctionnalités sont :

- Importer une instance du problème SAT (fichier de test) et afficher son contenu (clauses et ses littéraux) dans un tableau.
- Choix de la méthode de résolution et ses paramètres (propres à chacune d'elles).
- Options globales qui déterminent la durée totale du test.
- Bouton de début de résolution en utilisant l'approche choisie.

2.2 Résultats et statistiques

Après la résolution, un rapport final en détails est présenté :

- Le taux (pourcentage) de satisfiabilité et le temps moyen par essai.
- Un histogramme avec les clauses satisfaites et le temps pris par chaque tentative.

3 Conclusion

Ce projet nous a permis d'approfondir nos connaissances en se familiarisant avec des approches sophistiquées pour résoudre des problèmes complexes. Ces derniers, souvent NP-Complets, n'étant pas solvables avec les techniques classiques (exactes). Pour cela, les méthodes méta-heuristiques sont utilisées cherchant à trouver un compromis entre les ressources utilisées, temps d'exécution et la qualité de la solution.

De plus, l'étude approfondie du problème SAT nous a donné la chance de découvrir son importance capitale dans l'étude théorique de la complexité.